Application Development Solutions

COBOL ANSI-85
Programming Reference Manual

Volume 1:
Basic Implementation

ClearPath MCP Release 8.0

February 2003

Printed in USA
8600 1518–307

Application Development Solutions

COBOL ANSI-85
Programming Reference Manual

Volume 1:
Basic Implementation

UNISYS

ClearPath MCP Release 8.0

February 2003

Printed in USA
8600 1518–307

Correspondence regarding this publication can be e-mailed to **doc@unisys.com**.

Application Development Solutions

COBOL ANSI-85

Programming Reference Manual

Volume 1: Basic Implementation

ClearPath MCP Release 8.0

8600 1518–307

Application Development Solutions

COBOL ANSI-85

Programming Reference Manual

Volume 1: Basic Implementation

ClearPath MCP Release 8.0

8600 1518–307

Bend here, peel upwards and apply to spine.

Contents

Section 1. Program Structure and Language Elements

Section 2. Identification Division

Section 3. Environment Division

Section 4. Data Division

Section 5. Procedure Division Concepts

Section 6. Procedure Division Statements A–H

Section 8. Procedure Division Statements S–Z

Section 9. Intrinsic Functions

Section 10. Interprogram Communication

Section 13. Tasking in COBOL85

Section 14. Report Writer

Section 15. Compiler Operations

Section 16. Internationalization

Appendix A. Output Messages

Appendix B. Reserved Words

Appendix C. Interpreting General Formats

Appendix D. Using the Checkpoint/Restart Utility

Appendix E. COBOL Binding

Appendix F. Comparison of COBOL Versions

Appendix G. COBOL Migration

Appendix H. Migrating V Series Intrinsics

Figures

Tables

Tables

Section 1
Program Structure and Language Elements

About This Manual

Common Business-Oriented Language (COBOL) is a programming language that enables a programmer to write computer instructions in a language much like standard English. This implementation of COBOL85 follows the American National Standard Programming Language COBOL ANSI X3.23-1985.

This manual provides the complete COBOL85 syntax and the extensions to COBOL85.

Information concerning the interface between COBOL ANSI-85 and various products is located in Volume 2 of this manual, subtitled, "Product Interfaces."

This section describes the

- Components of a source program
- Rules for entering the components in the source program
- The COBOL character set
- Punctuation characters used as separators
- Various elements that make up the language, such as COBOL words, identifiers, literals, and figurative constants

Purpose

This manual explains the syntax and concepts of this implementation of the Common Business-Oriented Language (COBOL) ANSI-85.

Audience

The primary audience for this manual includes programmers and systems analysts who are experienced in developing, maintaining, and reading COBOL programs. The secondary audience consists of technical support personnel and information systems management. A possible tertiary audience includes programmers who are learning COBOL; however, note that the manual is not designed for this audience.

Conventions

Throughout this manual, Unisys extensions to the American National Standard for Programming Language COBOL, ANSI X3.23-1985 are highlighted.

In addition, the term *ClearPath MCP servers* refers to ClearPath NX, LX, CS, and Libra Series servers.

Unless otherwise stated, the term *Windows* is used in this book to refer to Windows NT Server 4.0; Windows NT Server 4.0, Enterprise Edition; Windows 2000 Server; and Windows 2000 Advanced Server.

Acknowledgments

COBOL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations.

No warranty, expressed or implied, is made by any contributor or by the CODASYL Programming Language Committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contributor, or by the committee, in connection therewith.

The authors and copyright holders of the copyrighted material used herein have specifically authorized the use of this material, in whole or in part, in the COBOL specifications. These authors or copyright holders are the following:

IBM: IBM Commercial Translator Form No. F 28-8013 (copyright 1959)

Minneapolis-Honeywell: FACT, DSI 27A5260-2760 (copyright 1960)

Sperry Rand Corporation: FLOW-MATIC, Programming for the UNIVAC (R) I and II, Data Automation Systems (copyright 1958, 1959)

Such authorization extends to the reproduction and use of COBOL specifications in programming manuals or similar publications.

Program Overview

The program written in COBOL is called the *source program*. Before a computer can read a source program, the COBOL instructions must be translated into machine language. This translation is the job of the COBOL compiler. First, the COBOL compiler verifies that the source program satisfies all the rules of the COBOL language. Then the compiler translates the COBOL instructions to machine language and produces an *object program* that contains the translated instructions.

After compilation, the COBOL compiler prints a copy of the source program and lists any compilation errors on that printout. If program corrections are necessary, you can make the appropriate changes in the source program and then recompile it.

Divisions of a Source Program

A COBOL source program consists of four parts called *divisions*. Each division has a heading and can contain one or more sections or paragraphs, which are constructed and combined according to specific rules. The divisions of a COBOL source program must occur in the order shown.

Division	Purpose
Identification Division	Identifies and describes the program.
Environment Division	Identifies file processing requirements, hardware requirements, computers used, nonstandard internal memory allocation for files, and notation used throughout the program. This division is optional in certain programming situations.
Data Division	Describes the data elements that the object program is to manipulate or create. These data elements can be constants or items within files, records, or program work areas. This division is optional in certain programming situations.
Procedure Division	Defines the steps needed to accomplish a desired task using the data defined in the Data Division. This division is optional in certain programming situations.

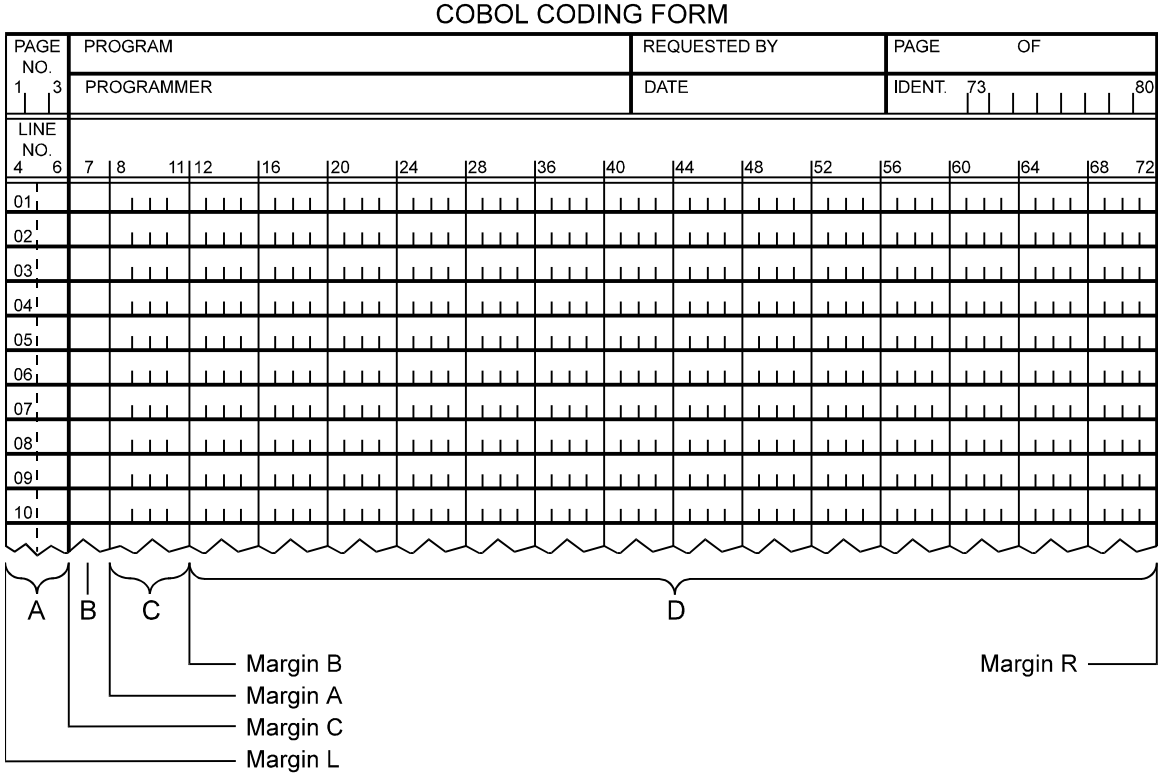
Contents of a Division

Divisions can contain one or more sections. A section is made up of paragraphs, which are formed by a variety of sentences, statements, clauses, phrases, and words. The following table describes the language elements that make up a COBOL85 division.

Element	Description
Section	A section consists of a section header optionally followed by one or more entries in the Data Division or one or more paragraphs in the Environment and Procedure divisions.
Paragraph	In the Identification and Environment Divisions, a paragraph consists of a paragraph header optionally followed by one or more entries. In the Procedure Division, a paragraph consists of a paragraph-name with a separator period at the end, optionally followed by one or more sentences.
Clause	A clause is an ordered set of consecutive COBOL character-strings that specify an attribute of an entry.
Phrase	A phrase is an ordered set of consecutive COBOL character-strings that form a portion of a COBOL procedural statement or a COBOL clause.
Sentence	A sentence is a sequence of one or more statements, the last of which is terminated by a separator period.
Statement	A statement is a syntactically valid combination of words, literals, and separators that begins with a verb.
Word	A COBOL word is a string of a maximum of 30 characters. The valid types of COBOL words and the rules for forming them are described later in this section under the heading "Types of COBOL Words."
Separator	A character or a space that is used to punctuate a portion of a COBOL program.

Reference Format

The COBOL compiler expects the components of your source program to appear in specific areas along a line of code. Each line has 72 columns, which are grouped into five areas. This line-formatting scheme is referred to as the *reference format*. Specific portions of a program must be placed in each area on the coding form. Predesigned coding forms are available to assist you in structuring lines of code in the correct way. An example of a coding form is shown in Figure 1–1.



A.	1 - 6	Sequence field. Used for a sequence number.
B.	7	Indicator area. Used to denote a continuation line (-), a comment line (*or /), a debugging line (D), or a dollar option ($).
C.	8 - 11	Area A. Items which must begin in this area are: division headers; section headers; paragraph headers; the key words DECLARATIVES and END DECLARATIVES; level indicators FD and SD; and the level numbers 01 and 77.
D.	12 - 72	Area B. All other items must begin and end at some position in this area.

Figure 1–1. Sample of COBOL Coding Form

Table 1–1 describes the areas in which specific information must be placed in a line of code.

Table 1–1. Areas of a Line of Code for Columns 1–72

Columns	Area	Description
1–6	Sequence field	You can put a sequence number in this area to label a source program line. The sequence number can consist of any character in the character set of the computer. The content of the sequence number area does not need to be unique or to have any particular sequence.
7	Indicator area	You can place a symbol in this area to indicate that the succeeding line is of a specific type. The types of lines and the symbols used to denote them are as follows: To denote a . . . Continuation line, use a hyphen (-) Comment line, use an asterisk (*) or a slash (\) Debugging line, use the letter D Compiler control option, use a dollar symbol ($) When the FREE compiler option is • Set, any character in column 7 other than an asterisk, slash, hyphen, dollar sign, or space is treated as part of the source image. • Reset, any character in column 7 other than an asterisk, slash, hyphen, or dollar sign is treated as a space.
8–11	Area A	The items that must begin in this area are as follows: • Division, section, and paragraph headers • The keywords DECLARATIVES and END DECLARATIVES • The level indicators FD and SD • The level numbers 01 and 77
12–72	Area B	Items not placed in the other areas must begin and end at some position in this area.

Division and Section Headers

A division header is a combination of words, followed by a separator period, that indicates the beginning of a division.

A section header is a combination of words, followed by a separator period, that indicates the beginning of a section. Section headers are used in the Environment, Data, and Procedure Divisions. In the Environment and Data Divisions, a section header is composed of reserved words followed by a separator period.

The valid section headers for the Environment Division are

```
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
```

The valid section headers for the Data Division are

```
FILE SECTION.
DATA-BASE SECTION.
WORKING-STORAGE SECTION.
LINKAGE SECTION.
COMMUNICATION SECTION.
LOCAL-STORAGE SECTION.
REPORT SECTION.
PROGRAM-LIBRARY SECTION.
```

In the Procedure Division, a section header consists of a user-defined section-name followed by the reserved word SECTION. The section header must end with a period.

Note: *The compiler ignores segment numbers that follow the reserved word SECTION in a section header. You can retain the segment numbers, but it is recommended that you make the Segmentation module a comment for the sake of clarity. Note that the Segmentation module has been placed in the obsolete element category.*

Level-Numbers

A level-number is a one- or two-digit number that indicates the hierarchical position or a special characteristic of a data item. Level-number 1 is typically used for a line that identifies a record. Level-numbers 2 through 49 typically specify fields within the record. Level numbers 66, 77, and 88 have specific meanings in COBOL and identify a special property of the data in the field.

When specific level-numbers appear in a general format, COBOL requires that you use those level-numbers in your COBOL program. For more information on level-numbers, refer to Section 4.

Example

The following example shows an input record for a magazine subscription list. The 01-level entry identifies the record. The 05-level entries identify a name and an address field within the record. The 07-level entries specify the content of each field in the record. Note that you could use any numbers from 02 through 49 in place of 05 and 07.

```
01  MAGAZINE-SUBSCRIPTION-INPUT-RECORD.
     05  NAME.
         07  FIRST                          PIC X(10).
         07  MIDDLE-INITIAL                 PIC X(2).
         07  LAST                           PIC X(13).
     05  ADDRESS.
         07  STREET                         PIC X(12).
         07  CITY                           PIC X(10).
         07  STATE-ABBREV                   PIC X(2).
         07  ZIP-CODE                       PIC x(10).
```

Special-Purpose Lines—Fixed Indicators

In addition to standard lines of code, there are several special-purpose lines that you can include in a source program. Special-purpose lines are usually designated by a special character in the indicator area (column 7) of the line. The types of special-purpose lines and their associated characters are as follows:

Type of Line	Special Character
Comment line	Asterisk (*) or slash (/)
Continuation line	Hyphen (-)
Debugging line	Letter D (D)
Compiler control option line	Dollar sign ($)
Blank line	Blank

Continuation Lines

Sometimes a line of code requires more than the 72 characters allocated on a coding form. You can continue any entry, including a sentence, phrase, clause, word, literal, or PICTURE character-string onto a subsequent line. The subsequent line is called a continuation line.

Designating a Continuation Line

You designate a continuation line by placing a hyphen (-) in the indicator area (column 7) of a line. The hyphen indicates that the first nonblank character in area B (columns 1272) of the continuation line follows the last nonblank character of the preceding line (with no spaces). If the indicator area of a line does not contain a hyphen, the compiler assumes that a space precedes the first nonblank character in the line. Area A of a continuation line must be blank.

Rules

You can use successive continuation lines. Also, you can place comment lines and blank lines between a line and its continuation lines.

Double-byte names must be placed completely on a single line. You cannot continue some of the characters of a double-byte name onto a continuation line.

When you use continuation lines with pseudocode, note that the characters that compose the pseudocode designator (==) must be on the same line.

When you use a continuation line with a nonnumeric literal, an undigit literal, or a national literal, observe the following rules:

- Use all 72 columns of the line to be continued. All spaces at the end of the line are considered to be part of the literal.
- Do not place a quotation mark (") or a commercial at sign (@) in column 72 of the line to be continued. Doing so delimits the literal and prevents it from being continued.
- Enter a quotation mark (for a nonnumeric literal), a commercial at sign (for an undigit literal), or the delimiter *N"* (for a national literal) as the first nonblank character in area B. The literal continues with the character immediately following the quotation mark or commercial at sign.

Examples

The following example shows how a SELECT statement is continued over two lines.

```
200100    SELECT MASTERFILE ASSIGN TO DISK OR
200110-   GANIZATION IS SEQUENTIAL.
```

The following example assumes that the Y in the word KEY is in column 72, the end of Area B. The literal must end with a quotation mark. Thus, a continuation line is needed that begins with a quotation mark (to signify a nonnumeric literal) and ends with a quotation mark (to end the literal).

```
200120  01   WARNING-MESSAGE PIC X(24) VALUE IS "WRONG ENTRY FOR THIS KEY
200130-      "  " .
```

Comment Lines

A comment line is any line with an asterisk (*) or a slash (/) in the indicator area (position 7) of the line. A comment line can appear as any line in a source program after the Identification Division header and as any line in library text of a COBOL library. You can include any combination of the characters from the computer's character set, including national standard data format characters, in area A and area B of a comment line.

A slash in the indicator area causes page ejection before the comment line if the listing of the source program is printed. An asterisk in the indicator area causes the line at the next available line position in the listing to be printed. The asterisk or slash and the characters in area A and area B appear on the listing but serve as documentation only. For example, if you want a heading at the top of a page, type a slash in the indicator area and the heading in areas A and B. The compiler does not perform a syntax check on comment lines.

Floating Comment Indicator

A comment indicator, signified by the symbols *>, is used to indicate the following:

- A comment line, when specified as the first character-string in the program-text area
- A floating inline comment, when specified following one or more character-strings in the program-text area, subject to the following conditions:
  - The floating comment indicator of an inline comment must be preceded by a separator space; it can be specified wherever a separator space can be specified.
  - For purposes of analyzing the text of a compilation group, a space is implied immediately following a floating comment indicator.
  - When a floating comment indicator is present, the rest of the line is treated as a comment.
  - All the characters that form a multiple-character floating comment must be specified on the same line.

Debugging Lines

A debugging line is any line with a D in the indicator area (column 7) of the line. A debugging line with spaces in columns 8 through 72 is considered to be the same as a blank line. You can enter a debugging line anywhere after the OBJECT-COMPUTER paragraph.

Debugging lines are used when the debugging module is activated. The debugging module is activated when you specify the WITH DEBUGGING MODE clause in the SOURCE-COMPUTER paragraph. If you do not activate the debugging module, the compiler treats a debugging line like a comment line. Thus, you should make sure that your program is syntactically correct when the debugging lines are considered to be comment lines. $FREE must be reset to compile debugging lines.

You can use successive debugging lines, and you can continue debugging lines. Each continued debugging line must contain a D in the indicator area. Character-strings cannot be continued across multiple lines.

Example

The following example shows the use of debugging lines.

```
010000 IDENTIFICATION DIVISION.
100000 ENVIRONMENT DIVISION.
100050 CONFIGURATION SECTION.
100100 SOURCE-COMPUTER  MICROA WITH DEBUGGING MODE.
        .
        .
        .
100600 WORKING-STORAGE Section.
100700D77  PERFORMANCE-COUNT          PIC 9(4).
100800D77  BAD-RECORDS                PIC 9(4).
```

```
100900D77  RATIO                      PIC 9(4) 99.
        .
        .
        .
101000 PROCEDURE DIVISION.
102000 OPEN-IT.
102100     OPEN INPUT GUEST-FILE.
103000D    MOVE ZEROS TO PERFORMANCE-COUNT, BAD-RECORDS, RATIO.
104000 READ-IT.
104100     READ GUEST-FILE AT END GO TO FINISH-IT.
105000D    ADD 1 TO PERFORMANCE-COUNT.
106000D    IF IN-KEY NOT NUMERIC ADD 1 TO BAD-RECORDS.
        .
        .
        .
107000     GO TO READ-IT.
108000 FINISH-IT.
108100     CLOSE GUEST-FILE.
109000D    DIVIDE PERFORMANCE-COUNT BY BAD-RECORDS GIVING RATIO.
        .
        .
        .
```

Compiler Control Option Lines

A compiler control option is designated by a line that has a dollar sign ($) in the indicator area (column 7) of the line. Such a line specifies the compiler control options to be used during the compilation process. For details about compiler control options, refer to Section 15.

Blank Lines

A blank line is a line that has no characters except blanks in Area B (positions 8 through 72). You can include blank lines anywhere in the source program to help make it more readable.

Pseudotext

Pseudotext is a sequence of text words, comment lines, or the separator space in a source program or COBOL library bounded by, but not including, pseudotext delimiters. Pseudotext delimiters are two contiguous equal sign (=) characters that surround the pseudotext. Pseudotext must not consist entirely of a separator comma or a separator semicolon.

The text or space that makes up pseudotext can start in either area A or area B. If, however, a hyphen (-) is in the indicator area of a line that follows the opening pseudotext delimiter, area A of the line must be blank, and the normal rules for continuation lines apply to the formation of text. For more information on the use of the hyphen, refer to "Continuation Lines" earlier in this section.

COBOL Character Set

The characters you use to write a COBOL source program include the letters of the alphabet, digits, and special characters. The standard character set is shown in Table 1–2.

Certain characters of the COBOL character set might not be represented graphically in definitions of national and international standard character sets. In these instances, you can specify a substitute graphic to replace the character or characters not represented.

Table 1–2. Areas of a Line of Code for Characters

Character	Meaning
0 through 9	Digit
A through Z	Uppercase letter
a through z	Lowercase letter
(Blank)	Space
+	Plus sign
-	Minus sign (hyphen)
*	Asterisk
/	Slant (slash)
=	Equal sign
$	Currency (dollar) sign
,	Comma (decimal point)
;	Semicolon
.	Period (decimal point, full stop)
"	Quotation mark
(	Left parenthesis
)	Right parenthesis
>	Greater than symbol
<	Less than symbol
:	Colon
_	Underscore

Using Separator Characters for Punctuation

When writing the text of a source program, you often need to show where one language element ends and the next one begins. You can differentiate between language elements by using separator characters. Sometimes separators are required by a general format. Other times, you can use separators at your discretion to improve the readability of your program. The characters you can use as separators and the rules for using them are described in Table 1–3. Note that the rules provided in Table 1–2 do not apply to the characters contained in nonnumeric literals, comment-entries, or comment lines.

Table 1–3. Valid Separator Characters

Separator	Guidelines for Use
(space)	Spaces can precede or follow all other separators except when restricted by reference format rules as discussed in this section. All spaces that immediately follow the comma, semicolon, or period are recognized as part of that separator and are not recognized as the space separator. A space is required before the opening pseudotext delimiter. A space that follows the opening quotation mark (") of a nonnumeric literal is considered to be part of the literal. A space that precedes the ending quotation mark of a nonnumeric literal is considered to be part of the literal.
.	A period marks the end of a COBOL entry. The period must be followed by a space, which is interpreted as part of the period separator.
, ;	You can use the comma and semicolon as separators anywhere you would use a space, with the exception that you cannot use the comma as a separator in a PICTURE character-string. You should include a space after the comma or semicolon separators. Although the compiler may permit the omission of the trailing space if the resulting code is not ambiguous, it is recommended that you include the space to prevent encountering problems when a space is required, but not supplied.
( )	A pair of parentheses (left and right) delimits subscripts, reference modifiers, arithmetic expressions, and conditions. They must appear only in balanced pairs of left and right.

Table 1–3. Valid Separator Characters

Separator	Guidelines for Use
" "	Quotation marks delimit nonnumeric literals. They must appear in balanced pairs, except when the literal is continued onto another line. A line that is to be continued must contain opening quotation marks preceding the literal. Each continuation line contains opening quotation marks as the first nonblank character in Area B. The last continuation line contains closing quotation marks following the literal. An opening quotation mark must be immediately preceded by a space, left parenthesis, comma, or semicolon. A closing quotation mark must be followed immediately by a space, right parenthesis, comma, semicolon, or period.
==	Two contiguous equal signs are pseudotext delimiters. You must place two contiguous equal signs at the beginning of a line of pseudotext and at the end of the line. An opening pseudotext delimiter must be immediately preceded by a space. A closing pseudotext delimiter must be immediately followed by a space, comma, semicolon, or period.
:	The colon is a required separator when it appears in general formats.
@	The at-sign character delimits undigit literals. An opening at-sign character must be preceded immediately by a space, comma, semicolon, or left parenthesis. A closing at-sign character must be followed immediately by a space, comma, semicolon, period, or right parenthesis.
B"	The letter B followed by a quotation mark is an opening separator for a Boolean literal. You must use another quotation mark to end the Boolean literal. The B" separator must be preceded by a space or a left parenthesis.
N"	The letter N followed by a quotation mark is an opening separator for a national literal. You must use another quotation mark to end the national literal. The N" separator must be preceded by a space or a left parenthesis.

Note: *Any punctuation character that you use in a PICTURE character-string or a numeric literal is considered to be part of the string or literal rather than a punctuation character. You can delimit PICTURE character-strings with spaces, commas, semicolons, or periods.*

Types of COBOL Words

A COBOL word is a character-string that contains a maximum of 30 characters. Words can be classified into three categories:

- Reserved words (compiler-defined)
- System-names
- User-defined words

You **cannot** use a reserved word as a system-name or a user-defined word.

You **can** use the same word for a system-name and a user-defined word. The compiler can determine how the word is to be used by the context of the clause or phrase in which the word occurs.

The following paragraphs describe the types of COBOL words.

Reserved Words

A reserved word is a COBOL85 word that has a specific meaning to the compiler and is reserved for use only as indicated by a general format. A reserved word appears in uppercase letters in the general formats. When the reserved word is a required part of the syntax, it appears underlined. Underlined reserved words are called *keywords*. If a reserved word is not underlined, you can omit it from the syntax.

A reserved word cannot appear in the program as a user-defined word or a system-name. Table 1–4 shows the way reserved words are used by the COBOL language.

Table 1–4. Types of Reserved Words

Word Types	Purpose
Connectives	Qualify data, link operands in a series, or link logical operators to form conditions.
Figurative constants	Associate names with commonly used values.
Functions	Associate names with commonly used calculations.
Special registers	Serve as compiler-generated, read-only storage areas that access specific COBOL85 features.
Arithmetic and relational operators	Indicate arithmetic operation or quantify a relation.
Keywords and optional words	Satisfy the requirements of the syntax and improve the readability of your program.

The following paragraphs describe each of the types of reserved words. A complete list of reserved words is provided in Appendix B.

Connectives

Connectives are reserved words that you can use in one of the following ways:

- As qualifiers to associate data-names, condition-names, text-names, or paragraph-names. Examples of qualifier connectives are *OF* and *IN*.
- As logical connectives to form conditions. Examples of logical connectives are *AND* and *OR*.

Figurative Constants

A figurative constant is a reserved word, such as ALL or SPACES, that takes on the value implied by the word.

You can use figurative constants in place of a literal in a general format. However, if the literal is restricted to a numeric literal, you are limited to the figurative constant ZERO or its alternate forms ZEROS and ZEROES.

When you use a figurative constant in a context that requires national data, the figurative constant represents a national literal value.

The figurative constants you can use and the values they imply are described in Table 1-5. Note that the singular and plural forms of figurative constants are equivalent, so you can use them interchangeably.

Table 1–5. Figurative Constants

The figurative constant . . .	Represents . . .
ZERO, ZEROS, or ZEROES	The numeric value 0 or one or more of the 0 characters from the computer's character set. For national data, it represents the national literal @A3B0@.
SPACE OR SPACES	One or more space characters. For national data, it represents the national literal @A1A1@.
HIGH-VALUE or HIGH-VALUES	One or more of the characters that has the highest ordinal position in the program collating sequence. For national data, it represents the national literal @FFFF@ . The actual characters associated with each figurative constant depend upon the program collating sequence specified. To define HIGH-VALUE in the SPECIAL-NAMES paragraph of the Environment Division (invalid for national literals), you must use the ALPHABET clause. For details, refer to "SPECIAL-NAMES Paragraph" in Section 3.

Table 1–5. Figurative Constants

The figurative constant . . .	Represents . . .
LOW-VALUE or LOW-VALUES	One or more of the characters that has the lowest ordinal position in the program collating sequence. For national data, it represents the national literal @0000@ . The actual characters associated with each figurative constant depend upon the program collating sequence specified. To define LOW-VALUE in the SPECIAL-NAMES paragraph of the Environment Division (invalid for national literals), you must use the ALPHABET clause. For details, refer to "SPECIAL-NAMES Paragraph" in Section 3.
QUOTE or QUOTES	One or more quotation marks ("). For national data, it represents the national literal @A1C9@ . You can use the following statement to avoid using a literal: `MOVE QUOTE TO PRINT-LINE` You cannot use the word QUOTE or QUOTES in place of the quotation mark in a source program to enclose a nonnumeric literal. Thus, QUOTE ABD QUOTE is incorrect as a way of stating the nonnumeric literal "ABD".
ALL literal	A continuous sequence of an alphanumeric or a national literal. The literal must be nonnumeric and must not be a figurative constant. Note that associating the figurative constant [ALL] literal, and a literal of length greater than one, with a data item that is numeric or numeric-edited is becoming obsolete in COBOL ANSI-85 and will be deleted from the next revision of the COBOL standard.
[ALL] symbolic-character	The name of a position in the collating sequence. For example, the end-of-text position does not have a name. If you designate ETX IS 14, the name ETX becomes the same as position 14 in the collating sequence. Then, you can use the name ETX in your program. You designate the symbolic-character in the SYMBOLIC CHARACTERS clause of the SPECIAL-NAMES paragraph in the Environment Division. Refer to "SPECIAL-NAMES Paragraph" in Section 3.

When you use a figurative constant other than the [ALL] literal, using the word ALL is redundant and is for readability purposes only.

When a figurative constant represents a string of one or more characters, the compiler determines the length of the string from context according to the following rules:

- When a figurative constant is specified in a VALUE clause, or when a figurative constant is associated with another data item (for example, when the figurative constant is moved to or compared with another data item), the string of characters specified by the figurative constant is repeated character by character on the right until the size of the resultant string is greater than or equal to the number of character positions in the associated data item.

  The resultant string is then truncated from the right until it is equal to the number of character positions in the associated data item. This truncation is done before, and independently of, the application of any JUSTIFIED clause that might be associated with the data item.
- When a figurative constant, other than the [ALL] literal, is not associated with another data item (for example, when the figurative constant appears in a DISPLAY, STOP, STRING, or UNSTRING statement), the length of the string is one character.
- When the figurative constant [ALL] literal is not associated with another data item, the length of the string is the length of the literal.
- When a figurative constant is used in conjunction with the VALUE clause as part of a data description entry, editing characters in the PICTURE clause are included in determining the size of the data, but they have no effect on the initialization of the data item.

Functions

A function is a temporary data item whose value is derived automatically at the time of reference during the execution of the object program. Functions are specified by a function-identifier, which consists of the reserved word FUNCTION, a reserved function-name, and optional user-defined arguments. Functions are described in detail in Section 9.

Special Registers

Special registers are compiler-generated storage areas whose primary use is to store information produced by specific COBOL features. Table 1–6 describes the special registers.

Table 1–6. Special Registers

The register . . .	Contains . . .
DATE	If DATE is followed by the qualifier "YYYYMMDD", the system date is formatted as an unsigned, 8-digit elementary numeric integer made up of the year (four digits), month of the year (two digits) and the day of month (two digits). For example, July 1, 1993 is expressed as 19930701. If DATE is not qualified, the system date is formatted as an unsigned, 6-digit elementary numeric integer made up of the year of the century (two digits), the month of year (two digits), and the day of the month (two digits). For example, July 1, 1993 is expressed as 930701. To query this special register, use Format 2 of the ACCEPT statement.
DAY	If DAY is followed by the qualifier "YYYYDDD", the system date is formatted as an unsigned, 7-digit elementary numeric integer made up of the year (four digits) followed by the number of days since the beginning of the year (three digits). For example, July 1, 1993 is expressed as 1993183. If DAY is not qualified, the system date is formatted as an unsigned 5-digit elementary numeric integer made up of the year of the century (two digits) followed by the number of days since the beginning of the year (three digits). For example, July 1, 1993 is expressed as 93183. To query this special register, use Format 2 of the ACCEPT statement.
DAY-OF-WEEK	A single data element that represents the day of the week. A value of 1 represents Monday, a value of 2 represents Tuesday, and so on. When accessed by a COBOL program, this register behaves as an unsigned elementary numeric integer 1 digit in length (PIC9(1) COMP). To query this special register, use Format 2 of the ACCEPT statement.
LINAGE-COUNTER	The number of lines advanced within a printed page. LINAGE-COUNTER is a fixed data-name for a line counter suitable for computation. It is generated by the presence of a LINAGE clause in a file description (FD) entry. The implicit class of a LINAGE-COUNTER is numeric. No data item is referenced; it is treated as a LINENUMBER attribute for purposes of retrieval. The compiler automatically supplies one LINAGE-COUNTER for each file in the File Section that has a LINAGE clause in its FD entry. For more information, refer to “LINAGE” Clause in Section 4.

Table 1–6. Special Registers

The register . . .	Contains . . .
LINE-COUNTER	The vertical position in a report. LINE-COUNTER is a fixed data-name for a line counter suitable for computation. It is generated for each report description (RD) entry in the Report Section. The compiler automatically provides one LINE-COUNTER register for each report in the RD entry. You can query this special register by using the Report Writer facility.
PAGE-COUNTER	Page numbers within a report group. PAGE-COUNTER is a fixed data-name for a page counter suitable for computation. It is generated for each report-description (RD) entry in the Report Section. The compiler automatically supplies one PAGE-COUNTER for each report that has the word PAGE-COUNTER as a source data item in an RD entry. You can query this special register by using the Report Writer facility.
TIME	The elapsed time after midnight based on a 24-hour clock in hours, minutes, seconds, and hundredths of a second. TIME is an unsigned, 8-digit, elementary numeric integer. For example, 2:41 p.m. is expressed as 14410000. The maximum value of TIME is 23595999. You can query this special register by using Format 2 of the ACCEPT statement.
TIMER	The number of 2.4-microsecond intervals since midnight. TIMER is a single, unsigned 11-digit numeric integer. It is composed of the current value of the computer's interval timer. You can query this special register by using Format 2 of the ACCEPT statement.
TODAYS-DATE	If TODAYS-DATE is followed by the qualifier "MMDDYYYY", the system date is formatted as an unsigned, 8-digit elementary numeric integer made up of the month of the year (two digits), the day of the month (two digits), and the year (four digits). For example, July 1, 1993 is expressed as 07011993. If TODAYS-DATE is not qualified, the system date is formatted as an unsigned, 6-digit elementary numeric integer made up of the month of the year (two digits), the day of the month (two digits), and the year of the century (two digits). For example, July 1, 1993 is expressed as 070193. To query this special register, use Format 2 of the ACCEPT statement.
TODAYS-NAME	The current day of the week. TODAYS-NAME is an elementary, 9-character, alphanumeric item. If the day of the week is less than nine characters long, it is left-justified in the 9-character area provided, with space-fill on the right. You can query this special register by using Format 2 of the ACCEPT statement.

Arithmetic and Relational Operators

Arithmetic and relational operators are symbols used to imply a mathematical operation or to compare the value of two operands. Table 1–7 lists these COBOL operators. Note that the operators are required when they appear in a general format even though they are not underlined.

Table 1–7. Special Character Words

Type of Operator	Symbol	Meaning
Arithmetic	+	Addition
	-	Subtraction
	*	Multiplication
	/	Division
	**	Exponentiation
Relational	>	Greater than
	<	Less than
	=	Equal to
	>=	Greater than or equal to
	<=	Less than or equal to

System-Names

A system-name is a word that you use to communicate with the operating system. A system-name can be one of two types, as shown in the following table.

Type of System-Name	Description
Computer-name	This is the name of the computer, for example MICROA or A17, on which the COBOL program is to be compiled or executed.
Implementor-name	This is a name that refers to a particular feature, such as ODT or SW1.

You can use the same word as a system-name and a user-defined word. The compiler determines the class of a specific occurrence of the word by the context of the clause or phrase in which the word occurs.

Rules

Observe the following rules when you form a system-name:

- Make the system-name no more than 30 characters long.
- Select each character from the set of characters A through Z, 0 through 9, the underscore (_), and the hyphen (-). (Each lowercase letter is equivalent to its corresponding uppercase letter.)
- Do not use the underscore or the hyphen as the first or last character of a system-name.
- Do not use a reserved word as a system-name.

User-Defined Words

A user-defined word is a word that you supply to complete the syntax of a clause or statement. You can use the same word as a user-defined word and a system-name. The compiler determines the class of a specific occurrence of the word by the context of the clause or phrase in which the word occurs.

Rules

Observe the following rules when you form a user-defined word:

- Make the user-defined word no more than 30 characters long.
- Select each character from the set of characters A through Z, 0 through 9, the underscore (_), and the hyphen (-). (Each lowercase letter is equivalent to its corresponding uppercase letter.)
- Do not use the underscore or the hyphen as the first or last character of a user-defined word.
- Do not use a reserved word.
- Make sure that all user-defined words, except level-numbers and segment-numbers, are unique. You can use qualification to make similar words unique. (Qualification is discussed in Section 4.)
- Include at least one alphabetic character in all user-defined words, except in the following types of words:
  - Family-names
  - Level-numbers
  - Library-names
  - Paragraph-names
  - Section-names
  - Segment-numbers
  - Text-names

Double-Byte Names

Double-byte names are user-defined words made up of 16-bit characters and are used with national languages that require a 16-bit coded character set. Double-byte names include an SDO (start of double octet) character, one or more 16-bit characters, and an EDO (end of double octet) character. SDO and EDO are control characters that distinguish double-byte names from standard single-byte names.

As with national literals, you must enter double-byte names from a keyboard and terminal that uses a 16-bit coded character set and that automatically inserts control characters. Also, printer backup files that contain images of double-byte names must be printed by a printer that automatically interprets control characters. The SDO and EDO control characters appear as space characters on terminals and printers that use 16-bit coded character sets.

The types of user-defined words that can be specified in 16-bit characters are

- Alphabet-name
- Class-name
- Condition-name
- Data-name
- Index-name
- Mnemonic-name
- Paragraph-name
- Record-name
- Section-name
- Symbolic-character

Rules

Observe the following rules when you form a double-byte name:

- Include any character from the 16-bit character set.
- Make names no more than 14 16-bit characters long. The maximum length allowed is 28 bytes plus 2 bytes for the SDO and EDO control characters.
- Use 16-bit characters only. You cannot mix standard 8-bit characters with 16-bit characters to form a double-byte name.
- Make names unique. You can use qualification to make similar words unique. Refer to Section 4, "Data Division," for information on qualification.
- Place names completely on a single line. You cannot continue some of the characters of a double-byte name to a continuation line.

Table 1–8 lists and describes the types of user-defined words that most frequently appear in COBOL85 general formats.

Table 1–8. Types of User-Defined Words

Type	Purpose
Alphabet-name	Assigns a name to a specific character set and collating sequence.
Class-name	Assigns a name to any group of characters in the computer's character set in the SPECIAL-NAMES paragraph of the Environment Division. You can use a class-name in a conditional expression.
Condition-name	Assigns a name to a specific value, set of values, or range of values from a complete set of values that a conditional variable can have. (A conditional variable is a data item that can assume more than one value.) A condition-name can also assign a name to a switch or device. You define condition-names in the Data Division or in the Special-Names paragraph of the Environment Division. You can use a condition-name as an abbreviation for a relation condition. A relation condition assumes that the associated conditional variable is equal to one of the set of values to which that condition-name is assigned. You can also use a condition-name in a SET statement to indicate that the associated value is to be moved to the conditional variable.
Data-name	Names a data item described in a data description entry. A data-name must not have a reference-modifier, qualifier, or subscript unless specifically permitted by the rules of the general format. A data-name that has a reference-modifier, qualifier, or subscript is referred to as an identifier. Identifiers are described in detail later in this section.
File-name	Names a file described in a file description entry or a sort-merge file description (FD) entry in the File Section of the Data Division.
Index-name	Names an index associated with a specific table.
Level-number	Defines a one- or two-digit number that indicates the hierarchical position of a data item or the special properties of a data description entry.
Library-name	Names a COBOL library that is to be used by the compiler for a given source program compilation.
Mnemonic-name	Assigns a user-defined word to an implementor-name in the SPECIAL-NAMES paragraph of the Environment Division. An implementor-name is a system-name that refers to a particular feature available on the COBOL85 compiler.
Paragraph-name	Identifies and begins a paragraph in the Procedure Division. Paragraph-names are equivalent only if they consist of the same sequence of the same number of digits and/or characters.

Table 1–8. Types of User-Defined Words

Type	Purpose
Program-name	Identifies a COBOL source program in the Identification Division and the end-program header.
Record-name	Names a record described in a record description (RD) entry in the Data Division.
Section-name	Names a section in the Procedure Division. Section-names are equivalent only if they consist of the same sequence of the same number of digits and/or characters.
Symbolic-character	Specifies a user-defined figurative constant. Refer to "Figurative Constants" in this section for more information.
Text-name	Specifies the external identification of a file in the COBOL library.

Note that within a source program, excluding nested programs, user-defined words are grouped into disjoint sets. A disjoint set is a set that has no common elements. Thus the user-defined words within the set must be unique. In addition, all user-defined words, except level numbers, can belong to only one disjoint set.

User-defined words are grouped into the following disjoint sets:

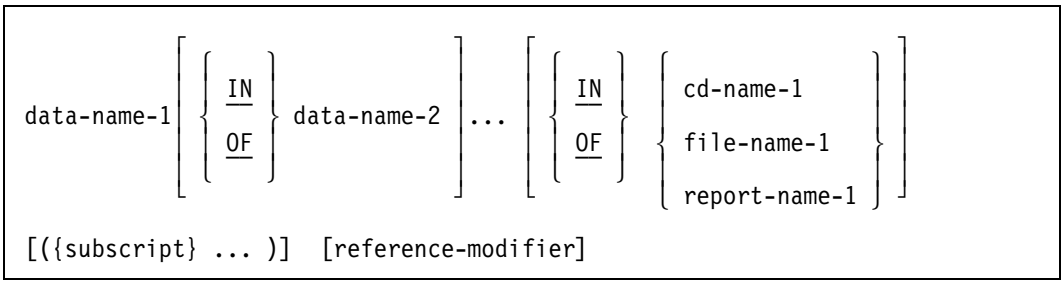
- Alphabet-names
- Class-names
- Condition-names, data-names, and record-names
- File-names
- Index-names
- Library-names
- Mnemonic-names
- Paragraph-names
- Program-names
- Section-names
- Symbolic-characters
- Text-names

Identifiers

An identifier is a syntactically correct sequence of character-strings and separators used to uniquely identify a data item.

When a data item other than a function is specified, the term identifier is used in a general format to indicate a data-name that must be either unique in a program or must be followed by a syntactically correct combination of qualifiers, subscripts, or reference modifiers to make it unique. (Qualifiers and reference modifiers are described in Section 4. Subscripts are discussed in Section 5.)

The general syntax for an identifier is as follows:



Note that the words IN and OF are equivalent in this syntax.

Some special identifiers do not exactly follow the ANSI COBOL85 format for an identifier. These identifiers and the sections in this manual in which they are described are shown in the following table:

For information about . . .	Refer to . . .
Event-identifiers	Section 6 (CAUSE statement)
File attribute identifiers	Section 12
Function identifiers	Section 9
Task attribute identifiers	Section 6 (CHANGE statement) and Section 13

Literals

A literal is a word, number, or symbol that names, describes, or defines itself and not something else that it might represent. For example, consider the following general format for the ADD statement:

```
ADD   { identifier-1 }
      { literal-1    } ... TO  { identifier-2 [ ROUNDED ] } ...
```

Assume that you want the value for identifier-2 to be "TOTAL." If you choose to use a literal as shown in the preceding syntax, your program line might read "ADD 1 TO TOTAL." The computer adds the actual value of 1 to the value stored in the TOTAL field.

If you want to add the value stored in the EXTRA-INCOME field to the value stored in the TOTAL field, you would use an identifier instead of a literal. Your program line might read "ADD EXTRA-INCOME TO TOTAL."

Every literal belongs to one of the following types:

- Nonnumeric
- National
- Numeric
- Undigit
- Floating-point
- Boolean

The types of literals are described in the following paragraphs.

Nonnumeric Literals

A nonnumeric literal is an alphanumeric value from 1 through 160 characters in length. The characters can include any character in the alphanumeric and national character sets. National characters and the control characters SDO and EDO, which are used to distinguish national characters from nonnumeric characters, can be mixed with nonnumeric characters to form nonnumeric literals.

To indicate that a value is a nonnumeric literal, you must place quotation marks (") before and after the value. The quotation marks are not considered to be part of the value of the literal. The general format for a nonnumeric literal is as follows:

```
" { character-1 } ... "
```

Character-1 can be any character in the computer's character set.

Details

To use the quotation mark as a literal, use two contiguous quotation marks. For example, assume that you want to produce the name William "Bud" Smith, with the name Bud in quotation marks. You would use the following code:

```
"William ""Bud"" Smith".
```

Note that all punctuation characters are part of the value of the nonnumeric literal and are not used as separators.

The value of a nonnumeric literal in the object program is the value represented by character-1.

Examples

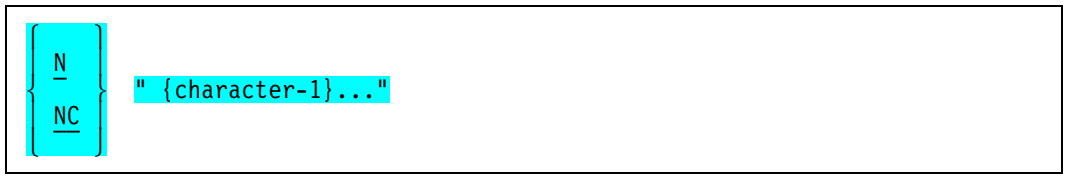
The following table provides examples of the coding of nonnumeric literals.

Coding	Result
"MY NAME"	MY NAME
""""	"
"FEET/SQ. IN."	FEET/SQ. IN.
"THIS IS ""EDITED"" OUTPUT"	THIS IS "EDITED" OUTPUT

National Literals

A national literal is a character string in a language other than standard American English. A national literal is of the national class and category.

The general format for a national literal is as follows:



The letter N and the quotation mark (") serve only as delimiters and are not part of the value of the national literal. Character-1 is a string of 8-bit or 16-bit characters in national standard data format. The 16-bit national literal must be keyed in from a terminal that uses a national character set and automatically inserts control characters.

Details

The 8-bit national characters do not require the insertion of control characters at the beginning and end of the literal.

The 16-bit national characters are distinguished from 8-bit national or nonnumeric characters by the insertion of control characters at the beginning and end of the literal. The control characters SDO (start of double octet) must follow the first double quotation mark and immediately precede character-1. The control character EDO (end of double octet) must immediately follow character-1 and precede the ending double quotation mark. Control characters cannot be used within the character-1 string. Each control character occupies one byte of space, so the length of a national literal can be from 1 to 79 16-bit characters, or from 2 to 158 bytes long. Multi Octet Character Set support for the COBOL85 compiler supports only the Kanji 16-bit character set.

Note: *COBOL74 uses the delimiter NC instead of N to specify a 16-bit national literal. Both the NC and N delimiters can be used for this release of COBOL85.*

Example

An example of a national literal declaration that is 10 national characters long is as follows:

```
01 NAME PIC N(10) VALUE N"AAAAAAAAAA".
```

Numeric Literals

A numeric literal is a literal composed of one or more numeric characters. Numeric literals do not have delimiters.

COBOL85 acknowledges two types of numeric literals:

- Standard numeric literals (1 to 23 digits)
- Long numeric literals (24 to 160 digits)

The rules for forming both types of literals are explained in the following paragraphs.

Rules for All Numeric Literals

The following rules apply to both standard numeric literals and long numeric literals:

- Every numeric literal is in the numeric category.
- The value of a numeric literal is the algebraic quantity represented by the characters in the numeric literal.
- The size of a numeric literal in standard data format characters equals the number of digits specified in the character-string.
- If the literal conforms to the rules for the formation of numeric literals but is enclosed in quotation marks ("), it is a nonnumeric literal and the compiler treats it as such. For example, "1234" is a nonnumeric literal.

Rules for Standard Numeric Literals

Observe the following rules when forming standard numeric literals:

- The literal can contain only one sign character.

  If you use a sign, it must appear as the leftmost character of the literal.
- If the literal is unsigned, it is assumed to be positive.
- The literal can contain only one decimal point.

  The decimal point can appear anywhere within the literal except in the rightmost character position.

  The decimal point is treated as an assumed decimal point, which means that it does not involve the existence of an actual character in a data item. The assumed decimal point has logical meaning with no physical representation.
- If the literal does not contain a decimal point, the literal is an integer.

Rules for Forming Long Numeric Literals

Observe the following rules when forming long numeric literals:

- Long numeric literals must be described as unsigned integers, so they **cannot** have operational signs or decimal points.
- Arithmetic operations and relative arithmetic comparisons are not permitted on long numeric literals.
- You can use long numeric literals only with the following Procedure Division statements: CALL, IF, INITIALIZE, INSPECT, MERGE, MOVE, READ, SORT, and WRITE. For details, refer to the description of each statement in Sections 6 through 8 of this manual.

Undigit Literals

An undigit literal is a string of hexadecimal characters delimited at the beginning and end by the at-sign (@). A hexadecimal character is a character from the set composed of the digit characters 0 through 9 and the uppercase letters A through F.

Whether an undigit literal is interpreted as 4-bit numeric characters, 8-bit alphanumeric characters, 8-bit national characters, or 16-bit national characters depends upon the data item to which it is associated:

- If the undigit literal appears in the VALUE clause of a data item whose usage is COMPUTATIONAL, the undigit literal is interpreted as 4-bit numeric characters.
- If the undigit literal is associated with an alphanumeric data item, the undigit literal is interpreted as 8-bit alphanumeric characters. The undigit literal must contain an even number of hexadecimal characters because two hexadecimal characters are required for each alphanumeric character.
- If the undigit literal is associated with a national data item and the CCSVERSION phrase is specified in the program, the undigit literal is interpreted as 8-bit national characters. The undigit literal must contain an even number of hexadecimal characters because two hexadecimal characters are required for each 8-bit national character.
- If the undigit literal is associated with a national data item and no CCSVERSION phrase is specified, the undigit literal is interpreted as 16-bit national characters. The undigit literal must contain a number of characters that is divisible by four because four hexadecimal characters are required for each 16-bit national character.

An undigit literal is interpreted as national in the following cases:

- In the INSPECT statement where the inspected data item is national
- In the STRING statement where the receiving data item is national
- In the UNSTRING statement where the sending data item is national
- In the MOVE statement where the receiving field is national
- In the VALUE clause associated with a national data item or in the VALUE clause of a condition-name associated with national data items
- In the conditional expression of an EVALUATE, IF, PERFORM, or SEARCH statement where the category of the other relational operand is national
- In the ALL figurative constant if it occurs in the situations described for the preceding cases

An undigit literal cannot be treated as national in a DISPLAY or STOP statement.

Floating-Point Literals

Floating-point literals provide an alternate means of representing REAL and DOUBLE data items. The general format of a floating point literal is

mantissa E exponent

The mantissa is the decimal part of the number. The mantissa can be signed and must have one decimal point. The exponent signifies a power of 10 used as a multiplier. The exponent can be signed and must be an integer.

The value represented by a floating-point literal is the mantissa multiplied by 10 raised to the power of the exponent. For single-precision, the permissible range for the value of a floating point literal is

$8.75811540203 * 10^{-47}$ to $4.31359146673 * 10^{68}$

For double-precision, the permissible range for the value magnitude is

$1.9385458571375858335564 * 10^{-29581}$
to
$1.9488283820502807912 4467 * 10^{29603}$

Floating-point literals can be used in the language anywhere a noninteger numeric literal is permitted.

Examples

```
1.Eû40
û.0023E29
+.0012345Eû5
+1.2E9500
2.E40
+123.45678901234E20
```

Boolean Literals

A Boolean literal is a character string delimited on the left by the separator B" and on the right by the separator quotation mark.

General Format

```
B"Boolean-character"
```

The Boolean-character is a "0" or a "1".

Examples

```
B"1"
B"0"
```

Section 2
Identification Division

This section presents and explains the syntax of the Identification Division, the first division of a COBOL program.

General Format

The general format of the Identification Division is as follows:

```
IDENTIFICATION DIVISION.
[ PROGRAM-ID. program-name. ]
[  AUTHOR. [ comment-entry ] ... ]
[ INSTALLATION. [ comment-entry ] ... ]
[ DATE-WRITTEN. [ comment entry ] ... ]
[ DATE-COMPILED. [ comment-entry ] ... ]
[ SECURITY. [ comment-entry ] ... ]
```

Except for the DATE-COMPILED paragraph, the entire Identification Division is copied from the input source program and is included on the output listing. The object program, however, is not affected by the information included in this division.

Note: *The AUTHOR, INSTALLATION, DATE-WRITTEN, DATE-COMPILED, and SECURITY paragraphs are obsolete elements in COBOL ANSI-85 and will be deleted from the next revision of the COBOL standard.*

Identification Division Header

The following header identifies and must begin the Identification Division:

```
IDENTIFICATION DIVISION.
```

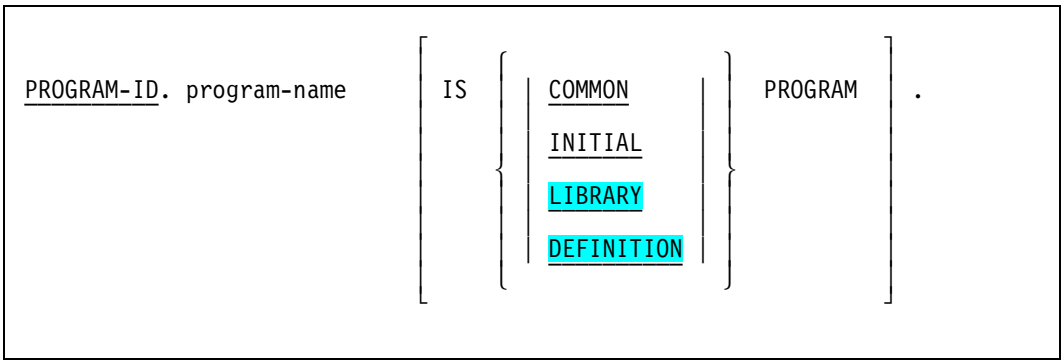
These keywords begin in area A and must be followed by a period.

PROGRAM-ID Paragraph

The PROGRAM-ID paragraph is the only required paragraph in the Identification Division of a program nested within another program. Otherwise, this paragraph is optional, and the compiler will implicitly generate a PROGRAM-ID record if it is missing. However, multiple source programs arranged sequentially in a single source program must be delimited by the PROGRAM-ID paragraph, which specifies the name of the program and assigns selected program attributes to that program.

Refer also to "Using the ANSI IPC Constructs," "The Run Unit," "Nested Source Programs," and "Common and Initial Programs" in Section 10.

The format of the PROGRAM-ID paragraph is as follows:



PROGRAM-ID

This keyword begins in area A and must be followed by a period.

program-name

This name is a user-defined word that identifies the source program, the object program, and all listings that pertain to a particular program.

The program-name in the PROGRAM-ID paragraph is not necessarily the same as the source program name or object program name, which are determined by the method of compilation. For example, if TESTSOURCE/C85/XYZ were the source file name, its PROGRAM-ID could be PROGRAM-ID xyz COMMON), and, if compiled through CANDE, its object code file name could be OBJECT/TESTSOURCE/C85/XYZ.

When a sequence of programs is compiled, the second program is named <program-name>-1, the third program is named <program-name>-2, and so on.

Note that a nested program must not be assigned the same program-name as that of any other program contained in the separately compiled program that contains the nested program.

IS COMMON PROGRAM Clause

You can use this clause if the program is contained in another program. When used, this clause specifies that the program can be called from programs other than the one containing it. Refer also to "Common and Initial Programs" in Section 10.

IS INITIAL PROGRAM Clause

This clause specifies that the program (and any programs it contains) will be placed in its initial state each time it is called. Refer also to "Common and Initial Programs" in Section 10.

IS LIBRARY PROGRAM Clause

This clause identifies a program as a library program. The program that contains this clause must be the outermost program of a collection of programs; the library program cannot be nested within another program.

A program that contains an IS LIBRARY PROGRAM clause must also contain an export definition in the Program-Library Section of the Data Division. For more information, refer to "Program-Library Section" in Section 4. For information on library programs, refer to Section 11.

Note: *If the IS LIBRARY PROGRAM clause is present in a source program, the compiler control options LIBRARYPROG and LEVEL cannot be set.*

IS DEFINITION PROGRAM Clause

This clause identifies the program as a definition program. The program that contains this clause must be the first program. It is followed by a list of multi-procedure programs separated by a LIBRARY control option. The BINDSTREAM compiler control option must be set.

A program that contains an IS DEFINITION PROGRAM can contain only the Working-Storage Section, Local-Storage Section, and the import definitions in the Program-Library Section of the Data Division. For more information, refer to BINDSTREAM and LIBRARY compiler control options.

AUTHOR Paragraph

The AUTHOR paragraph gives the name of the person who wrote the program. Use of this paragraph is optional.

The format of the AUTHOR paragraph is as follows:

```
[ AUTHOR. [ comment-entry ] ... ]
```

AUTHOR

This keyword begins in area A and must be followed by a period.

comment-entry

This can be any combination of characters from the computer's character set. You must not continue a comment-entry with a hyphen in the indicator area; however, a comment-entry can extend beyond one line. The comment-entry is becoming obsolete in COBOL85 and will be deleted from the next revision of the ANSI COBOL standard.

Note: *The AUTHOR paragraph is an obsolete element in COBOL ANSI-85 and will be deleted from the next revision of the COBOL standard.*

INSTALLATION Paragraph

The INSTALLATION paragraph gives the name of the site where the program will be used. Use of this paragraph is optional.

The format of the INSTALLATION paragraph is as follows:

```
[ INSTALLATION. [ comment-entry ] ... ]
```

INSTALLATION

This keyword begins in area A and must be followed by a period.

comment-entry

This can be any combination of characters from the computer's character set. You must not continue a comment-entry with a hyphen in the indicator area; however, a comment-entry can extend beyond one line. The comment-entry is becoming obsolete in COBOL85 and will be deleted from the next revision of the ANSI COBOL standard.

Note: *The INSTALLATION paragraph is an obsolete element in COBOL ANSI-85 and will be deleted from the next revision of the COBOL standard.*

DATE-WRITTEN Paragraph

The DATE-WRITTEN paragraph gives the date that the program was written. Use of this paragraph is optional.

The format of the DATE-WRITTEN paragraph is as follows:

```
[ DATA-WRITTEN. [ comment-entry] ... ]
```

DATE-WRITTEN

This keyword begins in area A and must be followed by a period.

comment-entry

This can be any combination of characters from the computer's character set. You must not continue a comment-entry with a hyphen in the indicator area; however, a comment-entry can extend beyond one line. The comment-entry is becoming obsolete in COBOL85 and will be deleted from the next revision of the ANSI COBOL standard.

Note: *The DATE-WRITTEN paragraph is an obsolete element in COBOL ANSI-85 and will be deleted from the next revision of the COBOL standard.*

DATE-COMPILED Paragraph

The DATE-COMPILED paragraph gives the date that the program was compiled. If this paragraph is present, the system automatically updates the compilation date in the source program listing. Use of this paragraph is optional.

The format of the DATE-COMPILED paragraph is as follows:

```
[ DATE-COMPILED. [ comment-entry ] ... ]
```

DATE-COMPILED

This keyword begins in area A and must be followed by a period.

This keyword causes the current date to be inserted in the source program listing during program compilation.

comment-entry

This can be any combination of characters from the computer's character set. You must not continue a comment-entry with a hyphen in the indicator area; however, a comment-entry can extend beyond one line. The comment-entry is becoming obsolete in COBOL85 and will be deleted from the next revision of the ANSI COBOL standard.

Details

If a DATE-COMPILED paragraph is present, it is replaced during compilation with a paragraph of the following form:

```
DATE-COMPILED.  year month day hh:mm
```

Example

```
DATE-COMPILED. 1988 FEBRUARY 11 10:15.
```

This example shows how the DATE-COMPILED paragraph would appear on the output listing of a program that was compiled on February 11, 1988, at 10:15.

Note: *The DATE-COMPILED paragraph is an obsolete element in COBOL ANSI-85 and will be deleted from the next revision of the COBOL standard.*

Security Paragraph

The SECURITY paragraph identifies the security restrictions under which the program can be accessed. Use of this paragraph is optional.

The format of the SECURITY paragraph is as follows:

```
[ SECURITY. [ comment-entry ] ... ]
```

SECURITY

This keyword begins in area A and must be followed by a period.

comment-entry

This can be any combination of characters from the computer's character set. You must not continue a comment-entry with a hyphen in the indicator area; however, a comment-entry can extend beyond one line. The comment-entry is becoming obsolete in COBOL85 and will be deleted from the next revision of the ANSI COBOL standard.

Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. IDEX.
AUTHOR. WATSINA NAM.
INSTALLATION. YOUR CORPORATION.
DATE-WRITTEN. FEBRUARY 11, 1988.
DATE-COMPILED.
SECURITY. CONFIDENTIAL.
```

The Identification Division in this example includes all five optional paragraphs. Because the DATE-COMPILED paragraph is included, the compilation date will be provided on the source listing.

Note: *The SECURITY paragraph is an obsolete element in COBOL ANSI-85 and will be deleted from the next revision of the COBOL standard.*

Section 3
Environment Division

This section illustrates and explains the syntax of the Environment Division, the second division of a COBOL program.

General Format

The general format of the Environment Division is as follows:

```
ENVIRONMENT DIVISION.
[ CONFIGURATION SECTION.
[ SOURCE-COMPUTER. [ computer-name [ WITH DEBUGGING MODE ] ]
[ OBJECT-COMPUTER. [ object computer entry ] ]
[ SPECIAL-NAMES. [ special names entry ] ]
[ INPUT-OUTPUT SECTION.
  FILE-CONTROL. { file control entry } ...
[ I-O-CONTROL. [ input output control entry ] ] ] ]
```

Environment Division Header

The following header identifies and must begin the Environment Division:

```
ENVIRONMENT DIVISION.
```

ENVIRONMENT DIVISION

These keywords begin in area A and must be followed by a period.

Configuration Section

The Configuration Section identifies the source computer, the object computer, and the mnemonic-names that are substituted for system-names in the program. Use of this section is optional.

Note that the Configuration Section must not be included in a program that is contained directly or indirectly in another program. Refer to "Nested Source Programs" in Section 10.

The Configuration Section includes a header and the following three optional paragraphs:

- SOURCE-COMPUTER Paragraph

  Describes the computer configuration on which the source program will be compiled.

- OBJECT-COMPUTER Paragraph

  Describes the computer configuration on which the object program produced by the compiler will be run.

- SPECIAL-NAMES Paragraph

  Provides a means of specifying the currency sign, choosing the decimal point, specifying symbolic-characters, relating implementor-names to user-specified mnemonic-names, relating alphabet-names to character sets or collating sequences, relating class-names to sets of characters, and specifying the default sign position for all signed data items whose usage is DISPLAY or COMPUTATIONAL.

Configuration Section Header

The following header identifies and must begin the Configuration Section:

```
CONFIGURATION SECTION.
```

CONFIGURATION SECTION

These keywords begin in area A and must be followed by a period.

SOURCE-COMPUTER Paragraph

The SOURCE-COMPUTER paragraph identifies the computer on which the program will be compiled. Use of this paragraph is optional.

```
SOURCE-COMPUTER. [ computer-name [ WITH DEBUGGING MODE ] . ]
```

SOURCE-COMPUTER

This keyword begins in area A and must be followed by a period.

computer-name

This name is a system-name (any COBOL word) that identifies the computer on which the source program is to be compiled.

The computer-name is for documentation purposes only.

WITH DEBUGGING MODE

This clause serves as a compile time switch over the debugging lines written in a separately compiled program. When the WITH DEBUGGING MODE clause is specified in a separately compiled program, all debugging lines are compiled as specified in the program. When the WITH DEBUGGING MODE clause is not specified in a program and the program is not contained within a program including a WITH DEBUGGING MODE clause, then the debugging lines are compiled as comment lines. $FREE must be reset to compile debugging lines.

Details

All clauses of the SOURCE-COMPUTER paragraph apply to the program in which they are explicitly or implicitly specified and to any program contained in that program.

If you specify the SOURCE-COMPUTER paragraph but not the computer-name (refer to "General Format of the Environment Division" in this section), the computer on which the source program is compiled is the source computer.

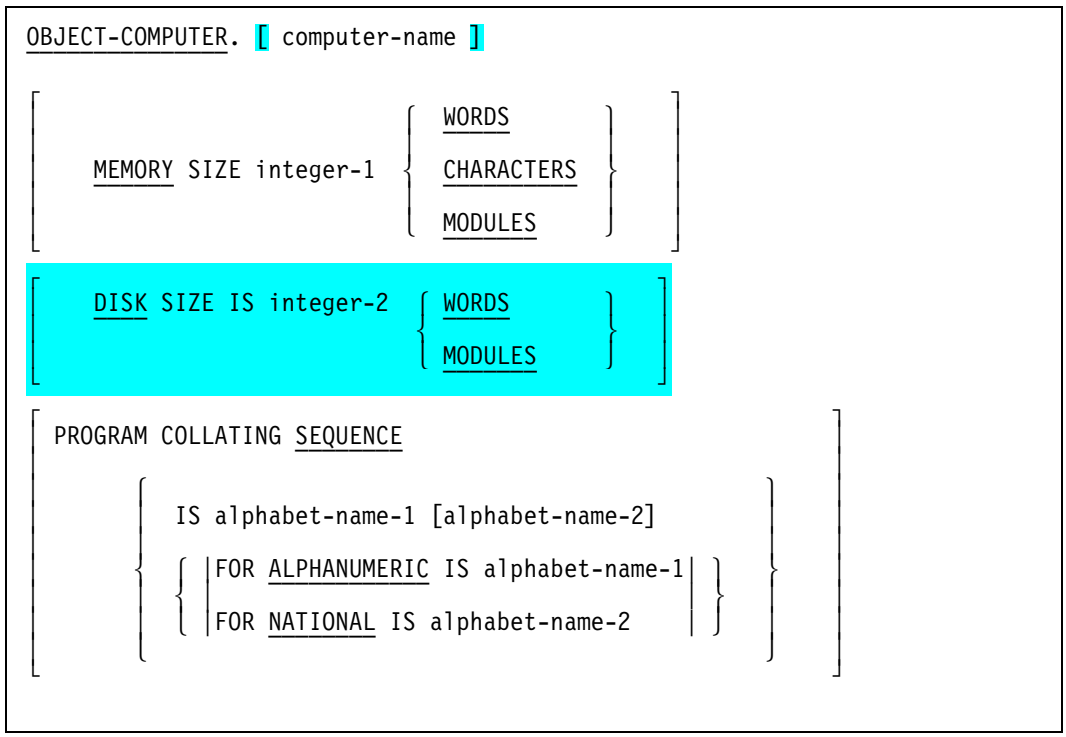
If you specify the SOURCE-COMPUTER paragraph but the program is not contained in a program that includes a SOURCE-COMPUTER paragraph, again, the computer on which the source program is compiled is the source computer.

OBJECT-COMPUTER Paragraph

The OBJECT-COMPUTER paragraph identifies the computer on which the program will be executed. Use of this paragraph is optional.

All clauses of the OBJECT-COMPUTER paragraph apply to the program in which they are explicitly or implicitly specified and to any program contained in that program.

The format of the OBJECT-COMPUTER paragraph is as follows:



OBJECT-COMPUTER

This keyword begins in area A and must be followed by a period.

computer-name

This name is a system-name (any COBOL word) that identifies the hardware for which object code is to be generated. The computer-name is optional.

MEMORY SIZE Clause

The SORT and MERGE statements can also specify MEMORY SIZE and take precedence over the OBJECT-COMPUTER paragraph. Refer to "MERGE Statement" in Section 7 and "SORT Statement" in Section 8 for more information. This clause specifies the actual main storage requirement needed for execution.

If you use this clause but a SORT or MERGE statement does not appear in the program, the clause is ignored. If you do not use this clause in either a SORT or MERGE statement or the OBJECT-COMPUTER paragraph, a default memory size of 12,000 words is assumed. (One module of memory is equivalent to 16,384 words of memory.)

Note that the MEMORY SIZE clause is an obsolete element in COBOL ANSI-85 and will be deleted from the next revision of the COBOL standard.

integer-1

The value contained in integer-1 specifies the number of bytes, words, or modules of main storage, exclusive of control program requirements that are available for object program execution.

WORDS
CHARACTERS
MODULES

You can specify the memory size in words with WORDS, in bytes with CHARACTERS, and in 16,384-word units with MODULES.

DISK SIZE Clause

This clause specifies the amount of disk space to be used for SORT operations.

The DISK SIZE clause is used only in conjunction with the SORT statement. If you omit the DISK SIZE clause from a program containing a SORT statement, DISK SIZE is assumed to be 900,000 words. If you use the DISK SIZE clause, but a SORT statement does not appear in the program, the DISK SIZE is ignored.

The DISK SIZE can be specified in either MODULES or WORDS. A module of disk is equivalent to 1.8 million words of disk.

PROGRAM COLLATING SEQUENCE Clause

If you use this clause, the program-collating sequence is the collating sequence associated with the alphabet-name specified in this clause. The same collating sequence is also applied to any nonnumeric merge or sort keys, unless the COLLATING SEQUENCE phrase of the respective MERGE or SORT statement is specified.

If this clause is not specified, the EBCDIC collating sequence is used.

alphabet-name-1

This name is a user-defined word.

The collating sequence associated with alphabet-name-1 is used to determine the truth value of any nonnumeric comparisons that are explicitly specified in relation conditions or condition-name conditions.

alphabet-name-2

This name is a user-defined word.

The collating sequence associated with alphabet-name-2 is used to determine the truth value of any national comparisons that are explicitly specified in relation conditions or in condition-name conditions.

When the PROGRAM COLLATING SEQUENCE clause is specified, the initial alphanumeric program collating sequence is the collating sequence associated with alphabet-name-1 and the initial national program collating sequence is the collating sequence associated with alphabet-name-2. When alphabet-name-1 is not specified, the initial alphanumeric program collating sequence is the native alphanumeric collating sequence, EBCDIC. When alphabet-name-2 is not specified, the initial program collating sequence is the native national collating sequence, JAPAN EBCDIC D1-2.

For localization purposes, the program can specify the PROGRAM COLLATING SEQUENCE clause and a CCSVERSION collating sequence associated with an alphabet-name. In this case, the truth value of the alphabetic characters that are explicitly specified in the class condition does not always consist entirely of the letters A through Z and the space character. The class of alphabetic characters is determined by the system collating sequence when the CCSVERSION collating sequence is specified.

When the PROGRAM COLLATING SEQUENCE clause is not specified for a given program, and the program is not contained within a program for which a PROGRAM COLLATING SEQUENCE clause is specified, the initial program collating sequences are the native alphanumeric collating sequence and the native national collating sequence.

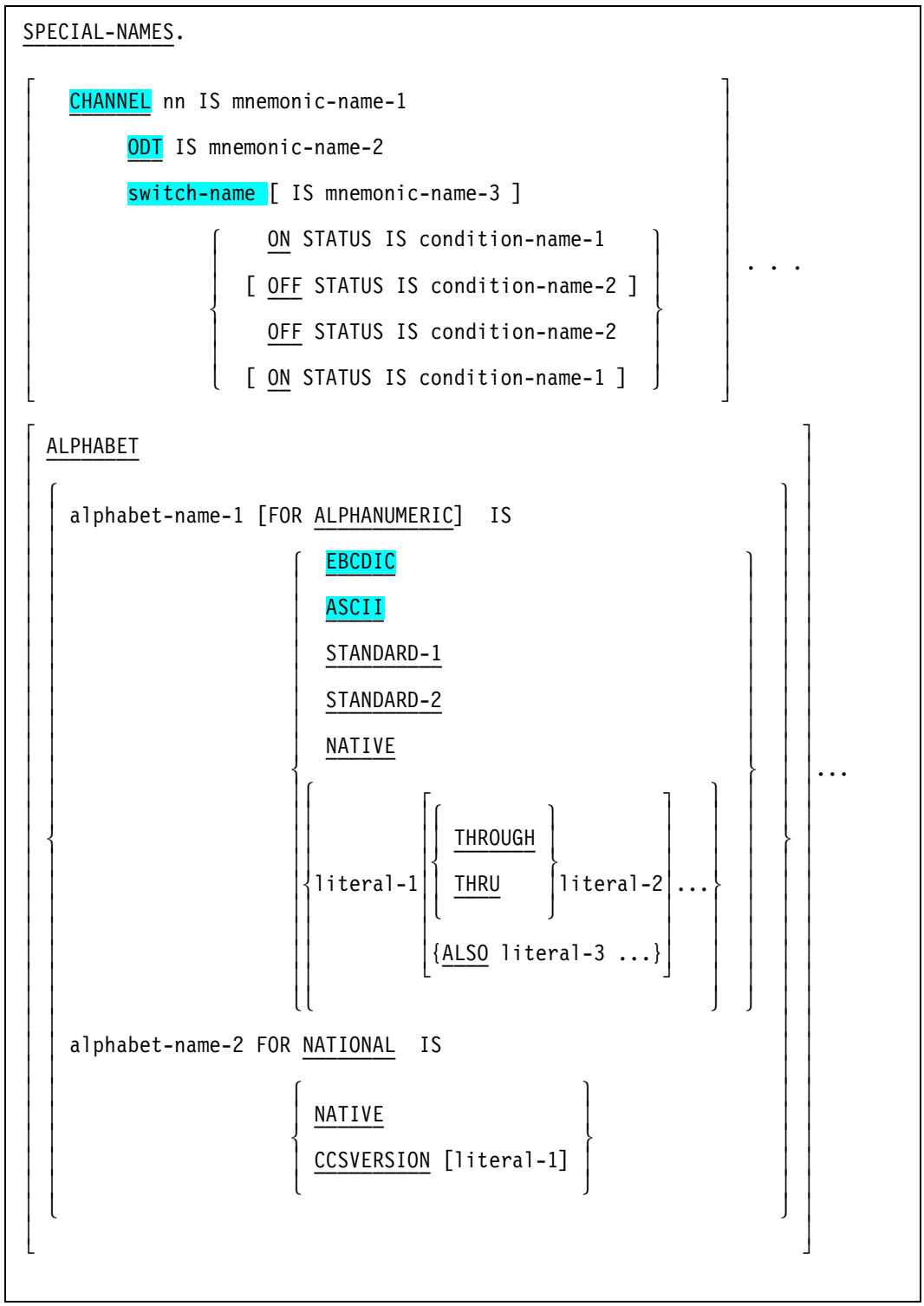
SPECIAL-NAMES Paragraph

The SPECIAL-NAMES paragraph does the following:

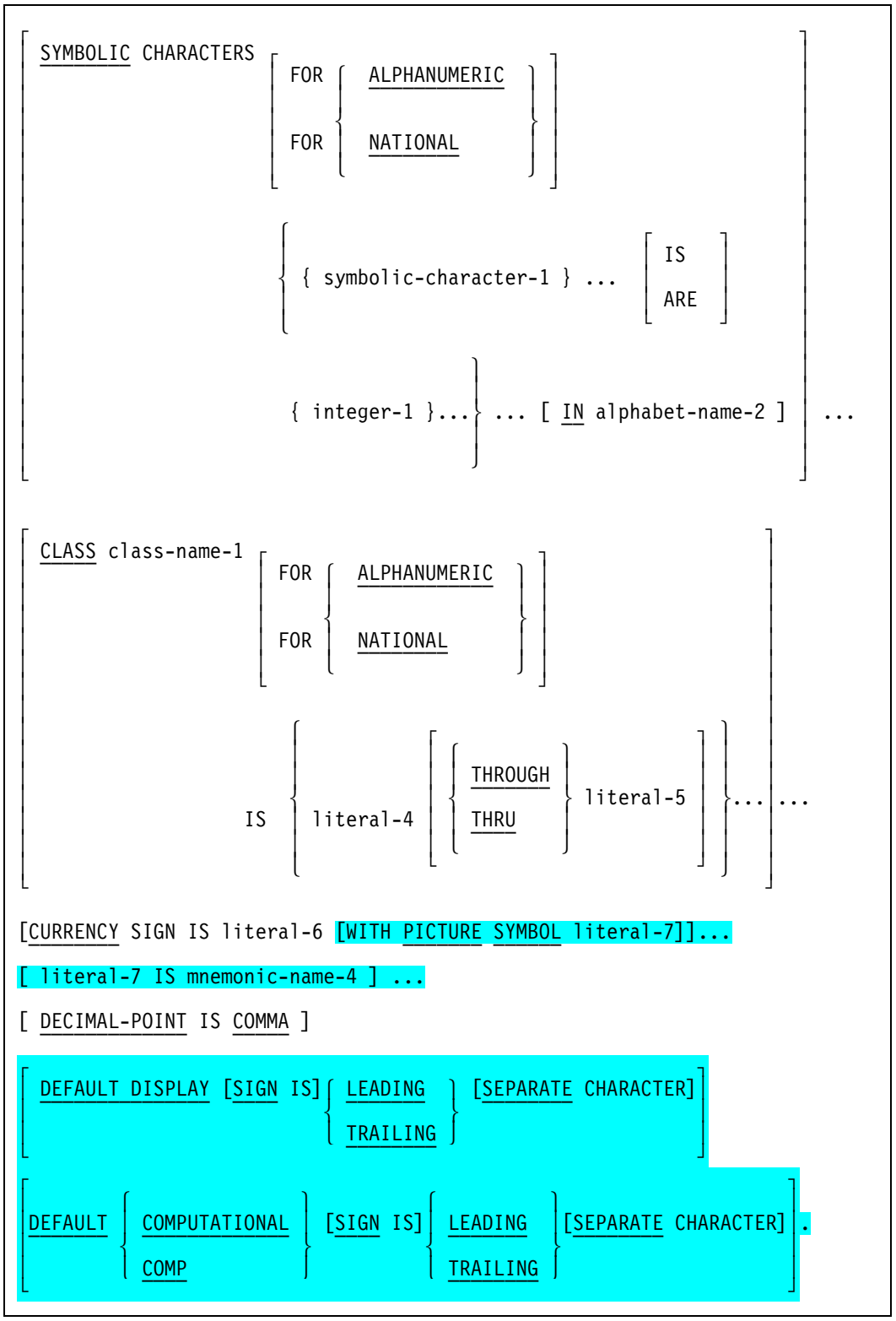
- Relates implementor-names used by the compiler to mnemonic-names used by the source program
- Assigns condition-names to the status of switches
- Relates alphabet-names to character sets or collating sequences
- Specifies symbolic-characters
- Relates class-names to sets of characters
- Exchanges the functions of the comma and the period in the PICTURE character string and in numeric literals
- Specifies a substitution character for the currency symbol in the PICTURE character string
- Changes default editing characters
- Specifies the default sign position for all signed data items whose usage is DISPLAY or COMPUTATIONAL
- Associates a mnemonic-name with an object program

This paragraph is optional. All clauses specified in the SPECIAL-NAMES paragraph for a program also apply to programs contained in that program.

The format of the SPECIAL-NAMES paragraph is as follows:



continued


SYMBOLIC CHARACTERS
FOR ALPHANUMERIC
FOR NATIONAL
{ symbolic-character-1 } ...
IS
ARE
{ integer-1 }...
... [IN alphabet-name-2]
...
CLASS class-name-1
FOR ALPHANUMERIC
FOR NATIONAL
IS
literal-4
THROUGH
THRU
literal-5
...
...
[CURRENCY SIGN IS literal-6 [WITH PICTURE SYMBOL literal-7]]...
[literal-7 IS mnemonic-name-4] ...
[DECIMAL-POINT IS COMMA]
DEFAULT DISPLAY [SIGN IS]
LEADING
TRAILING
[SEPARATE CHARACTER]
DEFAULT
COMPUTATIONAL
COMP
[SIGN IS]
LEADING
TRAILING
[SEPARATE CHARACTER].

CHANNEL Clause

This clause relates a mnemonic-name to a particular channel number. You can then use the mnemonic-name in a WRITE or SEND statement in place of CHANNEL nn. (WRITE and SEND statements are discussed in Section 8.)

nn

This is an integer from 01 to 11.

mnemonic-name-1

This name is a user-defined word that is associated with the channel number specified in the CHANNEL clause.

ODT Clause

This clause relates a mnemonic-name to the Operator Display Terminal (ODT). You can then use the mnemonic-name in an ACCEPT or DISPLAY statement. (The ACCEPT and DISPLAY statements are discussed in Section 6.)

mnemonic-name-2

This name is a user-defined word that is associated with the ODT.

SWITCH-NAME Clause

This clause associates mnemonic-names and condition-names with program switches.

switch-name

The switch-names you can use to specify the switches are SW1, SW2, SW3, SW4, SW5, SW6, SW7, and SW8.

mnemonic-name-3

This name is a user-defined word that can be associated with the switch-name.

This name can be referenced only in the SET statement. (The SET statement is discussed in Section 8.)

condition-name-1
condition-name-2

These condition-names are user-defined words that specify the status of a switch. One condition-name can be associated with the ON status, another with the OFF status. The condition-name associated with ON STATUS is TRUE when the switch is set, and FALSE when the switch is not set. The condition-name associated with OFF STATUS is TRUE when the switch is not set, and FALSE when the switch is set.

The status of the switch can be interrogated by testing these condition-names in the program's Procedure Division. The status of the switch can be altered by execution of a Format 3 SET statement, which specifies as its operand the mnemonic-name associated with that switch.

The condition-names specified in the containing program's SPECIAL-NAMES paragraph can be referred to from any contained program.

Details

Switches provide a means of communicating with the external environment. The meaning associated with each switch is user-defined. Switches can be set at program initiation time or through Work Flow Language (WFL) using the task attributes SW1, SW2, SW3, SW4, SW5, SW6, SW7, and SW8.

Refer to "Condition-Name Conditions" and "Switch-Status Conditions" in Section 5 for more information.

ALPHABET Clause

This optional clause relates alphabet-names to character sets or collating sequences.

alphabet-name-1
alphabet-name-2

This is a user-defined word that assigns a name to a specific character code set or collating sequence.

An alphabet name can consist of the characters *A* through *Z*, *a* through *z*, *0* through *9*, and the hyphen (-). You cannot use the hyphen or *0* through *9* as the first character, and you cannot use the hyphen as the last character.

When alphabet names are referred to in the PROGRAM COLLATING clause of the OBJECT-COMPUTER paragraph or in the COLLATING SEQUENCE phrase of a SORT or MERGE statement, the ALPHABET clause specifies a collating sequence.

When alphabet names are referred to in the SYMBOLIC CHARACTERS clause or in a CODE-SET clause in a file description entry, the ALPHABET clause specifies a character code set.

EBCDIC
ASCII
STANDARD-1
STANDARD-2
NATIVE

STANDARD-1 and ASCII indicate that alphabet-name-1 is the character code set and collating sequence defined by the American National Standard Code for Information Interchange, X3.4-1977.

STANDARD-2 indicates that alphabet-name-1 is the character code set and collating sequence defined by the International Reference Version of the ISO 7-bit code defined in International Standard 646, 7-Bit Coded Character Set for Information Processing Interchange.

If the NATIVE phrase is specified, the native character code set and native collating sequence are identified with alphabet-name-1. The native character code set is the character code set associated with DISPLAY usage, EBCDIC.

When the NATIVE phrase is specified for a national alphabet name, the native national coded character set and native national collating sequence are defined as JAPAN EBCDIC D1-2.

The correspondence between characters of the ASCII character code set and characters of the EBCDIC character code set is determined by the standard translation tables for EBCDIC-to-ASCII and ASCII-to-EBCDIC.

literal-1
literal-2
literal-3

If the literal phrase of the ALPHABET clause is specified:

- A given character must not be specified more than once in that clause.
- The alphabet-name cannot be referred to in a CODE-SET clause.

The following syntax rules apply to the literals specified in the literal phrase of the ALPHABET clause:

- If numeric, the literals must be unsigned integers and must have values in the range of 1 through 256.
- If nonnumeric and associated with a THROUGH (THRU) or ALSO phrase, each literal must be one character in length.

Note that literal-1, literal-2, and literal-3 must not specify a symbolic-character figurative constant.

THROUGH
THRU

These keywords are equivalent.

ALSO

If you specify the ALSO phrase, the characters of the native character set specified by the value of literal-1 and literal-3 are assigned to the same ordinal position in the collating sequence being specified or in the character code set that is used to represent the data. If alphabet-name-1 is referenced in a SYMBOLIC CHARACTERS clause, only literal-1 is used to represent the character in the native character set.

Refer to "OBJECT-COMPUTER Paragraph" in this section, "SORT Statement" in Section 8, and "MERGE Statement" in Section 7.

CCSVERSION

If the CCSVERSION option is specified, the character code set and the collating sequence identified with the alphabet-name is the system collating sequence. If the CCSVERSION phrase is specified without literal-1, the collating sequence identified with the alphabet-name is the internationalized system default collating sequence. If the CCSVERSION phrase is specified with literal-1, the collating sequence is identified by literal-1, provided that literal-1 is valid. The alphabet-name cannot be referred to in a CODE-SET clause.

If the CCSVERSION "ASERIESNATIVE" is specified, the native national coded character set and native national collating sequence are referenced as JAPAN EBCDIC D1-2.

Example of CCSVERSION Defined at RUN Time

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
OBJECT-COMPUTER.
    PROGRAM COLLATING SEQUENCE FOR NATIONAL IS CCS.
SPECIAL-NAMES.
    ALPHABET CCS FOR NATIONAL IS CCSVERSION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 DATA1 PIC N(10).
PROCEDURE DIVISION.
BEGIN.
    MOVE HIGH-VALUES TO DATA1.
    IF DATA1 = HIGH-VALUES
    DISPLAY "OK".
    STOP RUN.
```

Example of CCSVERSION Specified by <literal-1>

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
OBJECT-COMPUTER.
    PROGRAM COLLATING SEQUENCE FOR NATIONAL IS CCS.
SPECIAL-NAMES.
    ALPHABET CCS FOR NATION AL IS CCSVERSION "FRANCE".
DATA DIVISION.
WORKING-STORAGE SECTION.
01 DATA1 PIC N(10).
PROCEDURE DIVISION.
BEGIN.
    MOVE HIGH-VALUES TO DATA1.
    IF DATA1 = HIGH-VALUES
    DISPLAY "OK".
    STOP RUN.
```

The CCSVERSION phrase can be specified only once in a program.

Note: *Using the internationalized system-default ccsversion can produce unexpected results for the HIGH-VALUE and LOW-VALUE figurative constants if a program is run on a host with a system-default ccsversion that differs from the ccsversion compiled into the program. In this case, the HIGH-VALUE and LOW-VALUE figurative constants contain values that are correct for the ccsversion compiled into the program. For example, if the program is compiled on a host with a system-default ccsversion of SPANISH and the program is run on a host with a default ccsversion of FRANCE, the HIGH-VALUE and LOW-VALUE constants define their values from the SPANISH ccsversion at compile time, not from the FRANCE ccsversion.*

Rules for the ALPHABET Clause

The collating sequence identified in the ALPHABET clause is defined according to the following rules:

- If the ALPHABET clause is specified without either the ALPHANUMERIC or the NATIONAL phrase, the ALPHANUMERIC phrase is used.
- The value of each literal specifies the following:
  - If numeric, the literal defines the ordinal number of a character in the native character set. This value must not exceed 256.
  - If nonnumeric, the literal defines the actual character in the native character set. If the value of the nonnumeric literal contains multiple characters, each character in the literal, starting with the leftmost character, is assigned successive ascending positions in the specified collating sequence.
- The order in which the literals appear in the ALPHABET clause determines, in ascending sequence, the ordinal numbers of the characters in the specified collating sequence.

- Any characters in the native collating sequence that are not explicitly defined in the literal phrase assume a position in the specified collating sequence that is greater than any of the explicitly specified characters. The relative order in the set of these unspecified characters is unchanged from the native collating sequence.
- If the THROUGH (THRU) phrase is used, the set of contiguous characters in the native character set, beginning with the character defined by the value of literal-1 and ending with the character defined by the value of literal-2, is assigned a successive ascending position in the specified collating sequence. In addition, the set of contiguous characters defined by a given THROUGH (THRU) phrase can specify characters of the native character set in either ascending or descending sequence.
- If the ALSO phrase is used, the characters of the native character set specified by the value of literal-1 and literal-3 are assigned to the same ordinal position in the specified collating sequence or in the character code set that is used to represent the data. If alphabet-name-1 is referred to in a SYMBOLIC CHARACTERS clause, only literal-1 is used to represent the character in the native character set.

The character that has the highest ordinal position in the program collating sequence is associated with the figurative constant HIGH-VALUE, except when this figurative constant is defined as a literal in the SPECIAL-NAMES paragraph. If more than one character has the highest position in the program collating sequence, the last character specified is associated with the figurative constant HIGH-VALUE.

The character that has the lowest ordinal position in the specified program-collating sequence is associated with the figurative constant LOW-VALUE, except when this figurative constant is defined as a literal in the SPECIAL-NAMES paragraph. If more than one character has the lowest position in the program-collating sequence, the first character specified is associated with the figurative constant LOW-VALUE.

When defined as literals in the SPECIAL-NAMES paragraph, the figurative constants HIGH-VALUE and LOW-VALUE are associated with those characters having the highest and lowest positions, respectively, in the native collating sequence.

SYMBOLIC CHARACTERS Clause

This optional clause specifies symbolic characters.

symbolic-character-1
integer-1

There must be a one-to-one correspondence between occurrences of symbolic-character-1 and occurrences of integer-1.

The internal representation of symbolic-character-1 is the internal representation of the character that is used in the native character set.

A symbolic-character-1 can appear only once in a SYMBOLIC CHARACTERS clause.

The relationship between each symbolic-character-1 and the corresponding integer-1 is determined by position in the SYMBOLIC CHARACTERS clause. The first symbolic-character-1 is paired with the first integer-1, the second symbolic-character-1 is paired with the second integer-1, and so on.

The ordinal position specified by integer-1 must exist in the native character set.

There must be a one-to-one correspondence between occurrences of symbolic-character-1 and occurrences of integer-1.

When the SYMBOLIC CHARACTERS clause is specified with neither the ALPHANUMERIC nor the NATIONAL phrase, the ALPHANUMERIC phrase is implied.

When the NATIONAL phrase is specified, the following conditions apply:

- When the IN phrase is specified, alphabet-name-2 references an alphabet that defines a single-octet national character set; the ordinal position specified by integer-1 exists in that character set.
- When the IN phrase is not specified, the ordinal position specified by integer-1 exists in the national character set specified with the "ALPHABET FOR NATIONAL IS CCSVERSION" clause.

IN alphabet-name-2

The alphabet name is a user-defined word.

If the IN phrase is used, integer-1 determines the ordinal position of the character that is represented in the character set named by alphabet-name-2.

If the IN phrase is not used, symbolic-character-1 represents the character whose ordinal position in the native character set is determined by integer-1.

CLASS Clause

This optional clause relates a name to the set of characters listed in the clause.

class-name-1

This name is a user-defined word that can be referred to only in a class condition.

The characters specified by the values of the literals of this clause define the exclusive set of characters of which this name consists.

literal-4
literal-5

When the CLASS clause is specified without the ALPHANUMERIC or the NATIONAL phrase, the ALPHANUMERIC phrase is implied.

When the ALPHANUMERIC phrase is specified or implied the following conditions apply:

- If literal-4 is numeric, the literal specifies the ordinal number of a character in the native character set. This value cannot exceed 256.
- If literal-4 is nonnumeric, the literal specifies the actual character in the native character set. If the value of the literal contains multiple characters, each character in the literal is included in the set of characters identified by class-name-1.

When the NATIONAL phrase is specified the following conditions apply:

- If literal-4 is numeric, it is an unsigned integer and has a value within the range of one through the number of characters in the national character set specified with the ALPHABET FOR NATIONAL IS CCSVERSION clause.
- Each non-integer literal is a national literal.
- The THROUGH (THRU) phrase cannot be specified for a national character.
- The number of characters specified cannot exceed the number of characters in the national character set specified with the ALPHABET FOR NATIONAL IS CCSVERSION clause.

Note: *The aforementioned literals cannot specify a symbolic-character figurative constant.*

The following syntax rules apply to the literals specified in the literal phrase of the CLASS clause:

- If numeric, the literals must be unsigned integers and must have values in the range of 1 through 256.
- If nonnumeric and associated with a THROUGH (THRU) phrase, each literal must be one character in length.

THROUGH
THRU

These keywords are equivalent.

If the THROUGH (THRU) phrase is used, the contiguous characters in the native character set, beginning with the character specified by the value of literal-4 and ending with the character specified by the value of literal-5, are included in the set of characters identified by class-name-1. In addition, the contiguous characters identified by a given THROUGH (THRU) phrase can specify characters of the native character set in either ascending or descending sequence.

CURRENCY SIGN Clause

The CURRENCY SIGN clause specifies a currency string that is placed into numeric-edited data items when they are used as receiving items. The CURRENCY SIGN clause also specifies a currency string that is placed into de-edited data items when they are used as sending items that have a numeric or numeric-edited receiving item. In addition, the clause is used to determine which symbol will be used in a picture character string to specify the presence of a currency string. This symbol is referred to as the currency symbol.

If the CURRENCY SIGN clause is specified without the PICTURE SYMBOL phrase, literal-6 is used as the currency symbol. If the CURRENCY SIGN clause is specified with the PICTURE SYMBOL phrase, literal-7 is used as the currency symbol.

literal-6

Literal-6 represents the value of the currency string. Literal-6 must be an alphanumeric or national literal that is not a figurative constant.

If the PICTURE SYMBOL phrase is not specified, then literal-6 is specified and can consist of only a single character. In this case, literal-6 can be any single character from the character set except for the following:

- Digits 0 through 9
- Alphabetic characters consisting of the uppercase letters A, B, C, D, E, N, P, R, S, V, X, Z; the lowercase form of these alphabetic characters; and the space character
- Special characters consisting of the plus sign (+), the minus sign (-), the comma (,), the period (.), the asterisk (*), the slant (/), the semicolon (;), parenthesis (*( )*), the double quotation mark ("), and the equal sign (=)

If the PICTURE SYMBOL phrase is specified, then literal-6 can be any number of characters. In this case, it must contain at least one non-space character and can consist of any characters from the character set except for the following:

- Digits 0 through 9
- Special characters consisting of the plus sign (+), the minus sign (-), the comma (,), the period (.), and the asterisk (*)

literal-7

Literal-7 can be any single character from the character set except for the following:

- Digits 0 through 9
- Alphabetic characters consisting of the uppercase letters A, B, C, D, E, N, P, R, S, V, X, Z; the lowercase form of these alphabetic characters; and the space character
- Special characters consisting of the plus sign (+), the minus sign (-), the comma (,), the period (.), the asterisk (*), the slant (/), the semicolon (;), parenthesis (( )), the double quotation mark ("), and the equal sign (=)

Literal-7 IS MNEMONIC-NAME Clause

This clause associates an object-program with a user-defined mnemonic name. It can be used to declare the file name of a program to be bound or a program to be initiated as a separate procedure.

In this clause, literal-7 must be a valid file title. It can be of the form AAA/BBB/CCC . . ., where each group of characters between two slashes is one directory, or file title node, of the file title. A directory name can contain a maximum of 17 characters. A file title can consist of a maximum of 14 directories. Mnemonic-name is a user-defined word of your choice.

For information on binding, refer to Appendix E. For details about tasking and structuring a COBOL85 program to initiate separately compiled programs, refer to Section 13.

DECIMAL-POINT Clause

This optional clause exchanges the functions of the comma and the period in the character string of the PICTURE clause and in numeric literals. All numeric literals used in the program are affected by this clause.

With this clause in use, a comma used as a separator must be followed by a space. The space is required because a comma immediately followed by a numeric literal is interpreted as a decimal point by the compiler.

DEFAULT DISPLAY SIGN and DEFAULT COMPUTATIONAL SIGN Clauses

.These optional clauses specify the default sign position for all signed data items whose usages are DISPLAY or COMPUTATIONAL, respectively. The default sign position specified with these clauses is used when a signed data item is declared in the Data Division without the optional SIGN clause. If the SIGN clause is used in the Data Division, it overrides the DEFAULT clauses specified in this division. For more information about signed data items, refer to the discussion of the SIGN clause and character S of thePICTURE clause in Section 4.

Example of the SPECIAL-NAMES Paragraph

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PAYROL.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
         SW5 ON STATUS IS SW5-ON
                 OFF STATUS IS SW5-OFF;
         CURRENCY SIGN IS "E";
         DECIMAL-POINT IS COMMA.
```

In this example, the program PAYROL includes the optional Configuration Section in its Environment Division. The source and object computers are the same (both are A5s). A switch, SW5, is named in the SPECIAL-NAMES paragraph, and the condition-names SW5-ON and SW5-OFF are used to specify ON STATUS and OFF STATUS, respectively, of this switch. The one-character nonnumeric literal E defined in the CURRENCY SIGN clause will replace the dollar sign character ($) in the PICTURE clause. The DECIMAL-POINT clause is present, so the comma will replace the period in the PICTURE clause and in numeric literals, and the period will replace the comma.

Input-Output Section

The Input-Output Section includes the information needed to control transmission and handling of data between external media and the object program. This section is optional in a COBOL source program.

The Input-Output Section is divided into the following two paragraphs:

- The FILE-CONTROL paragraph names and associates the files with external media.
- The I-O-CONTROL paragraph defines special control techniques to be used in the object program.

Input-Output Section Header

The following header identifies and must begin the Input-Output Section:

```
INPUT-OUTPUT SECTION.
```

INPUT-OUTPUT SECTION

These keywords begin in area A and must be followed by a period.

FILE-CONTROL Paragraph

The FILE-CONTROL paragraph does the following:

- Names each file
- Identifies the file medium
- Specifies hardware
- Specifies alternate input/output areas
- Specifies the organization of the file

The FILE-CONTROL paragraph is required.

General Format of the FILE-CONTROL Paragraph

The general format of the FILE-CONTROL paragraph is as follows:

```
FILE-CONTROL. { file control entry } ...
```

FILE-CONTROL

This keyword begins in area A and must be followed by a period.

file control entry

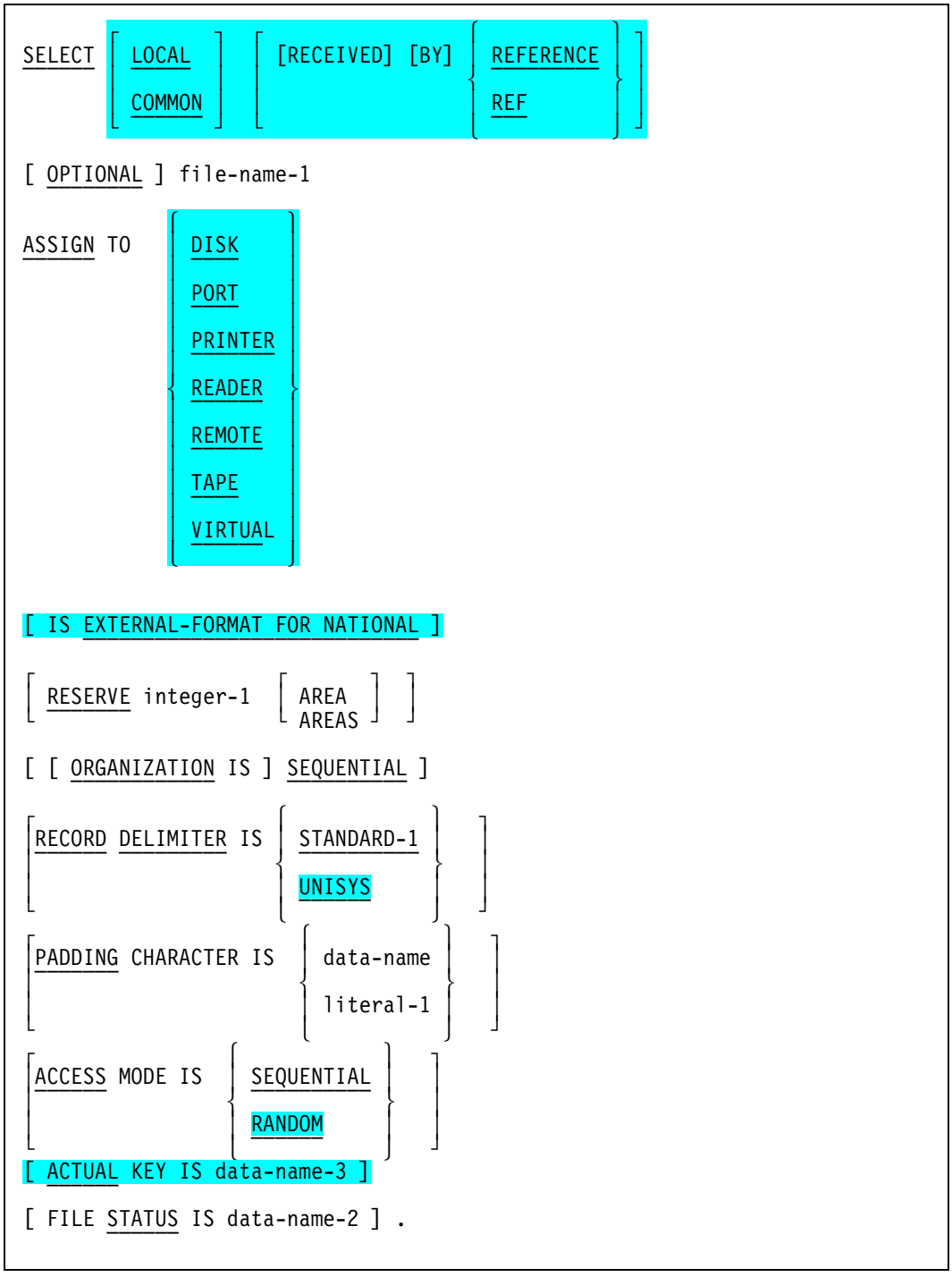
The file control entry has the following four formats:

- Format 1 declares the physical attributes of a sequential file.
- Format 2 declares the physical attributes of a relative file.
- Format 3 declares the physical attributes of an indexed file.
- Format 4 declares the physical attributes of a sort or merge file.

Each of these formats is discussed in the following pages.

File Control Entry Format 1: Sequential Organization

You can use this format to declare the physical attributes of a sequential file.



In the FILE-CONTROL paragraph, you must specify the SELECT clause first. The clauses that follow the SELECT clause can appear in any order.

SELECT Clause

LOCAL

This phrase specifies that the file is a formal parameter for a procedure. A file specified as LOCAL can be named only in the WITH clause and USING clause of one of the following:

- The ENTRY PROCEDURE clause associated with a procedure imported from a library
- The USE statement associated with a separately compiled, or bound procedure

COMMON

This phrase specifies that the file is declared in another module to which this program is to be bound. The file description and record description entries in each module in which the file is declared COMMON must match.

Note: *The compiler option COMMON does not affect entries in the Environment Division or in the File Section of the Data Division.*

RECEIVED BY REFERENCE
REF

This phrase allows two or more programs to use the file. Because access to the file is by reference, any program can perform input-output operations to the file.

OPTIONAL

This phrase only applies to files opened in input, I-O, or extend mode. It is required for files that are not necessarily present each time the object program is executed.

If you designate an input file with the OPTIONAL phrase in its SELECT clause, and the file is not present at the time the OPEN statement is executed, the operator is notified of this fact. At this time, the file can be loaded, or the operator can enter the system command OF. If the operator uses the OF command, the first READ statement for this file causes an AT END or INVALID KEY condition to occur. Refer to the *System Commands Operations Reference Manual* for information on the OF command.

file-name-1

This is a user-defined word that names a file connector.

Each file-name specified in the SELECT clause must have a file description entry in the Data Division of the same program. Also, each file-name in the Data Division must be specified only once in the FILE-CONTROL paragraph.

If the file connector referred to by file-name-1 is an external file connector (refer to "EXTERNAL Clause" in Section 4 and to "File Connectors" in Section 10), all file control entries in the run unit that refer to this file connector must have:

- The same specification for the OPTIONAL phrase
- A consistent specification in the ASSIGN clause
- A consistent specification in the RECORD DELIMITER clause
- The same value for integer-1 in the RESERVE clause
- The same organization
- The same access mode
- The same specification for the PADDING CHARACTER clause

ASSIGN Clause

This clause associates the file referenced by file-name-1 to a storage medium.

You can assign file-name-1 to the following:

- DISK
- PORT
- PRINTER
- READER
- VIRTUAL

IS EXTERNAL-FORMAT FOR NATIONAL Clause

The IS EXTERNAL-FORMAT FOR NATIONAL clause causes data items of the national class to be transmitted in *external format* to be suitable for display or printing. External format means that the control characters SDO (for "start of double octet") and EDO (for "end of double octet") are inserted at the beginning and the end of the data to distinguish it as national data.

This clause can be specified only for remote files and printer files.

Files with this clause cannot be referenced by a SAME clause (see "Input-Output Control Entry Format 1: Sequential I/O" later in this section for details about the SAME clause).

If the CCSVERSION clause is specified, the EXTERNAL-FORMAT FOR NATIONAL option is ignored and a warning is issued.

RESERVE Clause

This clause specifies the number of input-output areas allocated.

integer-1

If the RESERVE clause is specified, the number of input-output areas allocated is equal to the value of integer-1.

If the RESERVE clause is not specified, two input-output areas are automatically allocated.

ORGANIZATION IS SEQUENTIAL Clause

This clause specifies sequential organization as the logical structure of a file. If If this clause is not used, sequential organization is implied.

Details

Sequential organization is a permanent logical file structure in which a record is identified by a predecessor-successor relationship. This relationship is established when the record is placed into the file.

The file organization is established at the time a file is created and cannot subsequently be changed.

RECORD DELIMITER Clause

This clause indicates the method of determining the length of a variable-length record on the external medium. Any method used will not be reflected in the record area or the record size used in the program.

Note that this clause can be specified only for variable-length records.

STANDARD-1
UNISYS

If either STANDARD-1 or UNISYS is specified, the external medium must be a magnetic tape file.

If this phrase is specified, the method used for determining the length of a variable length record is that specified in American National Standard X3.27-1978, Magnetic Tape Labels and File Structure for Information Interchange, and in International Standard 1001 1979, Magnetic Tape Labels and File Structure for Information Interchange.

Details

At the time the OPEN statement that creates the file is executed, the record delimiter used is the one specified in the RECORD DELIMITER clause associated with the file-name specified in the OPEN statement.

If the associated file connector is an external file connector, all RECORD DELIMITER clauses in the run unit that are associated with that file connector must have the same specifications.

PADDING CHARACTER Clause

This clause is for documentation purposes only.

ACCESS MODE IS SEQUENTIAL Clause

This clause specifies the order in which records are to be accessed in the file. If you specify SEQUENTIAL, records are accessed sequentially. If you do not use this clause, sequential access is assumed.

ACCESS MODE IS RANDOM Clause

If you specify RANDOM, records are accessed randomly. Random access can be specified for mass-storage files only.

Details

Records in the file are accessed in the sequence dictated by the file organization. For sequential files, this sequence is specified by predecessor-successor record relationships established by the execution of WRITE statements when the file is created or extended.

If the associated file connector is an external file connector, every file control entry in the run unit that is associated with that file connector must specify the same access mode.

ACTUAL KEY Clause

For mass-storage files that specify an ACTUAL KEY, the value of the ACTUAL KEY data item specifies the logical ordinal position of the record in the file.

For port files, the value of the ACTUAL KEY data item specifies the subfile index of the port file. The ACTUAL KEY clause must be specified for a port file that contains more than one subfile.

For remote files, the value of the ACTUAL KEY data item specifies the ordinal number of the station within the station list of the remote file. A zero value specifies all stations within the station list of the remote file.

data-name-3

This name is a user-defined word that must refer to an unsigned integer data item whose description does not contain the PICTURE symbol *P*.

This name can be qualified.

Note that you can significantly improve the performance of all I/O statements that act upon a sequential file declared with an actual key by declaring the appropriate key as follows:

```
77 USERKEY     REAL.
```

FILE STATUS Clause

This clause specifies a data item that contains the status of an input-output operation.

data-name-2

This name is a user-defined word. This name must be defined in the Data Division as a two-character alphanumeric data item and must not be defined in the File Section. This name can be qualified.

The data item referred to by data-name-2 is the one specified in the file control entry associated with that statement. See "General Format of the Environment Division" in this section.

Details

When the FILE STATUS clause is specified, the data item referred to by data-name-2 is updated to contain the value of the I-O status whenever the I-O status is updated. This value indicates the status of execution of the statement. See"I-O Status Codes" in this section.

Example of File Control Entry Format 1: Sequential Organization

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PAYROL.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    SW5 ON STATUS IS SW5-ON
        OFF STATUS IS SW5-OFF;
    CURRENCY SIGN IS "E";
    DECIMAL-POINT IS COMMA.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT DATA-CAPTURE ASSIGN TO DISK;
        ORGANIZATION IS SEQUENTIAL;
        ACCESS MODE IS SEQUENTIAL;
        FILE STATUS IS FS-1.
        SELECT PRINTOUT ASSIGN TO PRINTER.
    .
    .
    .
DATA DIVISION.
WORKING-STORAGE SECTION.
77 FS-1            PIC XX.
```

The program PAYROL includes both a Configuration Section and an Input-Output Section. File Control Entry Format 1 is used in the FILE-CONTROL paragraph. The input file DATA-CAPTURE, a sequential file, will be stored on disk. The records of this file will be accessed sequentially. (If the ACCESS MODE clause is not specified, sequential access is assumed.) The data item FS-1 is specified in the FILE STATUS clause and is defined in the Data Division. A value will be moved by the operating system into FS-1 after the execution of every statement that refers to that file. This value indicates the status of execution of the statement. The output file PRINTOUT is assigned to a printer.

File Control Entry Format 2: Relative Organization

You can use this format to declare the physical attributes of a relative file.

```
SELECT [ LOCAL    ] [ [RECEIVED] [BY] { REFERENCE } ]
       [ COMMON   ] [                 { REF       } ]

[ OPTIONAL ] file-name-1

ASSIGN TO DISK

[ RESERVE integer-1 [ AREA  ] ]
                    [ AREAS ]

[ ORGANIZATION IS ] RELATIVE

[ACCESS MODE IS  { SEQUENTIAL [ RELATIVE KEY IS data-name-1 ] }   ]
                 {{RANDOM  }                                   }  |
                 {{DYNAMIC }     RELATIVE KEY IS data-name-1   }  |
                                                               ]

[ FILE STATUS IS data-name-2 ] .
```

Refer to File Control Entry Format 1 for information on the RESERVE clause and the FILE STATUS clause.

SELECT Clause

Refer to File Control Entry Format 1 for information on the SELECT clause, the LOCAL phrase, the COMMON phrase, the RECEIVED BY REFERENCE phrase, and the OPTIONAL phrase.

In addition, if the file connector referred to by file-name-1 is an external file connector (refer to "EXTERNAL Clause" in Section 4 and to "File Connectors" in Section 10), all file control entries in the run unit that reference this file connector must have:

- The same specification for the OPTIONAL phrase
- A consistent specification in the ASSIGN clause
- The same value for integer-1 in the RESERVE clause
- The same organization
- The same access mode
- The same external data item for data-name-1 in the RELATIVE KEY phrase

ASSIGN Clause

Refer to File Control Entry Format 1 for information on this clause.

In addition, in this format DISK specifies that mass storage is the storage medium of the file. You can define the medium more precisely in the VALUE OF clause of the FD entry in the Data Division or through the use of file equation.

ORGANIZATION IS RELATIVE Clause

In this format, this clause specifies relative organization as the logical structure of a file.

Details

Relative organization is a permanent logical file structure in which each record is uniquely identified by an integer value greater than zero, which specifies the record's logical ordinal position in the file.

The file organization is established at the time a file is created and cannot subsequently be changed.

ACCESS MODE Clause

This clause specifies the order in which records are to be accessed in the file.

There are three forms of the ACCESS MODE clause in this format: the ACCESS MODE IS SEQUENTIAL clause, the ACCESS MODE IS RANDOM clause, and the ACCESS MODE IS DYNAMIC clause.

If this clause is not used, sequential access is assumed.

ACCESS MODE IS SEQUENTIAL

If the access mode is sequential, records in the file are accessed in the sequence dictated by the file organization. For relative files, this sequence is the order of ascending relative record numbers of existing records in the file.

ACCESS MODE IS RANDOM

If the access mode is random, the value of the relative key data item for relative files indicates the record to be accessed.

Note that this access mode must not be specified for file-names specified in the USING or GIVING phrase of a SORT or MERGE statement.

ACCESS MODE IS DYNAMIC

If the access mode is dynamic, records in the file can be accessed sequentially and/or randomly.

RELATIVE KEY

If a relative file is referred to by a START statement, the RELATIVE KEY phrase within the ACCESS MODE clause must be specified for that file.

data-name-1

This name is a user-defined word that must refer to an unsigned integer data item whose description does not contain the PICTURE symbol *P*. The data item specified by data-name-1 is used to communicate a relative record number to the I-O handler.

This name can be qualified.

This name must not be defined in a record description entry associated with that file-name.

The relative key data item associated with the execution of an input-output statement is the data item referred to by data-name-1 in the ACCESS MODE clause.

Details

All records stored in a relative file are uniquely identified by relative record numbers. The relative record number of a given record specifies the record's logical ordinal position in the file. The first logical record has a relative record number of 1, and subsequent logical records have relative record numbers of 2, 3, 4, and so forth.

If the associated file connector is an external file connector, every file control entry in the run unit associated with that file connector must specify the same access mode. In addition, data-name-1 must reference an external data item and the RELATIVE KEY phrase in each associated file control entry must reference that same external data item in each case.

Example of File Control Entry Format 2: Relative Organization

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PAYROL.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    SW5 ON STATUS IS SW5-ON
        OFF STATUS IS SW5-OFF;
    CURRENCY SIGN IS "E";
    DECIMAL-POINT IS COMMA.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT DATA-CAPTURE-2 ASSIGN TO DISK;
    ORGANIZATION IS RELATIVE;
    ACCESS MODE IS DYNAMIC; RELATIVE KEY IS ACCT-NO.
```

In this example, the program PAYROL includes both a Configuration Section and an Input-Output Section in its Environment Division. File Control Entry Format 2 is used in the FILE-CONTROL paragraph. The input file DATA-CAPTURE-2, which is a relative file, will be stored on disk. The file organization is relative. The access mode is dynamic, so the records of DATA-CAPTURE-2 can be accessed either sequentially (that is, in ascending order by relative record number) or randomly (that is, in a sequence determined by use of the RELATIVE KEY phrase). The desired record is accessed by placing its relative record number in the RELATIVE KEY data item, ACCT-NO.

File Control Entry Format 3: Indexed I/O

You can use this format to declare the physical attributes of an indexed file.

```
SELECT [ LOCAL  ]  [ [ RECEIVED ] [ BY ] { REFERENCE } ]
       [ COMMON ]  [                     { REF       } ]

[ OPTIONAL ] file-name-1

ASSIGN TO DISK

[ RESERVE integer-1 [ AREA  ] ]
                    [ AREAS ]

[  ORGANIZATION IS ] INDEXED

[ACCESS MODE IS { SEQUENTIAL } ]
                { RANDOM     }
                { DYNAMIC    }
RECORD KEY IS data-name-1

[ ALTERNATE RECORD KEY IS data-name-2 [ WITH DUPLICATES ] ] ...

[ FILE STATUS IS data-name-3 ] .
```

Refer to File Control Entry Format 1 for information on the RESERVE clause.

SELECT Clause

Refer to File Control Entry Format 1 for information on the SELECT clause, the LOCAL phrase, the COMMON phrase, the RECEIVED BY REFERENCE phrase, and the OPTIONAL phrase.

In addition, in this format, if the file connector referred to by file-name-1 is an external file connector (refer to "EXTERNAL Clause" in Section 4 and to "File Connectors" in Section 10), all file control entries in the run unit that reference this file connector must have:

- The same specification for the OPTIONAL phrase
- A consistent specification in the ASSIGN clause
- The same value for integer-1 in the RESERVE clause
- The same organization
- The same access mode
- The same data description entry for data-name-1 with the same relative location within the associated record
- The same data description entry for data-name-2, the same relative location within the associated record, the same number of alternate record keys, and the same DUPLICATES phrase

ASSIGN Clause

Refer to File Control Entry Format 1 for information on this clause.

In addition, in this format, DISK specifies that mass storage is the storage medium of the file. You can define the medium more precisely in the VALUE OF clause of the FD in the Data Division or with file equation.

ORGANIZATION IS INDEXED Clause

In this format, this clause specifies indexed organization as the logical structure of a file.

Details

Indexed organization is a permanent logical file structure in which each record is identified by the value of one or more keys within that record.

The file organization is established at the time a file is created and cannot subsequently be changed.

The native character set is assumed for data on the external media.

For an indexed file, the collating sequence associated with the native character set is assumed. This is the sequence of values of a given key of reference used to process the file sequentially.

ACCESS MODE Clause

This clause specifies the order in which records are to be accessed in the file.

There are three forms of the ACCESS MODE clause in this format: the ACCESS MODE IS SEQUENTIAL clause, the ACCESS MODE IS RANDOM clause, and the ACCESS MODE IS DYNAMIC clause.

If this clause is not specified, sequential access is assumed.

ACCESS MODE IS SEQUENTIAL

If the access mode is sequential, records in the file are accessed in the sequence dictated by the file organization. For indexed files, this sequence is ascending within a given key of reference according to the collating sequence of the file.

ACCESS MODE IS RANDOM

If the access mode is random, the value of a record key data item for indexed files indicates the record to be accessed.

Note that this access mode must not be specified for file-names specified in the USING or GIVING phrase of a SORT or MERGE statement.

ACCESS MODE IS DYNAMIC

If the access mode is dynamic, records in the file can be accessed sequentially and/or randomly.

Details

If the associated file connector is an external file connector, every file control entry in the run unit that is associated with that file connector must specify the same access mode.

RECORD KEY Clause

This clause specifies a prime record key for the file with which this clause is associated. The values of the prime record key must be unique among the records of the file. The prime record key provides an access path to records in an indexed file.

If the indexed file contains variable length records, the prime record key must be contained within the first *x* character positions of the record, where *x* equals the minimum record size specified for the file (refer to "RECORD Clause" in Section 4).

data-name-1

This name is a user-defined word. It must reference an alphanumeric or a numeric data item in a record description entry associated with the file-name to which the RECORD KEY clause is subordinate.

This name can be qualified.

This name must not reference a group item that contains a variable occurrence data item.

The data description of data-name-1, as well as its relative location within a record, must be the same as that used when the file was created.

If the file has more than one record description entry, data-name-1 need be described only in one of these record description entries. The identical character positions referred to by data-name-1 in any one record description entry are implicitly referred to as keys for all other record description entries of that file.

Details

If the associated file connector is an external file connector, all file description entries in the run unit that are associated with that file connector must specify the same data description entry for data-name-1, with the same relative location within the associated record.

ALTERNATE RECORD KEY Clause

This clause specifies an alternate record key for the file with which this clause is associated. The alternate record key provides an alternate access path to the records in an indexed file.

If the indexed file contains variable length records, each alternate record key must be contained within the first *x* character positions of the record, where *x* equals the minimum record size specified for the file (refer to "RECORD Clause" in Section 4).

data-name-2

This name is a user-defined word. This name must be defined as an alphanumeric or a numeric data item in a record description entry associated with the file-name to which the ALTERNATE RECORD KEY clause is subordinate.

This name can be qualified.

This name must not reference a group item that contains a variable occurrence data item.

This name must not refer to an item whose leftmost character position corresponds to the leftmost character position of the prime record key or of any other alternate record key associated with this file.

The data description of data-name-2, as well as its relative location within a record, must be the same as that used when the file was created. The number of alternate record keys for the file must also be the same as that used when the file was created.

If the file has more than one record description entry, data-name-1 need be described only in one of these record description entries. The identical character positions referred to by data-name-2 in any one record description entry are implicitly referred to in keys for all other record description entries of that file.

WITH DUPLICATES

This phrase specifies that the value of the associated alternate record key can be duplicated in any of the records in the file.

If this phrase is not specified, the value of the associated alternate record key must not be duplicated among any of the records in the file.

Details

If the associated file connector is an external file connector, every file control entry in the run unit that is associated with that file connector must specify the same data description entry for data-name-2, the same relative location within the associated record, the same number of alternate record keys, and the same DUPLICATES phrase.

FILE STATUS Clause

Refer to File Control Entry Format 1 for information on this clause. Note that information about data-name-2 in Format 1 applies to data-name-3 in this format.

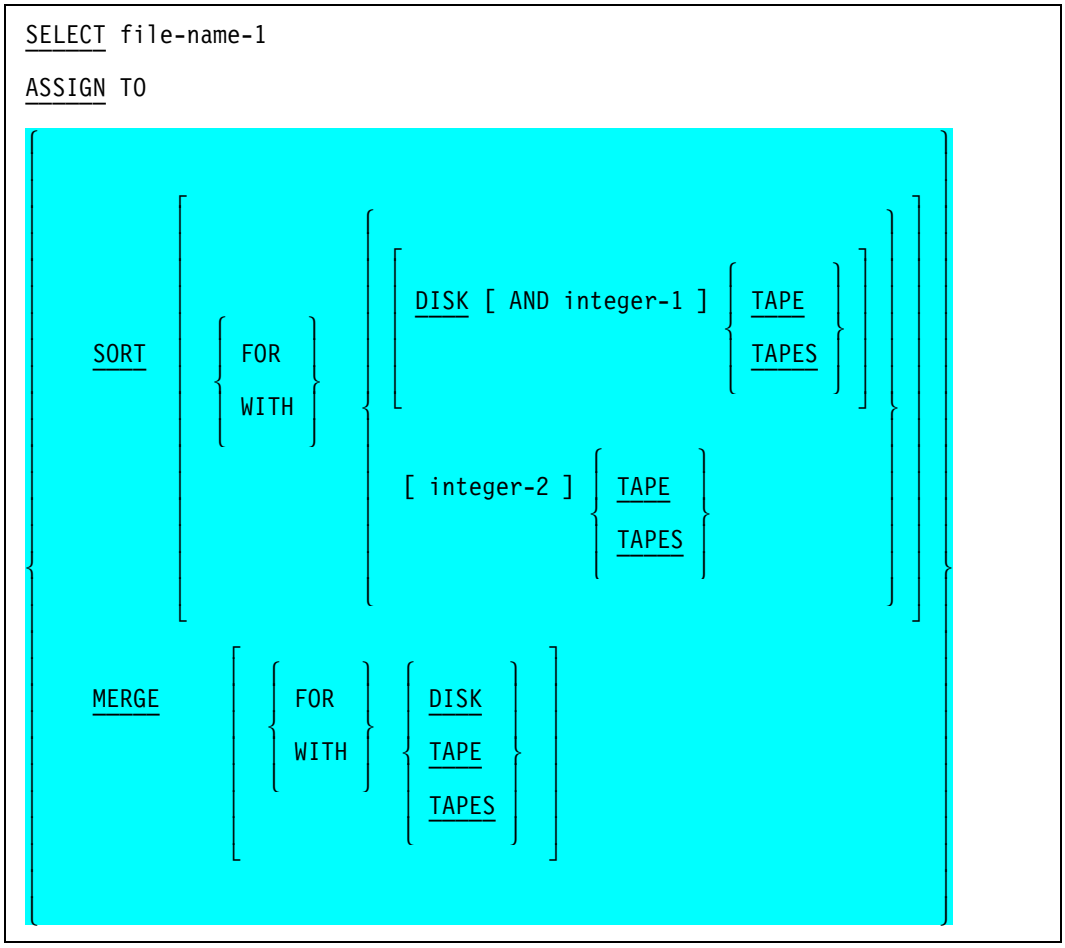
Example of File Control Entry Format 3: Indexed I/O

```
PROGRAM-ID. PAYROL.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    SW5 ON STATUS IS SW5-ON
        OFF STATUS IS SW5-OFF;
    CURRENCY SIGN IS "E";
    DECIMAL-POINT IS COMMA.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT DATA-CAPTURE-3 ASSIGN TO DISK;
    RESERVE 5 AREAS;
    ORGANIZATION IS INDEXED;
    ACCESS MODE IS RANDOM;
    RECORD KEY IS NAME;
    ALTERNATE RECORD KEY IS MODEL-NO WITH DUPLICATES.
```

In this example, the program PAYROL includes both a Configuration Section and an Input-Output Section in its Environment Division. File Control Entry Format 3 is used in the FILE-CONTROL paragraph. The input file DATA-CAPTURE-3, which is an indexed file, will be stored on disk. Five input-output areas are allocated with the RESERVE clause. The file organization is indexed. The access mode is random, so the records of DATA-CAPTURE-3 can be accessed in a sequence determined by use of the RECORD KEY clause. The desired record is accessed by placing the value of its prime record key in the RECORD KEY data item, NAME. MODEL-NO is specified as an alternate record key for the file.

File Control Entry Format 4: Sort-Merge

You can use this format to declare the physical attributes of a sort or merge file.



Refer to File Control Entry Format 1 for information on the SELECT clause.

File-name-1 represents a sort or merge file.

Details

In this format, since file-name-1 represents a sort or merge file, only the ASSIGN clause is permitted to follow file-name-1 in the FILE-CONTROL paragraph.

Each sort or merge file described in the Data Division must be specified only once as a file-name in the FILE-CONTROL paragraph.

ASSIGN Clause

This clause associates the sort or merge file referred to by file-name-1 with a storage medium.

DISK

When DISK is specified, mass storage is the primary work medium.

TAPE, TAPES

TAPE or TAPES can be specified to contain any overflow.

integer-1
integer-2

Integer-1 and integer-2 must have values within the range of 3 through 8.

If integer-1 is not specified, three tapes are assumed.

If TAPE or TAPES is specified as the primary work medium of the sort and integer-2 is not specified, the default number of tapes is three.

Example of File Control Entry Format 4: Sort-Merge

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PAYROL.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    SW5 ON STATUS IS SW5-ON
        OFF STATUS IS SW5-OFF;
    CURRENCY SIGN IS "E";
    DECIMAL-POINT IS COMMA.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT SORT-FILE ASSIGN TO SORT FOR DISK AND 3 TAPES.
```

The program PAYROL includes both a Configuration Section and an Input-Output Section. File Control Entry Format 4 is used in the FILE-CONTROL paragraph. The ASSIGN clause is the only clause that can be specified in this format. The input file SORT-FILE, a sort file, is assigned to the storage medium SORT, with mass storage (DISK) as the primary work medium of the sort and three tapes for overflow.

I-O-CONTROL Paragraph

The I-O-CONTROL paragraph specifies:

- The memory area that is to be shared by different files
- The location of files on a multiple file reel

This paragraph is optional. Clauses can appear in any order in this paragraph.

```
I-O-CONTROL. [ input-output control entry . ]
```

I-O-CONTROL

This keyword begins in area A and must be followed by a period.

input-output control entry

The input-output control entry should have a period only at the end. The input-output control entry has three formats:

- Format 1 is used for sequential I/O.
- Format 2 is used for relative and indexed I/O.
- Format 3 is used for sort-merge.

Input-Output Control Entry Format 1: Sequential I/O

```
I-O-CONTROL.
[ [ SAME  [ RECORD ] AREA FOR file-name-3 { file-name-4 } ... ] ...
[MULTIPLE FILE TAPE CONTAINS { file-name-5 [POSITION integer-3] } ... ] ...   .   ]
```

I-O-CONTROL

This keyword begins in area A and must be followed by a period.

SAME Clause

This clause specifies the memory area that is to be shared by different files.

There are two forms of the SAME clause: the SAME AREA clause and the SAME RECORD AREA clause.

SAME AREA

The SAME AREA clause specifies that two or more files that do not represent sort or merge files are to use the same memory area during processing. The area being shared includes all storage areas assigned to the files referred to by file-name-3 and file-name-4; thus, only one file can be open at a time.

SAME RECORD AREA

The SAME RECORD AREA clause specifies that two or more files are to use the same memory area for processing of the current logical record. Only the record work area is shared. All of the files can be open at the same time. A logical record in the SAME RECORD AREA is considered a logical record of each opened output file whose file-name appears in the SAME RECORD AREA clause and of the most recently read input file whose file-name appears in the SAME RECORD AREA clause. Like an implicit redefinition of the area, records are aligned at the leftmost character position.

file-name-3
file-name-4

These names must be specified in the FILE-CONTROL paragraph of the same program.

These names must not reference an external file connector.

These names must not reference a file that uses the IS EXTERNAL-FORMAT FOR NATIONAL clause.

Rules

More than one SAME clause can be included in a program, subject to the following restrictions:

- A file-name must not appear in more than one SAME AREA clause.
- A file-name must not appear in more than one SAME RECORD AREA clause.
- If one or more file-names of a SAME AREA clause appear in a SAME RECORD AREA clause, all of the file-names in the SAME AREA clause must appear in the SAME RECORD AREA clause. However, additional file-names not appearing in that SAME AREA clause can also appear in that SAME RECORD AREA clause. The rule that only one of the files mentioned in a SAME AREA clause can be open at any given time takes precedence over the rule that all files mentioned in a SAME RECORD AREA clause can be open at any given time.

MULTIPLE FILE TAPE Clause

This clause specifies the location of files on a multiple-file reel.

This clause is required when more than one file shares the same physical reel of tape. Regardless of the number of files on a single reel, only those files that are used in the object program need to be specified.

POSITION

If all file-names have been listed in consecutive order, the POSITION phrase need not be given. If any file in the sequence is not listed, the position relative to the beginning of the tape must be given. No more than one file on the same tape reel can be open at one time.

Note: *This clause is an obsolete element in COBOL ANSI-85 and will be deleted from the next revision of the COBOL standard.*

Example of Input-Output Control Entry Format 1: Sequential I/O

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PAYROL.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    SW5 ON STATUS IS SW5-ON
        OFF STATUS IS SW5-OFF;
    CURRENCY SIGN IS "E";
    DECIMAL-POINT IS COMMA.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INFIL ASSIGN TO TAPE.
    SELECT OUTFIL ASSIGN TO TAPE.
I-O-CONTROL.
    SAME AREA FOR INFIL OUTFIL.
```

In this example, the program PAYROL includes both a Configuration Section and an Input-Output Section in its Environment Division. The SELECT clause in the FILE-CONTROL paragraph assigns two sequential files, INFIL and OUTFIL, to the storage medium TAPE. The SAME clause in the I-O-CONTROL paragraph, which uses Input-Output Control Entry Format 1, specifies that INFIL and OUTFIL will share the same memory area during processing (but only one of these files can be open at a time).

Input-Output Control Entry Format 2: Relative and Indexed Organization

```
I-O-CONTROL.
[ [ SAME [ RECORD ] AREA FOR file-name-3 { file-name-4 } ... ]  ...  .  ]
```

Refer to Input-Output Control Entry Format 1 for information on the SAME clause.

I-O-CONTROL

This keyword begins in area A and must be followed by a period.

Examples of Input-Output Control Entry Format 2: Relative and Indexed Organization

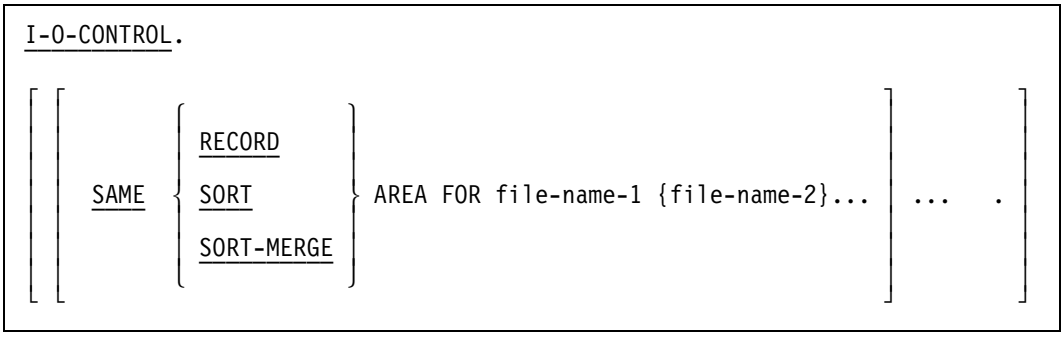
```
IDENTIFICATION DIVISION.
PROGRAM-ID. PAYROL.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    SW5 ON STATUS IS SW5-ON
        OFF STATUS IS SW5-OFF;
    CURRENCY SIGN IS "E";
    DECIMAL-POINT IS COMMA.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INFIL ASSIGN TO DISK;
        ORGANIZATION IS RELATIVE;
        ACCESS MODE IS SEQUENTIAL;        RELATIVE KEY IS ACCT-NO.
    SELECT OUTFIL ASSIGN TO DISK;
        ORGANIZATION IS RELATIVE;
        ACCESS MODE IS SEQUENTIAL;        RELATIVE KEY IS MODEL-NO.
I-O-CONTROL.
    SAME AREA FOR INFIL OUTFIL.
```

The program PAYROL includes a Configuration Section and an Input-Output Section. The SELECT clause assigns two relative files, INFIL and OUTFIL, to DISK. The SAME clause causes the INFIL and OUTFIL files to use the same memory area for file information. These two files must not be open at the same time.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PAYROL.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    SW5 ON STATUS IS SW5-ON
        OFF STATUS IS SW5-OFF;
    CURRENCY SIGN IS "E";
    DECIMAL-POINT IS COMMA.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INFIL ASSIGN TO DISK;
        ORGANIZATION IS INDEXED;
        ACCESS MODE IS SEQUENTIAL;
        RECORD KEY IS NAME;
        ALTERNATE RECORD KEY IS MODEL-NO WITH DUPLICATES.
    SELECT OUTFIL ASSIGN TO DISK;
        ORGANIZATION IS INDEXED;
        ACCESS MODE IS SEQUENTIAL;
        RECORD KEY IS ACCT-NO;
        ALTERNATE RECORD KEY IS DEPT-NO.
I-O-CONTROL.
    SAME RECORD AREA FOR INFIL OUTFIL.
```

In this example, the program PAYROL includes both a Configuration Section and an Input-Output Section in its Environment Division. The SELECT clause in the FILE-CONTROL paragraph assigns two indexed files, INFIL and OUTFIL, to the storage medium DISK. The SAME RECORD clause in the I-O-CONTROL paragraph, which uses Input-Output Control Entry Format 2, specifies that INFIL and OUTFIL will share the same record area for processing of the current logical record. Both of these files can be open at the same time.

Input-Output Control Entry Format 3: Sort-Merge



I-O-CONTROL

This keyword begins in area A and must be followed by a period.

SAME Clause

This clause specifies the memory area that is to be shared by different files. At least one of these files must be a sort or merge file.

There are two forms of the SAME clause in this format: the SAME RECORD AREA clause and the SAME SORT AREA or SAME SORT-MERGE AREA clause.

SAME RECORD AREA

This clause specifies that two or more files referred to by file-name-1 and file-name-2 are to use the same memory area for processing of the current logical record. All of these files can be in the open mode at the same time. A logical record in the SAME RECORD AREA is considered to be a logical record of each file that is open in the output mode and whose file-name appears in the SAME RECORD AREA clause, and of the most recently read file that is open in the input mode and whose file-name appears in the SAME RECORD AREA clause. This is equivalent to an implicit redefinition of the area; that is, records are aligned on the leftmost character position.

SAME SORT AREA
SAME SORT-MERGE AREA

These keywords are equivalent. If this clause is used, at least one of the file-names must represent a sort or merge file.

file-name-1
file-name-2

Each file-name specified in the SAME clause must be specified in the FILE-CONTROL paragraph of the same program. File-name-1 and file-name-2 must not reference an external file connector.

A file-name that represents a sort or merge file must not appear in the SAME clause unless the SORT, SORT-MERGE, or RECORD phrase is used.

The files referred to in the SAME clause need not all have the same organization or access.

Details

The SAME clause specifies that storage is shared as follows:

- The SAME SORT AREA or SAME SORT-MERGE AREA clause specifies a memory area that will be made available for use in sorting or merging each sort or merge file named. Thus, any memory area allocated for the sorting or merging of a sort or merge file is available for reuse in sorting or merging any of the other sort or merge files.
- In addition, storage areas assigned to files that do not represent sort or merge files can be allocated as needed for sorting or merging the sort or merge files named in the SAME SORT AREA or SAME SORT-MERGE AREA clause.
- Files other than sort or merge files do not share the same storage area with each other. For these files to share the same storage area with each other, the program must contain a SAME AREA or SAME RECORD AREA clause specifying file-names associated with these files.
- During the execution of a SORT or MERGE statement that refers to a sort or merge file named in this clause, any non-sort or non-merge files associated with file-names named in this clause must not be in the open mode.

Rules

More than one SAME clause can be included in a program. If more than one SAME clause is included in a program, the following restrictions apply:

- A file-name must not appear in more than one SAME RECORD AREA clause.
- A file-name that represents a sort or merge file must not appear in more than one SAME SORT AREA or SAME SORT-MERGE AREA clause.
- If a file-name that does not represent a sort or merge file appears in a SAME clause and one or more SAME SORT AREA or SAME SORT-MERGE AREA clauses, all of the files named in that SAME clause must be named in that SAME SORT AREA or SAME SORT-MERGE AREA clause(s).

Example of Input-Output Control Entry Format 3: Sort-Merge

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PAYROL.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    SW5 ON STATUS IS SW5-ON
        OFF STATUS IS SW5-OFF;
    CURRENCY SIGN IS "E";
    DECIMAL-POINT IS COMMA.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT DATA-CAPTURE-1 ASSIGN TO TAPE.
    SELECT DATA-CAPTURE-2 ASSIGN TO TAPE.
    SELECT SORT-FILE ASSIGN TO SORT.
    SELECT MERGE-FILE ASSIGN TO MERGE.
I-O-CONTROL.
    SAME SORT-MERGE AREA FOR SORT-FILE MERGE-FILE.
```

In this example, the program PAYROL includes both a Configuration Section and an Input-Output Section in its Environment Division. In the FILE-CONTROL paragraph, the input file SORT-FILE, which is a sort file, is assigned to the storage medium SORT, and the input file MERGE-FILE, which is a merge file, is assigned to the storage medium MERGE. The SAME clause in the I-O-CONTROL paragraph, which uses Input-Output Control Entry Format 3, specifies that SORT-FILE and MERGE-FILE will share the same memory area for sorting or merging.

I-O Status Codes

The I-O status is a two-character conceptual entity whose value is set to indicate the status of an input-output operation. Some status values indicate successful execution, while other values indicate unsuccessful execution. The value of the I-O status is made available to the COBOL program through the data item named in the FILE STATUS clause of the file control entry for the file.

The I-O status value is placed into the FILE STATUS data item in the following situations:

- During the execution of a CLOSE, DELETE, OPEN, READ, REWRITE, START or WRITE statement before the execution of any imperative statement associated with the statement
- Before the execution of any applicable USE AFTER STANDARD EXCEPTION procedure

Note: *To receive the standard COBOL ANSI-85 status codes, you must set the $ANSICLASS compiler option. For COBOL74 compatibility, this option must be reset (FALSE). The default value is FALSE. For details about this option, refer to Section 15.*

Tables 3–1 through 3–6 describe each of the possible I-O status codes. The symbols used in the File Organization column of these tables are as follows:

Symbol	Meaning
S	Organization is sequential.
R	Organization is relative.
I	Organization is indexed.

The I-O status values in Table 3–1 indicate that the input-output statement was successfully executed.

Table 3–1. I-O Status Codes: Successful Execution

File Organization	I-O Status	Condition Description
SRI	00	The input-output statement was successfully executed and no further information is available concerning the input-output operation.
SRI	04	A READ statement was successfully executed, but the length of the record that was processed did not conform to the fixed file attributes for that file.
SRI	05	An OPEN statement was successfully executed, but the referenced optional file was not present when execution of the OPEN statement began. If the open mode was I-O or EXTEND, the file was created.

Table 3–1. I-O Status Codes: Successful Execution

File Organization	I-O Status	Condition Description
S	07	The input-output statement was successfully executed. However, for a CLOSE statement with the NO REWIND, REEL/UNIT, or FOR REMOVAL phrase or for an OPEN statement with the NO REWIND phrase, the referred to file was on a non-reel/unit medium.

The I-O status values in Table 3–2 indicate that a sequential READ statement was unsuccessfully executed as a result of an end-of-file condition.

Table 3–2. I-O Status Codes: Unsuccessful READ—End-of-File Condition

File Organization	I-O Status	Condition Description
SRI	10	The execution of a sequential READ statement was unsuccessful because: • The end of the file has been reached (no next logical record was present in the file). • A sequential READ statement was attempted for the first time on an optional input file that was not present when the associated OPEN statement was executed. • An attempt was made to sequentially read a port file when no next logical record existed and the communication path with the connected process was no longer established.
SR	14	A sequential READ statement was attempted for a relative file, but the number of significant digits in the relative record number is larger than the size of the actual or relative key data item described for the file.

The I-O status values in Table 3–3 indicate that the input-output statement was unsuccessfully executed as a result of an invalid key condition.

Table 3–3. I-O Status Codes: Unsuccessful I/O—Invalid Key Condition

File Organization	I-O Status	Condition Description
I	21	A sequence error exists for a sequentially accessed indexed file. Either the prime record key value was changed by the program between the successful execution of a READ statement and the execution of the next REWRITE statement for that file, or the ascending sequence requirements for successive record key values were violated.
R	22	An attempt was made to write a record that would create a duplicate key in a relative file.
I	22	An attempt was made to write or rewrite a record that would create a duplicate prime record key or a duplicate alternate record key without the DUPLICATES phrase in an indexed file.
SRI	23	This condition exists because: • An attempt was made to randomly access a record that does not exist in the file. • A START or random READ statement was attempted on an optional input file that is not present.
SR	24	A sequential WRITE statement was attempted, but the relative record number is larger than the size of the relative key data item described for the file.
SRI	25	No space is available on the disk, and the NORESOURCEWAIT file attribute is TRUE.

The I-O status values in Table 3–4 indicate that the input-output statement was unsuccessfully executed as a result of an error that precluded further processing of the file. Any specified exception procedures are executed. The permanent error condition remains in effect for all subsequent input-output operations on the file, until you take action to correct the error or your program performs error recovery.

Table 3–4. I-O Status Codes: Unsuccessful I/O—Permanent Error Condition

File Organization	I-O Status	Condition Description
SRI	30	A permanent error exists and no further information is available concerning the input-output operation.
S	34	A permanent error exists because of a boundary violation; an attempt was made to write beyond the externally defined boundaries of a sequential file.
SRI	35	A permanent error exists because an OPEN statement with the INPUT, I-O, or EXTEND phrase was attempted on a non-optional file that is not present. This error occurs only after you enter the command ?NF RSVP.
SRI	37	A permanent error exists because an OPEN statement is attempted on a file and that file will not support the open mode specified in the OPEN statement. The possible violations are: • The EXTEND or OUTPUT phrase was specified, but the file will not support write operations. • The I-O phrase was specified, but the file will not support the input and output operations that are permitted for a sequential, relative, or indexed file when opened in the I-O mode. • The INPUT phrase was specified, but the file will not support read operations.
SRI	38	A permanent error exists because an OPEN statement was attempted on a file previously closed with lock.
SRI	39	The OPEN statement was unsuccessful because a conflict has been detected between the fixed file attributes and the attributes for that file in the program.

The I-O status values in Table 3–5 indicate that the statement was unsuccessfully executed either as a result of an improper sequence of operations that were performed on the file, or as a result of violating a limit defined by the user.

Table 3–5. I-O Status Codes: Unsuccessful I/O—Invalid Operations

File Organization	I-O Status	Condition Description
SRI	41	An OPEN statement was attempted for a file in the open mode.
SRI	42	A CLOSE statement was attempted for a file not in the open mode.
SRI	43	For a mass-storage file in the sequential access mode the last input-output statement executed for the associated file prior to the execution of a DELETE or REWRITE statement was not a successfully executed READ statement.
SRI	44	A boundary violation exists because: • An attempt was made to write or rewrite a record that is larger than the largest or smaller than the smallest record allowed by the RECORD IS VARYING or RECORD CONTAINS clause of the associated file-name. • An attempt was made to rewrite a record to a sequential file and the record is not the same size as the record being replaced. (Sequential)
SRI	46	A sequential READ statement was attempted on a file open in the input or I-O mode and no valid next record has been established because: • The preceding READ statement was unsuccessful but did not cause an at end condition. • The preceding READ statement caused an at end condition. • The preceding START statement was unsuccessful. (Relative/Indexed). This does not apply to port files or remote files for which a non-zero TIMELIMIT is specified.
SRI	47	The execution of a READ or START statement was attempted on a file opened in a mode other than input or I-O.
S	48	The execution of a WRITE statement was attempted on a file opened in a mode other than output.
SRI	48	The execution of a WRITE statement was attempted on a file opened in a mode other than I-O, output, or extend.

Table 3–5. I-O Status Codes: Unsuccessful I/O—Invalid Operations

File Organization	I-O Status	Condition Description
S	49	The execution of a REWRITE statement was attempted on a file opened in a mode other than output.
RI	49	The execution of a DELETE or REWRITE statement was attempted on a file opened in a mode other than I-O.

The I-O status values in Table 3–6 indicate Unisys defined conditions.

Table 3–6. I-O Status Codes: Unisys Defined Conditions

File Organization	I-O Status	Condition Description
SRI	91	Short Block. A physical block shorter than the physical blocksize declared for the file was received from the hardware device. The operation completed successfully because an I/O status value of 91 is to be considered a warning, not an error.
SRI	92	Data Error. When logical records are declared variable in length and the logical record length is supplied by the programmer (by the RECORD CONTAINS clause), a data error occurs on a READ, WRITE, or REWRITE statement if the logical record length supplied is less than the minimum record size or greater than the maximum record size declared for the file. This condition initiates no I/O operation and does not cause data to be transferred to or from the record area.
S	93	Port File. A broadcast write operation failed on one or more subports.
S	94	No Data. The WITH NO WAIT clause was used with the READ statement and no data was available.
S	95	No Buffer. The WITH NO WAIT clause was used with the WRITE statement and no buffer was available.
SRI	96	Timeout. A time limit elapsed before the transfer of data to or from the hardware device.

Table 3–6. I-O Status Codes: Unisys Defined Conditions

File Organization	I-O Status	Condition Description
I	82	Form Not Found. A READ FORM or WRITE FORM statement to a file that is designated ASSIGN TO REMOTE requested a form that does exist in the form library or requested a form for which the compile-time version does not equal the run-time version.
I	96	Timeout. A deadly embrace (or "deadlock") occurred because multiple programs tried to lock the same records in a different order.
SRI	97	Break on Output. For an output or I-O file, this condition occurs if the hardware device is equipped with a break so that the transfer of data in process can be halted.
	98	Deadlock.
SRI	99	Unexpected I/O Error. An error might have occurred in the I/O operation, but its nature cannot be determined.
S	9A	You specified a file with the LOCK statement that does not support locking.
S	9B	An existing locked region of the file is blocking the LOCK request, and the resulting wait timed out.
S	9C	The LOCK statement failed, because the number of locked regions met the system-imposed maximum.
S	9D	An UNLOCK statement was issued but failed, because no locked records matched its record specification.
S	9E	The record being written is unlocked by the current user.

Recovering from I-O Errors

You can enable a COBOL85 program to recover from an I-O error by specifying a particular action for the program to take if an error occurs during the execution of an I-O statement. To specify the alternate action, use one of the following syntaxes:

- FILE STATUS clause in the Environment Division

  The FILE STATUS clause associates a file status variable with the file. You can use this method to detect and recover from any I-O error for the file.

- USE AFTER STANDARD EXCEPTION statement in the Procedure Division

  You can associate a USE procedure with the file to detect and recover from an I-O error in a file

  – That is named in the USE statement

  – That has the same open mode as that of the file named in the USE statement (for files not explicitly named in any USE AFTER STANDARD EXCEPTION statement)

- AT END or INVALID KEY phrase with the READ and WRITE statements as allowed by the specific formats

  You can use this method to detect and recover from either an AT END or an INVALID KEY error condition. For details on these error conditions, see the READ statement in Section 7 and the WRITE statement in Section 8.

How the Recovery Process Occurs

Recovery from an I-O error occurs in the following way:

1. If the FILE STATUS clause is present in the program, the specified data item is updated to reflect the error condition.
2. The next action depends upon the following conditions:

If an AT END or INVALID KEY phrase is . . .	And . . .	Then . . .
Present	The error code in the FILE STATUS data item indicates AT END or an INVALID key,	The specified imperative statement is executed.
Present	The error code in the FILE STATUS data item does not indicate AT END or an INVALID key,	The USE AFTER STANDARD EXCEPTION procedure is executed, if present.
Not Present		The USE AFTER STANDARD EXCEPTION procedure is executed, if present.

3. Finally, the next statement in the program is executed.

Modifying the Recovery Process for COBOL74 Compatibility

The $FS4XCONTINUE compiler option is available to provide error semantics similar to COBOL74 if you are migrating COBOL74 code to COBOL85. When reset (FALSE), this compiler option causes a program to be terminated if it issues

- An OPEN statement for an open file
- A CLOSE statement for a file that is already closed
- A READ, SEEK, or START statement for a file opened in a mode other than INPUT or I-O
- A WRITE statement for a file opened in a mode other than EXTEND, I-O, or OUTPUT
- A DELETE or REWRITE statement for a file opened in a mode other than I-O

Before the program terminates, it executes either the imperative statement named in the AT END or INVALID KEY phrase, if specified, or the USE AFTER STANDARD EXCEPTION statement, if specified.

Because the semantics provided by the $FS4XCONTINUE option often conflict with COBOL85, you can modify the settings of the option (and thus, the semantics of the compiled code) at any point in the source program listing.

When the $FS4XCONTINUE option is TRUE, error recovery occurs as described in this section under "How the Recovery Process Occurs."

Note: *For COBOL74 compatibility, the $ANSICLASS option must be reset (FALSE), which is the default value. For details about this option, refer to Section 15.*

Section 4
Data Division

This section illustrates and explains the concepts and syntax of the Data Division, the third division of a COBOL program. The Data Division describes the data that the object program is to accept as input, to manipulate, to create, or to produce as output.

Use of the Data Division is optional in a COBOL source program.

Structure of the Data Division

The Data Division consists of the header DATA DIVISION, followed by eight optional sections:

Section	Function
File Section	This section describes the physical structure of data files used by the program.
Data-Base Section	Refer to Section 3 of *MCP/AS COBOL ANSI-85 Programming Reference Manual, Volume 2: Product Interfaces*.
Working-Storage Section	This section describes the records and the subordinate data items that are developed and processed internally in the program and that have values assigned in the source program that do not change during execution of the object program.
Linkage Section	Located in a called program, this section describes the data in the calling program that is to be referenced by both programs. The Linkage Section is meaningful only if the object program functions under the control of a calling program that contains a CALL statement with a USING phrase.
Communication Section	Refer to Section 1 of *MCP/AS COBOL ANSI-85 Programming Reference Manual, Volume 2: Product Interfaces*.
Local-Storage Section	Located in a calling program, this section describes parameters in the calling program that are to be referenced by both the calling and the called programs.

Section	Function
Report Section	Refer to Section 14.
Program-Library Section	This section defines the interface between a user program and a library program.

Data entries in a section can take the following forms:

- File description entries

  File description entries represent the highest level of organization in the File Section. File description entries follow the File Section header. Each entry begins in area A with a level indicator, followed by a space, followed by a file-name, followed by a set of file clauses, as required.

- Record description entries

  A record description entry consists of one or more data description entries that describe one record in the file.

- Data description entries

  Data description entries begin with a level-number, followed by a space, followed by a data-name (if required), followed by a set of data clauses, as required. The total set of data description entries associated with a particular record is a record description entry. The syntax for various data description entries is provided in this section.

Record Concepts

To separate the logical characteristics of data from the physical characteristics of the data storage media, separate clauses or phrases are used. In a COBOL program, the input or output statements refer to one logical record of a file, as opposed to a physical record. A physical record is a physical unit of information whose size and recording mode is convenient to a particular computer for the storage of data on an input or output device. The size of a physical record is hardware dependent and does not bear a direct relationship to the size of the file contained on a device.

A COBOL logical record is a group of related information that is uniquely identifiable and treated as a unit. A logical record can be contained in a single physical unit; several logical records can be contained in a single physical unit; or a logical record can be contained in more than one physical unit.

The concept of a logical record is not restricted to file data but is carried over into the definition of working-storage. Thus, working-storage is grouped into logical records and defined by a series of record description entries.

Level Concepts

The concept of levels is inherent in the structure of a logical record. Levels indicate the subdivision of a record for the purpose of data reference. Once a subdivision has been specified, it can be further divided for more detailed data referral.

The most basic subdivisions of a record (that is, those that cannot be further subdivided) are called elementary items. A record can consist of a sequence of elementary items or can itself be an elementary item. For reference purposes, the elementary items are combined into groups. Each group consists of a named sequence of one or more elementary items. Groups, in turn, can be combined into groups of two or more groups, and so forth. Thus, an elementary item can belong to more than one group.

A group includes all group and elementary items following it until a level-number less than or equal to the level-number of that group is encountered. All items that are immediately subordinate to a given group item must be described by using identical level-numbers greater than the level-number used to describe that group item.

A true concept of levels does not exist for the following types of entries:

- Entries that specify elementary items or groups introduced by a RENAMES clause
- Entries that specify noncontiguous working-storage and linkage data items
- Entries that specify condition-names

Example

```
01  PRIMARY.
    03  ACCT-NO          PIC 9(8).
    03  NAME              PIC X(20).
    03  OTHER-NAMES.
        05  NME     PIC X(20).
        05  FLAG    PIC 9.
```

This data description entry defines the group item PRIMARY. PRIMARY consists of the elementary items ACCT-NO and NAME, and the group item OTHER-NAMES. OTHER-NAMES consists of the elementary items NME and FLAG.

Level-Numbers

A system of level-numbers shows the organization of elementary items and group items (that is, the hierarchy of data in a logical record). Level-numbers identify entries for working-storage items, linkage items, condition-names, and the RENAMES clause.

A one- or two-digit level-number is required as the first element in each data description entry. At least one space must follow a level-number. Because records are the most inclusive data items, level-numbers for records start at 01. Multiple level-01 entries that are subordinate to any given level indicator represent implicit redefinitions of the same area. Less inclusive data items are assigned higher (not necessarily successive) level-numbers not greater in value than 49. Separate entries are written in the source program for each level-number used. Level-numbers can identify special properties of a data description entry, as shown next.

Number . . .	Identifies entries that . . .
66	Describe items through RENAMES clauses for the purpose of regrouping data items
77	Specify noncontiguous data items that are not subdivisions of other items and are not themselves subdivided
88	Specify condition-names to be associated with particular values of a conditional variable

The syntax for these types of entries is provided later in this section.

For data description entries that begin with a level-number 01 or 77, the level-number must begin in area A. Data description entries that begin with other level-numbers can begin any number of positions to the right of margin A.

Note that the extent of indentation is determined only by the width of the physical medium. The entries on the output listing need to be indented only if the input is indented. Indentation does not affect the magnitude of a level-number.

Level Indicators (FD, SD)

A level indicator consists of two alphabetic characters that identify a specific type of file. The level indicators FD and SD are used in file description entries in the Data Division. FD identifies the beginning of a file description entry, and SD identifies the beginning of a sort-merge file description entry. The level indicator must precede the file-name.

Classes and Categories of Data Items

Every data item is considered to belong to one of five classes: alphabetic, numeric, alphanumeric, national, or Boolean

Each class is further subdivided into categories:

- Alphabetic
- Numeric
- Alphanumeric
- National
- Boolean
- Alphanumeric-edited
- National-edited
- Numeric-edited

The relationship between the class and category of data items is shown in Table 4–1. Information on how to define the different categories of items is presented under "PICTURE Clause" in this section.

Table 4–1. Relationship between Class and Category of Data Items

Level of Item	Class	Category
Elementary	Alphabetic	Alphabetic
	Numeric Alphanumeric	Numeric Numeric-edited Alphanumeric-edited Alphanumeric
	National	National National-edited
	Boolean	Boolean
Nonelementary (Group)	Alphanumeric	Alphabetic Numeric Numeric-edited Alphanumeric-edited Alphanumeric National National-edited

Note that the class of a group item is treated as alphanumeric at object time regardless of the class of elementary items subordinate to that group item.

Class and Category of Figurative Constants and Intrinsic Functions

Following are the class and category for figurative constants and intrinsic functions:

- When moved to a national or national-edited field, all figurative constants belong to the national class and category.
- The figurative constant space, except when moved to a national or national-edited field, belongs to the alphabetic class and category.
- The figurative constant ZERO (ZEROS, ZEROES), except when moved to a Boolean, national, or national-edited category, belongs to the numeric class and category when moved to a numeric field and the numeric-edited class and the alphanumeric category when moved to a nonnumeric field.
- In all other cases, figurative constants belong to the alphanumeric class and category.
- Intrinsic functions belong to either the numeric class and category or the alphanumeric class and category. For details, refer to Section 9.

  The PICTURE clause describes the general characteristics and editing requirements of an elementary data item. When the PICTURE clause of the item contains a picture character N, the usage is implicitly NATIONAL.

  The USAGE clause specifies the manner in which a data item is represented in the storage of a computer, and can affect the type of character representation of the item. If the USAGE clause is not specified for an elementary item, or for any group to which the item belongs, the usage is implicitly DISPLAY. The syntax for these clauses is provided in this section.

When a computer provides more than one means of representing data, the standard data format or national standard data format must be used for data items other than integer and numeric functions, if not otherwise specified by the data description.

Note that an alphanumeric function is always represented in the standard data format.

The size of an elementary data item or a group item is the number of characters in the standard data format of the item. Synchronization and usage can cause a difference between this size and that required for internal representation.

The size of a national data item is the number of national characters in the national standard data format of the item.

Long Numeric Data Items

Standard numeric data items are limited to 23 digits. To ease the moving of programs from V Series systems, a longer data item is supported. The "long numeric data item" consists of 24 to 99,999 digits. It is intended to be used primarily to initialize structures and arrays that overlay the long numeric data item itself.

The general rules for forming and using long numeric data items are as follows:

- You can declare a long numeric data item with a usage of DISPLAY or COMPUTATIONAL. If the usage is COMPUTATIONAL, then the long numeric data item must contain an even number of digits.
- A long numeric data item cannot have a usage of BINARY, DOUBLE, or REAL.
- You must define a long numeric data item as an unsigned integer without editing or decimal point characters.
- A long numeric data item cannot appear in a database record.
- A long numeric data item cannot appear as a file key, sort key, or search key.
- You cannot explicitly reference long numeric data items in COBOL85 TADS statements.

For specific details about declaring long numeric data items, refer to "PICTURE Clause" and "VALUE Clause" later in this section.

You can reference long numeric data items in certain comparison operations. Refer to "Comparison of Numeric Operands" under "Relation Conditions" in Section 5 for more information.

You can reference long numeric data items in the following Procedure Division statements:

CALL	MOVE
IF	READ
INITIALIZE	SORT
INSPECT	WRITE
MERGE	

For more specific information, refer to the discussion of each statement in Sections 6 through 8.

Algebraic Signs

There are two categories of algebraic signs: operational signs and editing signs. Operational signs are associated with signed numeric data items and signed numeric literals to indicate their algebraic properties. Editing signs appear on edited reports, for example, to identify the sign of the item.

The SIGN clause, discussed in this section, allows the programmer to state explicitly the location of the operational sign.

Editing signs are inserted into a data item through the sign control symbols of the PICTURE clause, which is also defined under the data description entry formats in this section.

Standard Alignment Rules

The standard rules for positioning data in an elementary item depend on the category of the receiving item.

Category of Receiving Item	Positioning of Data
Numeric	The data is aligned by decimal point and is moved to the receiving digit positions with zero fill or truncation on either end as required. If an assumed decimal point is not explicitly specified, the data item is treated as if it has an assumed decimal point immediately following its rightmost digit and is aligned in the same way.
Numeric-edited	The data moved to the edited data item is aligned by decimal point with zero fill or truncation at either end as required in the receiving character positions of the data item, except where editing requirements cause replacement of the leading zeros.
Alphanumeric (other than a numeric-edited data item), alphanumeric-edited, or alphabetic	The sending data is moved to the receiving character positions and aligned at the leftmost character position in the data item with space fill or truncation to the right, as required.
National or national-edited	The sending data is moved to the receiving character positions and is aligned at the leftmost character position in the data item with national space fill or truncation to the right, as required.

Note that if the JUSTIFIED clause is specified for the receiving item, these standard rules are modified (refer to "JUSTIFIED Clause" in this section).

Increasing Object-Code Efficiency

Certain uses of data (for example, in arithmetic operations or in subscripting) can be made easier if the data is stored so that it is aligned on the natural addressing boundaries in the computer memory (for example, word boundaries and byte boundaries). Specifically, additional machine operations in the object program can be required for the accessing and storage of data if portions of two or more data items appear between adjacent natural boundaries, or if certain natural boundaries divide a single data item. Data items that are aligned on these natural boundaries in such a way as to avoid such additional machine operations are said to be synchronized. With increases in machine speeds, the measurable effect of SYNCHRONIZE might be recognized only at the level of millions of calculations. Synchronization can be accomplished in two ways:

- By using the SYNCHRONIZED clause
- By recognizing the appropriate natural boundaries and organizing the data suitably without using the SYNCHRONIZED clause

Uniqueness of Reference

Every user-defined name in a COBOL program is assigned by the programmer to name a resource that will be used in solving a data processing problem. To use a resource, a statement in a COBOL program must contain a reference that uniquely identifies that resource. To ensure that a user-defined name is unique, you can add a subscript, a qualifier, or a reference modifier. Qualifiers and reference modifiers are discussed in this section. Subscripts are discussed in Section 5.

When the same name has been assigned in separate programs to two or more occurrences of a resource of a given type, and when qualification by itself does not allow the reference in one of those programs to differentiate between the two identically named resources, then certain conventions that limit the scope of names apply. These conventions ensure that the resource identified is the one described in the program that contains the reference (refer to "Scope of Names" in Section 10). When the resource is an ANSI intrinsic function, the values assigned to the arguments of each function help differentiate between the two functions.

Unless otherwise specified, subscripts and reference modifiers are evaluated only when a statement is executed.

Qualification

Every user-defined name explicitly referred to in a COBOL source program must be unique in one of the following ways:

- No other name has the identical spelling and hyphenation.
- The name is unique within the context of a REDEFINES clause (refer to "REDEFINES Clause" in this section).
- The name exists in a hierarchy of names so that reference to the name can be made unique by mentioning one or more of the higher-level names in the hierarchy.
- The name is contained in a program that is contained in another program or contains another program (refer to "Scope of Names" in Section 10).

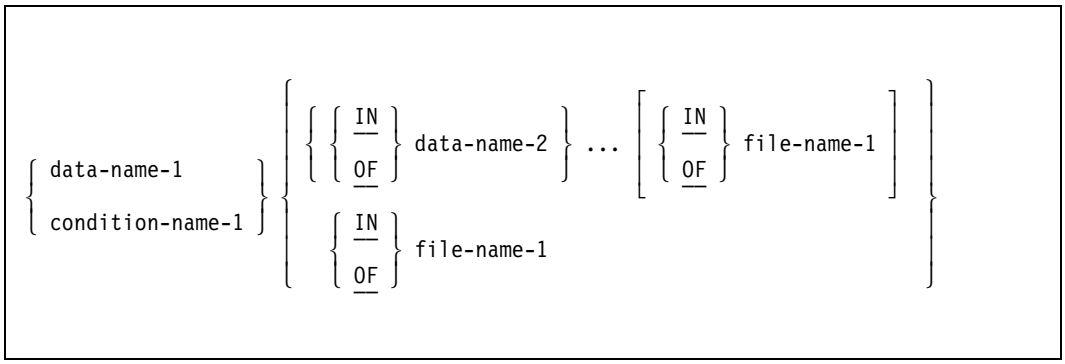
Higher-level names in a hierarchy of names are called qualifiers, and the process that specifies uniqueness is called qualification. The formats on the following pages can be used for qualification.

Identical user-defined names can appear in a source program. However, uniqueness must then be established through qualification for each user-defined name that is explicitly referred to (except in the case of redefinition). As long as uniqueness is established, all available qualifiers do not need to be specified.

Reserved words that name the special registers require qualification to provide uniqueness of reference whenever a source program would result in more than one occurrence of any of these special registers. A paragraph-name or section-name in one program cannot be referred to from any other program.

The same data-name must not be used as the name of an external record and as the name of any other external data item described in any program that is contained in or contains the program that describes that external data record. Also, the same data-name must not be used as the name of an item that possesses the global attribute and as the name of any other data item described in the program that describes that global data item.

Qualification Format 1



In this format, each qualifier must be the name associated with a level indicator, the name of a group item to which the item being qualified is subordinate, or the name of the conditional variable with which the condition-name being qualified is associated. Qualifiers are specified in the order of successively more inclusive levels in the hierarchy.

data-name-1
data-name-2

A data-name is a user-defined word that names a data item described in a data description entry. When used in the general formats, *data-name* represents a word that must not be reference-modified, subscripted, or qualified unless specifically permitted by the rules of the format.

In this format, either of these data-names can be a record-name.

condition-name-1

A condition-name is a user-defined word that assigns a name to a subset of values that a conditional variable can assume; or a user-defined word assigned to a status of a switch or device. When *condition-name* is used in the general formats, it represents a unique data item reference consisting of a syntactically correct combination of a condition-name, together with qualifiers and subscripts, as required for uniqueness of reference.

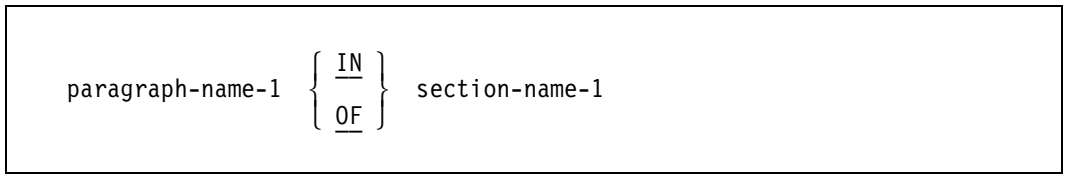
file-name-1

A file-name is a user-defined word that names a file connector described in a file description entry or a sort-merge file description entry in the File Section of the Data Division.

IN
OF

These keywords are logically equivalent.

Qualification Format 2



paragraph-name-1

A paragraph-name is a user-defined word that identifies and begins a paragraph in the Procedure Division.

If explicitly referenced, a paragraph-name must not be duplicated in a section.

When a paragraph-name is qualified by a section-name, the word SECTION must not appear.

A paragraph-name does not need to be qualified when it is referred to within the same section.

A paragraph-name in one program cannot be referred to from any other program.

IN
OF

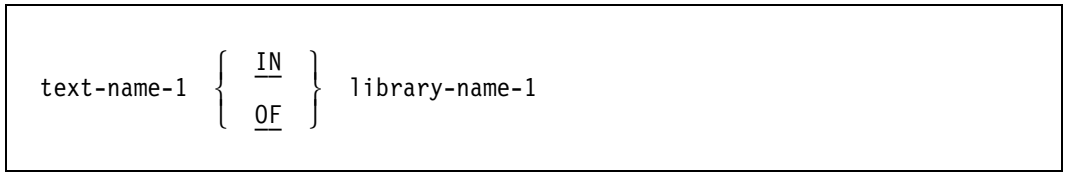
These keywords are logically equivalent.

section-name-1

A section-name is a user-defined word that names a section in the Procedure Division.

A section-name in one program cannot be referred to from any other program.

Qualification Format 3



text-name-1

A text-name is a user-defined word that identifies library text.

If more than one COBOL library is available to the compiler during compilation, text-name-1 must be qualified each time it is referred to.

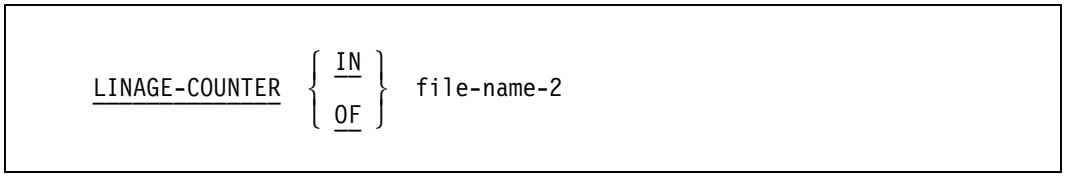
IN
OF

These keywords are logically equivalent.

library-name-1

A library-name is a user-defined word that names a COBOL library that is to be used by the compiler for a given source program compilation.

Qualification Format 4



LINAGE-COUNTER

LINAGE-COUNTER must be qualified each time it is referred to if more than one file description entry containing a LINAGE clause have been specified in the source program.

IN
OF

These keywords are logically equivalent.

file-name-2

A file-name is a user-defined word that names a file connector described in a file description entry or a sort-merge file description entry in the File Section of the Data Division.

Details

For each nonunique user-defined name that is explicitly referred to, uniqueness must be established through a sequence of qualifiers that precludes any ambiguity of reference.

A name can be qualified even though it does not need qualification; if there is more than one combination of qualifiers that ensures uniqueness, then any such set can be used.

Reference Modifiers

A reference modifier identifies a function or a data item by specifying a leftmost character and a length for the function or data item. Unless otherwise specified, a reference modifier is allowed only when the function-name or data-name references an alphanumeric function or data item.

```
{data-name-1                                  }
{FUNCTION funct-name-1 [({argument-1}...)]    }

             (leftmost-char-position:[length])
```

Note: *The reference modifier consists only of the leftmost-character-position and the length. The other elements in the preceding syntax are provided only for context.*

data-name-1

This data-name must refer to a data item whose usage is DISPLAY or NATIONAL. It can be qualified or subscripted.

FUNCTION function-name-1 (argument-1)

This is an alphanumeric function. For information about functions, refer to Section 9.

leftmost-character-position

This must be an arithmetic expression. For details about arithmetic expressions, refer to Section 5.

Evaluation of the leftmost-character-position specifies the ordinal position of the leftmost character of the unique data item in relation to the leftmost character of the data item or function specified in this format.

Evaluation of the leftmost-character-position must result in a positive nonzero integer less than or equal to the number of characters in the data item or function specified in this format.

length

This must be an arithmetic expression. For details, see Section 5.

The evaluation of the length specifies the size of the data item to be used in the operation.

The evaluation of the length must result in a positive nonzero integer.

The sum of the leftmost-character-position and the length minus the value 1 must be less than or equal to the number of characters in the data item or function specified in this format.

If the length is not specified, the unique data item extends from and includes the character identified by the leftmost-character-position up to and including the rightmost character of the data item or function specified in this format.

Details

Reference modification creates a unique data item that is a subset of the data item or function specified in this format. The syntax descriptions for the leftmost-character-position and the length contain the definitions of the unique data item.

The unique data item is considered an elementary data item without the JUSTIFIED clause. If a function is specified, the data item has the class and category of alphanumeric. When a data item is specified, the unique data item has the same class and category as defined for the data item referred to by data-name-1, with the exceptions shown in the following table.

The category . . .	Is considered to be class and category . . .
Numeric	Alphanumeric
Numeric-edited	Alphanumeric
Alphanumeric-edited	Alphanumeric
National-edited	National

Each character of a data item or a function specified in this format is assigned an ordinal number that is incremented by one from the leftmost position to the rightmost position. The leftmost position is assigned the ordinal number one. Note that if the data description entry for data-name-1 contains a SIGN IS SEPARATE clause, the sign position is assigned an ordinal number within that data item.

The type of data item specified by data-name-1 determines how that data item is treated for purposes of reference modification:

If the data item specified by data-name-1 is described as . . .	Then it is treated in reference modification as if it were redefined as . . .
Numeric, numeric-edited, alphabetic, or alphanumeric-edited	An alphanumeric data item of the same size as the data item referred to by data-name-1
National-edited	A national data item of the same size as the data item referred to by data-name-1

Reference modification for an operand is evaluated as follows:

- If subscripting is specified for the operand, the reference modification is evaluated immediately after evaluation of the subscripts.
- If subscripting is not specified for the operand, the reference modification is evaluated at the time subscripting would be evaluated if subscripts had been specified.
- If the subscript ALL is specified for an operand, the reference modifier is applied to each of the implicitly specified elements of the table.

If a reference modifier is specified in a function reference, the reference modifier is evaluated immediately after the function is evaluated.

General Format

The general format of the Data Division is as follows:

```
DATA DIVISION.

[ FILE SECTION.

   [ file description entry
     { record description entry } ... ] ...
   [ sort-merge file description entry
     { record description entry }... ] ... ]

[ DATA-BASE SECTION.

   [ 01  [ internal-set name ]  INVOKE set-name ]  ...   ]

[ WORKING-STORAGE SECTION.

   [ 77-level description entry ]
      record description entry      ... ]

[ LINKAGE SECTION.

   [ 77-level description entry ]
      record description entry     ... ]

[ COMMUNICATION SECTION.

   [ COMS headers ]  ...  ]

[ LOCAL-STORAGE SECTION.

   local-storage description entry
     { 77-level description entry }
     {                            } ... ]
     { record description entry   }

[ REPORT SECTION.

   [ A report description entry ... ] ...
   [ A report-group description entry ... ] ... ]

[ PROGRAM-LIBRARY SECTION.

    [ library description entry ] ...    ]
```

Because the record description entry is used in each of the sections of the Data Division, it is described in the following subsection rather than as an element of each section.

For information on the Data-Base Section and the Communication Section, refer to *COBOL ANSI-85 Programming Reference Manual, Volume 2: Product Interfaces*.

Record Description Entry

A record description consists of a set of data description entries that describe the characteristics of a particular record. Because a record description can have a hierarchical structure, the clauses used with an entry can vary considerably, depending on whether it is followed by subordinate entries.

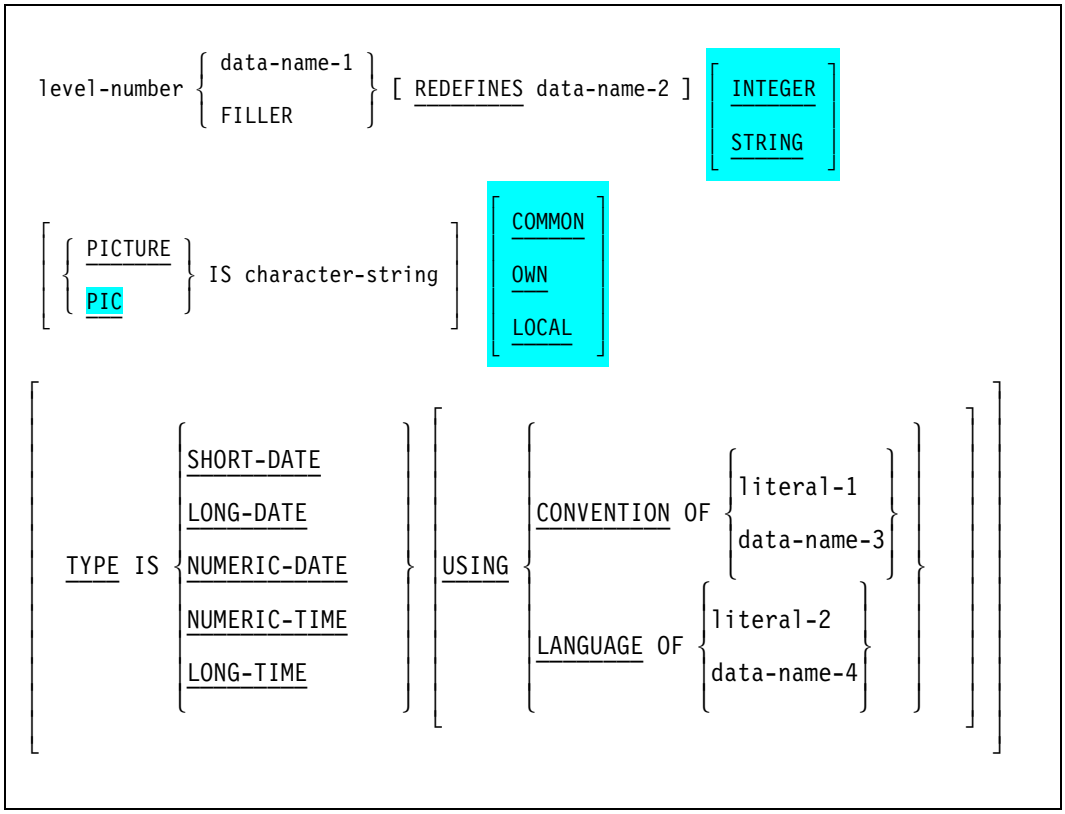
One or more record description entries must follow the file description entry.

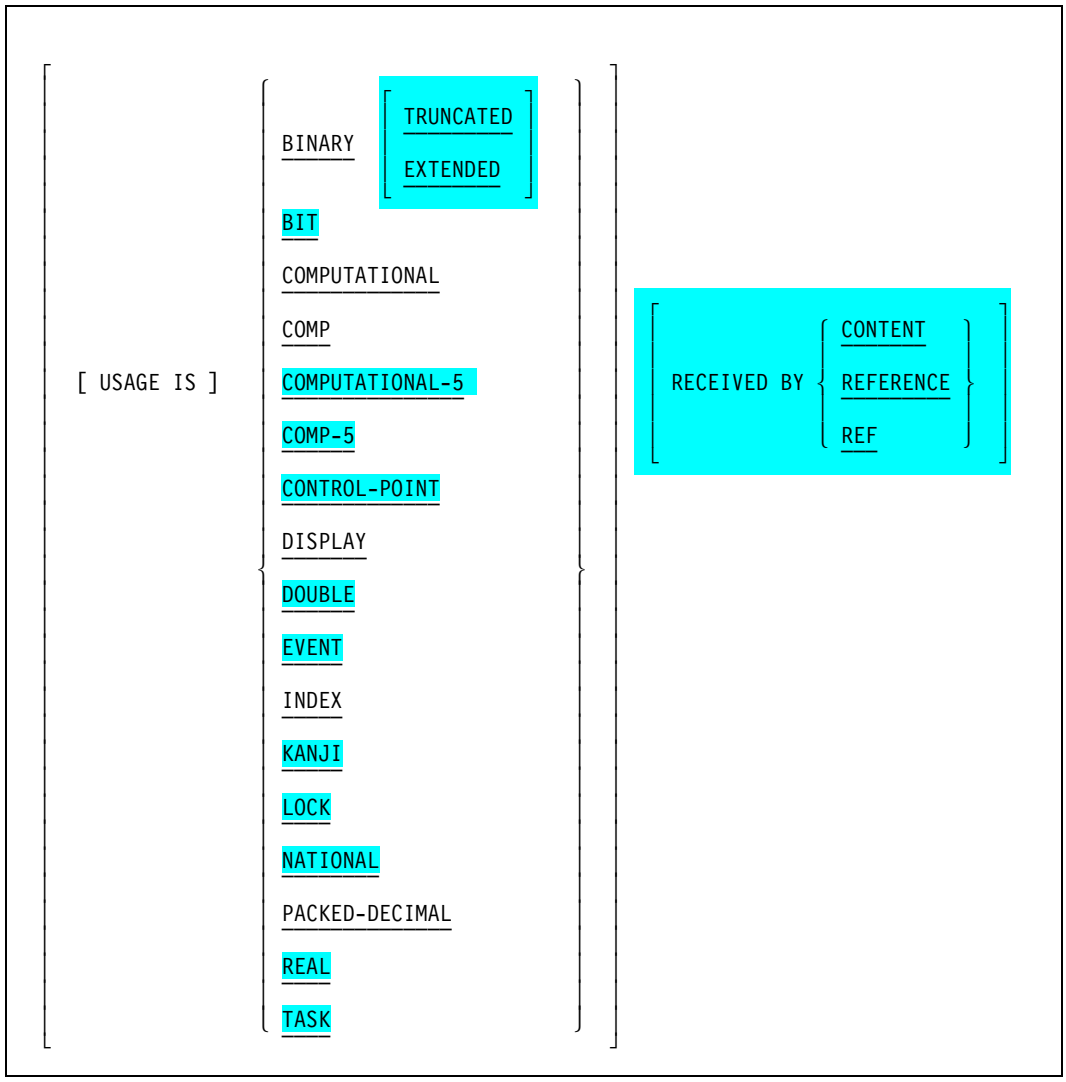
Format	Use
Format 1	This format identifies noncontiguous working-storage data items and noncontiguous linkage data items.
Format 2	This format renames a data-name or range of data-names.
Format 3	This format defines condition-names associated with conditional variables.
Format 4	This format is used for interprogram communication. It determines whether the data record and its subordinate data items have local names or global names, and it determines the internal or external attribute of the data record and its subordinate data items.

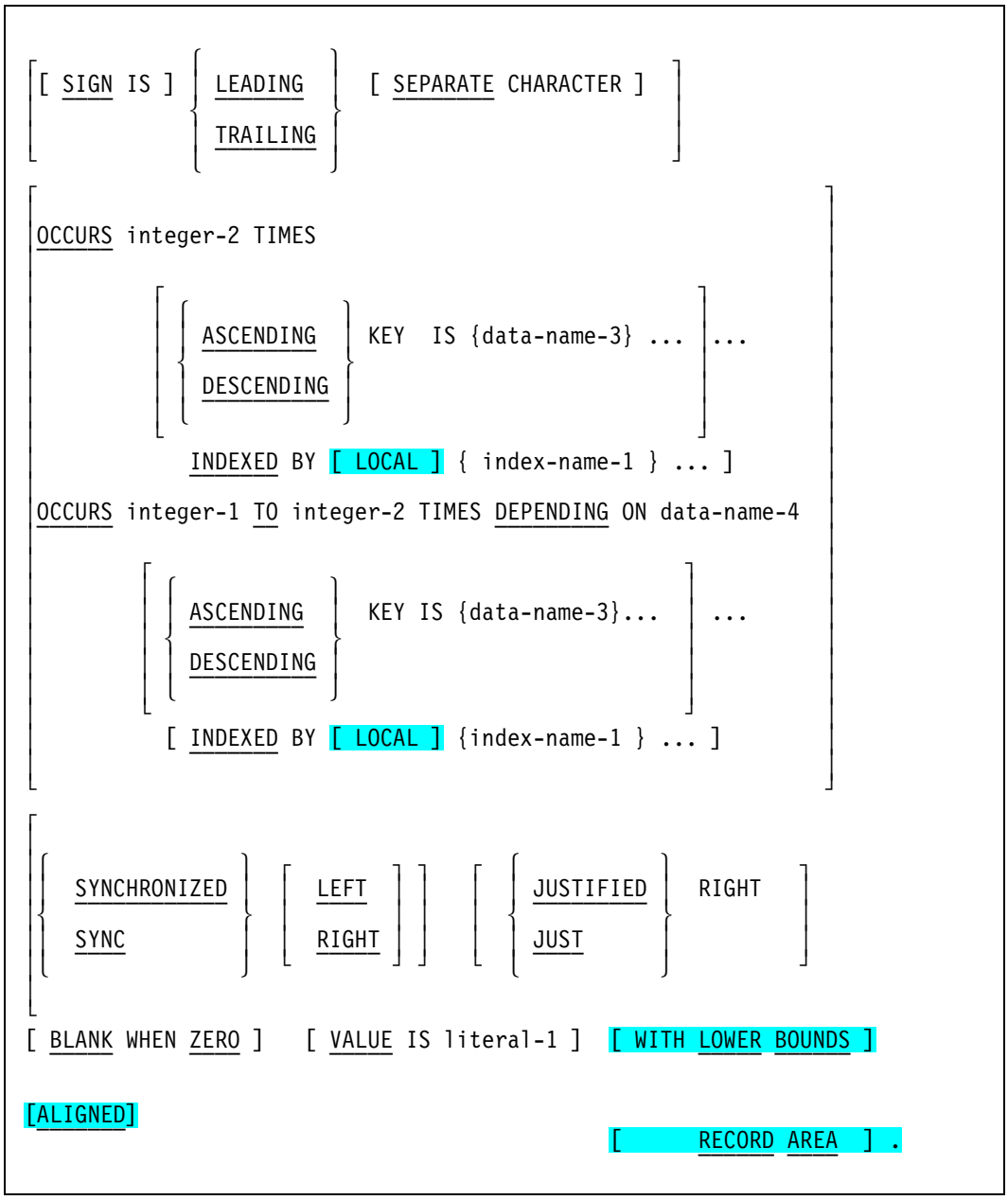
The syntax of each data description entry is illustrated and explained on the following pages.

Data Description Entry Format 1

The format for Data Description Entry Format 1 is shown on the following three pages. A user will use this syntax for most data items.




[USAGE IS]
BINARY
TRUNCATED
EXTENDED
BIT
COMPUTATIONAL
COMP
COMPUTATIONAL-5
COMP-5
CONTROL-POINT
DISPLAY
DOUBLE
EVENT
INDEX
KANJI
LOCK
NATIONAL
PACKED-DECIMAL
REAL
TASK
RECEIVED BY
CONTENT
REFERENCE
REF


[SIGN IS] { LEADING / TRAILING } [SEPARATE CHARACTER]
OCCURS integer-2 TIMES
{ ASCENDING / DESCENDING } KEY IS {data-name-3} ... ...
INDEXED BY [LOCAL] { index-name-1 } ...]
OCCURS integer-1 TO integer-2 TIMES DEPENDING ON data-name-4
{ ASCENDING / DESCENDING } KEY IS {data-name-3}... ...
[INDEXED BY [LOCAL] {index-name-1 } ...]
{ SYNCHRONIZED / SYNC } [LEFT / RIGHT]
{ JUSTIFIED / JUST } RIGHT
[BLANK WHEN ZERO] [VALUE IS literal-1] [WITH LOWER BOUNDS]
[ALIGNED]
[RECORD AREA] .

The level-number, the data-name, and the REDEFINES clause must appear in the order in which they are described in the syntax diagram. All other clauses of Format 1 can appear in any order.

The first three clauses of this data description entry are described in order on the following pages. The remaining clauses are described in alphabetical order.

level-number

In this format, this number can be one of the following:

- Any number from 01 through 49
- 77

Level-number 77 identifies noncontiguous working-storage data items and noncontiguous linkage data items. Level-number 77 is used only in this format of a data description entry.

Data-Name or FILLER Clause

data-name-1

This name is a user-defined word that specifies the name of the data item being described.

FILLER

This keyword can be used to name an elementary item in a record.

A FILLER item can never be referred to explicitly. However, the keyword FILLER can be used to name a conditional variable because such use does not require explicit reference to the FILLER item itself, but only to the value of the FILLER item.

Example

```
03 FILLER  PIC 9.
   88 ANNUAL VALUE 1.
   88 WEEKLY VALUE 2.
   88 DAILY VALUE 3.
```

Details

If this clause is omitted, the data item being described is treated as though FILLER had been specified.

REDEFINES Clause

This clause allows the same computer storage area to be described by different data description entries.

When specified, this clause must immediately follow the data-name-1 or FILLER clause if either is specified; otherwise, it must immediately follow the level-number. The remaining clauses can be written in any order.

Note that this clause must not be used in level 01 entries in the File Section.

data-name-2

The level-numbers of data-name-2 and the subject of this entry (the data-name-1 or FILLER clause) must be identical, but must not be 66 or 88.

The data description entry for data-name-2 cannot contain an OCCURS clause; however, data-name-2 can be subordinate to an item whose data description entry contains an OCCURS clause. In this case, the reference to data-name-2 in the REDEFINES clause must not be subscripted. Neither the original definition nor the redefinition can include a variable-occurrence data item.

If the data item referred to by data-name-2 is declared to be an external data record or is specified with a level-number other than 01, the number of character positions it contains must be greater than or equal to the number of character positions in the data item referred to by the subject of this entry (the data-name-1 or FILLER clause). If the data-name referred to by data-name-2 is specified with a level-number of 01 and is not declared to be an external data record, there is no such constraint.

Data-name-2 must not be qualified, even if it is not unique. (No ambiguity of reference exists in this case because of the required placement of the REDEFINES clause in the source program.)

Rules

Multiple redefinitions of the same character positions are permitted. These redefinitions must all use the data-name of the entry that originally defined the area.

The entries that give the new description of the character positions must not contain a VALUE clause, except in condition-name entries.

No entry having a level-number numerically lower than the level-number of data-name-2 and the subject of this entry (data-name-1 or FILLER) can occur between the data description entries of data-name-2 and the subject of this entry (data-name-1 or FILLER).

The entries that give the new descriptions of the character positions must follow the entries defining the area of data-name-2, without intervening entries that define new character positions.

Details

Data-name-2 can be subordinate to an entry that contains a REDEFINES clause.

Storage allocation starts at data-name-2 and continues over a storage area sufficient to contain the number of character positions in the data item referred to by the data-name-1 or FILLER clause.

When the same character position is defined by more than one data description entry, the data-name associated with any of those data description entries can be used to refer to that character position.

ALIGNED Clause

This clause can be specified only for an elementary bit data item. The ALIGNED clause specifies that an elementary bit data item be aligned at the leftmost bit of the next character boundary in storage. The ALIGNED clause and the SYNCHRONIZED clause are mutually exclusive.

BLANK WHEN ZERO Clause

This clause fills a data item with spaces when the value of the data item is zero.

This clause can be specified only for an elementary item whose PICTURE is specified as numeric or numeric-edited (refer to "PICTURE Clause" in this section).

The numeric or numeric-edited data description entry to which this clause applies must be described, either implicitly or explicitly, as USAGE IS DISPLAY.

When this clause is used for an item whose PICTURE is numeric, the category of the item is considered to be numeric-edited.

Example

```
01  PRODUCT.
        05  PART-NO      PIC IS 99999.
        05  QUANTITY     PIC S9999   SIGN IS LEADING SEPARATE.
        05  UNIT-PRICE   PIC IS 999V99.
        05  TOTAL-PRICE  PIC 9(5)V99  BLANK WHEN ZERO.
```

In this example, the data item TOTAL-PRICE is set to blank spaces when its value is zero.

COMMON Clause

COBOL procedures compiled at lexicographic level 3 or higher can use untyped procedures, files, and certain variables in the outer block of the host program when those procedures, files, and variables are declared as COMMON.

Any 77-level item or 01-level item declared in the Working-Storage Section of a host program can be passed as a parameter. You can declare such a data item as COMMON in a bound procedure by using the COMMON clause in the data description entry for the item.

COMMON declarations are matched by name and type to the global directory of the host.

Using the COMMON clause in the host program is redundant, because each data item declared in a host program is placed in the global directory of the host. However, no error or warning message is issued if the COMMON clause is declared in the host program.

Index-names generated for a COMMON array are COMMON items themselves.

If you must declare most or all of the variables in the Working-Storage Section as COMMON, you can set the COMMON compiler option to TRUE. This option affects only variables in the Working-Storage Section that can be declared COMMON. For more information about compiler options, refer to Section 15. You can use the LOCAL or OWN clause to override the COMMON compiler option.

If the data item you are declaring as COMMON has an OCCURS clause, you can specify the index-name as LOCAL by using the INDEXED BY LOCAL clause within the OCCURS clause.

Examples

```
77  GLASTATUS COMMON COMP PIC 9(11).
77  GL-RCD COMMON.
01  GL-EBCARY COMMON.
    03  CMP-ITM COMP PIC 9(11) OCCURS 100 INDEXED BY I.
```

INTEGER and STRING Clauses

These clauses identify the type of the data item used as a library parameter. INTEGER will identify the data type as an integer, and STRING will identify the data type as a string.

To use the INTEGER clause, the level-number of the data item must be either 01 or 77, and the USAGE of the data item must be COMPUTATIONAL.

To use the STRING clause, the level-number of the data item must be either 01 or 77, and the USAGE of the data item must be DISPLAY or NATIONAL.

Note: *The INTEGER clause and the STRING clause are ignored for data items not referenced as formal parameters.*

JUSTIFIED (JUST) Clause

This clause permits alternate (nonstandard) positioning of data in a receiving data item. "JUST" and "JUSTIFIED" are equivalent words.

Rules

- This clause can be specified only at the elementary item level.
- This clause cannot be specified for any data item described as numeric or for which editing is specified.
- This clause must not be specified for an index data item.

Details

When the receiving data item is described with the JUSTIFIED clause and the sending data item is larger than the receiving data item, the leftmost characters are truncated. When the receiving data item is described with the JUSTIFIED clause and it is larger than the sending data item, the data is aligned at the rightmost character position in the data item with space fill for the leftmost character positions.

When the JUSTIFIED clause is omitted, the standard rules for aligning data in an elementary item apply (refer to "Standard Alignment Rules" earlier in this section).

LOCAL Clause

A local variable is a variable that is referenced in the same procedure in which it is declared. Any value stored in a local variable is lost upon exit from that procedure.

For COBOL procedures compiled at level 3 or higher, the variables are declared implicitly LOCAL unless the COMMON or OWN clause is specified.

Index-names for a LOCAL array are treated as LOCAL variables.

LOWER-BOUNDS Clause

The LOWER-BOUNDS clause enables COBOL85 programs to handle array parameters that generate a stack item to pass the lower bound of the array.

This clause is used in the data description of a 01 item in the Linkage Section or Working-Storage Section (if LOWER-BOUNDS are received), or the Local-Storage Section (if LOWER-BOUNDS are to be passed).

This clause declares formal parameters to be compatible with FORTRAN and ALGOL. It must be used when communicating with ALGOL programs with formal array parameters declared with a variable lower-bound description (for example, ARRAYNAME [*]).

The actual lower bound passed by a COBOL program as a parameter to another program will always have a value of zero. The actual lower bound received by a COBOL program as a parameter will not be used in addressing the array. Therefore, the LOWER-BOUNDS clause affects bound and library procedures. ALGOL programs that call a library passing an array parameter with a lower-bound [*] send two parameters: a by-reference array followed by a by-value integer. Thus, the COBOL85 library must declare a data description with LOWER-BOUNDS to receive the lower-bound parameter.

Note: *This clause is ignored for data items not referenced as formal parameters.*

Examples

COBOL Parameter	Corresponding ALGOL Parameter
01 BINARY	REAL array [<integer >]
01 BINARY WITH LOWER-BOUNDS	REAL array [*]
01 COMP	HEX array[<integer>]
01 COMP WITH LOWER-BOUNDS	HEX array[*]
01 DISPLAY	EBCDIC char array[<integer>]
01 DISPLAY WITH LOWER-BOUNDS	EBCDIC char array[*]

The use of <integer> in the preceding examples refers to a specified lower bound associated with the array.

OCCURS Clause

This clause eliminates the need for separate entries for repeated data items and supplies information required for the application of subscripts or indexes.

This clause must not be specified in a data description entry that has either:

- A level-number of 01, 66, 77, or 88
- A variable-occurrence data item subordinate to the entry

Except for the OCCURS clause, all data description clauses associated with an item whose description includes an OCCURS clause apply to each occurrence of the item described.

OCCURS integer-2 TIMES

This form of the OCCURS clause specifies that the subject of this entry (the data-name-1 or FILLER clause) occurs the number of times indicated by the integer in this clause.

integer-2

The value of this integer represents the exact number of occurrences of the subject of this entry (the data-name-1 or FILLER clause).

OCCURS integer-1 TO integer-2 TIMES

This form of the OCCURS clause specifies that the subject of this entry (the data-name-1 or FILLER clause) has a variable number of occurrences. (Note that the length of the subject of this entry is not variable, just the number of occurrences.)

A data description entry that contains this form of the OCCURS clause can only be followed, within that record description, by data description entries that are subordinate to it.

If this form of the OCCURS clause is specified in a record description entry and the associated file description entry or sort-merge description entry contains the VARYING phrase of the RECORD clause, the records are variable length. If the DEPENDING ON phrase of the RECORD clause is not specified, the content of the data item referred to by data-name-4 of the OCCURS clause must be set to the number of occurrences to be written before the execution of any RELEASE, REWRITE, or WRITE statement.

integer-1

The value of this integer represents the minimum number of occurrences of the subject of this entry (the data-name-1 or FILLER clause).

When both integer-1 and integer-2 are used, integer-1 must be greater than or equal to zero, and integer-2 must be greater than integer-1.

integer-2

The value of this integer represents the maximum number of occurrences of the subject of this entry (the data-name-1 or FILLER clause).

DEPENDING ON data-name-4

Data-name-4 can be qualified.

Data-name-4 must describe an integer.

The data item defined by data-name-4 must not occupy a character position in the range between the first character position defined by the data description entry containing the OCCURS clause and the last character position defined by the record description entry containing that OCCURS clause.

If the OCCURS clause is specified in a data description entry that is included in a record description entry containing the EXTERNAL clause, data-name-4, if specified, must reference a data item that possesses the external attribute that is described in the same Data Division.

If the OCCURS clause is specified in a data description entry subordinate to one containing the GLOBAL clause, data-name-4, if specified, must be a global name and must reference a data item that is described in the same Data Division.

The data item identified by data-name-4 must not contain an OCCURS clause except when data-name-4 is the subject of this entry.

The current value of the data item referred to by data-name-4 represents the number of occurrences of the subject of this entry.

At the time the subject of this entry is referred to or any data item subordinate or superordinate to the subject of this entry is referred to, the value of the data item referred to by data-name-4 must fall in the range of integer-1 through integer-2. The contents of the data items whose occurrence numbers exceed the value of the data item referred to by data-name-4 are undefined.

When a group item is referred to that has subordinate to it an entry that contains this form of the OCCURS clause, the part of the table area used in the operation is determined as follows:

- If the data item referred to by data-name-4 is outside the group, only that part of the table area that is specified by the value of the data item referred to by data-name-4 at the start of the operation will be used.
- If the data item referred to by data-name-4 is included in the same group and the group data item is referred to as a sending item, only that part of the table area that is specified by the value of the data item referred to by data-name-4 at the start of the operation will be used in the operation. If the group is a receiving item, the maximum length of the group will be used.

ASCENDING/DESCENDING data-name-3

No entry that contains an OCCURS clause can appear between the descriptions of the data items identified by the data-names in the KEY IS phrase and the subject of this entry.

When the KEY IS phrase is specified, the repeated data must be arranged in ascending or descending order according to the values contained in data-name-3. The ascending or descending order is determined according to the rules for the comparison of operands (see "Conditional Expressions" in Section 5). The data-names are listed in their descending order of significance.

Data-name-3 can be qualified. The first specification of data-name-3 must be the name of either the entry containing the OCCURS clause or an entry subordinate to the entry containing the OCCURS clause. Subsequent specification of data-name-3 must be subordinate to the entry containing the OCCURS clause.

Data-name-3 must be specified without the subscripting normally required.

INDEXED BY

This phrase is required if the subject of this entry, or an entry subordinate to this entry, is to be referred to by indexing. The index-name identified by this phrase is not defined elsewhere, since its allocation and format are dependent on the hardware and, not being data, cannot be associated with a data hierarchy.

LOCAL

This option designates the specified index-name as a LOCAL item along with the other LOCAL host and subprogram items. LOCAL indicates that the item is declared in the procedure that references it. If multiple index-names are specified in an INDEXED BY clause, the LOCAL clause applies to each index-name.

If the COMMON or OWN clause is specified for a data item, you can specify the index-name as LOCAL by using the INDEX BY LOCAL clause within the OCCURS clause.

index-name-1

This name must be a unique word in the program.

Example

```
01  PRIMARY.
    03  ACCT-NO          PIC 9(8).
    03  NAME              PIC X(20).
    03  OTHER-NAMES  OCCURS 1 TO 5 TIMES
          DEPENDING ON ALTERNATE-NAMES.
        05  NME     PIC X(20).
        05  FLAG    PIC 9.
```

This data description entry defines the group item PRIMARY. PRIMARY consists of the elementary items ACCT-NO and NAME, and the group item OTHER-NAMES. OTHER-NAMES consists of the elementary items NME and FLAG. OTHER-NAMES can occur up to five times depending on the value of ALTERNATE-NAMES.

OWN Clause

COBOL procedures compiled at level 3 or higher can declare certain variables to be OWN. These variables retain their values through repeated exit and reentry of the procedure in which they are declared. You can declare any item in the Working-Storage Section as OWN by using the OWN clause or by setting the OWN compiler control option.

All related index-names for OWN items are also OWN; redefinitions of OWN items are implicitly OWN and do not require the OWN clause. If most or all of the variables declared in the Working-Storage Section must be declared OWN, then you can set the OWN compiler control option to TRUE throughout the compilation. To override the OWN compiler control option, use the LOCAL or COMMON clause.

Index-names for an OWN array are treated as OWN variables.

If the data item you are declaring as OWN has an OCCURS clause, you can specify the index-name as LOCAL by using the INDEXED BY LOCAL clause within the OCCURS clause.

Example

```
77 X       PIC X(10) OWN.
77 Y        REDEFINES X PIC 9(10).
01 A OWN.
    03 CMP-ITEM     COMP PIC 9 (11) OCCURS 100 INDEXED BY J.
```

PICTURE Clause

This clause describes the general characteristics and editing requirements of an elementary data item.

PICTURE
PIC

These keywords are equivalent.

IS character-string

A character-string consists of certain allowable combinations of characters in the COBOL character set used as symbols. The allowable combinations determine the category of the elementary item.

The allowable PICTURE clause symbols are *A*, *B*, *I*, *N*, *P*, *S*, *V*, *X*, *Z*, *0*, *1*, *9*, slant (/), comma (,), period (.), plus sign (+), minus sign (-), asterisk (*), currency symbol (usually $), *CR*, and *DB*. Refer to the paragraphs under the heading "Symbols" in this section for information on each of these symbols.

The lowercase letters that correspond to the uppercase letters that represent the PICTURE clause symbols *A*, *B*, *I*, *N*, *P*, *S*, *V*, *X*, *Z*, *CR*, and *DB* are the same as their uppercase representations in a PICTURE character-string. However, all other lowercase letters are not equivalent to their corresponding uppercase representations.

The maximum number of characters allowed in the character-string is 30.

Restrictions

The PICTURE clause has the following restrictions:

- A PICTURE clause can be specified only at the elementary item level.
- A group item must not have a PICTURE clause.
- Every elementary data item except an index data item or the subject of a RENAMES clause must have a PICTURE clause. The PICTURE clause is prohibited for an index data item and the subject of a RENAMES clause.
- The asterisk, when used as the zero suppression symbol, and the BLANK WHEN ZERO clause cannot appear in the same entry.

Symbols

The PICTURE clause symbols and their functions are described in Table 4–2.

Table 4–2. Picture Clause Symbols

Symbol	Function
A	Each A in the character-string represents a character position that can contain only an alphabetic character. This symbol is counted in the size of the item.
B	Each B in the character-string represents a character position into which the space character will be inserted and is counted in the size of the item. For an alphanumeric-edited item, each B represents an alphanumeric character position in the item into which an alphanumeric space character will be inserted. For a national-edited item, each B represents a national character position in the item into which a national space character is to be inserted.
I	Each I in the character string indicates that the nonblank character immediately following it is treated as a simple insertion character. Specifying the character I as the currency symbol overrides its use to indicate simple insertion characters. The I itself is not counted in the size of the item, but the single, nonblank character following it is counted in the size of the item. The 30-character limit for the size of a PICTURE string includes both the I symbol and the character that follows it.
N	Each N in the character-string represents a character position that contains a national character. Each N is counted in the size of the data item being described. The size is considered to be the total number of character positions defined for the data item.

Table 4–2. Picture Clause Symbols

Symbol	Function
P	Each P in the character-string indicates an assumed decimal scaling position, which is used to specify the location of an assumed decimal point when the point is not in the number that appears in the data item. The scaling position character P is not counted in the size of the data item. Scaling position characters are counted in determining the maximum number of digit positions (23) in numeric-edited items or numeric items. The scaling position character P can appear only as a continuous string of Ps in the leftmost or rightmost digit positions in a PICTURE character-string, because the scaling positions character P implies an assumed decimal point (to the left of Ps if Ps are leftmost PICTURE symbols and to the right if Ps are either the leftmost or rightmost character in such a PICTURE description). The symbol P and the insertion symbol period (.) cannot both occur in the same PICTURE character-string. The symbol P and the symbol V cannot both occur in the same PICTURE character-string (unless they are immediately adjacent, indicating the same character position). In certain operations that refer to a data item whose PICTURE character-string contains the symbol P, the algebraic value of the data item is used rather than the actual character representation of the data item. This algebraic value assumes the decimal point in the prescribed location and zero in place of the digit position specified by the symbol P. The size of the value is the number of digit positions represented by the PICTURE character-string. These operations can be: • Any operation that requires a numeric sending operand • A MOVE statement in which the sending operand is numeric and its PICTURE character-string contains the symbol P • A MOVE statement in which the sending operand is numeric-edited and its PICTURE character-string contains the symbol P and the receiving operand is numeric or numeric-edited • A comparison operation in which both operands are numeric In all other operations the digit positions specified with the symbol P are ignored and not counted in the size of the operand.

Table 4–2. Picture Clause Symbols

Symbol	Function
S	The letter S is used in a character string to indicate the presence of an operational sign in the internal representation of a numeric data item. A single S must be the first (leftmost) character in the character string and there cannot be more than one S character in a PICTURE clause character-string. The symbol S can be used in the PICTURE character string of any data item with the USAGE clause equal to DISPLAY, COMPUTATIONAL, or BINARY. The SIGN clause can be used to specify the exact representation and position of the operational sign. When an operational sign is specified for a DISPLAY data item and a SIGN clause is not specified, the sign is maintained and expected in the zone of the least significant (rightmost) character. When the data item is in the receiving field in an arithmetic statement and when the native character set is EBCDIC, the four zone bits are set to binary 1101 for negative values and to binary 1100 or 1111 for positive values. When the data item is used in an algebraic comparison or operation to supply an algebraic value, specification of the least significant zone as binary 1101 causes the value to be considered negative. Only the zone values 1100, 1101, and 1111 qualify the data item as NUMERIC if it is tested by the numeric class condition. For DISPLAY data items, the presence or absence of an operational sign has no effect on the amount of storage required to contain the data item, unless the SIGN SEPARATE clause is specified. When an operational sign is specified for a COMPUTATIONAL data item and a SIGN clause is not specified, the sign is maintained and expected as a leading, separate 4-bit character to the left of the most significant digit position. When the native character set is EBCDIC, the binary pattern of the sign character is 1101 for negative values and 1100 for positive values. Like DISPLAY data items, only these values allow the item to be considered NUMERIC in the class condition test. Unlike DISPLAY data items, the specification of an operational sign for COMPUTATIONAL data items increases by one the number of 4-bit character positions occupied by the data item in storage.
V	The V is used in a character-string to indicate the location of the assumed decimal point and can appear only once in a character-string. The V does not represent a character position and is not counted in the size of the elementary item. The V is redundant if the assumed decimal point is to the right of the rightmost symbol in the string that represents a digit position or scaling position.
X	Each X in the character-string represents a character position that contains any allowable character from the computer's character set. The X is counted in the size of the item.

Table 4–2. Picture Clause Symbols

Symbol	Function
Z	Each Z in a character-string can be used to represent only the leftmost leading numeric character positions that will be replaced by a space character when the content of that character position is a leading zero. Each Z is counted in the size of the item.
0	Each 0 (zero) in the character-string represents a character position into which the character 0 is to be inserted. The 0 is counted in the size of the item. For an alphanumeric-edited item, each 0 represents an alphanumeric character position into which the alphanumeric character 0 is to be inserted. For a national-edited item, each 0 represents a national character position into which the national character 0 is to be inserted.
1	The 1 in a PICTURE character-string represents a Boolean position that contains a Boolean character and can occur only once in a character-string. The 1 is counted in the size of the item.
9	Each 9 in the character-string represents a digit position that contains a numeric character. The 9 is counted in the size of the item.
/	Each slant (/) in the character-string represents a character position into which the slant character is to be inserted. The slant is counted in the size of the item. For an alphanumeric-edited item, each slant represents an alphanumeric character position into which an alphanumeric character slant is to be inserted. For a national-edited item, each slant represents a national character position into which a national character slant is to be inserted.
,	Each comma (,) in the character-string represents a character position into which the comma will be inserted. The comma is counted in the size of the item.
.	The period (.) in the character-string is an editing symbol that represents the decimal point for alignment purposes, and in addition, represents a character position into which the period will be inserted. The period is counted in the size of the item. For a given program, the functions of the period and comma are exchanged if the clause DECIMAL-POINT IS COMMA is stated in the SPECIAL-NAMES paragraph. In this exchange, the rules for the period apply to the comma, and the rules for the comma apply to the period wherever they appear in a PICTURE clause.
+ - CR DB	These symbols are editing sign control symbols. When used, they represent the character position into which an editing sign control symbol will be placed. The symbols are mutually exclusive in any one character-string. Each character used in the symbol is counted in determining the size of the data item.

Table 4–2. Picture Clause Symbols

Symbol	Function
*	Each asterisk (*) in the character-string represents a leading numeric character position into which an asterisk will be placed when the content of that position is a leading zero. Each * is counted in the size of the item.
cs	The currency symbol in the character-string represents a character position into which a currency symbol will be placed. The currency symbol in a character-string is represented by either the currency sign ($) or by the single character specified in the CURRENCY SIGN clause in the SPECIAL-NAMES paragraph. The currency symbol is counted in the size of the item.

Categories of Items

The PICTURE clause can describe the following categories of data items:

- Alphabetic
- Numeric
- Alphanumeric
- National
- Boolean
- Alphanumeric-edited
- National-edited
- Numeric-edited

Table 4–3 contains information on how to define the different categories of items.

Table 4–3. Specification of Data Item Categories in the PICTURE Clause

Item	Definition
Alphabetic	The PICTURE character-string for an alphabetic item can contain only the symbol A. The content of the character-string, when represented in standard data format, must be one or more alphabetic characters.
Numeric	The PICTURE character-string can describe two types of numeric data items: standard numeric items and long numeric items. The PICTURE character-string for standard numeric items can contain from 1 through 23 digits. The valid symbols for the PICTURE character-string are 9, P, S, and V. An unsigned numeric item, when represented in standard data format, must be one or more numeric characters. A signed numeric item can also contain a plus sign (+), minus sign (-), or other representation of an operational sign. The PICTURE character-string for long numeric items can contain from 1 to 99,999 digits. A long numeric item must be described as an unsigned integer, so operational signs, editing symbols, and the symbols P and V are not valid in its PICTURE character-string.
Alphanumeric	The PICTURE character-string for an alphanumeric item is restricted to certain combinations of the symbols A, X, and 9. The item is treated as if the character-string contained all Xs. An alphanumeric PICTURE character-string cannot consist entirely of As or entirely of 9s. The PICTURE character-string, when represented in standard data format, must be one or more characters in the computer's character set.

Table 4–3. Specification of Data Item Categories in the PICTURE Clause

Item	Definition
National	The PICTURE character-string for a national data item can contain only the letter N or X. When the letter N is used, the PICTURE clause must be accompanied by the USAGE IS NATIONAL clause. When the letter X is used, the PICTURE clause must be accompanied by the USAGE IS KANJI clause. (Note that the USAGE IS KANJI clause might become obsolete in a future release.) The PICTURE character-string, when represented in national standard data format, must be one or more characters in the national character set of the computer.
Boolean	Symbol 1 is the only symbol that the PICTURE character-string for a Boolean item can contain.
Alphanumeric-edited	The PICTURE character-string for an alphanumeric-edited item is restricted to certain combinations of simple insertion editing symbols and the symbols A, X, and 9. The PICTURE character string must contain at least one A or X, and must contain at least one simple insertion editing symbol. The PICTURE character-string, when represented in standard data format, must be two or more characters in the computer's character set.
National-edited	The PICTURE character-string for national-edited data items is restricted to certain combinations of the symbols X, N, I, B, 0, and slant (/). When the letter X is used, the PICTURE clause must be accompanied by the USAGE IS KANJI clause. Note that this clause might become obsolete in a future release. The PICTURE character-string, when represented in national standard data format, must be one or more characters in the national character set of the computer.

Table 4–3. Specification of Data Item Categories in the PICTURE Clause

Item	Definition
Numeric-edited	The PICTURE character-string for a numeric-edited item is restricted to certain combinations of simple insertion editing symbols; the symbols P, V, Z, 9, comma (,), period (.), plus sign (+), minus sign (-), CR, and DB; and the currency symbol ($). The allowable combinations are determined from the order of precedence of symbols and the editing rules. Refer to the paragraphs headed "Precedence Rules" and "Editing Rules" in this section. The number of digit positions that can be represented in the PICTURE character-string must range from 1 to 23 inclusive. The character-string must contain at least one simple insertion editing symbol, asterisk (*), plus sign (+), comma (,), period (.), minus sign (-), slant (/), CR, DB, or currency symbol ($). The content of each character position must be consistent with the corresponding PICTURE symbol. The size of an elementary item refers to the number of character positions occupied by the item in standard data format. The number of allowable symbols that represent character positions determines the size of an elementary item. The following symbols can appear only once in a given PICTURE: S, V, period (.), CR, and DB.

Determining the Size of an Elementary Item

The size of an elementary item is the number of character positions it occupies in standard data format. You indicate the size of an elementary item by using the number of allowable symbols that represent character positions. For example, 9999 indicates a field with four digits.

The symbols *A*, *B*, *P*, *X*, *Z*, *9*, *0* (zero), asterisk (*), slant (/), comma (,), plus sign (+), minus sign (-), or currency symbol ($) can appear more than once in a given PICTURE clause. You can specify a number of consecutive occurrences of a symbol by using an unsigned integer enclosed in parentheses after the symbol. For example, X(8) indicates eight alphanumeric characters.

Editing Rules

Editing in the PICTURE clause can be done either by insertion or by suppression and replacement. The four types of insertion editing are

- Simple insertion
- Special insertion
- Fixed insertion
- Floating insertion

The two types of suppression and replacement editing are

- Zero suppression and replacement with spaces
- Zero suppression and replacement with asterisks

The category to which an item belongs determines the type of editing that can be used, as shown in Table 4–4.

Table 4–4. Types of Editing for Data Item Categories

Category	Type of Editing
Alphabetic	None.
Numeric	None.
Alphanumeric	None.
National	None.
Alphanumeric-edited	Simple insertion.
National-edited	Simple insertion (B, slash (/), and zero (0) only).
Numeric-edited	All. Floating insertion editing and editing by zero suppression and replacement are mutually exclusive in a PICTURE clause. Only one type of replacement can be used with zero suppression in a PICTURE clause.

Simple Insertion Editing

Unisys supports the following two forms of simple insertion editing:

- ANSI simple insertion editing

  The space character (B), slash (/), zero (0), and comma (,) are used as insertion characters.

  When the STRICTPICTURE compiler control option is set, PICTURE character strings are expected to conform to the rules set forth in ANSI X3.23-1985; syntax errors are issued for any PICTURE character strings that do not conform. When the STRICTPICTURE compiler control option is reset, certain additional variations on simple insertion editing are allowed as detailed in the following explanations for Manual insertion editing and automatic insertion editing.

- Manual insertion editing

  When the AUTOINSERT option is RESET, a Unisys extension to ANSI X3.23-1985 COBOL allows the symbol I to be used to introduce any nonblank character as a simple insertion character. Therefore, any nonblank character that immediately follows the symbol I in a PICTURE character string is treated as a simple insertion character. This feature is intended as a replacement for automatic insertion editing, which is scheduled for deimplementation in a future software release.

- Automatic insertion editing

  When the AUTOINSERT compiler control option is SET, a Unisys extension to ANSI X3.23-1985 COBOL allows any character within a PICTURE character string that is not recognized by the compiler as valid in its particular immediate context is treated as a simple insertion character. This rule applies whether or not the particular character has a meaning in another context within a PICTURE character string (such as specified in Table 4-5, "Precedence Rules").

  > ***Note:*** *The AUTOINSERT compiler control option and the Automatic insertion editing extension are scheduled for deimplementation in a future software release. Refer to the preceding explanation of Manual insertion editing for an improved method of using arbitrary symbols as simple insertion characters.*

  For example, an extra left or right parenthesis that is not part of a valid parenthetical expression indicating multiple occurrences of the same symbol is by this rule an insertion character. Any digit other than 9 is also an insertion character when it does not occur within such a parenthetical expression. Plus and minus signs are insertion characters when they appear outside of the contexts in which they are correctly interpreted as signs. The Z and * characters in a PICTURE clause in which floating insertion editing or zero suppression is already indicated by other specifications are insertion characters. CR and DB sequences before the end of the PICTURE clause or after another sign has already appeared in the PICTURE clause are treated as insertions.

  This list is illustrative and is not intended to cover all of the possibilities. The extension applies to all characters in the EBCDIC character set in any context within the PICTURE clause, except the character B, the space character, and those characters for which a role has been defined in that particular context.

The character B already specifies simple insertion editing according to ANSI standards in that it causes the insertion of a space into the output string. This same functionality applies, according to this extension, even when the B character appears outside of ANSI-defined contexts. The B character in a PICTURE string never results in the insertion of a B into the output even if it appears in a context in which it would otherwise be treated as invalid.

The space character always indicates that the character immediately preceding it is the last character in the PICTURE character string.

A period followed by a space character always serves to indicate the end of the PICTURE string, as it does for ANSI-compliant PICTURE character strings.

The only simple insertion editing functionality that Unisys supports is as stated in the preceding descriptions. The results of using characters in a PICTURE character string that does not conform to either ANSI or automatic insertion editing rules are unpredictable.

If the insertion character comma (,) is the last symbol in the PICTURE character-string, then the PICTURE clause must be the last clause of the data description entry and must be immediately followed by the separator period. As a result, the combination of a comma and a period (,.) appears in the data description entry (or, if the DECIMAL POINT IS COMMA clause is used, two consecutive periods (..) will appear).

Special Insertion Editing

The period (.) is used as the insertion character. In addition to being an insertion character, it also represents the decimal point for alignment purposes. The insertion character used for the actual decimal point is counted in the size of the item. The use of the assumed decimal point, represented by the symbol *V*, and the actual decimal point, represented by the insertion character, in the same PICTURE character-string is not allowed.

If the insertion character period (.) is the last symbol in the PICTURE character-string, the PICTURE clause must be the last clause of that data description entry and must be followed by the separator period. As a result, two consecutive periods (..) appear in the data description entry (or the combination of a comma and a period (,.) if the DECIMAL-POINT IS COMMA clause is used). The result of special insertion editing is the appearance of the insertion character in the item in the same position as shown in the character-string.

Fixed Insertion Editing

The currency symbol and the editing sign control symbols plus sign (+), minus sign (-), *CR*, and *DB* are the insertion characters. Only one currency symbol and only one of the editing sign control symbols can be used in a given PICTURE character-string. When the symbols *CR* or *DB* are used, they represent two character positions in determining the size of the item and they must represent the rightmost character positions that are counted in the size of the item. If these character positions contain the symbols *CR* or *DB,* the uppercase letters are the insertion characters. The plus sign (+) or minus sign (-), when used, must be either the leftmost or rightmost character position to be counted in the size of the item. The currency symbol must be the leftmost character position to be counted in the size of the item except that it can be preceded by either a plus sign (+) or minus sign (-). Fixed insertion editing results in the insertion character occupying the same character position in the edited item as it occupied in the PICTURE character-string. Editing sign control symbols produce the following results depending upon the value of the data item:

Editing Symbol in PICTURE Character-string	Result	
	Positive or Zero Data Item	Negative Data Item
+	+	–
–	space	–
CR	2 spaces	CR
DB	2 spaces	DB

Floating Insertion Editing

The currency symbol and editing sign control symbols plus sign (+) and minus sign (-) are the floating insertion characters. They are mutually exclusive in a given PICTURE character-string.

Floating insertion editing is indicated in a PICTURE character-string by using a string of at least two of the floating insertion characters. This string can contain any of the simple insertion characters or have simple insertion characters immediately to the right of this string. These simple insertion characters are part of the floating string. When the floating insertion character is the currency symbol, the string of floating insertion characters can have the fixed insertion characters *CR* and *DB* immediately to the right of this string.

The leftmost character of the floating insertion string represents the leftmost limit of the floating symbols in the data item. The rightmost character of the floating string represents the rightmost limit of the floating symbols in the data item.

The second floating character from the left represents the leftmost limit of the numeric data that can be stored in the data item. Nonzero numeric data may replace all the characters at or to the right of this limit.

In a PICTURE character-string, there are only two ways of representing floating insertion editing. One way is to represent any or all of the leading numeric character positions on the left of the decimal point by the insertion character. The other way is to represent all the numeric character positions in the PICTURE character-string by the insertion character.

If the insertion character positions are only to the left of the decimal point in the PICTURE character-string, the result is that a single floating insertion character will be placed into the character position immediately preceding either the decimal point or the first nonzero digit in the data represented by the insertion symbol string, whichever is farther to the left in the PICTURE character-string. The character positions preceding the insertion character are replaced with spaces.

If all numeric character positions in the PICTURE character-string are represented by the insertion character, at least one of the insertion characters must be to the left of the decimal point.

When the floating insertion character is the plus sign (+) or minus sign (-), the character inserted depends on the value of the data item:

Editing Symbol in PICTURE Character-string	Result	
	Positive or Zero Data Item	Negative Data Item
+	+	–
–	space	–

If all numeric character positions in the PICTURE character-string are represented by the insertion character, the result depends on the value of the data. If the value is zero, the entire data item will contain spaces. If the value is not zero, the result is the same as when the insertion character is only to the left of the decimal point.

To avoid truncation, the minimum size of the PICTURE character-string for the receiving data item must be the number of characters in the sending data item, plus the number of nonfloating insertion characters being edited into the receiving data item, plus one for the floating insertion character. If truncation does occur, the value of the data that is used for editing is the value after truncation. Refer to "Standard Alignment Rules" in this section for more information.

Zero-Suppression Editing

The suppression of leading zeros in numeric character positions is indicated by the use of the alphabetic character *Z* or the character asterisk (*) as suppression symbols in a PICTURE character-string. These symbols are mutually exclusive in a given PICTURE character-string. Each suppression symbol is counted in determining the size of the item. If *Z* is used, the replacement character will be the space and if the asterisk is used, the replacement character will be an asterisk (*).

Zero-suppression and replacement is indicated in a PICTURE character-string by using a string of one or more of the allowable symbols to represent leading numeric character positions that are to be replaced when the associated character position in the data contains a leading zero. Any of the simple insertion characters embedded in the string of symbols or to the immediate right of this string are part of the string.

In a PICTURE character-string, there are only two ways of representing zero suppression. One way is to represent any or all of the leading numeric character positions to the left of the decimal point by suppression symbols. The other way is to represent all of the numeric character positions in the PICTURE character-string by suppression symbols.

If the suppression symbols appear only to the left of the decimal point, any leading zero in the data that corresponds to a symbol in the string is replaced by the replacement character. Suppression terminates at the first nonzero digit in the data represented by the suppression symbol string or at the decimal point, whichever is encountered first.

If all numeric character positions in the PICTURE character-string are represented by suppression symbols and the value of the data is not zero, the result is the same as if the suppression characters were only to the left of the decimal point. If the value is zero and suppression symbol is *Z*, the entire data item, including any editing characters, is spaces. If the value is zero and the suppression symbol is an asterisk (*), the entire data item, including any insertion editing symbols except the actual decimal point, will be an asterisk (*). In this case, the actual decimal point will appear in the data item.

When the symbols plus sign (+), minus sign (-), asterisk (*), *Z*, and the currency symbol (usually $) are used as floating replacement characters, they are mutually exclusive within a given character-string. The zero-suppression editing characters Z and asterisk (*) can be used as simple insertion characters in limited situations. When they are the trailing characters to the right of the decimal point in what would otherwise be a valid floating insertion editing picture, the Z or asterisk is treated as a simple insertion character.

Precedence Rules

The following table shows the order of precedence for characters as symbols in a character-string. An *X* at an intersection indicates that the symbol or symbols at the top of the column can precede (not necessarily immediately), in a given character-string, the symbol or symbols at the left of the row. Arguments in braces {} indicate that the symbols are mutually exclusive. The currency symbol is indicated by the symbol *cs*.

At least one of the symbols *A*, *X*, *N*, *Z*, *9*, or asterisk (*), or at least two occurrences of one of the symbols plus sign (+), minus sign (-), or a currency symbol (for example, *$*) must be present in a PICTURE character-string.

The nonfloating insertion symbols plus sign (+) and minus sign (-); the floating insertion symbols *Z*, asterisk (*), plus sign (+), minus sign (-), and currency symbol; and symbol *P* appear twice in the PICTURE character precedence in the tables that follow. The leftmost column and uppermost row for each symbol represent its use to the left of the decimal point position. The second appearance of the symbol in the chart represents its use to the right of the decimal point position.

Note: *The symbol* I *in the following table represents any manual insertion character when $AUTOINSERT is set, and the nonblank character following the* I *in the PICTURE string when $AUTOINSERT is reset.*

Table 4–5. Precedence Rules

First Symbol / Second Symbol		Nonfloating Insertion Symbols										Floating Insertion Symbols						Other Symbols						
		B	I	O	/	,	.	+ –	+ –	CR DB	CS	Z *	Z *	+ –	+ –	CS	CS	9	A X	S	V	P	P	N
Nonfloating Insertion Symbols	B	X	X	X	X	X	X	X			X	X	X	X	X	X	X	X	X		X		X	X
	I	X	X	X	X	X	X	X			X	X	X	X	X	X	X	X	X		X		X	X
	O	X	X	X	X	X	X	X			X	X	X	X	X	X	X	X	X		X		X	X
	/	X	X	X	X	X	X	X			X	X	X	X	X	X	X	X	X		X		X	X
	,	X	X	X	X	X	X	X			X	X	X	X	X	X	X	X			X		X	
	.	X	X	X	X	X		X			X	X		X		X		X						
	+ –																							
	+ –	X	X	X	X	X	X				X	X	X			X	X	X			X	X	X	
	CR DB	X	X	X	X	X	X				X	X	X			X	X	X			X	X	X	
	CS							X																
Floating Insertion Symbols	Z *	X	X	X	X	X		X			X	X												
	Z *	X	X	X	X	X	X	X			X	X	X								X		X	
	+ –	X	X	X	X	X					X			X										
	+ –	X	X	X	X	X	X				X			X	X						X		X	
	CS	X	X	X	X	X		X								X								
	CS	X	X	X	X	X	X	X								X	X				X		X	
Other Symbols	9	X	X	X	X	X	X	X			X	X		X		X		X	X	X	X		X	
	A X	X	X	X	X													X	X					
	S																							
	V	X	X	X	X	X		X			X	X		X		X		X		X		X		
	P	X	X	X	X	X		X			X	X		X		X		X		X		X		
	P							X			X									X	X		X	
	N	X	X	X	X																			X

RECEIVED BY Clause

This clause identifies the way in which parameters and results are passed between two procedures or between a user program and an imported library procedure.

- If the parameter or result is declared with the RECEIVED BY CONTENT clause, the parameter is passed by value.

  When parameters are passed by value, the value of the actual parameter is assigned to the formal parameter, which is handled as a local variable by the receiving procedure. Any change made to the value of a RECEIVED BY CONTENT parameter has no effect outside of the receiving procedure.
- If the parameter or result is declared with the RECEIVED BY REFERENCE clause, the parameter is passed by reference.
- When parameters are passed by reference, the address of the actual parameter is evaluated once and passed to the formal parameter. Every reference to the formal parameter within the receiving procedure references this address. Any change made to the value of the formal parameter within the receiving procedure changes the value of the actual parameter.

If the RECEIVED BY clause is not specified, all data items and files are RECEIVED BY REFERENCE. Exceptions to this are 77-level USAGE IS BINARY, DOUBLE, and REAL parameters to bound procedures which are assumed to be RECEIVED BY CONTENT, if not otherwise specified.

REF is a synonym for REFERENCE.

Note: *This clause is ignored for data items not referenced as formal parameters.*

RECORD AREA Clause

This clause specifies that the record being described is to be used for DIRECT I–O buffering. This clause may only appear on the 01 level in a WORKING–STORAGE SECTION or a LOCAL–STORAGE SECTION.

Areas described with the RECORD AREA clause become non–overlayable until the area is specified in a DEALLOCATE statement.

An area described with the RECORD AREA clause must not be declared to be binary.

SIGN Clause

Every numeric data description entry whose PICTURE clause contains the character *S* is considered to be a signed numeric data description entry. The *S* indicates only the presence of the operational sign. To indicate the position and mode of representation of the operational sign, you can use the SIGN clause.

A numeric data description entry with an *S* in the PICTURE clause, but to which no optional SIGN clause applies, has an operational sign that is positioned and represented according to the standard default position and representation of operational signs.

(If you do not specify a SIGN clause, the sign is assumed to be in the trailing position for a DISPLAY data item or in the leading position for a COMPUTATIONAL data item unless a default sign is specified by the DEFAULT DISPLAY SIGN clause or the DEFAULT COMP SIGN clause in the Special-Names paragraph of the Environment Division.)

If a SIGN clause is specified in a group item, each item subordinate to the group item is affected.

If a SIGN clause is specified in a group item subordinate to a group item for which a SIGN clause is specified, the SIGN clause specified in the subordinate group item takes precedence for that subordinate group item.

If a SIGN clause is specified in an elementary numeric data description entry that is subordinate to a group item for which a SIGN clause is specified, the SIGN clause specified in the subordinate elementary numeric data description entry takes precedence for that elementary numeric data item.

Note that when the SIGN clause is used, any conversion necessary for computation or comparisons takes place automatically.

SIGN IS SEPARATE

If the CODE-SET clause is specified in a file description entry, any signed numeric data description entries associated with that file-description entry must be described with this form of the SIGN clause.

SEPARATE CHARACTER

If a SIGN clause with a SEPARATE CHARACTER phrase applies to a numeric data description entry, the following rules apply:

- The operational sign is presumed to be the leading or, respectively, trailing character position of the elementary numeric data item. This character position is not a digit position.
- The letter *S* in a PICTURE character-string is counted in determining the size of the item (in standard data format characters).
- The operational signs for positive and negative are the standard data format characters plus sign (+) and minus sign (-), respectively.
- When the usage of the data item is DISPLAY, the operational sign is maintained and expected as a LEADING or TRAILING character separate from, and in addition to, the numeric character positions. The operational sign for negative values is the character minus sign (-) , and for positive values, plus sign (+).
- When the usage of the data item is COMPUTATIONAL, the operational sign is maintained and expected as a binary 1100, 1111, or 1101 in the zone of the LEADING or TRAILING character. Adding this binary character increases by one 4-bit character the amount of storage allocated for the data item, in addition to the storage allocated for an unsigned COMPUTATIONAL data item. The binary numbers 1100 and 1111 represent a positive sign, whereas the binary number 1101 represents a negative sign. The presence or absence of the SEPARATE CHARACTER phrase has no effect on the position or representation of the operational sign for COMPUTATIONAL data items.

If a SIGN clause without a SEPARATE CHARACTER phrase applies to a numeric data description entry, the following rules apply:

- The operational sign will be presumed to be associated with the leading (or, respectively, trailing) digit position of the elementary numeric data item.
- The letter *S* in a PICTURE character-string is not counted in determining the size of the item (in standard data format characters).
- When the usage of the data item is DISPLAY, the operational sign is maintained and expected as a binary 1100 or 1101 in the zone of the LEADING or TRAILING character and does not cause additional storage to be allocated for the data item.
- When the usage of the data item is COMPUTATIONAL, the operational sign is maintained and expected as a binary 1100 or 1101 LEADING or TRAILING 4-bit character. Adding this binary character increases by one 4-bit character the amount of storage allocated for the data item, in addition to the storage allocated for an unsigned COMPUTATIONAL data item. The presence or absence of the SEPARATE CHARACTER phrase has no effect on the position or representation of the operational sign for COMPUTATIONAL data items.

SYNCHRONIZED Clause

This clause specifies the alignment of an elementary item on the natural boundaries of the computer memory (refer to "Increasing Object-Code Efficiency" in this section).

This clause specifies that the subject data item is to be aligned in the computer so that no other data item occupies any of the character positions between the leftmost and rightmost natural boundaries that delimit this data item. If the number of character positions required to store this data item is less than the number of character positions between those natural boundaries, the unused character positions (or portions thereof) must not be used for any other data item. Such unused character positions, however, are included in the following:

- The size of any group item or items to which the elementary item belongs.
- The number of character positions allocated when any such group item is the object of a REDEFINES clause. The unused character positions are not included in the character positions redefined when the elementary item is the object of a REDEFINES clause.

This clause can appear only with an elementary item.

SYNCHRONIZED
SYNC

These keywords are equivalent.

The SYNCHRONIZED keyword not followed by either RIGHT or LEFT specifies that the elementary item is to be positioned between natural boundaries in such a way as to effect efficient utilization of the elementary data item. LEFT and RIGHT have no effect on alignment and are treated only as commentary.

If the subject data item is of type COMP, it is aligned on a byte boundary. If it is a single word type (REAL or PIC 9(11) or less in BINARY or COMP-5) or a double word type (DOUBLE, or PIC (12) or larger in BINARY or COMP-5), it is aligned on a word boundary. If the previous data item did not end on a byte (or word) boundary, an implicit FILLER is generated. This unused filler is included in the size of any group item or items to which the elementary item belongs.

Whenever a SYNCHRONIZED item is referred to in the source program, the original size of the item, as shown in the PICTURE clause, the USAGE clause, and the SIGN clause, is used in determining any action that depends on size, such as justification, truncation, or overflow.

If the data description of an item contains an operational sign and any form of the SYNCHRONIZED clause, the sign of the item appears in the sign position explicitly or implicitly specified by the SIGN clause.

When the SYNCHRONIZED clause is specified in a data description entry of a data item that also contains an OCCURS clause, or in a data description entry of a data item that is subordinate to a data description entry that contains an OCCURS clause, then the data description entry is affected as follows:

- Each occurrence of the data item is SYNCHRONIZED.
- Any implicit FILLER generated for other data items within that same table is generated for each occurrence of those data items.

BINARY, REAL, and DOUBLE data items that are subordinate to a data description entry containing an OCCURS clause are not SYNCHRONIZED.

TYPE Clause

The TYPE clause provides automatic date and time editing based on the CONVENTION and LANGUAGE options specified. The TYPE clause can be used only for internationalization purposes. The desired format for the five date and time data items can be obtained via the ACCEPT or MOVE statements.

Data items can be declared as one of the following date or time types:

Type	Example
SHORT-DATE	Fri, Aug 31, 1998
LONG-DATE	Friday, August 31, 1998
NUMERIC-DATE	08/31/98
NUMERIC-TIME	13:37:20
LONG-TIME	14:37:20.0000

Data items can also be declared with an associated LANGUAGE or CONVENTION option.

Each convention defined by Unisys has a specified format for the five date and time data items. The program formats an item that is declared to be one of the five date and time types according to the predefined format of the specified convention. For the SHORT-DATE, LONG-DATE, and LONG-TIME options, the specified language is also used in formatting the output. If the convention or language is not specified, the system determines the language and convention to be used based on system-defined hierarchy.

The only clauses that can be used with the TYPE clause are the PICTURE clause and the USAGE clause. If the PICTURE clause is specified, the TYPE clause can designate only PICTURE X or PICTURE N. If the USAGE clause is specified, the TYPE clause can designate only USAGE IS DISPLAY or USAGE IS NATIONAL. If the date or time items are edited in the PICTURE clause, the TYPE clause overrides the edit and the compiler issues a warning message.

The total length of the data item must be greater than or equal to the length required by the format of the specified conventions. If the length of a data item is shorter than the required length, the compiler issues a truncation warning message.

Example

The following example shows TYPE clause coding. The NUM-DATE-ITEM is declared as a NUMERIC-DATE type and it is formatted by using the ASERIESNATIVE convention. The NUM-DATE-ITEM language is determined by the system hierarchy. The LONG-DATE-ITEM data is formatted according to the convention and language determined by the system hierarchy. The LONG-TIME-ITEM is declared as the LONG-TIME type and is formatted using the UNITEDKINGDOM1 convention and the ENGLISH language.

```
01 NUM-DATE-ITEM PIC X(8) TYPE IS NUMERIC-DATE
                        USING CONVENTION OF "ASERIESNATIVE".
01 LONG-DATE-ITEM  PIC X(20) TYPE IS LONG-DATE.
01 LONG-TIME-ITEM  PIC X(20) TYPE IS LONG-TIME
                           USING CONVENTION OF "UNITEDKINGDOM1"
                                 LANGUAGE OF "ENGLISH".
```

USAGE Clause

This clause specifies the manner in which a data item is represented in the storage of a computer. The USAGE clause does not affect the use of the data item, although certain statements in the Procedure Division might restrict the USAGE clause to certain operands. For example, the PROCESS statement requires a data item to be declared with the USAGE IS TASK clause. The USAGE clause can affect the type of character representation of the item.

This clause can be written in any data description entry except those defined with a level-number of 66 or 88.

If this clause is written in the data description entry for a group item, it can also be written in the data description entry for any subordinate elementary item or group item, but the same usage must be specified in both entries. Note that if the USAGE clause is written at a group level, it applies to each elementary item in the group.

An elementary data item (or an elementary data item subordinate to a group item) whose declaration contains a USAGE clause that specifies BINARY, COMPUTATIONAL, or PACKED-DECIMAL must be declared with a PICTURE character-string that describes a numeric item (that is, a PICTURE character-string that contains only the symbols *P*, *S*, *V*, and *9*. Refer to "PICTURE Clause" in this section.)

An elementary data item declaration that contains a USAGE clause that specifies BIT is specified only with a PICTURE character-string that describes a Boolean data item.

USAGE IS BINARY

This form of the USAGE clause indicates that the data is in a binary-coded format. A BINARY item is capable of representing a value to be used in computations and therefore is always numeric. A long numeric data item cannot have a usage of BINARY.

BINARY items occupy memory as follows:

- When the declared size is less than or equal to 11 decimal digits, the actual size is equal to one computer word (the equivalent of 6 DISPLAY digits or 12 COMPUTATIONAL digits). Note that the item is not necessarily aligned on a word boundary.
- When the declared size is greater than 11 digits, the actual size is equal to two computer words (the equivalent of 12 DISPLAY digits); however, the item is not necessarily aligned on a word boundary.
- The actual size is used for determining the size of a record and for testing for size error conditions.

Although BINARY items are not required to start at a word boundary, faster execution results when they do start at a word boundary.

USAGE IS BINARY TRUNCATED

USAGE IS BINARY TRUNCATED is synonymous with USAGE BINARY. This syntax is provided for compatibility with COBOL74.

USAGE IS BINARY EXTENDED

USAGE IS BINARY EXTENDED is similar to COBOL74 USAGE BINARY and is provided for compatibility with that language.

The value stored in a USAGE BINARY EXTENDED data item is maintained internally as an integer; if the associated PICTURE clause contains an explicit decimal point, the compiler takes this into account in any operations, as for USAGE BINARY.

High-order digit truncation of that internal integer value is limited to ensuring that its magnitude does not exceed the internal representation of a single-precision or double-precision integer on the underlying architecture.

If the PICTURE clause of the data item specifies a digit length from 1 to 11 digits inclusive, the maximum internal magnitude that can be stored in the item is 549,755,813,887. If the PICTURE clause specifies from 12 to 23 digits inclusive, the magnitude of the internal value stored in the item must not exceed 302,231,454,903,657,293,676,543.

When arithmetic statements with ON SIZE ERROR clauses produce internal results that exceed these values, the ON SIZE ERROR condition is set. For other statements, INTEGER OVERFLOW program terminations prevent the data corruption that would otherwise result (and that could occur with COBOL74 USAGE BINARY under similar circumstances).

USAGE IS BIT

The USAGE BIT clause specifies that Boolean data items be represented as bits.

The following criteria are used to determine the alignment of an elementary data item described with USAGE BIT:

- When an ALIGNED clause is specified:

  Alignment of elementary bit data items occurs at the leftmost bit position of the next available byte in storage.

- When an ALIGNED clause is not specified and a SYNCHRONIZED clause is not specified:

  Alignment of an elementary bit data item within a record occurs at the next bit position in storage if that item is an elementary bit data item that immediately follows an elementary bit data item.

  Alignment of all other bit data items within a record occurs at the leftmost bit position of the next available byte.

  Alignment of elementary bit data items of level 1 or level 77 occurs at the leftmost bit position of a word.

- When an ALIGNED clause is not specified and a SYNCHRONIZED clause is not specified:

  SYNCHRONIZED LEFT specifies that the elementary bit data item begin at the leftmost bit of the next available word in which the elementary item is placed. An implicit elementary filler bit data item with the unused bits of the word is generated after the bit data item.

  SYNCHRONIZED RIGHT specifies that the elementary bit data item terminate at the rightmost bit of the next available word in which the elementary item is placed. An implicit elementary filler bit data item with the unused bits of the word is generated before the bit data item.

  If you specify SYNCHRONIZED without LEFT or RIGHT, elementary bit data items are treated as if you specified SYNCHRONIZED LEFT.

USAGE IS COMPUTATIONAL and USAGE IS COMP

A COMPUTATIONAL item can represent a value to be used in computations and must be numeric. The system interprets COMPUTATIONAL fields as packed-decimal numeric items rather than hexadecimal strings. Thus, if nonnumeric values are assigned to a COMPUTATIONAL item, the content of the COMPUTATIONAL item is undefined.

A numeric literal can be described as a COMPUTATIONAL item. Valid characters for a numeric literal are the numbers 0 through 9, the plus sign (+), the minus sign (-), and the decimal point. The hexadecimal digits A through F are not valid in a numeric literal.

If a group item is described as COMPUTATIONAL, the elementary items in the group are COMPUTATIONAL, but the group item itself is not COMPUTATIONAL (that is, it cannot be used in computations).

Elementary COMPUTATIONAL data items are represented internally as contiguous 4-bit digits.

A long numeric data item with a usage of COMPUTATIONAL must contain an even number of digits.

The keywords COMP and COMPUTATIONAL are equivalent.

USAGE IS COMPUTATIONAL-5 and USAGE IS COMP-5

This form of the USAGE clause indicates that the data behaves as a binary item with the SYNCHRONIZED clause specified. (The SYNCHRONIZED clause specifies that a binary item is to be aligned on a word boundary. For more information, see "SYNCHRONIZED Clause" earlier in this section.)

A COBOL85 data item declared as COMP-5 PIC S9(4) maps to a C short integer.

A COBOL85 data item declared as COMP-5 PIC S9(9) maps to a C long integer.

A long numeric data item cannot have a USAGE of COMP-5.

COMP-5 is a valid abbreviation for COMPUTATIONAL-5.

USAGE IS CONTROL-POINT

This clause is an obsolete synonym for the USAGE IS TASK clause.

USAGE IS DISPLAY

This form of the USAGE clause, whether specified explicitly or implicitly, indicates that a standard data format is used to represent a data item in the storage of the computer, and that the data item is aligned on a character boundary.

If the USAGE clause is not specified for an elementary item, or for any group to which the item belongs, the usage is implicitly DISPLAY.

DISPLAY data items are represented internally as contiguous 8-bit characters represented in the EBCDIC character set.

Every occurrence of a DISPLAY data item begins and ends on a byte boundary. In a record description, the declaration of a DISPLAY data item immediately following a COMPUTATIONAL or INDEX data item that does not end on a byte boundary causes automatic generation of a 4-bit filler between the two items. This filler area between the two data items is not included in the size of either item but is included in the size of all group items to which the two items are subordinate. Similarly, if the last item declared in a group item at the next-lowest hierarchic level is a COMPUTATIONAL or INDEX data item that does not end on a byte boundary, automatic generation of a 4-bit filler occurs. This filler is included in the size of a group item.

USAGE IS DOUBLE

This form of the usage clause specifies that a data item is a double-precision real number. All real numbers are represented internally in floating-point format.

A DOUBLE data item can represent a value that can be used in computations, and is always numeric. The actual size of a DOUBLE data item is equal to two computer words. DOUBLE data items are not necessarily word-aligned. Whether the data item is aligned on a word boundary is dependent on the context in which it is declared.

When a DOUBLE data item represents a value that the machine must approximate, and it is assigned to a DISPLAY, COMP, or BINARY data item, then precision might be lost.

Example

If A is declared as DOUBLE, and B is declared as PIC 9V99, the following statements yield the value of 1.79 for B, because B has the approximate value of 1.7999999999883584678:

```
MOVE 1.8 TO A
COMPUTE B = A
```

Although DOUBLE data items are not required to start at a word boundary, faster execution results when they do start at a word boundary.

USAGE IS EVENT

This clause specifies that the data item is an event item, which is used to provide synchronization and common interlocks between two or more tasks. An event item occupies two words of memory.

An event item has two states associated with it: available and happened.

The available state has two values: not available and available. The not available value is used to temporarily restrict access to a particular object so that only one process can access the object during a given period of time. The available value permits access to the object.

The happened state also has two values: not happened and happened. This state is used to allow one or more processes to wait without using any processor time while they wait.

You can specify the USAGE IS EVENT clause for a 77-level data item or a 01-level or subordinate data item.

If you specify the USAGE IS EVENT clause for a group item, the elementary items in the group are considered to be event items. The group itself is not an event item. You cannot used the group in any construct except the USING PHRASE of a CALL (Format 6), PROCESS, or RUN statement.

Event items cannot be doubly subscripted. This means that an event item with an OCCURS clause cannot have a subordinate event item with an OCCURS clause.

Event items cannot be redefined by items of any other usage.

You cannot use any other clauses with an item whose usage is EVENT.

USAGE IS INDEX

This form of the usage clause specifies that a data item is an index data item and contains a value that must correspond to an occurrence number of a table element.

If a group item is described with the USAGE IS INDEX clause, the elementary items in the group are all index data items. The group itself is not an index data item and cannot be used in the SEARCH or SET statement or in a relation condition.

A group item is also considered to be a group data item if its class is numeric, if its USAGE IS INDEX, and if it can be referred to at any place in the syntax that is acceptable for such an item. The size of the group item is considered in terms of DISPLAY characters (four characters for each subordinate index data item).

An index data item can contain a signed value. An index data item occupies the same space and has the same alignment as an item declared PICTURE S9(7) USAGE IS COMPUTATIONAL.

An elementary data item described with a USAGE IS INDEX clause must not be a conditional variable.

An index data item can be referred to explicitly only in a SEARCH or SET statement, a relation condition, the USING phrase of a Procedure Division header, or the USING phrase of a CALL statement.

When a MOVE statement or an input-output statement that refers to a group item that contains an index data item is executed, no conversion of the index data item takes place.

The BLANK WHEN ZERO, JUSTIFIED, PICTURE, SYNCHRONIZED, and VALUE clauses must not be specified for data items whose usage is INDEX.

USAGE IS LOCK

This clause specifies that the data item is a lock item, which is used to provide synchronization and common interlocks between two or more tasks. A lock item occupies two words of memory. A lock item has two states: not available and available.

The not available value is used to temporarily restrict access to a particular object so that only one process can access the object during a given period of time. The available value permits access to the object.

You can specify the USAGE IS LOCK clause for a 77-level data item or a 01-level or subordinate data item.

If you specify the USAGE IS LOCK clause for a group item, the elementary items in the group are considered to be lock items. The group itself is not a lock item. You cannot use the group in any construct except the USING PHRASE of a CALL (Format 6), PROCESS, or RUN statement.

Lock items cannot be doubly subscripted. This means that a lock item with an OCCURS clause cannot have a subordinate lock item with an OCCURS clause.

Lock items cannot be redefined by items of any other usage.

You cannot use any other clauses with an item whose usage is LOCK.

USAGE IS KANJI (Obsolete)

KANJI is a synonym for NATIONAL. Any data item that uses the USAGE IS KANJI clause must have the letter X in its PICTURE clause. The KANJI synonym might become obsolete in a future release of COBOL85; thus, NATIONAL is the preferred usage.

USAGE IS NATIONAL

This form of the USAGE clause, whether specified explicitly or implicitly, indicates that a national standard data format is being used to represent a data item. National data items are represented internally as contiguous 16-bit characters in the national character set. If the CCSVERSION clause is specified with options other than NATIVE or CCSVERSION "ASERIESNATIVE", national data items are represented internally as contiguous 8-bit characters in the national character set.

To have a usage of national, the data item must have an *N* or the characters *N*, *B*, *0*, and slant (/) in its PICTURE character-string.

Data items with a SIGN clause or a BLANK WHEN ZERO clause cannot be declared with a USAGE IS NATIONAL clause.

Every occurrence of a national data item begins and ends on a byte boundary. In a record description, if a national data item immediately follows a computational or an index data item that does not end on a byte boundary, the compiler automatically generates a 4-bit filler between the national item and the other item. This filler area between the two data items is not included in the size of either item, but is included in the size of all group items to which the two items are subordinate.

If the last item declared in a group item at the next-lowest hierarchic level is a computational or index data item that does not end on a byte boundary, automatic generation of a 4-bit filler occurs. This filler is included in the size of a group item.

USAGE IS PACKED-DECIMAL

This clause is the same as the USAGE IS COMPUTATIONAL/USAGE IS COMP clause.

USAGE IS REAL

This form of the usage clause specifies that a data item is a single-precision real number. All real numbers are represented internally in floating-point format.

A REAL data item can represent a value that can be used in computations and is always numeric. A long numeric cannot have a usage of REAL. The actual size of a REAL data item is equal to one computer word. A REAL data item can be used to store any item that is documented as being equivalent to a REAL data item without altering the bit pattern. REAL data items are not necessarily word-aligned. Whether the data item is aligned on a word boundary is dependent on the context in which it is declared. Although REAL data items are not required to start at a word boundary, faster execution results when they do start at a word boundary.

When a REAL data item represents a value that the machine must approximate, and it is assigned to a DISPLAY, COMP, or BINARY data item, then precision might be lost.

Example

If A is declared as REAL, and B is declared as PIC 9V999, the following statements yield the value of 1.119 for B, because A has the approximate value of 1.1199999999953433871:

```
MOVE 1.12 TO A
COMPUTE B = A
```

USAGE IS TASK

This form of the USAGE clause enables you to define a data item as a task variable. You can define a 77-level or a 01-level or subordinate data item as a task variable.

If you describe a group item with the USAGE IS TASK clause, all the elementary items in the group are considered to be task variables. The group itself is not a task variable. You can use the group item only as a parameter in the USING phrase of the CALL (Format 6), PROCESS, and RUN statements. For the syntax of these statements, refer to Section 7 and Section 8.

Note that a task variable with an OCCURS clause cannot have a subordinate task variable with an OCCURS clause.

Task variables cannot be redefined by variables of any other usage. No other clauses are allowed for a data item when the USAGE IS TASK clause is declared.

For details about task variables, refer to Section 13.

VALUE Clause

In this format, the VALUE clause defines the initial value of Working-Storage Section data items.

The following rules apply to the literals specified in a VALUE clause of an item:

If the item is . . .	Then the literal must . . .
Numeric	Have a value in the range of values indicated by the PICTURE clause, and must not have a value that would require truncation of nonzero digits
Signed Numeric	Have a signed numeric PICTURE character-string associated with it
Nonnumeric	Not exceed the size indicated by the PICTURE clause
Long numeric	• Be ZERO or 0 • Be a numeric literal of the same size in digits as the data item • Be an undigit literal of the same size in bytes as the item

The VALUE clause must not conflict with other clauses in the data description of the item or in the data description in the hierarchy of the item.

The following rules apply:

If the category of the item is . . .	Then all literals in the VALUE clause must be . . .
Numeric	Numeric. If a literal defines the value of a WORKING-STORAGE item, that literal is aligned in the data item according to the standard alignment rules (refer to "Standard Alignment Rules" in this section).
Numeric-edited	Numeric or nonnumeric. If the literal is a nonnumeric literal, it is aligned in the data item as if the data item had been described as alphanumeric. If the literal is a numeric literal, it is aligned on the data item according to the standard alignment rules for numeric literals. Refer to "Standard Alignment Rules" in this section. Editing characters in the PICTURE clause are included when determining the size of the data item but have no effect on initialization of the data item (refer to "PICTURE Clause" in this section). Therefore, the VALUE clause for an edited item must be specified in edited form.

If the category of the item is . . .	Then all literals in the VALUE clause must be . . .
Alphabetic Alphanumeric Alphanumeric-edited	Nonnumeric. The literal is aligned in the data item as if the data item had been described as alphanumeric (refer to "Standard Alignment Rules" in this section). Editing characters in the PICTURE clause are included in determining the size of the data item but have no effect on initialization of the data item (refer to "PICTURE Clause" in this section). Therefore, the VALUE clause for an edited item must be specified in edited form.
National National-edited	National. The literal is aligned in the data item as if the data item had been described as national (refer to "Standard Data Alignment Rules" in this section). Editing characters in the PICTURE clause are included in determining the size of the data item (refer to "PICTURE Clause" in this section). Therefore, the VALUE clause for an edited item must be specified in edited form.
Boolean	Boolean

Note that initialization is not affected by any BLANK WHEN ZERO or JUSTIFIED clause that might be specified.

Rules that govern the use of the VALUE clause differ depending on the section of the Data Division in which the VALUE clause occurs.

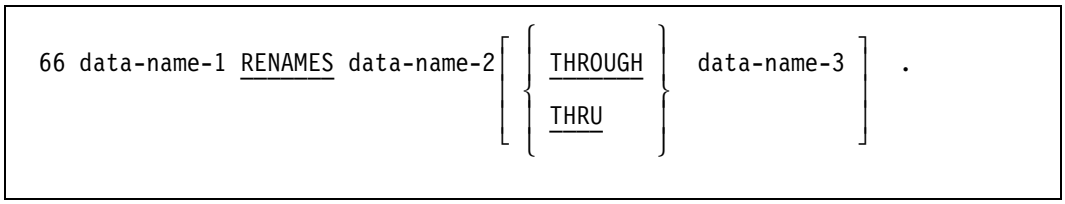
For Data Description Entry Format 1, the rules in the following table apply:

In . . .	The VALUE clause . . .
The File Section	Cannot be used.
The Linkage Section	Cannot be used, except for the data items which are not used as formal parameters.
The Local-Storage Section	Cannot be used.
The Working-Storage Section	Takes effect only when the program is placed into its initial state. If the VALUE clause is used in the description of the data item, the data item is initialized to the defined value. If the VALUE clause is not associated with a data item, the initial value of that data item is undefined. If the data item is defined as a formal parameter, then the VALUE clause will be ignored.

In . . .	The VALUE clause . . .
A data description entry that contains a REDEFINES clause, or an entry that is subordinate to an entry that contains a REDEFINES clause	Cannot be used.
A data description entry that is part of the description or redefinition of an external data record	Cannot be used.
A data description entry that contains an OCCURS clause, or an entry that is subordinate to an OCCURS clause	Causes every occurrence of the associated data item to be assigned the specified value.
An entry at the group level	Must contain a figurative constant or a nonnumeric literal, and the group area is initialized without consideration for the individual elementary or group items contained within this group. The VALUE clause cannot be stated at the subordinate levels within this group.
A group item containing subordinate items with descriptions include JUSTIFIED, SYNCHRONIZED, or USAGE (other than USAGE IS DISPLAY)	Cannot be used.

Data Description Entry Format 2: Level-66 RENAMES Entry

This format renames a data-name or range of data-names.



66

Level-number 66 identifies RENAMES entries. Level-number 66 is used only in this format of a data description entry.

A level-66 entry cannot rename another level-66 entry, nor can it rename a level-number 77, 88, or 01 entry.

data-name-1

This name is a user-defined word. It cannot be used as a qualifier and can be qualified only by the names of the associated level-01, FD, or SD entries.

When data-name-3 is specified, data-name-1 is a group item that includes all elementary items starting with data-name-2 (if data-name-2 is an elementary item) or with the first elementary item in data-name-2 (if data-name-2 is a group item), and concluding with data-name-3 (if data-name-3 is an elementary item) or with the last elementary item in data-name-3 (if data-name-3 is a group item).

When data-name-3 is not specified, data-name-1 assumes all characteristics of data-name-2 as determined from the data description of data-name-2, including usage, justification, synchronization, and editing requirements.

RENAMES Clause

This clause allows alternative, possibly overlapping, groupings of elementary items.

Any number of RENAMES entries can be written for a logical record.

All RENAMES entries that refer to data items in a given logical record must immediately follow the last data description entry of the associated record description entry.

data-name-2

This name is a user-defined word and must be the name of an elementary item or a group of elementary items in the same logical record.

This name cannot have an OCCURS clause in its data description entry and cannot be subordinate to an item that has an OCCURS clause in its data description entry.

This name cannot be the same name as data-name-3.

This name can be qualified.

THROUGH
THRU

These keywords are equivalent.

data-name-3

This name is a user-defined word and must be the name of an elementary item or a group of elementary items in the same logical record.

This name cannot have an OCCURS clause in its data description entry and cannot be subordinate to an item that has an OCCURS clause in its data description entry.

This name cannot be the same name as data-name-2.

This name can be qualified.

The beginning of the area described by this data-name must not be to the left of the beginning of the area described by data-name-2. Also, the end of the area described by this data-name must be to the right of the end of the area described by data-name-2. Therefore, data-name-3 cannot be subordinate to data-name-2.

Details

None of the items in the range of data-name-2 through data-name-3 (including data-name-2 and data-name-3) can be variable-occurrence data items.

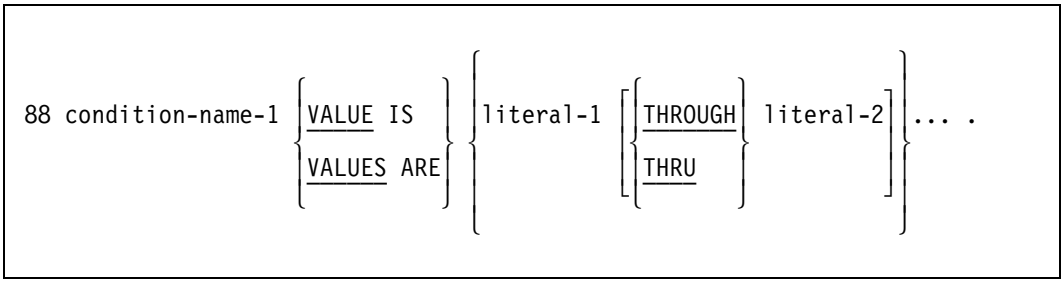
Example

```
03  NAME-PARTS.
    05  LAST1            PIC X(15).
    05  FIRST1           PIC X(15).
    05  MID              PIC X(10).
66  PARTIAL-NAME  RENAMES  LAST1 THROUGH FIRST1.
```

The RENAMES entry associates the user-defined name PARTIAL-NAME with the data descriptions for the elementary items LAST1 and FIRST1 of the group item NAME-PARTS.

Data Description Entry Format 3: Level-88 Condition-Name Entry

This format contains the name of the condition and the value, values, or range of values associated with the condition-name. This format is used for each condition-name.



88

Level-number 88 identifies entries that define condition-names associated with a conditional variable. Level-number 88 is used only in this format of a data description entry.

Note that each condition-name requires a separate entry with level-number 88.

condition-name-1

This name is a user-defined word. The Condition-name entries for a particular conditional variable must immediately follow the entry describing the item with which the condition-name is associated.

A condition-name can be associated with any data description entry that contains a level-number except the following:

- Another condition-name
- A level-66 item
- A group containing items with descriptions including JUSTIFIED, SYNCHRONIZED, or USAGE (other than USAGE IS DISPLAY)
- An index data item

VALUE
VALUES

This clause is explained in the following subsection under the heading "VALUE Clause."

literal-1 THROUGH literal-2

Whenever the THROUGH (THRU) phrase is used, literal-1 must be less than literal-2.

THROUGH
THRU

These keywords are equivalent.

VALUE Clause

In this format, the VALUE clause defines the values associated with condition-names.

The VALUE clause is required in a condition-name entry. The VALUE clause and the condition-name itself are the only two clauses permitted in the entry. The characteristics of a condition-name are implicitly those of its conditional variable.

The following rules apply to the literals specified in a VALUE clause of an item:

If the item is . . .	Then the literal must . . .
Numeric	Have a value in the range of values indicated by the PICTURE clause and must not have a value that would require truncation of nonzero digits
Signed numeric	Have a signed numeric PICTURE character-string associated with it
Nonnumeric	Not exceed the size indicated by the PICTURE clause

The VALUE clause must not conflict with other clauses in the data description of the item or in the data description within the hierarchy of the item. The following rules apply:

If the category of the item is . . .	Then all literals in the VALUE clause must be . . .
Numeric	Numeric. If the literal defines the value of a WORKING-STORAGE item, the literal is aligned in the data item according to the standard alignment rules (refer to "Standard Alignment Rules" in this section).
Numeric-edited	Numeric or nonnumeric.

If the category of the item is . . .	Then all literals in the VALUE clause must be . . .
Alphabetic Alphanumeric Alphanumeric-edited	Nonnumeric. The literal is aligned in the data item as if the data item had been described as alphanumeric (refer to "Standard Alignment Rules" in this section). Editing characters in the PICTURE clause are included in determining the size of the data item but have no effect on initialization of the data item (refer to "PICTURE Clause" in this section). Therefore, the VALUE clause for an edited item must be specified in edited form.
National National-edited	National. The literal is aligned in the data item as if the data item had been described as national (refer to "Standard Data Alignment Rules" in this section). Editing characters in the PICTURE clause are included in determining the size of the data item (refer to "PICTURE Clause" in this section). Therefore, the VALUE clause for an edited item must be specified in edited form.
Boolean	Boolean

Note that initialization is not affected by any BLANK WHEN ZERO or JUSTIFIED clause that might be specified.

Rules that govern the use of the VALUE clause differ depending on the section of the Data Division in which the VALUE clause occurs. In Data Description Entry Format 3, the rules in the following table apply:

In . . .	The VALUE clause . . .
The File Section	Can be used only in condition-name entries. Therefore, the initial value of the data items in the File Section is undefined.
The Linkage Section	Can be used only in condition-name entries (level 88).

In . . .	The VALUE clause . . .
The Working-Storage Section	Must be used in condition-name entries. VALUE clauses in the Working-Storage Section of a program take effect only when the program is placed into its initial state. If the VALUE clause is used in the description of the data item, the data item is initialized to the defined value. If the VALUE clause is not associated with a data item, the initial value of that data item is undefined.
A data description entry that contains a REDEFINES clause, or an entry that is subordinate to an entry that contains a REDEFINES clause	Can be used.
A data description entry that includes, or is subordinate to, an entry that includes the EXTERNAL clause	Can be used.
A data description entry that is part of the description or redefinition of an external data record	Can be used.
A data description entry that contains an OCCURS clause, or an entry that is subordinate to an OCCURS clause	Causes every occurrence of the associated data item to be assigned the specified value. (The OCCURS clause is described earlier in this section.)
An entry at the group level	Must contain a figurative constant or a nonnumeric literal, and the group area is initialized without consideration for the individual elementary or group items contained within this group. The VALUE clause cannot be stated at the subordinate levels within this group.
A data item referred to by a DEPENDING ON phrase	Can be used. The value is considered to be placed in the data item after the variable occurrence data item is initialized (refer to "OCCURS Clause" in this section).
A group item containing items subordinate to it with descriptions including JUSTIFIED, SYNCHRONIZED, or USAGE (other than USAGE IS DISPLAY)	Cannot be used.

Examples

```
01  MONTH      PIC 99.
    88  QI     VALUES ARE 01 02 03.
    88  QII    VALUES ARE 04 05 06.
    88  QIII   VALUES ARE 07 08 09.
    88  QIV    VALUES ARE 10 11 12.
```

These condition-name entries associate values with the conditions QI, QII, QIII, and QIV.

```
02  MONTH      PIC 99.
    88  MONTHS-WITH-31-DAYS
               VALUES ARE 01, 03, 05, 07
                          08, 10, 12.
```

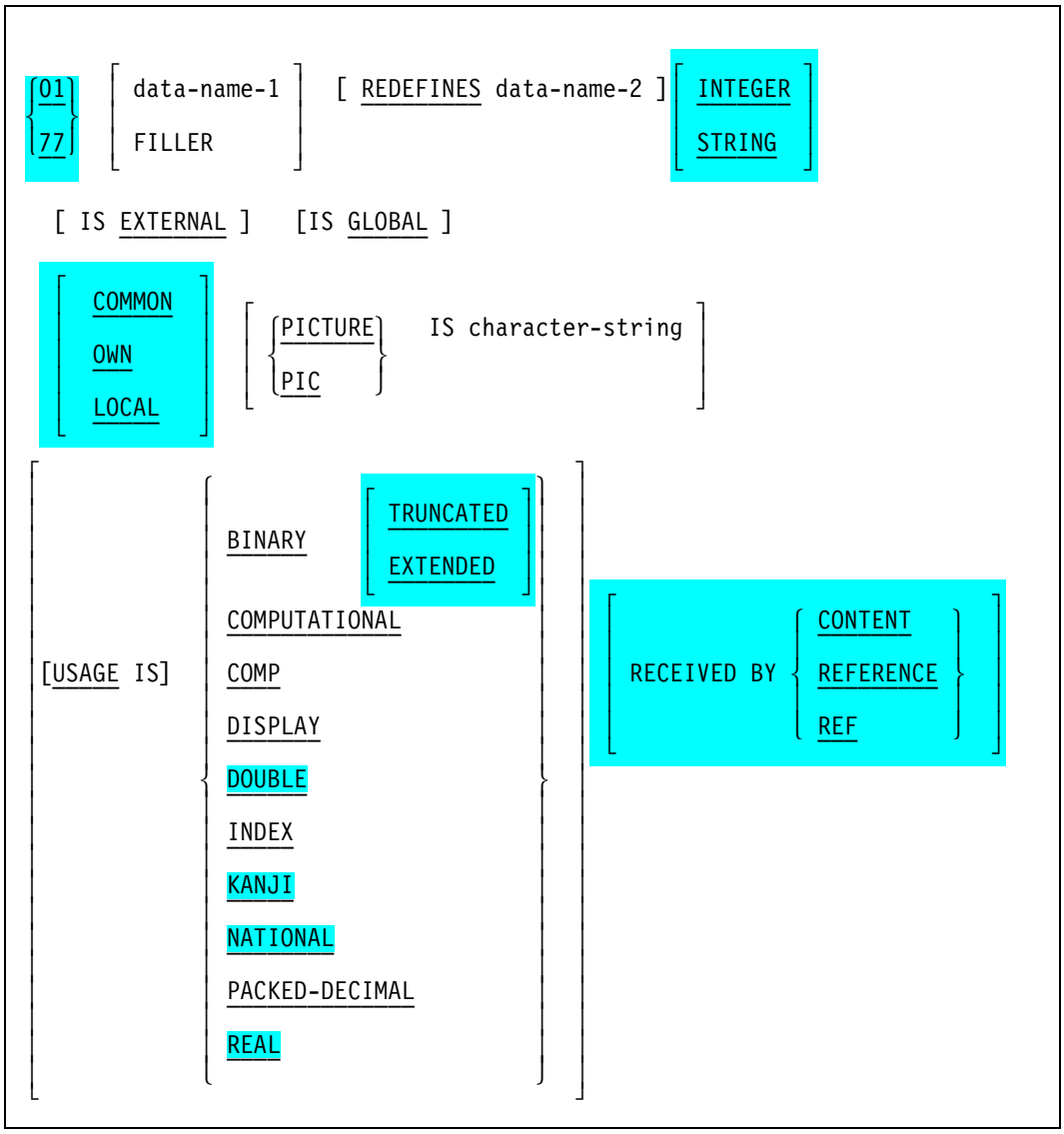
This condition-name entry associates values with the condition MONTHS-WITH-31-DAYS.

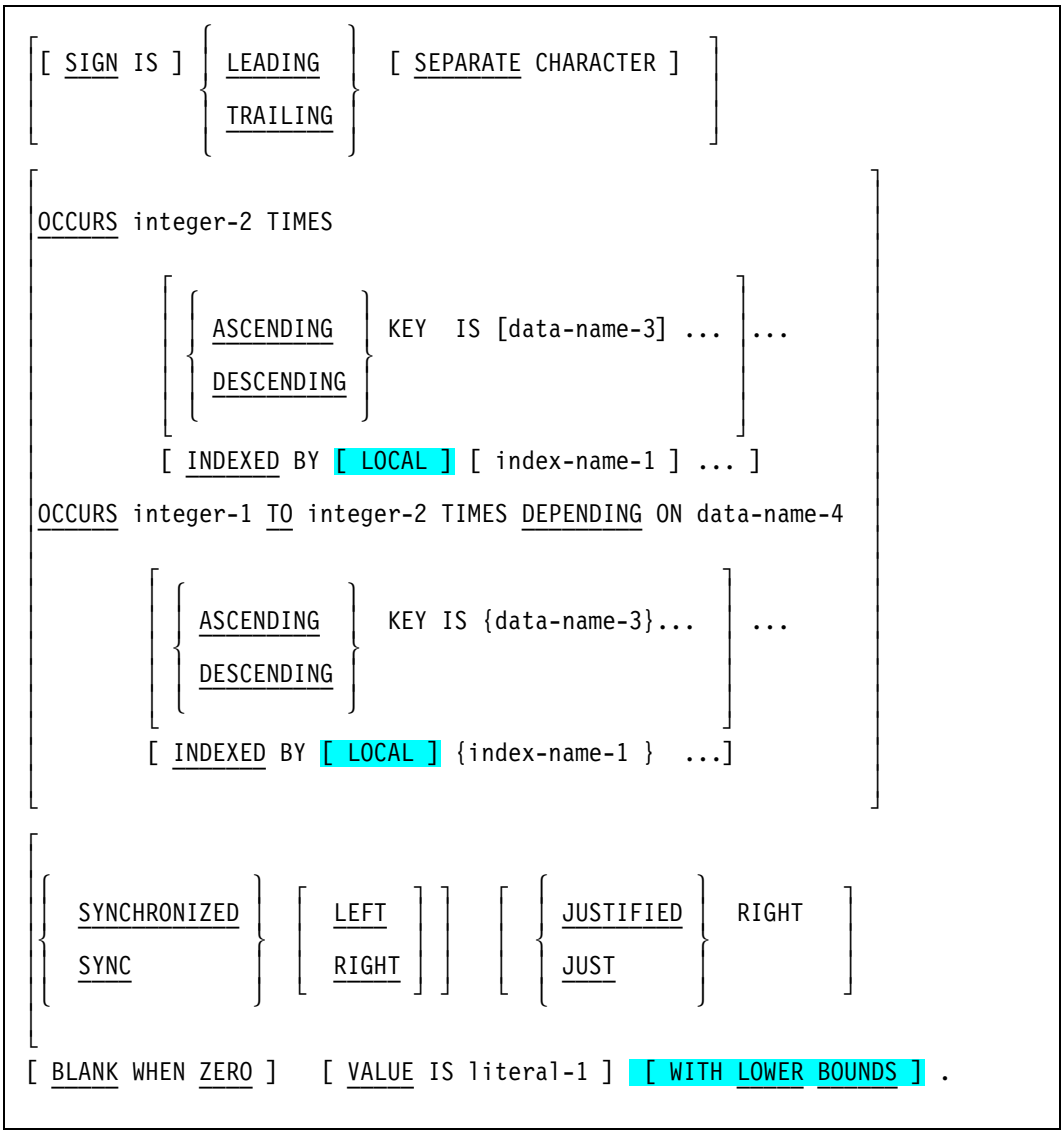
```
01 ITEM-1     PIC ZZ99  VALUE 1.
01 ITEM-2     PIC ZZ99  VALUE "  01".
```

In this example, the VALUE clauses define the same initial value for both ITEM-1 and ITEM-2.

Data Description Entry Format 4: IPC

In interprogram communication (IPC), a level-01 or level-77 data description entry in the Working-Storage Section or a level-01 data description entry in the File Section determines whether the data record and its subordinate data have local names or global names.





Refer to "Data Description Entry Format 1" for information on the BLANK WHEN ZERO, JUSTIFIED, LOCAL, LOWER BOUNDS, OCCURS, PICTURE, RECORD AREA, RECEIVED BY, SIGN, STRING, SYNCHRONIZED, and USAGE clauses.

Data-Name or FILLER Clause

Refer to "Data Description Entry Format 1" for information on this clause.

In Format 4, data-name-1 must be specified for any entry that contains the GLOBAL or EXTERNAL clause, or for record descriptions associated with a file description entry that contains the EXTERNAL or GLOBAL clause.

COMMON Clause

Refer to "Data Description Entry Format 1" for information on this clause.

The COMMON clause cannot be specified in the same data description entry as the EXTERNAL clause.

In addition, in Format 4, the COMMON clause can occur only at the outermost level of a group of nested programs.

Also, the compiler option COMMON does not apply to data-items declared in the Working-Storage Section of nested programs.

EXTERNAL Clause

The EXTERNAL clause specifies that a data item is external. The data items and group data items of an external data record are available to every program in the run unit that describes that record. This clause can be specified only in 01-level data description entries in the Working-Storage Section that are described as USAGE IS DISPLAY.

Rules

Observe the following guidelines when using the EXTERNAL clause:

- The EXTERNAL clause cannot be specified in a data description entry with the REDEFINES, COMMON, or OWN clause.
- Within a program, a data-name specified as the subject of a level-01 data description entry that includes the EXTERNAL clause cannot be specified for any other data description entry that includes the EXTERNAL clause.
- If two or more programs in a run unit describe the same external data record, the same record-name must appear in a record description entry in each program and the records must define the same number of standard data format characters.
- If a program contains a data description entry that includes the REDEFINES clause, which redefines the complete external record, this complete redefinition need not occur identically in other programs in the run unit (refer to "REDEFINES Clause" in this section).

Note that use of the EXTERNAL clause does not imply that the associated data-name is a global name. (Refer to "GLOBAL Clause" in this section.) For information on the EXTERNAL clause, refer to "File Description Entry Format 4: IPC and Sequential I–O."

GLOBAL Clause

Refer to "File Description Entry Format 4: IPC and Sequential I-O" for a complete description of this clause.

Rules

The following conditions apply to the GLOBAL clause when used in Data Description Entry Format 4:

- This clause can be specified only in data description entries whose level-number is 01 or 77.
- This clause specifies that a data-name is a global name. A global name is available to every program contained within the program that declares it.
- A data-name described using a GLOBAL clause is a global name. All data-names subordinate to a global name are global names. All condition-names associated with a global name are global names.
- In the same Data Division, the data description entries for any two data items for which the same data-name is specified must not include this clause.
- A statement in a program that is contained directly or indirectly in a program that describes a global name can reference that name without describing it again (refer to "Scope of Names" in Section 10).
- If this clause is used in a data description entry that contains the REDEFINES clause, only the subject of that REDEFINES clause possesses the global attribute.

OWN Clause

Refer to "Data Description Entry Format 1" for a complete description of this clause.

Rules

Observe the following rules when you use the OWN clause:

- The OWN clause can occur at any level of nested programs.
- The OWN clause cannot be specified in the same data description entry as the EXTERNAL clause.
- The compiler option OWN applies to all data-items declared in the Working-Storage Section of nested programs.

REDEFINES Clause

Refer to "Data Description Entry Format 1" for a complete description of this clause.

In Format 4, this clause and the EXTERNAL clause must not be specified in the same data description entry.

VALUE Clause

Refer to "Data Description Entry Format 1" for a complete description of this clause.

In Format 4, the VALUE clause must not be used in any data description entry that includes, or is subordinate to, an entry that includes the EXTERNAL clause. (The VALUE clause can be specified for condition-name entries associated with such data description entries.)

Data Division Header

The following header identifies and must begin the Data Division:

```
DATA DIVISION.
```

DATA DIVISION

These keywords begin in area A and must be followed by a period.

File Section

The File Section defines the structure of data files. Use of this section is optional.

Each file is defined by a file description entry and one or more record description entries. Record descriptions are written immediately following each file description entry. The format of record descriptions is described earlier in this section.

The general format of the File Section is as follows:

```
FILE SECTION.
[ file description entry  { record description entry } ... ] ...
```

FILE SECTION

These keywords begin in area A and must be followed by a period.

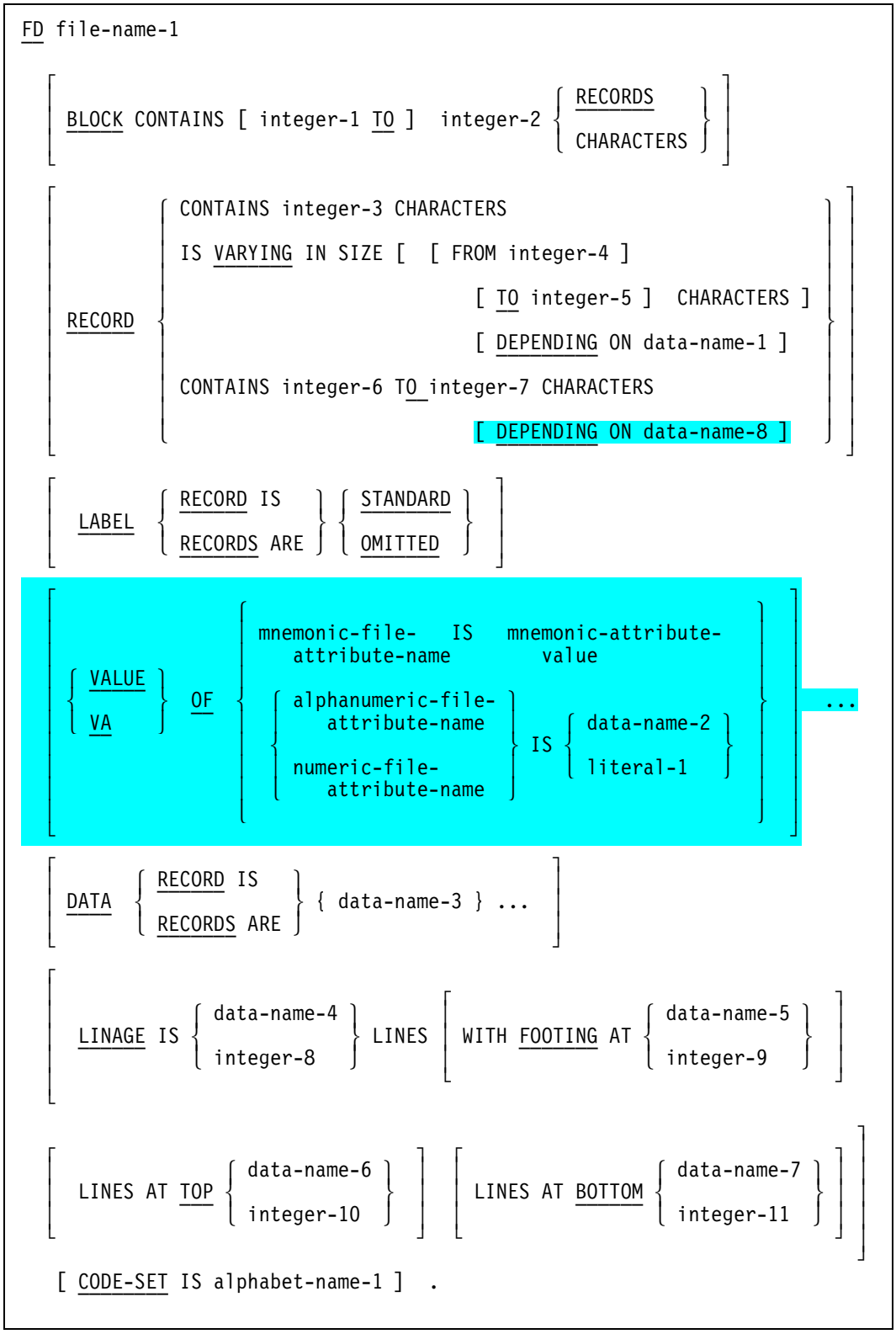
file description entry

A file description entry associates a file-name with a file connector.

Format	Use
Format 1	This format provides information on the physical structure, identification, and record-names that pertain to a sequential file.
Format 2	This format provides information on the physical structure, identification, and record-names that pertain to a relative or indexed file.
Format 3	This format provides information on the physical structure and record-names that pertain to a sort or merge file.
Format 4	This format is used for interprogram communication and sequential I-O. It determines the internal or external attributes of a file connector, of the associated data records, and of the associated data items. It also determines whether a file-name is a local name or a global name.
Format 5	This format is used for interprogram communication and relative I-O or indexed I-O. It determines the internal or external attributes of a file connector, of the associated data records, and of the associated data items. It also determines whether a file-name is a local name or a global name.

File Description Entry Format 1: Sequential I-O

This format provides information on the physical structure, identification, and record-names that pertain to a sequential file. The clauses that follow file-name-1 can appear in any order. They are described on the following pages in alphabetical order.


FD file-name-1
BLOCK CONTAINS [integer-1 TO] integer-2 { RECORDS / CHARACTERS }
RECORD { CONTAINS integer-3 CHARACTERS
IS VARYING IN SIZE [[FROM integer-4] [TO integer-5] CHARACTERS] [DEPENDING ON data-name-1]
CONTAINS integer-6 TO integer-7 CHARACTERS [DEPENDING ON data-name-8] }
LABEL { RECORD IS / RECORDS ARE } { STANDARD / OMITTED }
{ VALUE / VA } OF { mnemonic-file-attribute-name IS mnemonic-attribute-value
{ alphanumeric-file-attribute-name / numeric-file-attribute-name } IS { data-name-2 / literal-1 } } ...
DATA { RECORD IS / RECORDS ARE } { data-name-3 } ...
LINAGE IS { data-name-4 / integer-8 } LINES [WITH FOOTING AT { data-name-5 / integer-9 }]
[LINES AT TOP { data-name-6 / integer-10 }] [LINES AT BOTTOM { data-name-7 / integer-11 }]
[CODE-SET IS alphabet-name-1] .

FD

This level indicator identifies the beginning of a file description entry and must precede file-name-1.

FD refers to file description.

file-name-1

This name is a user-defined word.

The clauses that follow file-name-1 can appear in any order.

BLOCK CONTAINS Clause

The BLOCK CONTAINS clause specifies the size of a physical record. This clause is required except when one or more of the following conditions exist:

- A physical record contains only one complete logical record.
- The hardware device assigned to the file has only one physical record size.
- The number of records contained in a block is specified in the operating environment.

integer-1
integer-2

If integer-1 is not specified, integer-2 represents the exact number of RECORDS or CHARACTERS in the physical record.

If integer-1 and integer-2 are both specified, they refer to the minimum and maximum size of the physical record, respectively.

If the associated file connector is an external file connector, all BLOCK CONTAINS clauses in the run unit that are associated with that file connector must have the same values for integer-1 and integer-2.

RECORDS

The size of a physical record can be stated in terms of records unless one or more of the following conditions exists, in which case the RECORDS phrase must not be used:

- In mass-storage files, where logical records can extend across physical records.
- The physical record contains padding (area not contained in a logical record). Logical records are grouped in such a manner that an inaccurate physical record size would be implied.

When RECORDS is specified, the physical record size is considered to be integer-2 multiplied by the largest record specified for this file.

CHARACTERS

If this phrase is specified, the physical record size is specified in terms of the number of character positions required to store the physical record, regardless of the types of characters used to represent the items in the physical record.

When CHARACTERS is specified, the physical record size is considered to be integer-2 characters.

Details

If logical records of differing sizes are grouped into one physical record, the amount of data transferred from the record area to the physical record depends on the size of the record named in the WRITE or REWRITE statement. In this case, the logical records are aligned on maximum record-size boundaries. If the size of the record named does not equal the maximum record size specified for the file, the data is transferred to the physical record according to the rules specified for the MOVE statement without the CORRESPONDING phrase. The sending area is considered to be a group item.

If variable-length records are specified (refer to "RECORD Clause" in this section), then the physical record size is determined as follows:

If . . . is specified in the BLOCK CONTAINS clause	**Then the physical record size equals . . .**
Integer-2 RECORDS	Integer-2 multiplied by the maximum record size.
Integer-1 and integer-2 RECORDS	Either integer-1 multiplied by the maximum record size or integer-2 multiplied by the minimum record size, whichever is larger.
CHARACTERS	Either integer-2 or the maximum record size, whichever is larger. If the maximum record size is larger, a warning is issued. (Integer-1 is shown for documentation purposes only.)

CODE-SET Clause

This clause specifies the character code set used to represent data on the external media.

If this clause is specified, alphabet-name-1 specifies the algorithm for converting the character codes on the external media to or from EBCDIC during the execution of an input or output operation.

If this clause is not specified, the native character code set (EBCDIC) is assumed for data on the external media.

If this clause is specified for a file, all data in that file must be described as USAGE IS DISPLAY, and any signed numeric data must be described with the SIGN IS SEPARATE clause (refer to "Data Description Entry Format 1" for descriptions of the USAGE and SIGN clauses).

alphabet-name-1

This name is a user-defined word.

The alphabet-name clause referred to by the CODE-SET clause must not specify the literal phrase (refer to the "ALPHABET Clause" in Section 3).

If the CODE-SET clause is specified, upon successful execution of an OPEN statement, the character set used to represent the data on the external media is the one referred to by alphabet-name-1 in the file description entry associated with the file-name specified in the OPEN statement.

Details

If the associated file connector is an external file connector, all CODE-SET clauses in the run unit that are associated with that file connector must have the same character set.

DATA RECORDS Clause

This clause serves only as documentation for the names of data records in their associated file.

The DATA RECORDS clause is an obsolete element in COBOL ANSI-85 and will be deleted from the next revision of standard COBOL.

data-name-3

This name is a user-defined word.

This is the name of a data record that must have a level-01 record description (with the same name) associated with it.

The presence of more than one data-name indicates that the file contains more than one type of data record. These records can be different in size, format, and so forth. The order in which they are listed is not significant.

Details

Conceptually, all data records in a file share the same area, even if more than one type of data record is present in the file.

LABEL RECORDS Clause

This clause specifies the presence or absence of label information.

If this clause is not specified for a file, STANDARD is assumed.

The LABEL RECORDS clause is an obsolete element in COBOL ANSI-85 and will be deleted from the next revision of standard COBOL.

STANDARD

This specifies that labels exist for the file or the device to which the file is assigned and that the labels conform to the standard label specifications. STANDARD should be used if you wish to take advantage of the automatic file allocation and handling procedures in the operating system. (Note that disk devices maintain a directory instead of a system of labels.) The format of labels depends on the device containing the file. (Refer to the *I/O Subsystem Programming Guide* for label formats.)

OMITTED

OMITTED must be used if an input file does not have standard labels or if labels are not desired on output files.

Details

If the file connector associated with this file description entry is an external file connector (refer to the "EXTERNAL Clause" in this section, and to "File Connectors" in Section 10), all LABEL RECORDS clauses in the run unit associated with that file connector must have the same specification.

LINAGE Clause

This clause specifies the size of a logical page according to the number of lines. It also specifies the size of the top and bottom margins on the logical page, and the line number, at which the footing area begins in the page body. (The terms logical page and page body are defined under the paragraph headed "Details," which follows the description of syntax elements.)

data-name-4
integer-8

Integer-8 or the value of the data item referred to by data-name-4 specifies the number of lines that can be written and/or spaced on the logical page. The value must be greater than zero. The part of the logical page in which these lines can be written and/or spaced is called the page body.

At the time an OPEN statement with the OUTPUT phrase is executed for the file, either integer-8 or the value of the data item referred to by data-name-4, whichever is specified, is used to specify the number of lines that will make up the page body for the first logical page.

At the time a WRITE statement with the ADVANCING PAGE phrase is executed or a page overflow condition occurs, the value of the data item referred to by data-name-4, if specified, is used to define the page body for the next logical page.

FOOTING

This phrase specifies the line number in the page body at which the footing area begins.

If this phrase is not specified, the assumed value is equal to integer-8 or the contents of the data item referred to by data-name-4, whichever is specified.

data-name-5
integer-9

Integer-9 or the value of the data item referred to by data-name-5 specifies the line number in the page body at which the footing area begins. The value must be greater than zero and less than or equal to integer-8 or the value of the data item referred to by data-name-4.

Integer-9 must not be greater than integer-8.

At the time an OPEN statement with the OUTPUT phrase is executed for the file, either integer-9 or the value of the data item referred to by data-name-5, whichever is specified, is used to specify the number of lines that will make up the footing area for the first logical page.

At the time a WRITE statement with the ADVANCING PAGE phrase is executed or a page overflow condition occurs, the value of the data item referred to by data-name-5, if specified, is used to define the footing area for the next logical page.

LINES AT TOP

This phrase specifies the number of lines that make up the top margin on the logical page.

If this phrase is not specified, the value for this function is zero.

data-name-6
integer-10

Integer-10 or the value of the data item referred to by data-name-6 specifies the number of lines that make up the top margin on the logical page. This value can be zero.

At the time an OPEN statement with the OUTPUT phrase is executed for the file, either integer-10 or the value of the data item referred to by data-name-6, whichever is specified, is used to specify the number of lines that will make up the top margin for the first logical page.

At the time a WRITE statement with the ADVANCING PAGE phrase is executed or a page overflow condition occurs, the value of the data item referred to by data-name-6, if specified, is used to define the top margin for the next logical page.

LINES AT BOTTOM

This phrase specifies the number of lines that make up the bottom margin on the logical page.

If this phrase is not specified, the value for this function is zero.

data-name-7
integer-11

Integer-11 or the value of the data item referred to by data-name-7 specifies the number of lines that make up the bottom margin on the logical page. This value can be zero.

At the time an OPEN statement with the OUTPUT phrase is executed for the file, either integer-11 or the value of the data item referred to by data-name-7, whichever is specified, is used to specify the number of lines that will make up the bottom margin for the first logical page.

At the time a WRITE statement with the ADVANCING PAGE phrase is executed or a page overflow condition occurs, the value of the data item referred to by data-name-7, if specified, is used to define the bottom margin for the next logical page.

Details

The data-names used in this clause must refer to elementary unsigned numeric integer data items. All of the data-names can be qualified.

The logical page size is the sum of the values referred to by each phrase except the FOOTING phrase. Each logical page is contiguous to the next, and additional spacing is not provided.

Note that there is not necessarily a relationship between the size of the logical page and the size of a physical page.

The part of the logical page in which the lines can be written and/or spaced is called the page body.

The footing area is made up of the area of the page body between the line represented by integer-9 or the value of the data item referred to by data-name-5 and the line represented by integer-8 or the value of the data item referred to by data-name-4, inclusive.

If the file connector associated with this file description entry is an external file connector, all file description entries in the run unit that are associated with this file connector must have the following:

- A LINAGE clause, if any file description entry has a LINAGE clause
- The same corresponding values for integer-1, integer-2, integer-3, and integer-4, if specified
- The same corresponding external data items referred to by data-name-1, data-name-2, data-name-3, and data-name-4

A separate LINAGE-COUNTER register is generated for each file whose file description entry contains a LINAGE clause. Because more than one LINAGE-COUNTER can exist in a program, you must qualify LINAGE-COUNTER by file-name when necessary (refer to Format 4 under "Qualification" in this section). You can refer to a LINAGE-COUNTER only in Procedure Division statements.

The value in the LINAGE-COUNTER at any given time represents the line number at which the device is positioned in the current page body. Only the input-output control system can change the value of the LINAGE-COUNTER.

- When an OPEN statement with the OUTPUT phrase is executed for a file, the value of LINAGE-COUNTER is automatically set to 1.
- When a WRITE statement is executed, LINAGE-COUNTER is automatically modified according to the rules in the following table:

If the . . .	Then the LINAGE-COUNTER . . .
ADVANCING PAGE phrase of the WRITE statement is specified.	Is automatically reset to 1. During the resetting of the LINAGE-COUNTER to the value 1, the value of LINAGE-COUNTER is implicitly incremented to exceed the value specified by integer-1 or the data item referred to by data-name-1.
ADVANCING identifier-2 or integer-1 phrase of the WRITE statement is specified.	Is incremented by integer-1 or the value of the data item referred to by identifier-2.
ADVANCING phrase of the WRITE statement is not specified.	Is incremented by the value 1.
Device is repositioned to the first line that can be written on for each of the succeeding logical pages.	Is automatically reset to 1.

RECORD Clause

The RECORD clause specifies the number of character positions in a fixed-length record or the range of character positions in a variable-length record. If the number of character positions varies, you can specify the minimum and maximum number of character positions.

If the RECORD clause is omitted, the record-description entry completely defines the size of each record, and the file is considered to have fixed-length records. When multiple record-description entries are associated with this file, the record size for the file is that of the largest record-description entry. The other record descriptions merely represent a redefinition of the same memory area. As a result, each READ or WRITE statement for the file uses the full length of the record for data transfer.

There are three forms of the RECORD clause: the RECORD CONTAINS integer-3 CHARACTERS clause, the RECORD IS VARYING IN SIZE clause, and the RECORD CONTAINS integer-6 TO integer-7 clause.

RECORD CONTAINS integer-3 CHARACTERS

This form of the RECORD clause enables you to specify fixed-length records.

integer-3

This integer represents the exact number of character positions contained in each record of the file.

An error message is issued if the number of character positions specified by integer-3 does not match the record description entry.

RECORD IS VARYING IN SIZE

This form of the RECORD clause enables you to specify variable-length records.

integer-4

This integer specifies the minimum number of character positions that can be contained in any record of the file.

If this integer is not specified, the minimum number of character positions to be contained in any record of the file is equal to the least number of character positions described for a record in that file.

integer-5

This integer specifies the maximum number of character positions in any record of the file.

If this integer is not specified, the maximum number of character positions to be contained in any record of the file equals the greatest number of character positions described for a record in that file.

data-name-1

This name is a user-defined word. Data-name-1 can be qualified.

The number of character positions associated with a record description is determined by the sum of the number of character positions in all elementary data items excluding redefinitions and renamings, plus any implicit FILLER due to synchronization. If a table is specified,

- The minimum number of table elements described in the record is used in the summation to determine the minimum number of character positions associated with the record description.
- The maximum number of table elements described in the record is used in the summation to determine the maximum number of character positions associated with the record description.

  The contents of the data item referred to by data-name-1 and the number of character positions in the record depend upon whether data-name-1 is specified in the RECORD clause, as described in the following tables.

If data-name-1 is specified in the RECORD clause and . . .	Then . . .
A RELEASE, REWRITE, or WRITE statement has not yet been executed for the file.	The number of character positions in the record must be placed into the data item referred to by data-name-1 before any RELEASE, REWRITE, or WRITE statement is executed for the file.
A DELETE, RELEASE, REWRITE, START, or WRITE statement has been executed for the file.	The content of the data item referred to by data-name-1 is not altered.
A READ or RETURN statement has been unsuccessfully executed for the file.	The content of the data item referred to by data-name-1 is not altered.
A READ or RETURN statement has been successfully executed for the file.	The content of the data item referred to by data-name-1 indicates the number of character positions in the record just read.
A RELEASE, REWRITE, or WRITE statement is being executed for the file.	The number of character positions in the record is determined by the content of the data item referred to by data-name-1.
The INTO phrase is specified in the READ or RETURN statement.	The number of character positions in the current record that participate as the sending data items in the implicit MOVE statement is determined by the content of the data item referred to in data-name-1.

If data-name-1 is not specified in the RECORD clause and . . .	Then . . .
The record does not contain a variable-occurrence data item.	The number of character positions in the record is determined by the number of character positions in the record.
The record contains a variable-occurrence data item.	The number of character positions in the record is determined by the sum of the fixed portion and that portion of the table described by the number of occurrences at the time of execution of the output statement.
The INTO phrase is specified in the READ or RETURN statement.	The number of character positions in the current record that participate as the sending data items in the implicit MOVE statement is determined by the value that would have been moved into the data item referred to in data-name-1 if data-name-1 had been specified.

RECORD CONTAINS integer-6 TO integer-7 CHARACTERS [DEPENDING ON data-name-8]

This form of the RECORD clause enables you to specify the minimum and maximum number of character positions when the number of character positions varies. In this case, the logical records have variable lengths.

integer-6

This integer refers to the minimum number of characters in the smallest size data record.

integer-7

This integer refers to the maximum number of characters in the largest size data record.

data-name-8

Data-name-8 is a user-defined word that can be qualified.

Special Considerations for Sequential Files

The use of data-name-8 determines the BLOCKSTRUCTURE of the declared file. When data-name-8 is omitted from the RECORD CONTAINS clause or when it is internal to the record description of a file, the default is the same as that of data-name-1 as described earlier in this section. When data-name-8 is external to the record descriptions for a file, the file uses the BLOCKSTRUCTURE = EXTERNAL statement.

Details

How Record Size Is Determined

In this form of the RECORD clause, the size of each data record is completely defined in the record description entry. The size of each data record is specified according to the number of character positions required to store the logical record, regardless of the types of characters used to represent the items in the logical record.

The size of a record is determined by the sum of the number of characters in all fixed-length elementary items plus the sum of the maximum number of characters in any variable-length item subordinate to the record. This sum might be different from the actual size of the record (refer to "SYNCHRONIZED Clause" and "USAGE Clause" in this section). The size of the record is part of the record when the type of the file is DISK OR TAPE, but is not written if the file is PORT, PRINTER, or READER.

External File Connectors

If the associated file connector is an external file connector, all file description entries in the run unit that are associated with that file connector must specify the same values for integer-3 in the RECORD CONTAINS clause or integer-4 and integer-5 in the RECORD IS VARYING clause. If a RECORD clause is not specified, all record description entries associated with the file connector must be the same length.

Special Considerations for Relative and Indexed Files

The FROM integer-4 clause and the integer-6 TO clause are ignored for indexed or relative files unless the ANSI compiler control option has been set prior to the file description (FD) entry for the file. Only fixed length record files are created for indexed or relative files unless the ANSI compiler control option has been set. Specifying variable length records for indexed or relative files when the ANSI compiler control option is set makes these files incompatible with files created or retrieved through programs compiled with COBOL74. The DEPENDING ON clause is not allowed for indexed or relative files unless the ANSI compiler control option has been set prior to the file description (FD) entry for the file.

Special Considerations for Sequential Files

Data-name-1 of the DEPENDING phrase influences the file type of the file to which it applies, when it is a field internal to the file. Generally a BLOCKSTRUCTURE = VARIABLE file is created when the data item is a display numeric which occupies the first four characters of the record. Should the internal length field be elsewhere in the record or be of a different size, then a BLOCKSTRUCTURE = VARIABLEOFFSET file is created, with the supporting attributes SIZE2, SIZEOFFSET and SIZEMODE set accordingly. Should a single or double-word data item, which is used as the internal length field of the record, start on a character boundary, the resulting file will have BLOCKSTRUCTURE = EXTERNAL. Any other use of a double-word data item as an internal length field is invalid.

VALUE OF Clause

This clause defines the initial values for the attributes of a file.

The descriptive clauses and phrases of the Input-Output Section and the file record descriptions (other than the VALUE OF clause) implicitly determine the initial values for appropriate attributes of a file. These attribute values, however, can be overridden, or other attributes can be specified, by the VALUE OF clause.

File attributes provide access to functions not otherwise available within the language. Also, file attributes can be used to declare and access files. When both a file attribute and standard COBOL syntax are available to accomplish a desired function, it is always preferable to use the standard COBOL syntax, because changing the attribute can lead to unexpected results in cases when the attribute is also used or altered by the compiler.

Refer to the *I/O Subsystem Programming Guide* for a description of available attributes and their values.

Note: *The VALUE OF clause is an obsolete element in COBOL ANSI-85 and will be deleted from the next revision of standard COBOL. Unisys, however, will continue to support this element as an extension to the COBOL language.*

VALUE
VA

These keywords are equivalent.

mnemonic-attribute-value

This value must be associated with the attribute specified.

alphanumeric-file-attribute-name

If this is specified, the literal must be a nonnumeric literal, and the identifier must be a nonnumeric DISPLAY data item. Additionally, the contents of the data-name must be ended by a period.

numeric-file-attribute-name

If this is specified, the literal must be a numeric literal, and the identifier must be a numeric data item that represents an integer.

data-name-2

This name is a user-defined word.

This data-name should be qualified when necessary, but it cannot be subscripted, nor can it be described with the USAGE IS INDEX clause.

This data-name must be in the Working-Storage Section.

When an attribute is equated to the value of this data-name, the attribute is implicitly changed to this value just prior to execution of any explicit OPEN, SORT, or MERGE statement that refers to the file.

literal-1

When an attribute is equated to the value of this literal, the value becomes a part of the file description given by the file when first referred to at run time. Any specification in this file description can be overridden by a file-equation.

Details

If the associated file connector is an external file connector, all VALUE OF clauses in the run unit that are associated with that file connector must be consistent.

File titles must not contain special characters.

Using data-name-2 in file descriptions for port files is not recommended if your program specifies that subfiles will be opened independently and remain open simultaneously. The compiler explicitly sets all dynamic attributes for the entire file on each OPEN statement. The MCP will reject an OPEN statement for a subport of a file if any other subport of the file is open and the file declaration contains a dynamic file attribute that is permitted to be modified only when the file is closed.

You should use the CHANGE statement to dynamically change attributes of port files that have multiple subfiles explicitly opened. Note that the CHANGE statement must be executed while the port file is closed. Refer to "CHANGE Statement" in Section 6 for more information.

This restriction does not apply if your program opens the entire port file; if your program has only one subfile of a port open at any given time; or if there is no limitation on when a particular file attribute can be modified. For information on port files, refer to Section 12.

Examples

```
FD  SEQ-FILE
    BLOCK CONTAINS 10 RECORDS
    VALUE OF FILENAME IS "MASTER"
    DATA RECORDS ARE PRIMARY SECONDARY.
```

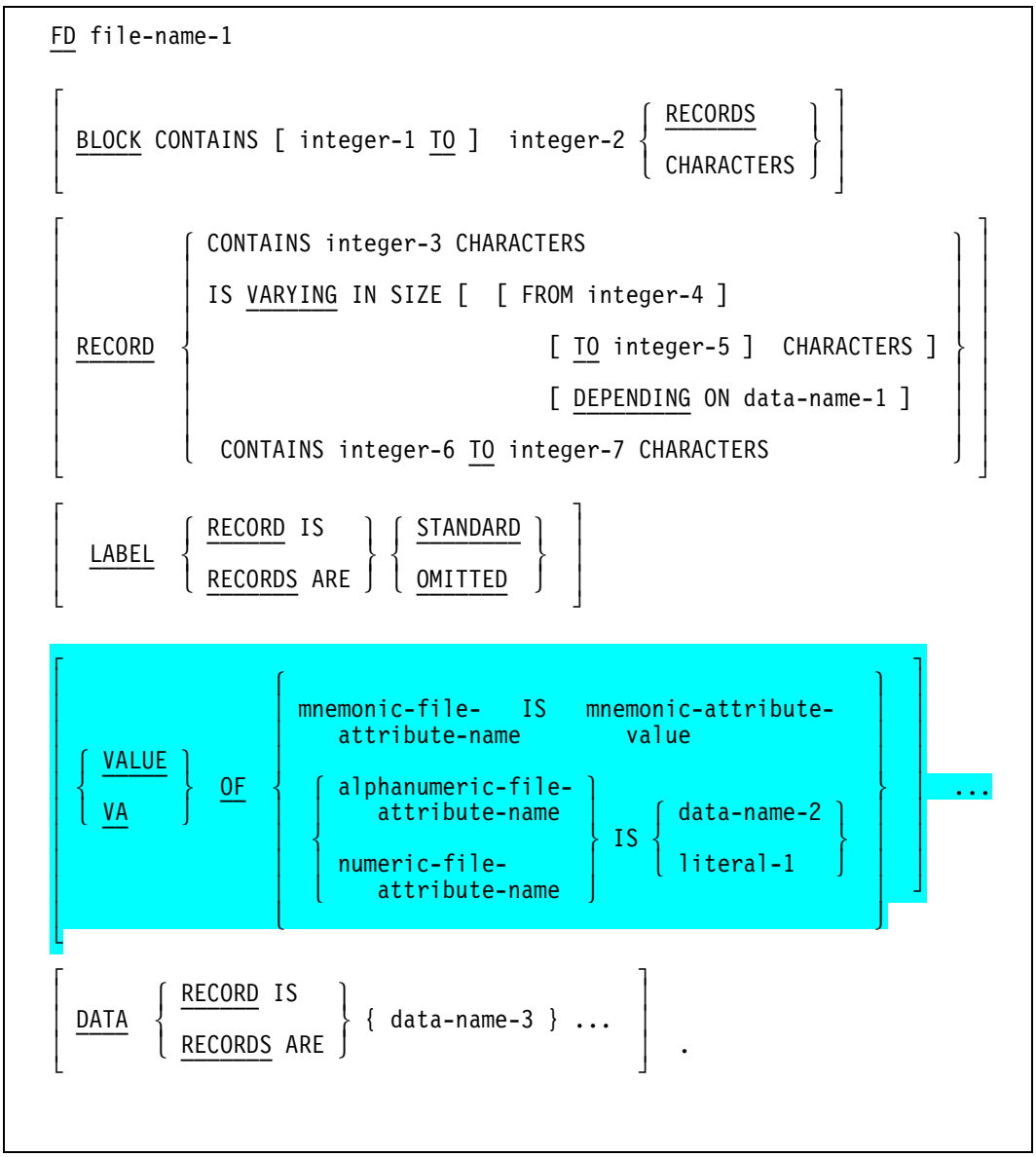
This file description entry defines a file with an internal file name of SEQ-FILE, and an external file name of MASTER. Each logical block of the file contains 10 physical file records. The records are identified as PRIMARY and SECONDARY for documentation purposes.

```
FD  PFILE
    LINAGE IS 40 LINES
    LINES AT TOP     5
    LINES AT BOTTOM 15.
```

This file description entry defines a file with an internal file name of PFILE. The logical page associated with PFILE is 40 lines in length with a top margin of 5 lines and a bottom margin of 15 lines.

File Description Entry Format 2: Relative I-O, Indexed I-O

This format provides information on the physical structure, identification, and record-names that pertain to a relative file or an indexed file.



Refer to "File Description Entry Format 1: Sequential I-O" for information on the DATA RECORDS, LABEL RECORDS, RECORD, and VALUE OF clauses.

FD

This level indicator identifies the beginning of a file description entry and must precede file-name-1. FD refers to file description.

file-name-1

This name is a user-defined word. The clauses that follow file-name-1 can appear in any order.

BLOCK CONTAINS Clause

Refer to "File Description Entry Format 1: Sequential I-O" for a complete description of this clause.

In this format (in the case of relative file organization) the physical record size is adjusted by the I/O subsystem to be integer-2 multiplied by six bytes larger than what would be determined by the methods stated in the BLOCK CONTAINS clause in "File Description Entry Format 1: Sequential I-O."

Examples

```
FD  REL-FILE
    BLOCK CONTAINS 2 RECORDS
    LABEL RECORD IS STANDARD
    VALUE OF  AREAS IS 10
              AREASIZE IS 1000
    DATA RECORDS ARE PRODUCT, PRODUCT-PART.
```

This file description entry defines a file with an internal file name of REL-FILE. Each logical block of the file contains 2 physical file records. The file attributes AREAS and AREASIZE associated with REL-FILE are assigned the values 10 and 1000, respectively. The records are identified as PRODUCT and PRODUCT-PART for documentation purposes.

```
FD  INDX-FILE
    BLOCK CONTAINS 10 RECORDS
    DATA RECORD IS ACCOUNT.
```

This file description entry defines a file with an internal file name of INDX-FILE. Each logical block of the file contains 10 physical file records. The record is identified as ACCOUNT for documentation purposes.

```
FD  CUSTOMER-FILE
    BLOCK CONTAINS 40 TO 60
    RECORD IS VARYING IN SIZE
      DEPENDING ON SIZE-VARIABLE.
```

This file description entry defines a file with an internal file name of CUSTOMER-FILE. Each logical block of the file contains between 40 and 60 physical file records. The records are variable length with the record length of each record stored in the variable SIZE-VARIABLE.

Variable Length Records

To provide the capability of variable length records, the compiler takes advantage of the I/O system implementation of files that have the following attributes:

```
BLOCKSTRUCTURE=VARIABLE
FILESTRUCTURE=STREAM
UNITS=CHARACTERS
FILEORGANIZATION=RELATIVE
```

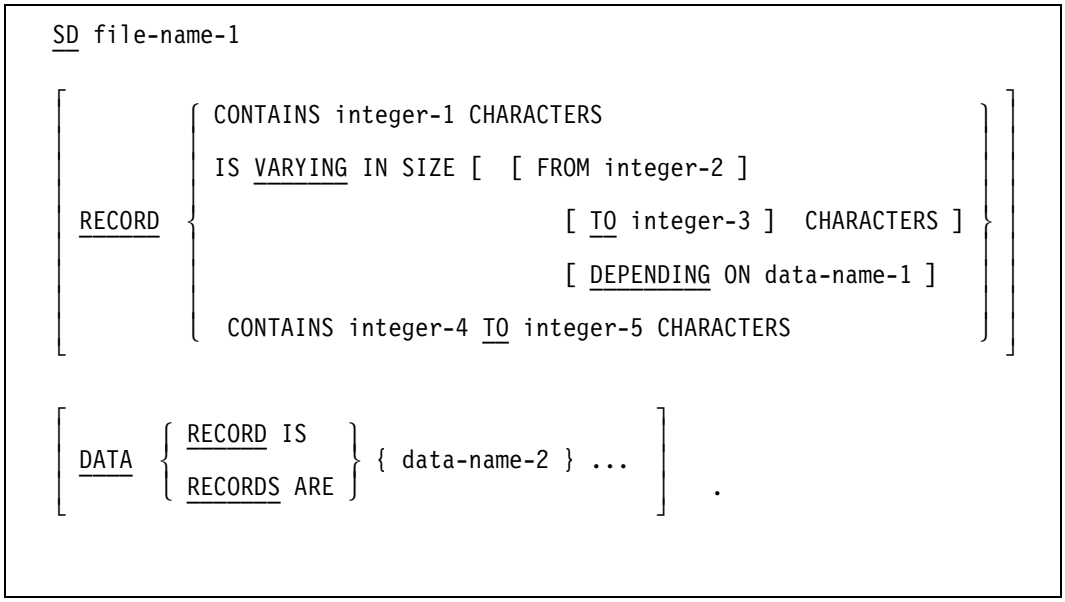
Although this is declared as a variable length record implementation, the I/O system actually maintains a fixed block and record size, where the length of each record is at least the length of the declared MAXRECSIZE value. For this reason, processor execution time, I/O transfer time and disk sector space requirements are larger than expected for files that have an average record size that is less than the declared MAXRECSIZE.

If the DEPENDING data-name is the leading 4–byte field of one of the record description entries for the file, the system sets the SIZEVISIBLE attribute to TRUE and maintains the length in the field. If the DEPENDING data-name is not a part of the record described for the file, SIZEVISIBLE is set to FALSE and the system maintains the length. In either case, before performing a WRITE or REWRITE operation, the intended record length must be programmatically established in the DEPENDING data-name. In the case of a READ followed by a REWRITE, the READ statement automatically returns the record length into the DEPENDING data-name.

Note that the DEPENDING clause is not available for indexed files.

File Description Entry Format 3: Sort-Merge

This format provides information on the physical structure and record-names that pertain to a sort or merge file.



SD

This level indicator identifies the beginning of a sort-merge file description entry and must precede file-name-1.

SD refers to sort-merge description.

Note that one or more record description entries must follow the sort-merge file description entry. However, input-output statements (except RELEASE and RETURN) cannot be executed for this sort or merge file.

file-name-1

This name is a user-defined word.

The clauses that follow file-name-1 can appear in any order.

DATA RECORDS Clause

Refer to "File Description Entry Format 1: Sequential I-O" for a complete description of this clause.

Note that information about data-name-3 in Format 1 applies to data-name-2 in this format.

RECORD Clause

Refer to "File Description Entry Format 1: Sequential I-O" for a complete description of this clause.

Note that information about integer-3, integer-4, integer-5, integer-6, and integer-7 in Format 1 applies to integer-1, integer-2, integer-3, integer-4, and integer-5, respectively, in this format.

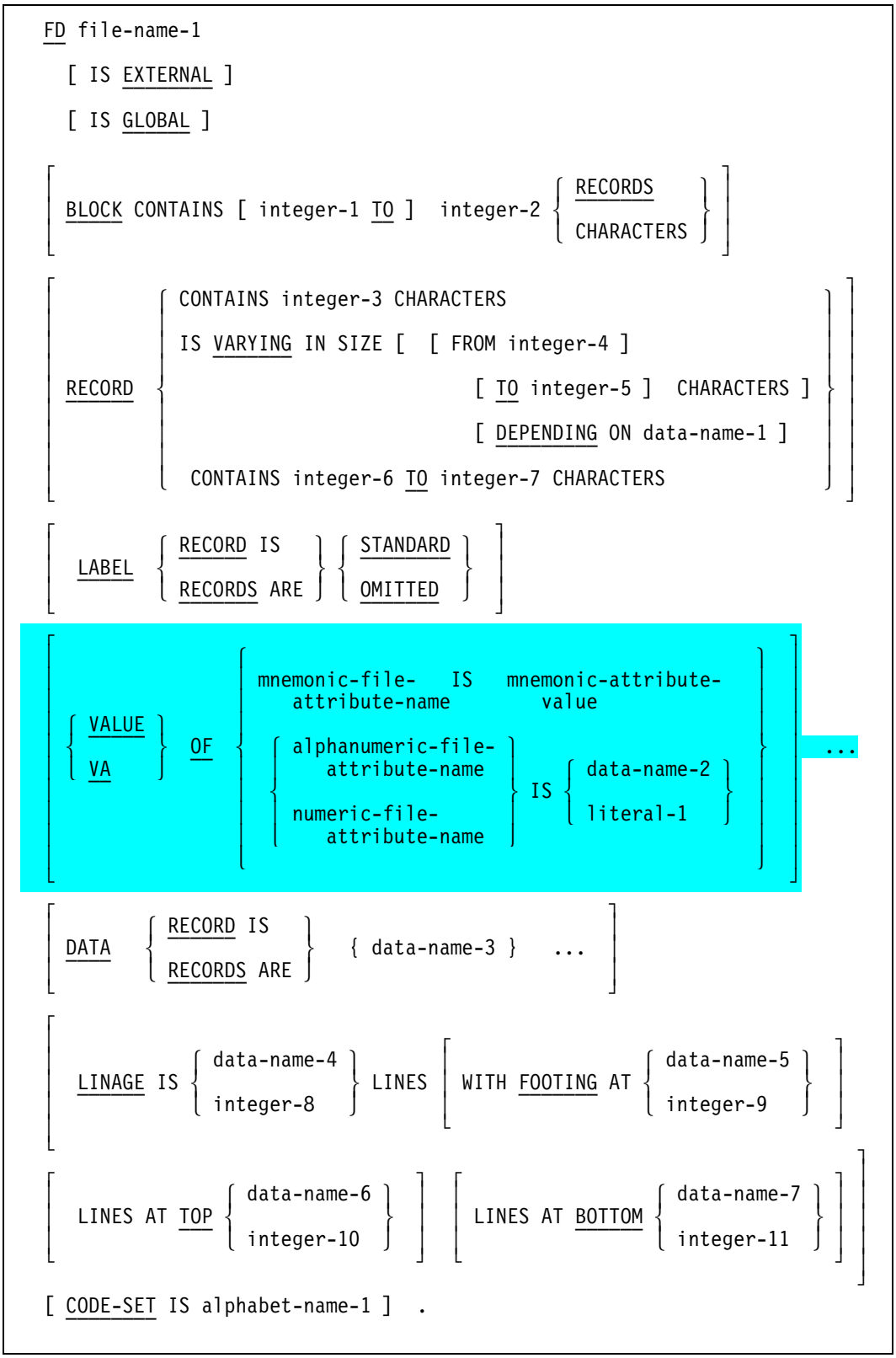
Example

```
SD  SORT-FILE
    RECORD CONTAINS 200 CHARACTERS
    DATA RECORD IS SORT-RECORD.
```

This sort-merge file description entry defines a file with an internal file name of SORT-FILE. Each record of the file contains 200 characters. The record is identified as SORT-RECORD for documentation purposes.

File Description Entry Format 4: IPC and Sequential I-O

This format is used for interprogram communication (IPC) and sequential I-O. This format determines the internal or external attributes of a file connector, of the associated data records, and of the associated data items. It also determines whether a file-name is a local name or a global name. Refer to the diagram on the following page.


FD file-name-1
[IS EXTERNAL]
[IS GLOBAL]
[BLOCK CONTAINS [integer-1 TO] integer-2 { RECORDS / CHARACTERS }]
[RECORD { CONTAINS integer-3 CHARACTERS / IS VARYING IN SIZE [[FROM integer-4] [TO integer-5] CHARACTERS] [DEPENDING ON data-name-1] / CONTAINS integer-6 TO integer-7 CHARACTERS }]
[LABEL { RECORD IS / RECORDS ARE } { STANDARD / OMITTED }]
[{ VALUE / VA } OF { mnemonic-file-attribute-name IS mnemonic-attribute-value / { alphanumeric-file-attribute-name / numeric-file-attribute-name } IS { data-name-2 / literal-1 } } ...]
[DATA { RECORD IS / RECORDS ARE } { data-name-3 } ...]
[LINAGE IS { data-name-4 / integer-8 } LINES [WITH FOOTING AT { data-name-5 / integer-9 }] [LINES AT TOP { data-name-6 / integer-10 }] [LINES AT BOTTOM { data-name-7 / integer-11 }]]
[CODE-SET IS alphabet-name-1] .

Refer to "File Description Entry Format 1: Sequential I-O" for information on the BLOCK CONTAINS, RECORD, LABEL RECORDS, VALUE OF, DATA RECORDS, LINAGE, and CODE-SET clauses.

FD

This level indicator identifies the beginning of a file description entry and must precede file-name-1.

FD refers to file description.

file-name-1

This name is a user-defined word.

The clauses that follow file-name-1 can appear in any order.

EXTERNAL Clause

In this format, this clause specifies that a file connector is external. The subordinate data items and group data items of an external data record are available to every program in the run unit that describes that record.

This clause can be specified only in file description entries in the File Section when used for interprogram communication.

If the file description entry contains the LINAGE clause and the EXTERNAL clause, the LINAGE-COUNTER data item is an external data item.

The file connector associated with this file description entry is an external file connector.

Note that use of the EXTERNAL clause does not imply that the associated file-name is a global name. (Refer to the GLOBAL Clause in this section.)

GLOBAL Clause

In this format, this clause specifies that a file-name is a global name. A global name is available to every program contained within the program that declares it. A statement in a program contained directly or indirectly within a program that describes a global name can refer to that name without describing it again (refer to "Scope of Names" in Section 10).

This clause can be specified only in file description entries.

If the file description entry contains the LINAGE clause and the GLOBAL clause, the special register LINAGE-COUNTER is a global name.

Note that if the SAME RECORD AREA clause is specified for several files, the file-description entries for these files must not include the GLOBAL clause.

Examples

```
FD  SEQ-FILE   IS EXTERNAL
    BLOCK CONTAINS 20 RECORDS
    RECORD CONTAINS 22 CHARACTERS
    VALUE OF ACCESSMODE IS SEQUENTIAL.
```

This file description entry defines an external file with an internal file name of SEQ-FILE. Each logical block of the file contains 20 physical file records, and each physical record contains 22 characters. The file attribute ACCESSMODE associated with SEQ-FILE is assigned the value SEQUENTIAL.

```
FD  SEQ-FILE   IS GLOBAL
    BLOCK CONTAINS 5 RECORDS
    DATA RECORD IS RECORD-NAME.
```

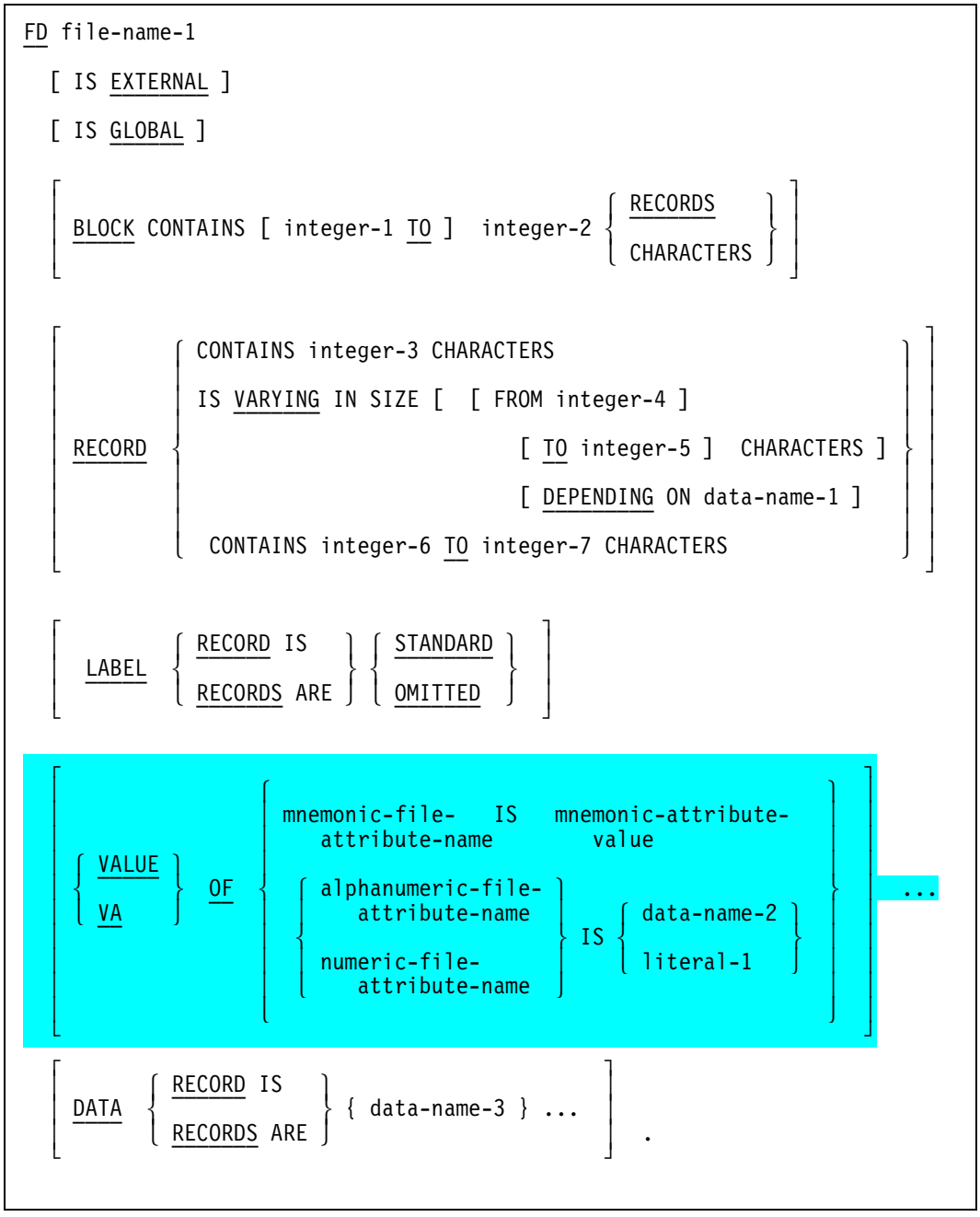
This file description entry defines a global file with an internal file name of SEQ-FILE. Each logical block of the file contains 5 physical file records. The record is identified as RECORD-NAME for documentation purposes.

```
FD  PFILE   IS EXTERNAL
    LINAGE IS 30
      WITH FOOTING AT 6.
```

This file description entry defines an external file with an internal file name of PFILE. The logical page associated with PFILE is 30 lines in length, with the footing area beginning at line number 6 of the page body.

File Description Entry Format 5: IPC, Relative I-O, and Indexed I-O

This format is used for interprogram communication (IPC) and relative I-O or indexed I-O. This format determines the internal or external attributes of a file connector, of the associated data records, and of the associated data items. It also determines whether a file-name is a local name or a global name.



Refer to "File Description Entry Format 1: Sequential I-O" for information on the BLOCK CONTAINS, RECORD, LABEL RECORDS, VALUE OF, and DATA RECORDS clauses.

Refer to "File Description Entry Format 4: IPC and Sequential I-O" for information on the EXTERNAL and GLOBAL clauses.

FD

This level indicator identifies the beginning of a file description entry and must precede file-name-1.

FD refers to file description.

file-name-1

This name is a user-defined word.

The clauses that follow file-name-1 can appear in any order.

Examples

```
FD  REL-FILE   IS GLOBAL
    BLOCK CONTAINS 3 RECORDS
    LABEL RECORDS ARE STANDARD.
```

This file description entry defines a global file with an internal file name of REL-FILE. Each logical block of the file contains 3 physical file records.

```
FD  CUSTOMER-FILE   IS EXTERNAL
    BLOCK CONTAINS 40 TO 60
    RECORD IS VARYING IN SIZE
      DEPENDING ON SIZE-VARIABLE.
```

This file description entry defines an external file with an internal file name of CUSTOMER-FILE. Each logical block of the file contains between 40 and 60 physical file records. The records are variable in length, with the record length of each record stored in the variable SIZE-VARIABLE.

```
FD  INDX-FILE   IS GLOBAL
    BLOCK CONTAINS  10 RECORDS
    DATA RECORD IS  ACCOUNT.
```

This file description entry defines a global file with an internal file name of INDX-FILE. Each logical block of the file contains 10 physical file records. The record is identified as ACCOUNT for documentation purposes.

Working-Storage Section

The Working-Storage Section describes records and subordinate data items that are not part of external data files, but are developed and processed internally. In addition, the Working-Storage Section describes data items that have values assigned in the source program that do not change during the execution of the object program. Use of this section is optional.

The Working-Storage Section is composed of the section header and record description entries (and/or data description entries) for noncontiguous data items.

The general format of the Working-Storage Section is as follows:

```
WORKING-STORAGE SECTION.

[ 77-level description entry ]
[                            ]  ...
[ record description entry   ]
```

WORKING-STORAGE SECTION

These keywords begin in area A and must be followed by a period.

77-level description entry

This is a data description entry that describes a noncontiguous data item with the level-number 77. Refer to "Data Description Entry Format 1" in this section for more information about this entry.

record description entry

This is the total set of data description entries associated with a particular record. Refer to "General Format of the File Section" in this section for more information about this entry.

Note that a record description entry is also referred to as a record description.

Noncontiguous Working Storage

Items and constants in working storage that do not have a hierarchical relationship to one another do not need to be grouped into records if they do not need to be further subdivided. Instead, they are classified and defined as noncontiguous elementary items. Each of these items is defined in a separate data description entry that begins with the special level-number 77.

The following data clauses are required in each data description entry:

- Level-number 77
- Data-name
- The PICTURE clause or the USAGE IS INDEX clause

Other data description clauses are optional but can be used to complete the description of the item, if necessary.

Working-Storage Records

Data elements in the Working-Storage Section that have a definite hierarchical relationship to one another must be grouped into records according to the rules for formation of record descriptions.

Data elements in the Working-Storage Section that do not have a hierarchical relationship to any other data item can be described as records that are single elementary items.

All clauses used in record descriptions in the File Section can be used in record descriptions in the Working-Storage Section.

Initial Values

The initial value of any data item in the Working-Storage Section, except an index data item, is specified by associating the VALUE clause with the data item. The initial value of any index data item or any data item not associated with a VALUE clause is undefined.

Example

```
WORKING-STORAGE SECTION.
77 DISK-CONTROL           PIC 9(8).
77 TOTAL-SALES             PIC 9(11)     VALUE IS ZERO.
01 STATE-TABLE.
   05 STATES.
      10 CA     PIC 9(4).
      10 NEVADA PIC 9(4).
      10 ORE    PIC 9(4).
01 HDG-LINE.
   03 FILLER    PIC X(58) VALUE IS SPACES.
   03 FILLER    PIC X(11) VALUE IS "PERFORMANCE".
   03 FILLER    PIC X(51) VALUE IS SPACES.
        .
        .
        .
```

In this example, DISK-CONTROL and TOTAL-SALES represent noncontiguous elementary items. STATE-TABLE and HDG-LINE represent working-storage records with subordinate entries (STATES and FILLER). This entire working-storage section describes the records in a sales performance report.

Linkage Section

The Linkage Section appears in a called program and describes data items that are referred to by the calling program and the called program. If a data item in the Linkage Section is accessed in a program that is not a called program, the effect is undefined.

The Linkage Section describes data that is available through the calling program, but will be referred to in both the calling and the called program.

The Linkage Section is meaningful in a program only if both of the following are true:

- The object program will function under the control of a CALL, PROCESS, or RUN statement.
- The CALL, PROCESS, or RUN statement in the calling program contains a USING phrase.

The way that data items described in the Linkage Section of a called program correspond to data items described in the calling program is discussed in Section 5 under "Procedure Division Header" and in Section 6 under "CALL Statement."

In the case of index-names, a correspondence is not established, and index-names in the called and calling programs always refer to separate indexes.

Note: *Data items defined in the Linkage Section but not referenced in a USING phrase will be treated in the same way as other Working-Storage Section data items.*

The Linkage Section consists of a section header and noncontiguous data items and/or record description entries.

The format for the Linkage Section is as follows:

```
LINKAGE SECTION.

[ { 77-level description entry  } ]
[ { record description entry    } ]
```

LINKAGE SECTION

These keywords begin in area A and must be followed by a period.

77-level description entry

This is a data description entry that describes a noncontiguous data item with the level-number 77. Refer to "Data Description Entry Format 1" in this section for more information about this entry.

record description entry

This is the total set of data description entries associated with a particular record. Refer to "Record Description Entry" in this section for more information about this entry.

A record description entry is also referred to as a record description.

Noncontiguous Linkage Storage

Items in the Linkage Section that do not have a hierarchical relationship to one another do not need to be grouped into records. Instead, they are classified and defined as noncontiguous elementary items. Each of these items is defined in a separate data description entry that begins with the special level-number 77.

The following data clauses are required in each data description entry:

- Level-number 77
- Data-name
- The PICTURE clause or the USAGE clause

Other data description clauses are optional but can be used to complete the description of the item, if necessary.

Linkage Records

Data elements in the Linkage Section that have a definite hierarchical relationship to one another must be grouped into records according to the rules for the formation of record descriptions.

Data elements in the Linkage Section that do not have a hierarchical relationship to any other data item can be described as records that are single elementary items.

Initial Values

The VALUE clause must not be specified in the Linkage Section, except in condition-name entries (level 88). Refer to "Data Description Entry Format 3: Level-88 Condition-Name Entry."

Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CALLER-PROGRAM.
DATA DIVISION.
      .
      .
      .
WORKING-STORAGE SECTION.
01 COLOR    PIC X(10).
01 LOC-SIZE PIC 99V99.

01 AMOUNT   PIC 999.

      .
      .
      .
PROCEDURE DIVISION.

PARA-1.
    CALL CALLED-PROGRAM
    USING LOC-SIZE, COLOR.
      .
      .
      .
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CALLED-PROGRAM.
DATA DIVISION.
.
.
.
WORKING-STORAGE SECTION.
01 WS-1    PIC 99V99.
.
.
.

LINKAGE SECTION.
01 HUE     PIC X(10).
01 MY-SIZE PIC 99V99.
PROCEDURE DIVISION
    USING MY-SIZE, HUE.
PARA-A.
    MOVE MY-SIZE TO WS-1.
    MOVE "RED" TO HUE.
EXIT PROGRAM.
```

The program on the left (CALLER-PROGRAM) is calling the program on the right (CALLED-PROGRAM). The identifiers (SIZE and COLOR) that will be passed to the called program are defined in the program that contains the CALL statement. These identifiers correspond to MY-SIZE and HUE, which are defined in the Linkage Section of the called program.

Local-Storage Section

The Local-Storage Section describes data that is available through the user program, but is referred to in both the user program and the imported library procedure. The data descriptions in the Local-Storage Section are the formal parameters expected by separate tasks, bound programs, or library procedures imported into the Program-Library Section. These data descriptions are used only to perform parameter matching when the separate task is started, when the host program and bound procedure are compared, or when the program and library are linked. Therefore, no space is allocated in the computer for the data described in this section. For more information on libraries, refer to "Program-Library Section" later in this section, and to Section 11.

The Local-Storage Section is optional.

The format for the Local-Storage Section is as follows:

```
LOCAL-STORAGE SECTION.

 LD { local-name } .
[  {  77-level description entry  }  ]
[  {                              }  ]
[  {  record description entry    }  ]
```

LOCAL-STORAGE SECTION

These keywords begin in area A and must be followed by a period.

LD

This level indicator identifies the beginning of a local-storage-description entry.

local-name

This name is a user-defined word.

77-level description entry

This is a data description entry that describes a noncontiguous data item with the level-number 77. Refer to "Record Description Entry" in this section for more information about this entry.

record description entry

This is the total set of data description entries associated with a particular record. Refer to "Record Description Entry" earlier in this section for more information about this entry.

Details

When an imported procedure is declared in the Program-Library Section, the Local-Storage Section must contain the data descriptions that describe the formal parameters of that procedure. Also, the data descriptions that describe the formal parameters of external procedures bound into the program must be placed in the Local-Storage Section.

The level indicator and its associated local-name provide a method of grouping data descriptions under a single heading. In the declaration of the imported procedure, the WITH clause permits specification of the local-name or local-names under which the data descriptions of the formal parameters can be found

Noncontiguous Local-Storage

If data items in the Local-Storage Section do not have a hierarchical relationship to one another, they do not need to be grouped into records, provided that they do not need to be further subdivided. Instead, the data items are classified and defined as noncontiguous elementary items. Each of these items is defined in a separate data-description entry, beginning with the special level-number 77.

The following data clauses are required in each data description entry for a noncontiguous elementary data item:

- Level-number 77
- Data-name
- The PICTURE clause or the USAGE clause

Local-Storage Records

If data items in the Local-Storage Section have a hierarchical relationship to one another, they must be grouped into records according to the rules for formation of record descriptions. All clauses used in record descriptions in the File Section can be used in record descriptions in the Local-Storage Section.

Initial Values

The VALUE clause must not be specified in the Local-Storage Section, except in condition-name entries (level 88). Refer to "Data Description Entry Format 3: Level-88 Condition-Name Entry."

Program-Library Section

The Program-Library Section defines the interface between a user program and a library program.

Each library is defined by a library description entry.

The general format of the Program-Library Section is as follows:

```
PROGRAM-LIBRARY SECTION.
  [ library description entry ] ...
```

PROGRAM-LIBRARY SECTION

These keywords begin in area A and must be followed by a period.

library description entry

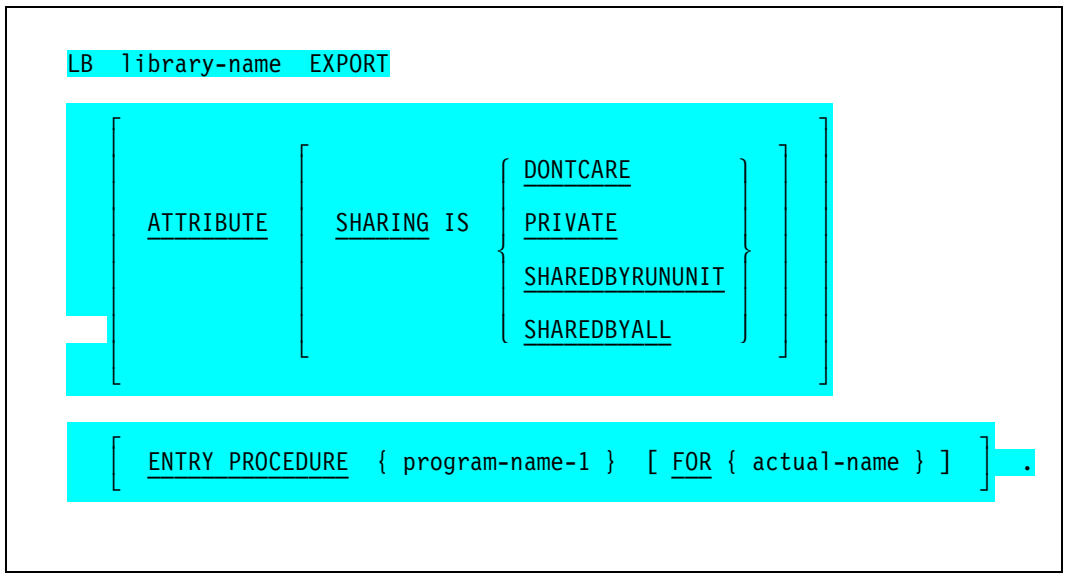
A library description entry defines the interface established for a particular library.

There are two formats for the library description entry:

Format	Use
Format 1	This format provides information required to make the current program into a library program. This format is also known as an export definition.
Format 2	This format provides information required to permit the current program to access a library program. This format is also known as an import definition.

Library Description Entry Format 1: Export Definition

This format provides information on the directly nested programs contained in this library that are to be exported as entry points into the library. Only the outermost program can contain an export definition.



LB

This level indicator identifies the beginning of a library description entry.

library-name

This name is a user-defined name, and must be the same as the program-name defined in the PROGRAM-ID paragraph of the current program.

EXPORT

This identifies the library description entry as an export definition. The procedures exported by this definition are assumed to be nested programs in the current program.

ATTRIBUTE Clause

The ATTRIBUTE clause specifies the setting of library attributes. For an export definition, only the SHARING attribute can be specified. The SHARING attribute controls how user programs share access to the library. For more information, see Section 11.

DONTCARE

The operating system determines the sharing.

PRIVATE

A copy of the library is invoked for each user (calling program). Any changes made to global items in the library by the actions of the user are visible only to that user.

SHAREDBYRUNUNIT

All invocations of the library within a run unit share the same copy of the library. A run unit consists of a program and all libraries that are initiated either directly or indirectly by that program.

SHAREDBYALL

All simultaneous users share the same instance of the library.

ENTRY PROCEDURE Clause

The ENTRY PROCEDURE clause defines the directly nested programs contained in this library that are to be exported as entry points when the library "freezes" (suspends execution and makes entry points available).

program-name

The program-name is defined in the PROGRAM-ID paragraph of a nested program to be made available as an entry point. The program defined by program-name must be directly nested within the program containing this Program-Library Section export definition.

FOR actual-name

The FOR phrase specifies the name of the entry point. If the FOR phrase is omitted, the name of the entry point is program-name.

If the FOR phrase is used, the actual-name is assigned as the name of the entry point, and is used by user programs to import the procedure. The actual-name item is a literal.

Details

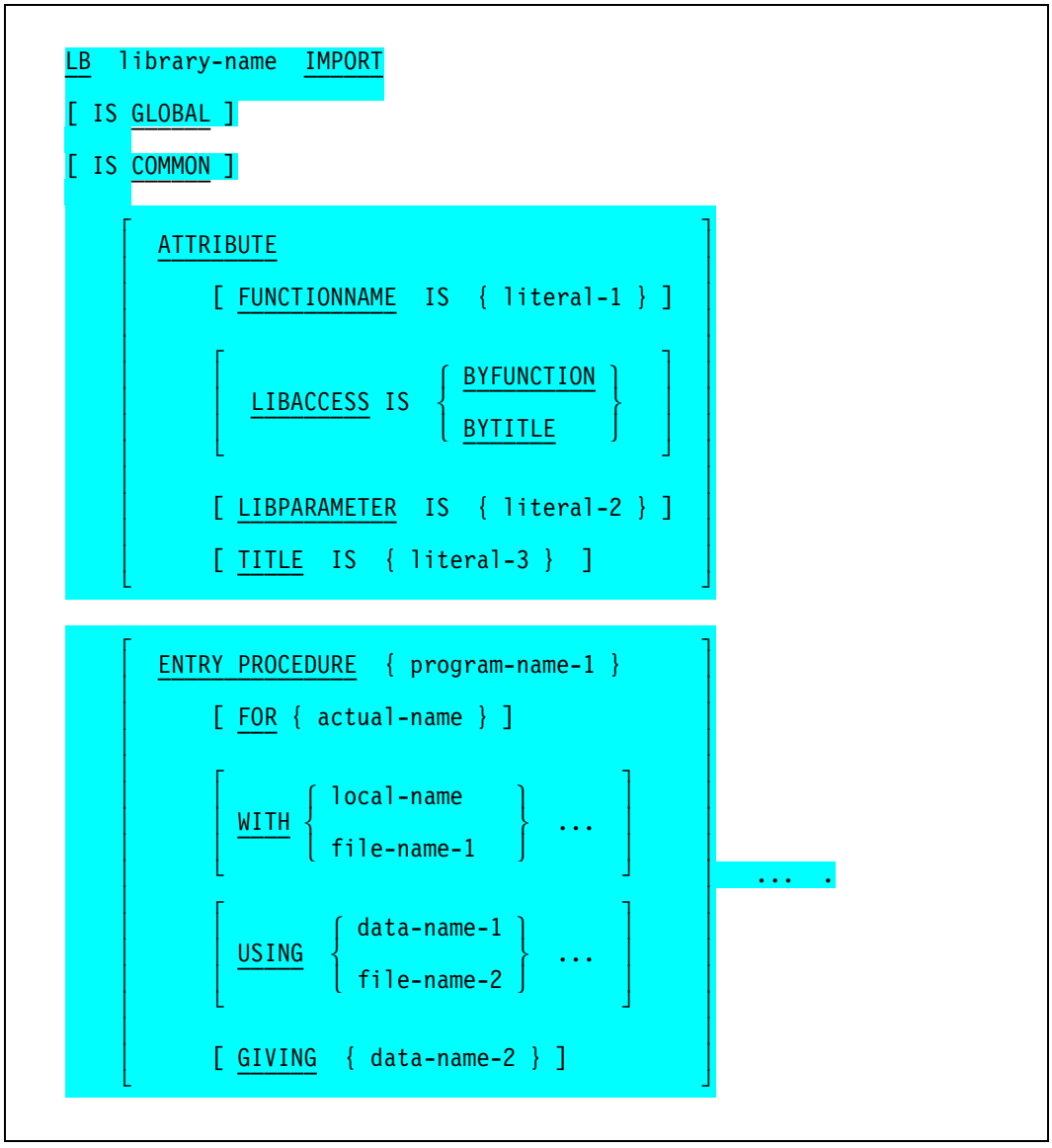
When the library executes a library FREEZE, the entry points defined by this export definition become available to user programs. The formal parameters of the nested program must be declared in the Linkage Section of the nested program, and must be specified in the Procedure Division header (with the USING and GIVING clauses) of the exported program. The information on formal parameters ensures that the interface provided by a user program matches the interface expected by the library procedure.

To access the programs within the library, a user program must include an import definition in its Program-Library Section (refer to "Library Description Entry Format 2: Import Definition" in this section).

For more information on COBOL85 libraries, refer to Section 11.

Library Description Entry Format 2: Import Definition

This format provides information on the external library procedures that are imported by this program. External library procedures are procedures (nested programs in COBOL85) that have been exported by a library program.



LB

This level indicator identifies the beginning of a library description entry.

library-name

This name is a user-defined name, and must be the same as the program-name defined in the PROGRAM-ID paragraph of the library program that contains the procedures to be imported.

IMPORT

This identifies the library description entry as an import definition. The procedures imported by this definition must be contained in and exported by the library program identified by library-name.

IS GLOBAL

This clause declares the imported library and its entry procedures to be global to any programs that might be contained in this program. If an imported library and its entry procedures are global to nested programs, those nested programs can access the imported library without having to duplicate the import definition for the library.

IS COMMON

This clause declares the imported library and its entry procedures to be common to both the bound program and the host program. If the IS COMMON clause is specified, the import definition for the library is assumed to be specified in the host program.

ATTRIBUTE Clause

The ATTRIBUTE clause specifies the setting of library attributes. For an import definition, the following library attributes can be specified. (See “Library Attributes” in Section 11).

FUNCTIONNAME

This specifies the system function name used to find the object code file for the library.

LIBACCESS

This specifies how the object code file for the library is located. LIBACCESS can be set to the following values:

- BYFUNCTION

The FUNCTIONNAME attribute locates the object code file for the library.

- BYTITLE

The TITLE attribute locates the object code file.

LIBPARAMETER

This is used to pass information from the user program to the selection procedures of libraries that provide entry points dynamically.

TITLE

This specifies the file title of the object code file of the library.

ENTRY PROCEDURE Clause

The ENTRY PROCEDURE clause defines the procedure or procedures to be imported from the library identified by library-name.

program-name

This is the name of a program referenced by a CALL statement in the user program that resides in the library identified by library-name. If the FOR clause is omitted, program-name must match the name of an entry point exported by the library identified by library-name.

FOR actual-name

The FOR clause specifies the name of the library entry point to be associated with program-name. If the FOR clause is used, actual-name must match the name of an entry point exported by the library identified by library-name.

WITH clause

The WITH clause specifies the LD clause in the Local-Storage Section that contains the description of the formal parameter data referenced by this imported procedure.

The Local-Storage Section appears in a user program and describes data items referred to by a user program and imported library procedures. Data type formal parameters must be described under the specified local-name. A file-name formal parameter must be defined as a LOCAL RECEIVED BY REFERENCE file in the FILE-CONTROL paragraph of the user program. For more information, refer to “Local-Storage Section” in this section.

USING clause

The USING clause specifies the formal parameters expected by the imported procedure. If the imported procedure is from a COBOL85 library, then the specification of the USING clause must match the Procedure Division header of the imported program. If the imported procedure is from a library written in another programming language, then the specification of the USING clause must match the formal parameter declaration for the imported procedure.

Any data parameters specified in the USING clause (data-name-1 to data-name-n) must be declared in the Local-Storage Section of the user program. Any file parameters must be declared as LOCAL RECEIVED BY REFERENCE files in the FILE-CONTROL paragraph of the user program.

GIVING clause

The GIVING clause is valid only for imported procedures that return a result. If the imported procedure is from a COBOL85 library, the specification of a GIVING clause must match the GIVING clause in the Procedure Division header of the imported program. If the imported procedure is from a library written in another programming language, then the specification of the GIVING clause must match the formal result declaration for the imported procedure.

The result returned by the program is identified by data-name-2, which must be declared in the Local-Storage Section of the user program. Data-name-2 must be a NUMERIC data item.

Details

When the library executes a library FREEZE, the entry points defined by this export definition are made available to user programs. The information concerning formal parameters (with the USING and GIVING clauses) is used to ensure that the interface provided by the user program matches the interface expected by the exported library procedure.

To access the programs in the library, a user program must include an import definition in its Program-Library Section. Import definitions are discussed later in this section.

For more information on COBOL85 libraries, refer to Section 11.

Section 5
Procedure Division Concepts

The Procedure Division is the fourth and last division of a COBOL source program. This division contains procedures that direct the system to perform certain actions in a given sequence.

The Procedure Division is optional in a COBOL source program. For example, you would not need a Procedure Division in a program that is to be nested in another program. You could also omit the Procedure Division when compiling part of a program to check syntax.

This section discusses the following concepts:

- The structure of the Procedure Division
- The general formats for the syntax of the Procedure Division
- The elements of a procedure
- The classifications of statements and sentences used in the Procedure Division
- The categories of COBOL verbs
- Information on common Procedure Division applications, such as arithmetic and conditional expressions, table handling, and sorting and merging

For the syntax of the elements and statements that comprise the Procedure Division, refer to Sections 6 through 8.

Structure of the Procedure Division

The Procedure Division begins with a header and can contain declarative and nondeclarative procedures. Declarative procedures are grouped at the beginning of the Procedure Division. The remainder of the division must consist of nondeclarative procedures.

An end program header can be used to indicate the end of the named COBOL source program. The end program header can be followed by a COBOL program that is to be compiled separately in the same invocation of the compiler.

Execution begins with the first statement of the Procedure Division, excluding declaratives. Statements are then executed in the order in which they are presented for compilation, except where the rules indicate some other order.

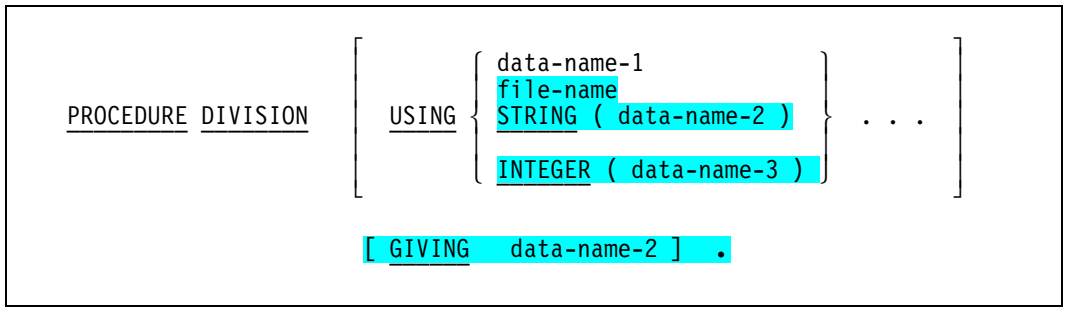
General Formats

The following general formats apply to the Procedure Division and are discussed in this section:

- Procedure Division header format
- Declarative procedure format
- Nondeclarative procedure format
- End program header format

Procedure Division Header

The following header identifies and must begin the Procedure Division:



PROCEDURE DIVISION

These keywords begin in area A.

USING

The USING clause names the identifiers that are received as parameters.

The data-name in the USING clause of the Procedure Division header must be defined in the Linkage Section of the program in which this header occurs, and it must have a 01 or 77 level-number and must not be a redefined item. If a file-name is specified, the file must be declared as a RECEIVED BY REFERENCE file in the SELECT clause.

When the USING clause is present, the object program operates as if each identifier in the list is replaced by the corresponding identifier from the USING clause of the CALL statement of the calling program.

When the RECEIVED BY REFERENCE clause appears in an identifier's data description, the corresponding identifier refers to a single set of data available to both the calling and called program.

When the data-name is RECEIVED BY CONTENT, the invocation of the procedure will initialize the corresponding data-name in the called program's USING clause to the current value in the initiating program. The correspondence is positional and not by symbolic name.

The calling program must contain a CALL, PROCESS, or RUN statement with a USING phrase. Section 8 provides detailed information about these statements.

data-name

This identifies a data item or items that will be shared by both the calling program and the called program.

The following rules apply to the data-name:

- The data-name must be defined as a level-01 or level-77 entry in the Linkage Section of the program in which this header occurs. The Linkage Section describes data to be shared when a program is communicating with another program.
- You cannot specify the same data-name more than once in a USING phrase.
- The record description entry for a data-name must not contain a REDEFINES clause.
- The description of the data item in both the calling and called program must describe the same number of character positions.
- The data-name in the called program and the data-name in the calling program do not have to be the same name. Correspondence between data-names is based on the data-name's position in the data description entry, not by name.
- Data items in a USING phrase (data-names and file-names) are separated by a comma.

file-name

This identifies a file to be shared by both the calling and called program. The file must be declared RECEIVED BY REFERENCE in the SELECT clause.

GIVING data-name-2

The GIVING clause identifies the procedure as a function that returns a value to the calling program in data-name-2. Data-name-2 must be a numeric data item.

Details

When the USING clause in the Procedure Division header is present, the object program operates as if each data-name in the list is replaced by the corresponding data-name from the USING phrase of the CALL, PROCESS, or RUN statement of the calling program.

You can refer to data items defined in the Linkage Section of the called program in the Procedure Division of that program.

Related Information

The following table provides references for additional information related to this topic:

For information about . . .	Refer to . . .
The Linkage Section	Section 4
The REDEFINES Clause	Section 4
The Working-Storage Section	Section 4
The CALL, PROCESS, or RUN statement	Sections 6 and 7
Interprogram communication	Section 10

Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PROGA.
.
.
.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  A.
    05  A1  PIC X(20).
    05  A2  PIC X(20).
    05  A3  PIC X(4).
01  B       PIC X(6).
01  C       PIC 9(4)V99.
.
.
.
PROCEDURE DIVISION.
     .
     .
     .
     CALL "PROGB" USING A, C.
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PROGB.
.
.
.
DATA DIVISION.
LINKAGE SECTION.
01  Employee-Data.
    05  Name    PIC X(20).
    05  Title   PIC X(20).
    05  Dept-no PIC X(4).
01  Hire-date   PIC X(6).
01  Salary      PIC 9(4)V99.
.
.
.
PROCEDURE DIVISION USING
     Employee-Data, Salary.
```

The statement *PROCEDURE DIVISION USING Employee-Data Salary* indicates the beginning of the Procedure Division and that the program containing this header, PROGB, is to be called by another program. The two programs will have access to the data in Employee-Data and Salary.

The CALL statement in PROGA contains a USING phrase.

Employee-Data and Salary are defined as level-01 items in the Linkage Section. The data-names A and C are defined as level-01 items in the Working-Storage Section.

Data-name A in PROGA corresponds to Employee-Data in PROGB; data-name C in PROGA corresponds to Salary in PROGB.

All corresponding data-names have the same number of characters, but do not have the same names.

Declarative Procedure Format

Declarative procedures consist of a set of one or more special-purpose sections that are grouped together at the beginning of the Procedure Division following the Procedure Division header. Each declarative procedure is composed of a section header, followed by a USE compiler-directing sentence, optionally followed by one or more associated paragraphs. Declarative procedures can be used when special conditions, such as input-output errors, occur during the execution of a program.

```
[ DECLARATIVES.
{section-name SECTION.
    declarative-sentence.
 [paragraph-name.
    [sentence]  ... ] ... }  ...
END DECLARATIVES. ]
 {section-name SECTION.
 [paragraph-name.
    [sentence]  ... ] ... }  ...
```

DECLARATIVES

This keyword must appear on a line by itself, begin in area A, and be followed by a period.

section-name

This user-defined word names a section of code. A section-name can consist of the characters *A* through *Z*, *a* through *z*, *0* through *9*, and the hyphen (-). The hyphen cannot appear as the first or last character of the section-name.

You can use the section-name in the nondeclarative portion of this syntax in Format 7 of the CALL statement to enter another program. Refer to the CALL statement in Section 6 for details.

paragraph-name

This user-defined word names a paragraph of code. A paragraph-name can consist of the characters *A* through *Z*, *a* through *z*, *0* through *9*, and the hyphen (-). The hyphen cannot appear as the first or last character of the paragraph-name.

sentence

This sentence consists of one or more compiler-directing statements and ends with a period.

END DECLARATIVES

These keywords must appear on a line by themselves, begin in area A, and be followed by a period (.).

The next source statement following the END DECLARATIVES statement must be a section-name.

Details

Each declarative consists of a single section.

A SORT statement cannot appear in the DECLARATIVES section of a COBOL program.

Refer also to "USE Statement" in Section 8 for more information.

Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DECL-IO-EXAMPLE.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
     SELECT INPUT-FILE ASSIGN TO DISK
     FILE STATUS IS INPUT-STATUS.
DATA DIVISION.
FILE SECTION.
FD  INPUT-FILE
      VALUE OF TITLE IS "DISK-FILE".
01  INPUT-REC PIC X(80).
WORKING-STORAGE SECTION.
77  INPUT-STATUS PIC XX.
PROCEDURE DIVISION.
DECLARATIVES.
DECL-1 SECTION.
    USE AFTER STANDARD ERROR PROCEDURE ON INPUT.
D-0010.
    DISPLAY "I/O ERROR READING FILE".
    DISPLAY "FILE STATUS IS" INPUT-STATUS.
    STOP RUN.
END DECLARATIVES.
MAIL-1000 SECTION.
P-1010.
    MOVE "00" TO INPUT-STATUS.
    OPEN INPUT INPUT-FILE.
    READ INPUT-FILE.
    STOP RUN.
```

This program illustrates the use of declaratives to handle input-output errors when reading the file INPUT-FILE. After an input-output error has occurred, and the system's standard retry procedures have been used, "I/O ERROR READING FILE" and "FILE STATUS IS" with the value of INPUT-STATUS will be displayed.

Nondeclarative Procedure Format

Nondeclarative procedures comprise the main portion of a COBOL85 program. A nondeclarative procedure consists of a paragraph-name followed by one or more statements that make a logical unit.

Once the program is compiled and initiated, execution begins with the first statement in the Procedure Division, excluding declaratives. Statements are then executed in the order in which they were compiled, except where the rules indicate some other order.

```
{paragraph-name.
    [sentence]... } ...
```

paragraph-name

This name is a user-defined word and can consist of the characters *A* through *Z*, *a* through *z*, *0* through *9*, and the hyphen (-). The hyphen cannot appear as the first or last character. A paragraph-name ends with a period.

sentence

This sentence consists of one or more statements and ends with a period. The Procedure Division statements are presented in Section 6.

Example

```
WRITE-PARA.
      MOVE IN-NAME TO OUT-NAME
      WRITE OUT-RECORD AFTER ADVANCING 2 LINES.
```

This example contains the paragraph-name WRITE-PARA and a sentence containing two statements—MOVE and WRITE.

End Program Header

The end program header indicates the end of the named COBOL source program. You must use the end program header if the COBOL source program

- Contains one or more other COBOL source programs.
- Is contained in another COBOL source program.
- Is in a sequence of programs to be separately compiled in the same invocation of the compiler. Only the last program in the sequence would not need an end program header.

```
END PROGRAM  program-name .
```

END PROGRAM

The end program header begins in area A.

program-name

This name is a user-defined word and must be identical to the program-name declared in the PROGRAM-ID paragraph of the source program's Identification Division.

A program-name must have at least one alphabetic character.

Details

If the source program does not have a Procedure Division, the end program would follow the end of the Data Division.

Example

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. EMPFIL.
 .
 .
 .
 END PROGRAM EMPFIL.
```

The END PROGRAM header indicates the end of the program EMPFIL.

Refer to "IPC Examples" in Section 10 for examples of the end program header used with a sequence of programs and nested programs.

Elements of a Procedure

A procedure is composed of a section, or a group of successive sections, or a paragraph, or a group of successive paragraphs. Paragraphs can be further broken down into sentences, statements, and verbs. Table 5–1 describes the elements that form a procedure.

Table 5–1. Elements of a Procedure

Element	Definition	Format
Section	A section contains a section-name followed by a period and zero, one, or more paragraphs. If one paragraph is in a section, then all paragraphs must be in sections.	A section-name begins in area A and must be unique throughout the program.
Paragraph	A paragraph consists of a paragraph-name followed by a period and zero, one, or more sentences.	A paragraph-name begins in area A. The paragraph-name must be unique within the section in which it appears.
Sentence	A sentence consists of one or more statements and ends with a period.	The first sentence in a paragraph begins either on the same line as the paragraph-name or in area B of the next nonblank line that is not a comment line. Successive sentences begin in either area B of the same line as the preceding sentence, or in area B of the next nonblank line that is not a comment line. When sentences require more than one line, they can be continued on a subsequent line or lines.
Statement	A statement begins with a verb and contains a syntactically valid combination of other words and symbols.	Statements are positioned similarly to sentences. For the specific format of COBOL statements, refer to Sections 6 through 8.
Verb	A verb is a word that indicates the way data will be manipulated or the actions to be taken by the COBOL compiler or object program.	Verbs appear in area B. Verbs are a subset of the COBOL reserved words. A list of COBOL reserved words is provided in Appendix B. To see how verbs are used in COBOL statements, refer to Sections 6 through 8.

Statement Scope Terminators

Scope terminators delimit the scope of certain Procedure Division statements. The scope of statements can be terminated either explicitly or implicitly.

Explicit Terminators

Explicit scope terminators are phrases that occur at the end of some Procedure Division statements to indicate the end of the statement. The presence of such a terminator indicates that the statement contains no more phrases. The explicit scope terminators are as follows:

END-ABORT-TRANSACTION	END-FREE	END-RETURN
END-ADD	END-GENERATE	END-REWRITE
END-ASSIGN	END-IF	END-SAVE
END-BEGIN-TRANSACTION	END-INSERT	END-SEARCH
END-CALL	END-LOCK	END-SECURE
END-CANCEL-TRANSACTION	END-MODIFY	END-SET
END-CLOSE	END-MULTIPLY	END-START
END-COMPUTE	END-OF-PAGE	END-STORE
END-CREATE	END-OPEN	END-STRING
END-DELETE	END-PERFORM	END-SUBTRACT
END-DIVIDE	END-READ	END-TRANSACTION
END-END	END-RECEIVE	END-UNSTRING
END-EVALUATE	END-RECREATE	END-WRITE
END-FIND	END-REMOVE	

Example

```
MULTIPLY RATE BY TIME GIVING DISTANCE
       ON SIZE ERROR
            DISPLAY "DISTANCE ERROR"
             PERFORM INIT-PROCEDURE
  END-MULTIPLY.
```

In this example, the END-MULTIPLY phrase explicitly terminates the scope of the MULTIPLY statement.

Implicit Terminators

Implicit scope terminators refer to the period at the end of a sentence that terminates the scope of all previous statements not yet terminated.

```
READ GFILE INTO New-Record
      AT END
         CLOSE GFILE
         PERFORM Search-Para.
```

The period at the end of *Search-Para* implicitly terminates the scope of the READ, CLOSE, and PERFORM statements.

A statement contained in another statement is called a nested statement. The scope of nested statements can be implicitly terminated by the ELSE, WHEN, or NOT AT END phrase of the containing statement.

```
IF Dept-No = 0113
      MOVE "Sales Department" TO Print-Dept-Name
      IF Emp-Name = SPACES
           PERFORM Proc-2
      ELSE MOVE Emp-Name TO Print-Emp-Name
 ELSE PERFORM Read-Proc.
```

The phrase *ELSE PERFORM Read-Proc* implicitly terminates the scope of the two IF and two MOVE statements. When statements are nested, the period that terminates the sentence also implicitly terminates all nested statements.

Types of Statements and Sentences

Statements and sentences can be one of the following types:

- Imperative, indicating a specific unconditional action to be taken by the object program
- Conditional, specifying that the truth value of a condition is to be determined and that the subsequent action of the object program depends on this truth value
- Compiler-directing, causing the compiler to take a specific action during compilation
- Delimited scope, which is a statement that includes its explicit scope terminator

Imperative Statements and Sentences

An imperative statement

- Begins with an imperative verb
- Specifies an unconditional action to be taken by the object program
- Is a conditional statement delimited by its explicit scope terminator (delimited scope statement)
- Can consist of a sequence of imperative statements, each separated from the next by a separator

An imperative statement, when it appears as a variable in the format of a Procedure Division statement, refers to that sequence of consecutive imperative statements that must be ended by a period or by any phrase associated with a statement containing that imperative statement.

```
An imperative sentence is an imperative statement terminated by the
separator period.
```

Examples

```
MOVE LNAME TO LNAME-PR.
```

The above example moves the data from the identifier LNAME to the identifier LNAME-PR.

```
PERFORM PARA-1.
```

The above example causes the statements specified in paragraph PARA-1 to be executed first. Then the statements immediately following this PERFORM statement will be executed.

Conditional Statements and Sentences

A conditional statement specifies that the truth-value of a condition will be determined and that the subsequent action of the object program depends on this truth value.

A conditional statement can be any one of the following:

- An EVALUATE, IF, or SEARCH statement, or a RETURN statement that includes the AT END phrase
- A LOCK statement with the AT LOCKED phrase
- A PERFORM statement with the UNTIL phrase
- A READ statement that includes the AT END, NOT AT END, INVALID KEY, or NOT INVALID KEY phrase
- A WRITE statement that includes the INVALID KEY, NOT INVALID KEY, END-OF-PAGE, or NOT AT END-OF-PAGE phrase
- A START, REWRITE, or DELETE statement that includes the INVALID KEY or NOT INVALID KEY phrase
- An arithmetic statement (ADD, COMPUTE, DIVIDE, MULTIPLY, or SUBTRACT) that includes the ON SIZE ERROR or NOT ON SIZE ERROR phrase
- A STRING or UNSTRING statement that includes the ON OVERFLOW or NOT ON OVERFLOW phrase
- A CALL statement that includes the ON OVERFLOW, ON EXCEPTION, or NOT ON EXCEPTION phrase
- A SORT or MERGE statement that includes the ON ERROR phrase

Conditional expressions are fully explained in this section. COBOL statements, including those that use conditional expressions, are explained in Sections 6 through 8.

Examples

```
IF A > B PERFORM PARA-1
 ELSE PERFORM PARA-2.
```

These statements cause the statements in paragraph PARA-1 to be executed if the value of A is greater than the value of B. If A is not greater than B, the statements in PARA-2 are executed.

```
ADD X, Y TO Z ON SIZE ERROR PERFORM SIZE-ERROR-PROC.
```

This statement adds the value of X to the value of Y and stores the result in Z. If, after decimal point alignment, the absolute value of the result exceeds the largest value that can be contained in Z, a size-error condition occurs. When a size-error condition occurs, the procedures in SIZE-ERROR-PROC are executed, and the value of Z is not changed.

Compiler-Directing Statements and Sentences

A compiler-directing statement, which consists of a compiler-directing verb and its operands, causes the compiler to take a specific action during compilation. The compiler-directing verbs are COPY, REPLACE, and USE.

A compiler-directing sentence is a single compiler-directing statement terminated by a period and followed by a space.

Examples

```
COPY ERROR-REC-PROC OF COB85.
```

This statement directs the compiler to copy the ERROR-REC-PROC portion of the library program COB85 into the source program that contains this COPY statement. The COPY statement is processed before the resulting source program is processed.

```
FILE-ERROR SECTION.
FILE1-ERROR PARA.
  USE AFTER STANDARD ERROR PROCEDURE ON FILE1.
  MOVE ERROR-MSG TO ERROR1
  WRITE PRINT-ERROR-REC.
```

This example directs the compiler to follow the error-handling procedures associated with the USE statement after it completes the standard error routine of the system.

Delimited Scope Statements

Delimited scope statements are statements that include their explicit scope terminators.

Examples

```
ADD A TO B GIVING C END-ADD
```

This example adds the value of A to the value of B, and stores the result in C. The END-ADD phrase explicitly terminates the scope of the ADD statement.

When a delimited scope statement is nested in another delimited scope statement with the same verb, each explicit scope terminator terminates the statement begun by the most recently preceding, and as yet unpaired, occurrence of that verb.

```
ADD A TO B GIVING C
      SIZE ERROR
      ADD A TO B GIVING OVER-SIZE END-ADD
      PERFORM SIZE-ERR-PARA
 END-ADD
```

This adds the value of A to the value of B, and stores the result in C. If the result exceeds the size specified for C, a size error occurs, and the result of A added to B is stored in OVER-SIZE and the procedures in SIZE-ERR-PARA are executed.

The first END-ADD in this series of statements terminates the scope of the ADD A TO B GIVING OVER-SIZE statement. The second END-ADD terminates the scope of the ADD A TO B GIVING C statement.

Categories of Verbs

Table 5–2 categorizes the COBOL verbs according to their functions. For a detailed discussion of the specific Procedure Division verbs with examples of syntax, refer to Sections 6 through 8. Refer to Volume 2: Product Interfaces for COBOL verbs intended for use with various products.

Table 5–2. Categories of COBOL Verbs

Category	Verb	Function
Arithmetic	ADD	Sums two or more numeric operands and stores the result.
	COMPUTE	Calculates an arithmetic expression and stores the result.
	DIVIDE	Divides a numeric operand into another operand and stores the quotient and remainder.
	INSPECT (TALLYING)	Searches for and tallies the occurrences of specified characters in a data item.
	MULTIPLY	Multiplies numeric operands and stores the result.
	SUBTRACT	Subtracts one or the sum of two or more numeric operands from one or more operands and stores the result.
Compiler-Directing	COPY	Incorporates text from a library program into a COBOL source program.
	REPLACE	Replaces source program text.
	USE	Specifies procedures for handling input- output errors in addition to the standard procedures provided by the input-output control system.

Table 5–2. Categories of COBOL Verbs

Category	Verb	Function
Conditional	ADD (ON SIZE ERROR, NOT ON SIZE ERROR)	Sums two or more numeric operands and stores the result. If a size-error conditions occurs, specific procedures are followed
	CALL (ON OVERFLOW, ON EXCEPTION, NOT ON EXCEPTION)	Transfers control from one program to another during program execution. If the called program is not present, specified procedures are followed.
	COMPUTE (ON SIZE ERROR, NOT ON SIZE ERROR)	Calculates an arithmetic expression and stores the result. If a size-error condition occurs, specified procedures are followed.
	DELETE (INVALID KEY, NOT INVALID KEY)	Removes a logical record from a relative or indexed file. If the file does not have the record indicated by the key, specified procedures are followed.
	DIVIDE (ON SIZE ERROR, NOT ON SIZE ERROR)	Divides one numeric operand into another and stores the quotient and remainder. If a size-error condition occurs, specified procedures are followed.
	EVALUATE	Causes multiple conditions to be evaluated. Subsequent action of the object program depends on the results of the evaluations.
	IF	Evaluates a condition. Subsequent action of the object program depends on whether the value of the condition is true or false.
	LOCK (with AT LOCKED)	Locks a common data storage area so that related processes cannot access it. The AT LOCKED phrase specifies statements to be performed if the storage area is already locked.
	MULTIPLY (ON SIZE ERROR, NOT ON SIZE ERROR)	Multiplies numeric operands and stores the result. If a size-error condition occurs, specified procedures are followed.
	PERFORM (UNTIL)	Transfers control to the specified subroutine until the condition in the UNTIL phrase is true.

Table 5–2. Categories of COBOL Verbs

Category	Verb	Function
Conditional	READ (AT END, NOT AT END, INVALID KEY, NOT INVALID KEY)	For sequential access, it makes the next logical record from a sequential file available. For random access, it makes a specific record from a mass-storage file available. If the end of the file is reached or if the file does not contain the indicated key, specified procedures are followed.
	RETURN (AT END, NOT AT END)	Causes the next record in a sort-merge file to be read. If the end of the file is reached, specified procedures are followed.
	REWRITE (INVALID KEY, NOT INVALID KEY)	Logically replaces a record in a mass-storage file. If the file does not contain the record identified by the indicated key, specified procedures are followed.
	SEARCH	Searches a table for a table element that satisfies a specified condition and adjusts the associated index-name to point to that table element.
	SORT (ON ERROR)	Sorts the contents of one or more input files. If an error condition is encountered, specific action can be performed.
	START (INVALID KEY, NOT INVALID KEY)	Provides a logical position for a relative or indexed file when the file will be read sequentially. If the file does not contain the indicated key, specified procedures are followed.
	STRING (ON OVERFLOW, NOT ON OVERFLOW)	Provides juxtaposition of the partial or complete contents of one or more data items into a single data item.
	SUBTRACT (ON SIZE ERROR, NOT ON SIZE ERROR)	Subtracts one or the sum of two or more numeric operands from one or more operands and stores the result. If a size-error condition occurs, specified procedures are followed.

Table 5–2. Categories of COBOL Verbs

Category	Verb	Function
Conditional	UNSTRING (ON OVERFLOW, NOT ON OVERFLOW)	Causes contiguous data items in a sending field to be separated and placed into multiple receiving fields. If the value of the pointer is less than 1 or greater than the sending field, or if all the receiving fields have been acted upon and the sending field contains characters that have not been examined, specified procedures are followed.
	WRITE (INVALID KEY, NOT INVALID KEY, END-OF-PAGE, NOT END-OF-PAGE)	Releases a logical record for an output or input-output file. If the file does not contain the indicated key, or if an end-of-page condition exists, specified procedures are followed.
	ACCEPT (DATE, DAY, DAY-OF-WEEK, TIME)	Makes low-volume data available to a specified data item. Data from the DATE, DAY, DAY-OF-WEEK, or TIME register is moved to the specified item.
	INITIALIZE	Sets selected types of data fields to predetermined values.
	INSPECT (REPLACING, CONVERTING)	Searches for and replaces occurrences of specified characters in a data item.
	MOVE	Transfers data, according to the rules of editing, to one or more data areas.
	STRING	Provides juxtaposition of the partial or complete contents or one or more data items into a single data item.
	UNSTRING	Causes contiguous data items in a sending field to be separated and placed into multiple receiving fields.
	ACCEPT	Makes low-volume data available to a specified data item from an ODT.
	ADD (without ON SIZE ERROR, NOT ON SIZE ERROR)	Sums two or more numeric operands and stores the result.
Imperative	ALLOW	Readies an interrupt procedure for execution when its associated events are activated.
	ATTACH	Associates an interrupt procedure with an event.

Table 5–2. Categories of COBOL Verbs

Category	Verb	Function
Imperative	ALTER	Modifies a predetermined sequence of operations.
	CALL (without ON OVERFLOW, ON EXCEPTION, NOT ON EXCEPTION)	Transfers control from one program to another during program execution.
	CANCEL	Ensures that the next time a program referenced in a CALL statement is called, the program will be in its initial state.
	CAUSE	Initiates specified events.
	CHANGE	Modifies a file, task, or library attribute.
	CLOSE	Ends the processing of a file and specifies the disposition of the file and the device to which the file is assigned.
	COMPUTE (without ON SIZE ERROR, NOT ON SIZE ERROR)	Calculates an arithmetic expression and stores the result.
	CONTINUE task-variable	Reinstates a synchronous, dependent process that was previously initiated by a CALL statement from another program and then exited by an EXIT PROGRAM statement.
	COPY	Incorporates text from a library program into the program that contains the COPY statement.
	DELETE (without INVALID KEY, NOT INVALID KEY)	Removes a logical record from a relative or indexed file.
	DETACH	Dissociates an interrupt procedure from an event or a task variable from a task.
	DISALLOW	Prevents an interrupt procedure from executing when its associated event is activated.
	DISPLAY	Causes low-volume data to be transferred to an ODT.
	DIVIDE (without ON SIZE ERROR, NOT ON SIZE ERROR)	Divides one numeric operand into another and stores the quotient and remainder.

Table 5–2. Categories of COBOL Verbs

Category	Verb	Function
Imperative	EXIT	Indicates a logical end for a series of sections or paragraphs referenced by a PERFORM statement.
	EXIT PROGRAM	Indicates the logical end of a called program.
	GO	Unconditionally transfers control from one procedure to another. Control is not implicitly returned to the statement following the GO statement.
	INITIALIZE	Sets selected types of data fields to predetermined values.
	INSPECT	Searches for and can tally or replace specified characters in a data item.
	LOCK (without AT LOCKED)	Locks a common data storage area so that related processes cannot access it
	MERGE	Merges two or more identically sequenced files on a set of specified keys. The merged records then become available to an output procedure or output file.
	MOVE	Transfers data, according to the rules of editing, to one or more data areas.
	MULTIPLY (without ON SIZE ERROR, NOT ON SIZE ERROR)	Multiplies numeric operands and stores the result.
	OPEN	Makes a file available for processing.
	PERFORM	Unconditionally transfers control to the specified subroutine and returns control to the statement following the PERFORM statement.
	PROCESS	Initiates a separately compiled program as an asynchronous, dependent process
	READ (without AT END or INVALID KEY, NOT AT END, NOT INVALID KEY)	For sequential access, it makes the next logical record from a sequential file available. For random access, it makes a specific record from a mass-storage file available.

Table 5–2. Categories of COBOL Verbs

Category	Verb	Function
	RELEASE	Transfers records to the initial phase of a sort operation and writes records to a sort file.
Imperative	REPLACE	Replaces source program text.
	RESET	Turns off specified events.
	REWRITE (without INVALID KEY, NOT INVALID KEY)	Logically replaces a record in a mass-storage file. If the file does not contain the indicated key, specified procedures are followed.
	RUN	Initiates a separately compiled program as an asynchronous, independent process.
	SEARCH (without AT END or WHEN)	Searches a table for a table element that satisfies a specified condition and adjusts the associated index-name to point to that table element.
	SEEK	Repositions a file to a specified record.
	SET	Establishes reference points for table handling operations by setting indexes associated with table elements; can alter the value of external switches; and can alter the value of the conditional variables.
	SORT	Sequences a file on a set of specified keys and makes the sort file available to output procedures or output files
	START (without INVALID KEY, NOT INVALID KEY)	Provides a logical position for a relative or indexed file when the file will be read sequentially. If the file does not contain the indicated key, specified procedures are followed.
	STOP	Suspends the execution of a program either permanently or temporarily.
	STRING (without ON OVERFLOW, NOT ON OVERFLOW)	Provides juxtaposition of the partial or complete contents of one or more data items into a single data item.
	SUBTRACT (without SIZE ERROR, NOT ON SIZE ERROR)	Subtracts one or the sum of two or more numeric operands from one or more operands and stores the result.

Table 5–2. Categories of COBOL Verbs

Category	Verb	Function
	UNLOCK	Frees a common storage area that was previously restricted by a LOCK statement.
	UNSTRING (without ON OVERFLOW, NOT ON OVERFLOW)	Causes contiguous data items in a sending field to be separated and placed into multiple receiving fields.
Imperative	WAIT	Suspends program execution for a specified period of time.
	WRITE (without INVALID KEY or END-OF-PAGE, NOT INVALID KEY, NOT AT END-OF-PAGE)	Releases a logical record for an output of input-output file.
Input-Output	ACCEPT (identifier)	Transfers low-volume data from an ODT to a specified data item.
	CLOSE	Ends the processing of a file, specifies the disposition of the file and of the device to which the file is assigned.
	DELETE	Removes a logical record from a relative or indexed file.
	DISPLAY	Causes low-volume data to be transferred to an ODT.
	OPEN	Makes a file available for processing.
	READ	For sequential access, it makes the next logical record from a sequential file available. For random access, it makes a specific record from a mass-storage file available.
	REWRITE	Logically replaces a record in a mass-storage file.
	SEEK	Repositions a file to a specified record.
	START	Provides a logical position for a relative or indexed file when the file will be read sequentially.
	STOP (literal)	Suspends the execution of a program. The literal is communicated to the operator, and execution continues with the next executable statement in the program.

Table 5–2. Categories of COBOL Verbs

Category	Verb	Function
	WRITE	Releases a logical record for an output or input-output file.
Interprogram Communication	CALL	Transfers control from one program to another during program execution.
	CANCEL	Ensures that the next time a program referenced in a CALL statement is called, the program will be in its initial state.
Interprogram Communication	EXIT PROGRAM	Indicates the logical end of a called program.
No Operation	CONTINUE	Indicates that no executable statement is present.
	EXIT	Indicates a logical end to a series of sections or paragraphs referenced by a PERFORM statement.
Ordering	MERGE	Merges two or more identically sequenced files on a set of specified keys. The merged records then become available to an output procedure or output file.
	RELEASE	Transfers records to the initial phase of a sort operation and writes records to a sort file.
	RETURN	Causes the next record in a sort-merge file to be read.
	SORT	Sequences a file on a set of specified keys and makes the sort file available to output procedures or to output files.
Procedure Branching	ALTER	Modifies a GO TO statement to a different destination.
	CALL	Transfers control from one program to another during program execution.
	EXIT	Indicates a logical end to a series of sections or paragraphs referenced by a PERFORM statement.
	EXIT PROGRAM	Indicates the logical end of a called program.

Table 5–2. Categories of COBOL Verbs

Category	Verb	Function
	GO	Unconditionally transfers control to a procedure-name. Control is not implicitly returned to the statement following the GO statement.
	PERFORM	Unconditionally transfers control to the specified subroutine and returns control to the next statement following the PERFORM statement.
Scope Termination	END-ADD END-CALL END-COMPUTE END-DELETE END-DIVIDE END-EVALUATE END-IF END-LOCK END-MULTIPLY END-PERFORM END-READ END-RETURN END-REWRITE END-SEARCH END-START END-STRING END-SUBTRACT END-UNSTRING END-WRITE	Delimits the scope of its corresponding statement.
String Handling	INSPECT (REPLACING, CONVERTING, TALLYING)	Searches for and replaces the occurrences of specified characters in a data item.
	STRING	Provides juxtaposition of the partial or complete contents or one or more data items into a single data item.
	UNSTRING	Causes contiguous data items in a sending field to be separated and placed into multiple receiving fields.
Table Handling	SEARCH	Searches a table for a table element that satisfies a specified condition and adjusts the associated index-name to point to that table element.
	SET	Establishes reference points for table handling operations by setting indexes associated with table elements; can alter the value of external switches; and can alter the value of the conditional variables.

Arithmetic Expressions

An arithmetic expression contains combinations of identifiers and literals, which are separated by arithmetic operators and parentheses.

Identifiers and literals that appear in arithmetic expressions must represent either numeric elementary items or numeric literals on which arithmetic operations can be performed. Numeric literals cannot exceed 23 digits when used for arithmetic operations.

An arithmetic operator is a single character or a fixed two-character combination that indicates a particular arithmetic operation. Binary operators link two variables together, such as in A + B. Unary operators contain only one variable, such as *+A* or *–B*.

The binary arithmetic operators are represented by specific characters, as follows:

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation

The unary arithmetic operators are represented by specific characters, as follows:

Operator	Meaning
+	The effect of multiplication by the numeric literal +1
-	The effect of multiplication by the numeric literal –1

Note that binary arithmetic operators must be preceded and followed by a space.

Any arithmetic expression can be preceded by a unary operator.

Parentheses can be used in arithmetic expressions to change the order in which expressions are evaluated. Refer to "Precedence in Evaluation of Arithmetic Expressions" in this section for more information.

There must be a one-to-one correspondence between left and right parentheses of an arithmetic expression. That is, each left parenthesis is to the left of its corresponding right parenthesis.

Allowed Combinations of Elements

Table 5–3 shows the permissible combinations of identifiers and literals, arithmetic operators, and parentheses in an arithmetic expression.

Table 5–3. Combination of Symbols in Arithmetic Expressions

First Symbol	Second Symbol				
	Identifier or Literal	*** / ** – +**	**Unary + or –**	**(**	**)**
Variable	—	OK	—	—	OK
*** / ** + –**	OK	—	OK	OK	—
Unary + –	OK	—	——	OK	—
(	OK	—	OK	OK	—
)	—	OK	—	——	OK

An arithmetic expression can begin only with a left parenthesis, plus sign, minus sign, or an identifier or literal and can end only with a right parenthesis or an identifier or literal.

Examples

The following examples show valid arithmetic expressions.

```
COMPUTE X = Y ** 10
```

This first example causes the value of the identifier Y to be raised to the power of the numeric literal 10 and stored in X.

```
MULTIPLY -6 BY Z
```

This second example causes the value of Z to be multiplied by –6. The –6 is a combination of a unary and a numeric literal.

```
SUBTRACT Discount FROM Item-Cost GIVING Sale-Price
```

This third example causes identifier Discount to be subtracted from identifier Item-Cost and the result to be stored in identifier Sale-Price.

Precedence in Evaluation of Arithmetic Expressions

Arithmetic expressions are evaluated as follows:

- Expressions in parentheses are evaluated first:

```
COMPUTE A = B + (C - D)
```

  D is subtracted from C and the result is added to B.

- Within nested parentheses, the sequence of operations works outward from the innermost parentheses:

```
COMPUTE A = B + (C - (D + E))
```

  D and E are added first. The result is subtracted from C. This result is then added to B.

- When parentheses are not used, or parenthesized expressions are at the same hierarchical level, the sequence of execution is as follows:
  - Unary plus and minus (+, –)
  - Exponentiation (**)
  - Multiplication and division (*, /)
  - Addition and subtraction (+, –)

```
COMPUTE A = B ** .5 * C - D
```

  Exponentiation is performed first, which results in the square root of B. The result is multiplied by C, and then D is subtracted.

- Parentheses help either to eliminate ambiguities in logic where consecutive operations of the same level appear, or to modify the normal sequence of execution in expressions where it is necessary to have some deviation from the normal precedence:

```
COMPUTE A = B + (C - D) + (E - F)
```

  D is subtracted from C; then F is subtracted from E. B is added to the result of C – D, which is then added to the result of E – F.

- When the sequence of execution is not specified by parentheses, the order of execution of consecutive operations of the same hierarchical level is from left to right:

```
COMPUTE X = A + B / C + (D ** E) * F - G
```

This example would be interpreted as:

```
((A + (B / C)) + ((D ** E) * F)) - G
```

First the exponentiation (D ** E) is performed; then the multiplication of (D ** E) by F and division (B / C). Addition and subtraction are last, proceeding from left to right, so A is first added to (B / C). The sum is added to the next group. Finally, G is subtracted from the resulting value.

Rules for Exponentiation

The following rules apply to the evaluation of exponents in an arithmetic expression:

- You cannot have the value of an expression be zero raised to any power. For example, you cannot have A ** 2, if the value of A is 0.

  You cannot have the value of an expression be raised to a power of zero. For example, you cannot have A ** B, if the value of B is 0.

  Either case causes a size-error condition. Refer to "SIZE ERROR Phrase" in this section for more information.

- If the evaluation yields both a positive and a negative real number, the value returned as the result is the positive number. For example, in 4 ** .5, which calculates the square root of 4, the result is +2 or –2. The value returned and stored as the result is +2.

- If the result of an evaluation is not a real number, a size-error condition exists. For example, in A ** .5, if the value of A is a negative number, the result would be an imaginary number, and a size-error condition would exist.

- If an identifier to store the value of a result is not associated with an expression, an intermediate data item will be used. Intermediate data items are described in the following subsection.

- For all noninteger operands, the operand value is scaled into a double-precision, floating-point value as part of the preparation for the operation. Various arithmetic operations are performed during the operation itself. The result of exponentiation should always be regarded as an approximation. Performing the appropriate calculations directly within the program might produce more precise results than exponentiation, particularly when the exponent is known to be an integer.

Intermediate Data Item

An intermediate data item is a signed numeric data item provided by the compiler to contain the values developed during evaluation of an arithmetic expression.

The contents of the intermediate data item are then moved to the resultant-identifier, which is a user-defined data item that contains the result of the arithmetic operation, according to the rules for the MOVE statement. (Refer to "MOVE Statement" in Section 7 for detailed information.) Rounding is performed, if specified, and the size-error condition determined only during this move.

An intermediate data item occurs when an arithmetic statement involves several operations. Consider the following example:

```
COMPUTE X = A * B + C
```

This example requires an intermediate item to contain the value of A * B; then C is added to this intermediate item to produce the final result.

Limitations on Intermediate Data Items

The length of an intermediate data item is limited to 23 decimal digits. It contains the leading zeros and the leftmost digits of the value produced in the arithmetic operation.

If the size of the result exceeds the size of the intermediate data item, the result is truncated on the right to the size of the intermediate data item. The truncated value is used in the remainder of the computation.

Addition and subtraction operations have the following limitation: When the two operands aligned on their decimal points require a field longer than 23 decimal digits, truncation occurs before the operation is performed. The right end of the longer operand will be truncated, with the most significant 23 decimal digits saved.

Example

```
COMPUTE X = A + B
```

The value of A is 11.000000000123456789012, and the value of B is 11111.23.

A contains 23 digits, which is the maximum allowed for an intermediate data item. When A and B are aligned on their decimal points, the sum will contain more than 23 digits, because B contains five digits before the decimal point. The value of A is truncated on the right end by three digits before the addition will be performed. The value of A becomes 11.000000000123456789.

General Rules for Arithmetic Statements

The COBOL arithmetic statements are the

- ADD statement, which sums two or more numeric operands and stores the result
- COMPUTE statement, which calculates an arithmetic expression and stores the result
- DIVIDE statement, which divides a numeric operand into another and stores the quotient and remainder
- MULTIPLY statement, which multiplies numeric operands and stores the result
- SUBTRACT statement, which subtracts one or the sum of two or more numeric operands from one or more operands and stores the result

These statements have features in common regarding data descriptions, operand size limit, multiple results, the ROUNDED phrase, and the ON SIZE ERROR phrase.

When a REAL or a DOUBLE data item, or an intermediate result, is assigned to a DISPLAY, COMP, or BINARY data item in an Arithmetic statement, precision can be lost if the REAL or DOUBLE data item, or the intermediate result, represents a value that the machine must approximate. For more information, refer to "USAGE IS REAL" and "USAGE IS DOUBLE" in "Data Description Entry Format 1" in Section 4.

Data Descriptions

The data descriptions of the operands need not be the same; any necessary conversion and decimal point alignment is supplied throughout the calculation.

Data to be used in arithmetic operations, and data that is to be edited for a report, must be defined in the Data Division as numeric data.

Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. ADD-EXAMPLE.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  MATH-ITEMS.
      05 AA PIC 99.
      05 BB PIC 9V999.
01  OUT-ITEM.
      05 CC PIC ZZZZ.999.

PROCEDURE DIVISION.
  .
  .
  .
  ADD AA TO BB GIVING CC END-ADD.
  STOP RUN.
```

The data items AA, BB, and CC are described in the Data Division. The values for AA and BB are:

AA = 02

BB = 1.005

CC would be 3.005 after the calculation and decimal point alignment.

Operand Size Limit

The maximum size of each operand is 23 decimal digits. An operand that exceeds the size limit causes a syntax error.

Multiple Results in Arithmetic Statements

The ADD, COMPUTE, DIVIDE, MULTIPLY, and SUBTRACT statements can have multiple results. Such statements behave as though they had been written in the following way:

- A statement whose execution accesses all data items that are part of the initial evaluation of the statement, performs any necessary arithmetic or combining of these data items and stores the result of this operation in a temporary location. See the individual statements for the rules indicating which items are part of the initial evaluation.
- A sequence of statements whose execution transfers or combines the value in this temporary location with each single resulting data item. These statements are considered to be written in the same left-to-right sequence that the multiple results are specified.

For example, assume that temp is an intermediate data item provided by the compiler. The multiple results of the statement ADD a, b, c TO c, d (c), e are equivalent to:

```
ADD a, b, c GIVING temp
ADD temp TO c
ADD temp TO d (c)
ADD temp TO e
```

And the multiple results of the statement MULTIPLY a (i) BY i, a (i) are equivalent to:

```
MOVE a (i) TO temp
MULTIPLY temp BY i
MULTIPLY temp BY a (i)
```

ROUNDED Phrase

The ROUNDED phrase is used to round the result from an arithmetic operation so that it fits into its specified data item. The ROUNDED phrase increases the absolute value of the result from the COMPUTE statement by adding 1 to its low-order digit whenever the absolute value of the most significant digit of the excess is greater than or equal to 5. (The excess refers to the number of digits greater than the size of the data item in which the result is to be stored.)

Assume, for example, that you created a data item that can have two numbers after the decimal point. The result of the COMPUTE statement yields four numbers after the decimal point, ".5678". To fit into the defined data item, the ROUNDED phrase rounds ".5678" to ".57".

The ROUNDED phrase often requires a resultant-identifier to store the final results of the arithmetic operation. Truncation occurs if, after decimal point alignment, the number of places in the fraction of the result of an arithmetic operation is greater than the number of places provided for the fraction of the resultant-identifier. When using large BINARY EXTENDED integers in the calculation of the source expression, an overly large result can occur while scaling for decimal alignment and subsequent rounding. It is suggested that the ON SIZE ERROR phrase be used with rounded COMPUTE results to detect and deal with this possibility.

Example

```
Data Division.
 .
 .
 .
 01 IN-RECORD.
    05 hourly-wage PIC 999V99.
    05 no-of-hours PIC 999V99.
 01 OUT-RECORD.
    05 Gross-pay PIC ZZZZZ9.99.
 .
 .
 .
 PROCEDURE DIVISION.

    MULTIPLY hourly-wage BY no-of-hours GIVING Gross-pay ROUNDED.
```

The values are as follows:

hourly-wage = 7.50
no-of-hours = 45.25

The actual result of the multiplication is 339.3750, and the result is rounded prior to being stored in Gross-pay as 339.38.

When the low-order integer positions in a resultant-identifier are represented by the character *P* in the PICTURE clause, which implies an assumed decimal point location, rounding or truncation occurs relative to the rightmost integer position for which storage is allocated. Refer to "PICTURE Clause" in Section 4 for more information.

Example

```
05 C1 PIC 9PP.
05 C2 PIC 9PP.
 .
 .
 .
ADD A B Giving C1.
ADD A B Giving C2 ROUNDED.
```

The values are as follows:

A = 100
B = 50

The result of the calculation would be

C1 = 100

C2 = 200

SIZE ERROR Phrase

The SIZE ERROR phrase enables you to specify procedures to be executed when a size error condition exists. Size error conditions occur under the following circumstances:

- If, after decimal point alignment, the absolute value of a result exceeds the largest value that can be contained in the associated resultant-identifier.
- If, in the case noted above, the USAGE IS BINARY clause is specified for the resultant-identifier, and the value exceeds what can be contained in the resultant-identifier implied by the associated decimal PICTURE character-string.
- Division by zero. The execution of the program is abnormally terminated if the SIZE ERROR phrase is not specified.
- Violation of the rules for evaluation of exponentiation. This terminates the arithmetic operation.

The size error condition applies only to the final results of an arithmetic operation and does not apply to intermediate results, except in the DIVIDE and COMPUTE statements. An intermediate result of a COMPUTE statement can exceed the 23-digit length limit of the intermediate data item, but a size error condition does not result unless the final results of the COMPUTE statement exceed the limit of the resultant-identifier. Such a condition can produce unexpected results.

If the SIZE ERROR phrase is not used and a size error condition occurs, the value of the affected resultant-identifiers is undefined. Values of resultant-identifier(s) for which no size error condition occurs are unaffected by size errors that occur for the other resultant-identifier(s) during the execution of this operation.

If the SIZE ERROR phrase is used and a size error condition occurs, then the values of resultant-identifier(s) affected by the size errors are not altered. After completion of this operation, the imperative statement in the SIZE ERROR phrase is executed.

If the ROUNDED phrase is specified, rounding takes place before checking for a size error. When such a size error condition occurs, the subsequent action depends on whether or not the SIZE ERROR phrase is specified.

If you use the CORRESPONDING phrase in an ADD or SUBTRACT statement and any of the individual operations produces a size error condition, the imperative statement in the SIZE ERROR phrase is not executed until all of the individual additions and subtractions are completed.

OFFSET Function

OFFSET is a numeric function that returns a count of the number of characters that precede a data item in the logical record in which the data item is defined.

If data-name refers to a packed numeric data item that is not aligned on a character boundary, then the returned value is equal to the number of characters preceding the character with which data-name begins. If data-name is a record-name or a 77-level item, the value returned is 0. Data-name can be qualified.

Example

Given the following data declaration:

```
77  CURRENT-OFFSET    PIC 9999.
01  PERSONNEL-GRP.
     05 NAME               PIC X(16).
     05 EMPLOYEE-NUMBER    PIC 9999.
     05 JOB-TITLE          PIC X(16).
     05 DATES              PIC X(16).
     05 SUPP-FILE-NO       PIC 9999.
```

The following statement sets CURRENT-OFFSET to a value of 20:

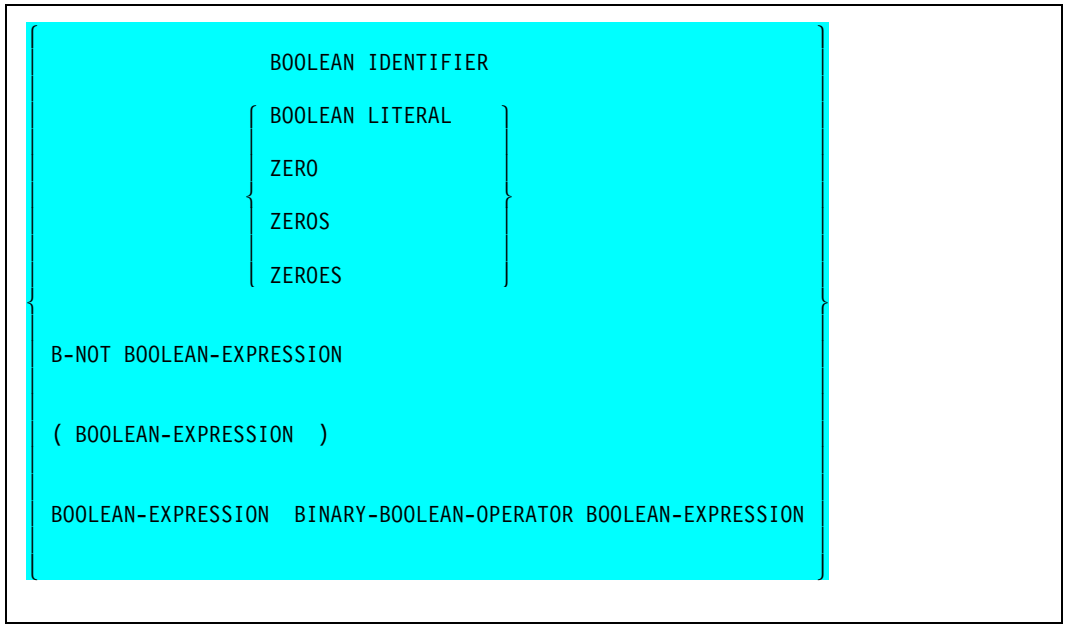
```
COMPUTE CURRENT-OFFSET = OFFSET (JOB-TITLE OF PERSONNEL-GRP).
```

Boolean Expressions

A Boolean expression can be any of the following:

- An identifier referencing a Boolean data item
- A Boolean literal
- The figurative constant ZERO (ZEROS, ZEROES)
- The figurative constant ALL literal, where literal is a Boolean literal
- A Boolean expression preceded by an unary Boolean operator
- Two Boolean expressions separated by a binary Boolean operator
- A Boolean expression enclosed in parentheses

General Format



Note: *The unary Boolean operator B-NOT cannot be immediately followed by another B-NOT.*

The following table shows the permissible combinations of operands, operators, and parentheses in a Boolean expression.

First Symbol	Second Symbol				
	Identifier or Literal	**B-AND B-OR B-XOR**	**B-NOT**	**(**	**)**
Identifier or literal	—	OK	—	—	—
B-AND, B-OR, B-XOR	OK	—	OK	OK	—
B-NOT	OK	—	—	OK	—
(	OK	—	OK	OK	—

Conditional Expressions

Conditional expressions contain conditions to be tested. The object program selects between alternate paths of control depending upon the truth value of the condition.

You can specify conditional expressions in the following statements:

Use a conditional expression in the . . .	To . . .
EVALUATE statement	Evaluate multiple conditions. Subsequent action of the object program depends on the results of the evaluations.
IF statement	Evaluate a single condition. Subsequent action of the object program depends on whether the value of the condition is TRUE or FALSE.
PERFORM statement (with the UNTIL phrase)	Transfer control to one or more procedures until a particular condition specified in the UNTIL phrase is TRUE.
SEARCH statement	Search a table for a table element that satisfies a specified condition and adjusts the associated index to indicate that table element.

Conditions associated with conditional expressions can be one of the following types:

- Simple conditions

  In a simple condition, a comparison is made. The value of the comparison is either TRUE or FALSE.

- Complex conditions

  In a complex condition, one or more logical operators (AND, OR, and NOT) act upon one or more conditions. The value of a complex condition is the truth value that results from the interaction of all the comparisons.

Conditions can be enclosed in any number of paired parentheses.

Simple Conditions

A simple condition has a truth-value of either TRUE or FALSE. The types of simple conditions that you can use are

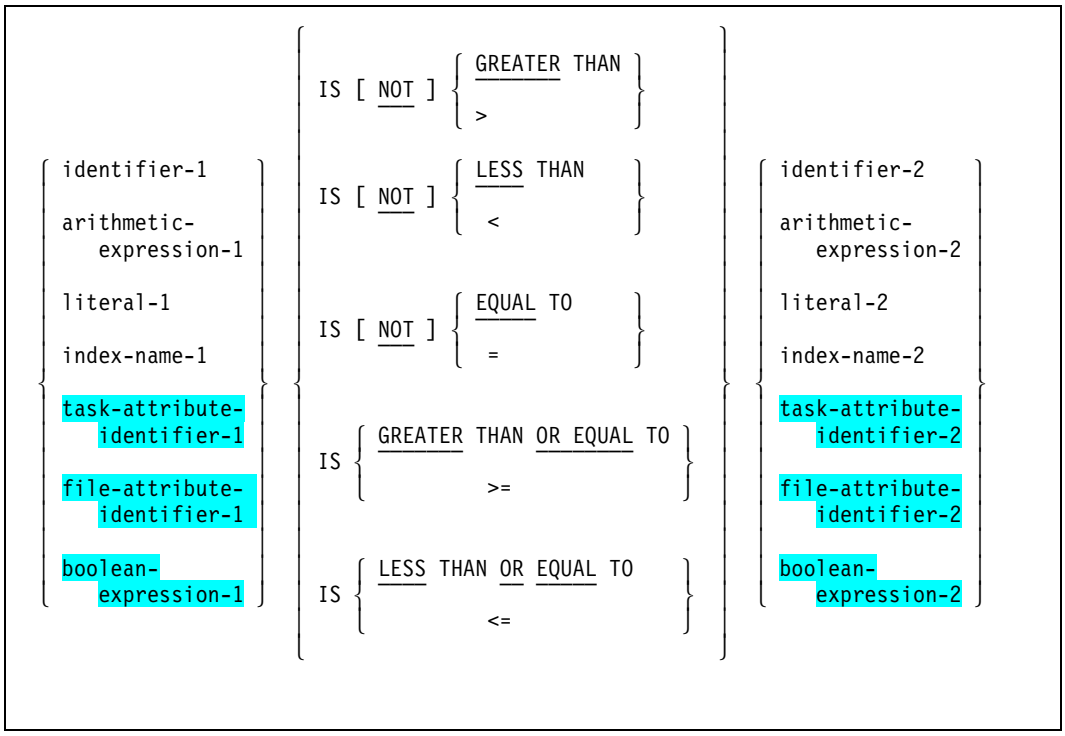
- Relation
- Class
- Condition-name
- Switch status
- Sign
- Event
- Boolean

Relation Conditions

A relation condition causes a comparison of two operands. The operands can be a literal, a task attribute, a data item referred to by an identifier, the value resulting from an arithmetic expression, or an index-name. A relation condition has a truth value of TRUE if the relation exists between the operands.

A relation condition must contain at least one reference to a variable. Otherwise, there is no question as to the truth-value of the condition, and no reason to have the condition. Consequently, you cannot compare one literal with another literal.

General Format for Relation Conditions



The first operand in the condition is called the *subject* of the condition.

identifier-1
literal-1
index-name-1

Identifiers, literals, and index-names are user-defined words. The hyphen (-) cannot appear as the first or last character in a user-defined word.

arithmetic-expression-1

Arithmetic-expression-1 refers to the result of an arithmetic operation.

task-attribute-identifier-1

Task-attribute-identifier-1 specifies one of the task attributes. For the syntax of the task attribute identifier, refer to Format 5 of the CHANGE statement in Section 6.

file-attribute-identifier-1

File-attribute-identifier-1 specifies one of the file attributes. For the syntax of the file attribute identifier, refer to Section 12.

boolean-expression-1

Boolean-expression-1 refers to the result of a Boolean expression.

IS [NOT] GREATER THAN
>
IS [NOT] LESS THAN
<
IS [NOT] EQUAL TO
=
IS GREATER THAN OR EQUAL TO
> =
IS LESS THAN OR EQUAL TO
< =

These relational operators specify the type of comparison to be made in a relation condition.

A space must precede and follow the first reserved word of the relational operator.

NOT, when used, determines the test to be true if the subject does not meet the specified relation. For example, IS NOT LESS THAN would be TRUE if the subject were equal to or greater than the object.

The second operand in the condition is called the *object* of the condition.

identifier-2
index-name-2
literal-2

Identifiers, literals, and index-names are user-defined words. The hyphen (-) cannot appear as the first or last character of a user-defined word.

arithmetic-expression-2

Arithmetic-expression-2 refers to the result of an arithmetic operation.

task-attribute-identifier-2

Task-attribute-identifier-2 is one of the task attributes, the value of which you are comparing to the value of the attribute specified by task-attribute-identifier-1. For the syntax of the task attribute identifier, refer to Format 5 of the CHANGE statement in Section 6.

file-attribute-identifier-2

File-attribute-identifier-2 is one of the file attributes, the value of which you are comparing to the value of the attribute specified by file-attribute-identifier-1. For the syntax of the file attribute identifier, refer to Section 12.

boolean-expression-2

Boolean-expression-2 refers to the result of a boolean expression.

Details

You can compare two numeric operands regardless of the formats indicated in their respective USAGE clauses. The USAGE clause in the Data Division specifies the format of a data item in computer storage, not the actual format of the numeric operand. For more information on the USAGE clause, refer to Section 4.

For comparisons that involve nonnumeric operands, index-names or index data items, the operands must have the same usage. If either of the operands is a group item, the nonnumeric comparison rules apply. Refer to "Comparison of Nonnumeric Operands" later in this section for more information.

A relation condition involving operands of class Boolean is a Boolean relation condition. An operand of class Boolean can be compared with another operand of class Boolean for equality (EQUAL and NOT EQUAL) only. Comparison of operands of class Boolean is a comparison of Boolean values, regardless of usage.

Examples

```
IF JOB-NO < 10 MOVE "ADMINISTRATIVE" TO CLASS.
```

This first example compares the identifier JOB-NO and the literal 10.

```
IF A + B >= C PERFORM C-PROC.
```

This second example compares the result of the arithmetic expression A + B and the identifier C.

```
PERFORM Year-End-Calc THRU Total-Proc VARYING Year FROM 1948 BY 1 UNTIL
Year = 1985 END-PERFORM.
```

This third example executes the procedures in Year-End-Calc through Total-Proc until the comparison of the index-name Year and the literal 1985 is TRUE.

Comparison of Numeric Operands

Numeric operands are compared according to their algebraic value, that is, the relation of the value to zero. Zero is a unique value regardless of the sign. That is, a plus or minus zero equals zero. The length of the literal or arithmetic expression operands, in number of digits represented, is not significant.

Numeric operands can be compared regardless of the formats described in their USAGE clauses. When needed, the numeric items are converted to their algebraic values. If the numeric item contains characters other than the digits 0 through 9, a conversion to valid numeric values occurs before the comparison is done.

Unsigned numeric operands are considered positive for purposes of comparison.

Comparisons involving long numeric operands are restricted to the following:

- Comparing a long numeric operand against 0 (zero) or an approximate figurative constant
- Comparing a long numeric operand for equality or nonequality against a long numeric operand of equal size and usage

Numeric Comparisons Involving HIGH-VALUES and LOW-VALUES

On Unisys V Series platforms, the rules and behavior for comparing these figurative constants to numeric data items is different than on Unisys A Series platforms. Tables 5-4 and 5–5 describe the results of moving HIGH-VALUES or LOW-VALUES to a data item of a certain type with a specific type of sign field.

Table 5–4. Numeric Comparisons Involving HIGH-VALUES

Data Value and Field Type	Platform			
	A Series COBOL74	A Series COBOL85	A Series COBOL85 with FIGCONST set	V Series COBOL74
COMP Unsigned	Syntax Error	Syntax Error	TRUE	Syntax Error
COMP TRAILING	Syntax Error	Syntax Error	Syntax Error	Syntax Error
COMP LEADING	Syntax Error	Syntax Error	Syntax Error	Syntax Error
COMP TRAILING SEPARATE	Syntax Error	Syntax Error	Syntax Error	Syntax Error
COMP LEADING SEPARATE	Syntax Error	Syntax Error	Syntax Error	Syntax Error
DISPLAY Unsigned	FALSE	FALSE	TRUE	TRUE
DISPLAY TRAILING	FALSE	FALSE	TRUE	TRUE
DISPLAY LEADING	FALSE	FALSE	TRUE	TRUE
DISPLAY TRAILING SEPARATE	FALSE	FALSE	TRUE	TRUE
DISPLAY LEADING SEPARATE	FALSE	FALSE	TRUE	FALSE

Table 5–5. Numeric Comparisons Involving LOW-VALUES

Data Value and Field Type	Platform			
	A Series COBOL74	A Series COBOL85	A Series COBOL85 with FIGCONST set	V Series COBOL74
COMP Unsigned	Syntax Error	Syntax Error	TRUE	Syntax Error
COMP TRAILING	Syntax Error	Syntax Error	Syntax Error	Syntax Error
COMP LEADING	Syntax Error	Syntax Error	Syntax Error	Syntax Error
COMP TRAILING SEPARATE	Syntax Error	Syntax Error	Syntax Error	Syntax Error
COMP LEADING SEPARATE	Syntax Error	Syntax Error	Syntax Error	Syntax Error
DISPLAY Unsigned	FALSE	FALSE	TRUE	TRUE
DISPLAY TRAILING	FALSE	FALSE	FALSE	FALSE
DISPLAY LEADING	FALSE	FALSE	FALSE	FALSE
DISPLAY TRAILING SEPARATE	FALSE	FALSE	FALSE	FALSE
DISPLAY LEADING SEPARATE	FALSE	FALSE	FALSE	FALSE

Comparison of Nonnumeric Operands

A comparison of nonnumeric operands, or one numeric and one nonnumeric operand, is made according to a specified collating sequence of characters. Refer to "OBJECT-COMPUTER Paragraph" in Section 3 for information on collating sequences.

The size of an operand is the total number of standard data format characters in the operand. A numeric and a nonnumeric operand can be compared only when their usage is the same, such as in a comparison of two operands whose usage is DISPLAY. If the numeric item contains characters other than the digits 0 through 9, no conversion to valid numeric values occurs before the comparison is done.

Compared operands need not be equal in size, because the comparison proceeds as though the shorter operand were extended on the right by enough spaces to make the operands of equal size.

Comparison of Undigit Literals and Numeric Operands

You can compare undigit literals with numeric operands on the basis of equality or nonequality. Comparisons involving greater than or less than operators are not allowed. Observe the following rules:

- Only unsigned integers can be compared with undigit literals.
- The PICTURE clause for the numeric operand cannot contain any editing characters or the characters S, V, or P.
- Binary and real items cannot be used.
- The undigit literal must be of the same length as the numeric data item with which it is being compared.
  - For packed items (COMP), there must be as many hex digits as specified in the PICTURE for the numeric data item.
  - For items with a usage of DISPLAY, there must be two hex digits for each number position in the item's definition.

Numeric Operands in Nonnumeric Comparisons

A numeric operand can be an integer data-item, non-integer data-item, or literal.

Example

```
01  Job-Data.
        05 Job-No PIC X(3).
        05 Job-Class PIC X(10).

Working-Storage Section.
01  Field-1 PIC 999 Value Is 200.
.
.
.
IF Job-No = Field-1 PERFORM 200-Proc.
```

In the IF statement, Job-No is a nonnumeric data item and Field-1 is a numeric field.

If the nonnumeric operand is an elementary data item or a nonnumeric literal, the numeric operand is treated as though it had been moved to an elementary alphanumeric data item of the same size as the numeric data item (in standard data format characters) ignoring the decimal point, if any. The contents of this alphanumeric data item are then compared to the nonnumeric operand. For detailed information on MOVE rules and data item descriptions, refer to "MOVE Statement" in Section 7 and the "PICTURE Clause" in Section 4.

In the preceding example, Job-No is an elementary alphanumeric data item that consists of three characters. For the comparison, the numeric operand Field-1 is also considered an alphanumeric data item of three characters.

If the nonnumeric operand is a group item, the numeric operand is treated as though it were moved to a group item of the same size as the numeric data item (in standard data format characters) ignoring the decimal point, if any. Then the contents of this group item were compared to the nonnumeric operand. Consider the following example, which uses data items from the preceding example:

```
IF Job-Data = Field-1 PERFORM Print-Proc.
```

Job-Data is a group item, and Field-1 will be considered as 13 alphanumeric characters in length.

A noninteger numeric operand cannot be compared to a nonnumeric operand.

If Job-No contains 102, the result of the comparison is true.

How Comparisons Are Made

The comparison proceeds by comparing characters in corresponding character positions. The evaluation starts from the high-order end and continues until either a pair of unequal characters is encountered or the low-order end of the operand is reached, whichever comes first.

The operands are equal if all pairs of characters compare equally when the low-order end is reached.

The first encountered pair of unequal characters is compared to determine their relative positions in the collating sequence. The operand that contains the character positioned higher in the collating sequence is recognized as the greater operand.

Comparisons Involving National Operands

A comparison of national operands is made according to a specified collating sequence of characters. For more information on collating sequences, refer to "OBJECT-COMPUTER Paragraph" in Section 3.

The size of an operand is the total number of national standard data format characters in the operand.

Compared operands need not be equal in size, because the comparison proceeds as though the shorter operand were extended on the right by enough space to make the operands of equal size.

Comparisons Involving Index-Names, Index Data Items

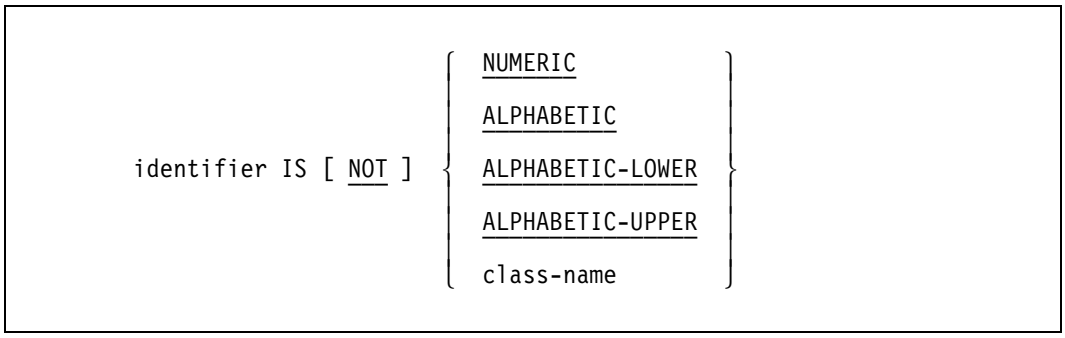
An index-name is a user-defined word that names an index associated with a specific table of data. Index-names are defined by an INDEXED BY phrase of the OCCURS clause. An index data item contains values associated with an index-name. It is an elementary data item described by a USAGE IS INDEX clause. For more information, refer to "OCCURS Clause" and "USAGE Clause" in Section 4.

For comparisons involving index-names and index data items, relation tests can be made only between:

- Two index-names. The result is the same as if the corresponding occurrence numbers were compared.
- An index-name and a data-item (other than an index data item) or literal. The occurrence number that corresponds to the value of the index name is compared to the data item or literal.
- An index data item and an index-name or another index data item. The actual values are compared without conversion, that is, according to their occurrence in the table. Refer to "Data-Names and Integers versus Index-Names" under "Table Handling" in this section for information on conversion.

Class Conditions

The class condition determines whether the operand is entirely numeric, alphabetic, contains only lowercase or only uppercase alphabetic characters, or contains only characters in a set specified by the CLASS phrase of the SPECIAL-NAMES paragraph of the Environment Division.



identifier

The identifier is a user-defined word that references a data item or a function that will be the object of the class test. Only alphanumeric functions can be used in class tests. For more information about identifiers, refer to Section 1.

NOT

NOT determines a test to be true if an operand is not of the specified class.

NUMERIC

This test classification determines whether the identifier consists entirely of the characters *0* through *9*, with or without an operational sign.

You cannot use the NUMERIC test with an identifier whose data description describes the identifier as alphabetic or with a group item composed of elementary items whose data description indicates the presence of operational sign(s).

If the data description of the identifier does not indicate the presence of an operational sign, the identifier is determined to be numeric only if the contents are numeric and an operational sign is not present.

If the data description of the identifier indicates the presence of an operational sign, the identifier is determined to be numeric only if the contents are numeric and a valid operational sign is present. Valid operational signs for data items described with the SIGN IS SEPARATE clause are the standard data format characters + and - .

For information on the position and representation of valid operational signs, refer to "PICTURE Clause" and "SIGN Clause" in Section 4.

ALPHABETIC

This test classification determines if the identifier consists entirely of any combination of characters *A* through *Z*, *a* through *z*, and spaces.

The ALPHABETIC test cannot be used with an identifier whose data description defines the item as numeric.

ALPHABETIC-LOWER

This test classification determines if the identifier consists entirely of the lowercase characters *a* through *z* and spaces.

The ALPHABETIC-LOWER test cannot be used with an identifier whose data description describes the item as numeric.

ALPHABETIC-UPPER

This test classification determines if the identifier consists entirely of the uppercase characters *A* through *Z* and spaces.

The ALPHABETIC-UPPER test cannot be used with an identifier whose data description describes the item as numeric.

Note: *For applications using the internationalization features, the data item being tested is determined to be alphabetic, alphabetic-upper, or alphabetic-lower only if the contents consist of any combination of the alphabetic characters in the truthset. To use a system collating sequence other than the characters A through Z, a through z, and the space, the program must use the ALPHABET FOR NATIONAL alphabet-name IS CCSVERSION phrase of the SPECIAL-NAMES paragraph.*

class-name

This test classification determines if the identifier consists only of the characters in the set specified by the CLASS phrase of the SPECIAL-NAMES paragraph of the Environment Division. For more information, refer to "SPECIAL-NAMES Paragraph" in Section 3.

The class-name test must not be used with an item whose data description describes the item as numeric.

Details

The USAGE of the operand used with the NUMERIC test must be DISPLAY or COMPUTATIONAL. The USAGE of the operand used with the ALPHABETIC tests must be DISPLAY. Refer to "USAGE Clause" in Section 4 for more information.

Examples

```
IF Item-Price IS NUMERIC PERFORM Price-Calc
      ELSE PERFORM Print-Error-Proc.
```

This first example tests the identifier Item-Price to see if it is entirely numeric. If it is, the procedures under Price-Calc are performed. If the test is not true, the procedures under Print-Error-Proc are performed.

```
SPECIAL-NAMES.
      CLASS A-to-K IS "A" THROUGH "K".
 .
 .
 .
      IF Element IS A-to-K GO TO Para-4.
```

This second example tests identifier Element to see if it contains only the characters specified in class-name A-to-K in the SPECIAL-NAMES paragraph.

Condition-Name Conditions

In a condition-name condition, a conditional variable is tested to determine whether or not its value is equal to one of the values associated with a condition-name.

```
condition-name
```

condition-name

This is a user-defined word that assigns a name to a subset of values that a conditional variable can assume.

A condition-name is defined as a level 88 entry in the Data Division.

If the condition-name is associated with a range or ranges of values, then the conditional variable is tested to determine whether or not its value falls in this range, including the end values.

Details

The rules for comparing a conditional variable with a condition-name value follow those specified for relation conditions. For more information, refer to "Relation Conditions" earlier in this section and to "Working-Storage Section" in Section 4.

The result of the test is true if one of the values corresponding to the condition-name equals the value of its associated conditional variable.

Example

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01  Dept-Code PIC X.
     88 Operations Values"A" Thru "D" .
     88 Programming Values "E" Thru "P" .
     88 Documentation Values "Q" Thru "T" .
     88 Personnel Values "U" Thru "Z" .
      .
      .
      .
PROCEDURE DIVISION.
     IF Programming PERFORM Prog-Para.
     IF NOT Personnel PERFORM Activity-Para.
```

This example tests conditional variable Dept-Code to see if it contains the range of values described for condition-name Programming. If it does, the procedure Prog-Para will be performed. The second statement tests Dept-Code for the values not described for condition-name Personnel, that is, if Dept-Code contains the values "A" through "T". If it does, then the procedure Activity-Para will be performed.

Switch-Status Conditions

A switch-status condition determines the ON or OFF status of a switch. The switch name and the ON or OFF value associated with the condition must be defined in the SPECIAL-NAMES paragraph of the Environment Division. For more information, refer to "SPECIAL-NAMES Paragraph" in Section 3.

```
condition-name
```

condition-name

This is a user-defined word assigned to the status of a switch or device.

Details

The result of the test is true if the switch is set to the specified position corresponding to the condition-name.

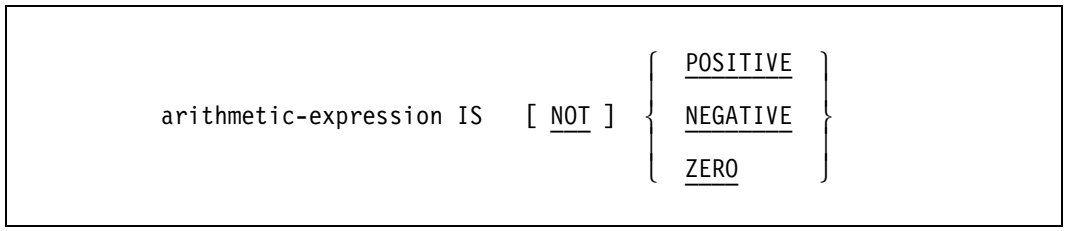
Example

```
ENVIRONMENT DIVISION.
 .
 .
 .
 SPECIAL-NAMES.
      SW5 ON STATUS IS SW5-ON.
 .
 .
 .
 PROCEDURE DIVISION.
      IF SW5-ON PERFORM SEARCH-PROC.
        .
        .
        .
```

This example tests switch SW5 for its ON or OFF status. If condition-name SW5-ON tests true, the procedures specified in SEARCH-PROC will be performed.

Sign Conditions

The sign condition determines whether or not the algebraic value of an arithmetic expression is less than, greater than, or equal to zero.



arithmetic-expression

This indicates an arithmetic operation and must contain at least one reference to a variable. For example, you could not have *COMPUTE A = 1 + 2*, which contains two literals.

NOT

NOT determines the test to be true if the arithmetic expression does not meet the specified sign condition.

POSITIVE

When you specify POSITIVE, the test is determined to be true if the arithmetic expression has a value greater than zero.

NEGATIVE

When you specify NEGATIVE, the test is determined to be true if the arithmetic expression has a value less than zero.

ZERO

When you specify ZERO, the test is determined to be true if the arithmetic expression has a value equal to zero.

Example

```
IF A / B Is Negative Add C To D Giving E.
```

This statement tests to see if the result of A divided by B has a value less than zero. If it does, C will be added to D and the value stored in E.

Event Condition

The event condition tests an event-valued file attribute, event-valued task attribute, or a data item declared with the USAGE IS EVENT clause to determine whether the event is TRUE or FALSE.

```
{ event-task-attribute  }
{ event-file-attribute  }
{ event-data-name       }
```

The use of an event-identifier as a condition returns the value TRUE when the event has been caused and not reset. It returns the value FALSE when the event is reset. For details, refer to the CAUSE statement in Section 6 and the RESET statement in Section 7.

Boolean Condition

A Boolean condition determines whether a Boolean expression is true or false.

General Format

```
[NOT] Boolean-expression-1
```

General Rules

- Boolean-expression-1 refers to Boolean items of length 1 only.
- Boolean-expression-1 evaluates true if the result of the expression is 1 and evaluates false if the result of the expression is 0.
- The condition NOT Boolean-expression-1 evaluates the reverse truth-value of Boolean-expression-1.

Negated Simple Conditions

The logical operator NOT negates a simple condition.

```
NOT simple-condition
```

NOT

This is a logical negator.

simple-condition

The simple-condition contains a comparison, the value of which is either TRUE or FALSE. The simple condition can be a relation, class, condition-name, switch-status, or sign condition.

Details

The negated simple condition produces the opposite truth value for a condition. Thus, the truth value of a negated simple condition is TRUE if the truth value of the condition is FALSE and FALSE if the truth value of the condition is TRUE.

Parentheses do not change the truth value of a negated condition.

Example

```
IF NOT A IS > = B
        MOVE ITEM-2 TO ITEM-3
 ELSE MOVE A TO ITEM-3.
```

This example tests the truth value of the relational condition "A is greater than or equal to B." If A is less than B, the condition is TRUE and ITEM-2 is moved to ITEM-3. If A is greater than or equal to B, the condition is FALSE, and A is moved to ITEM-3.

The statement *IF NOT A IS >= B* could be phrased as *IF A IS NOT >= B*. Both statements would cause the same results, but *IF NOT A IS >= B* is considered a negated condition and *IF A IS NOT >= B* is a relation condition that contains the optional word NOT.

Complex Conditions

A complex condition is a condition in which one or more logical operators act upon one or more conditions.

A logical operator is one of the reserved words AND, OR, or NOT. The reserved words AND and OR are called logical connectors; NOT is a logical negator. Logical operators must be preceded and followed by a space.

The logical operators and their meanings are as follows:

Logical Operator	Description	Effect on Condition
AND	Logical conjunction	The truth value is TRUE if both of the joined conditions are TRUE; FALSE if one or both of the joined conditions is FALSE.
OR	Logical inclusive OR	The truth value is TRUE if one or both of the included conditions are TRUE; FALSE if both included conditions are FALSE.
NOT	Logical negation or reversal of truth value	The truth value is TRUE if the condition is FALSE and FALSE if the condition is TRUE.

The truth value of a complex condition results from the interaction of all the stated logical operators on the individual truth values of simple conditions, or the intermediate truth values of conditions logically connected or logically negated.

Table 5–6 shows the truth table for complex conditions with logical operators. For example, the first line of Table 5–6 shows the following:

- A simple condition that results in the variable A is TRUE.
- A simple condition that results in the variable B is TRUE.
- If a complex condition uses the logical operator AND, and both A and B are TRUE, the result of that complex condition is TRUE.
- If a complex condition uses the logical operator OR, and both A and B are TRUE, the result of that complex condition is TRUE.
- If a complex condition uses the logical negator NOT to negate the simple condition that results in A, and both A and B are TRUE, the result of that complex condition is FALSE.

Table 5–6. Truth Table for Logical Operators

Values of Condition		Values of Complex Condition		
A	**B**	**A AND B**	**A OR B**	**NOT A**
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE

Example

```
EVALUATE TRUE ALSO TRUE
 WHEN B <= C AND D <= E ALSO E NOT = F OR 10
      COMPUTE A = C + 10
 WHEN OTHER PERFORM Para-3
 END-EVALUATE.
```

This example evaluates several conditions: The COMPUTE statement will be executed if B is equal to or less than C, D is equal to or less than E, and if E does not equal F or 10. For all other conditions, Para-3 will be executed.

Allowed Combinations of Elements

Complex conditions can include simple conditions, the logical operators AND and OR, the logical negator NOT, and parentheses.

Although parentheses are not needed when either AND or OR (but not both) is used exclusively in a combined condition, parentheses can be used to affect a final truth value when a mixture of AND, OR and NOT is used. Refer to "Precedence in Evaluation of Complex Conditions" later in this section for information on how parentheses affect a complex condition.

Table 5–7 shows the allowable combinations of conditions, logical operators, and parentheses.

Note that there must be a one-to-one correspondence between left and right parentheses.

Table 5–7. Combinations of Conditions, Logical Operators, and Parentheses

Given the Following Element:	First	Last	Element, When Not First, Can Be Immediately	
			Preceded Only By:	Followed Only By:
Simple-condition	Yes	Yes	OR, NOT, AND,)	OR, AND,)
OR or AND	No	No	Simple-condition,)	Simple-condition, NOT, (
NOT	Yes	No	OR, AND, (	Simple-condition, (
(	Yes	No	OR, NOT, AND, (	Simple-condition, NOT, (
)	No	Yes	Simple-condition,)	OR, AND,)

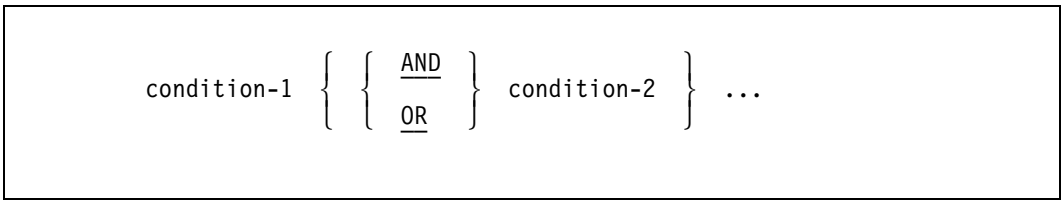
Example

```
SEARCH Tab1
      WHEN (Age IS < 45 OR Age is > 35)
      AND ( "V" is = Operations OR Dept)
      PERFORM Op-Proc
 END-SEARCH.
```

This example searches the table Tab1, and tests the "Age is less than or greater than" conditions. If one of these conditions is TRUE, Tab1 is searched to see if Operations or Dept equals "V". If either of these conditions is TRUE, the statements in Op-Proc will be executed.

Combined Condition Format

A combined condition results from connecting conditions with one of the logical operators AND or OR.



condition-1
condition-2

These elements can be any one of the following:

- A simple condition
- A negated simple condition
- A combined condition
- A negated combined condition; that is, the NOT logical operator followed by a combined condition enclosed in parentheses
- Combinations of the previous conditions that follow the rules summarized in Table 5–7.

AND

This is a logical connector. The value of the combined conditions is TRUE if both conditions are TRUE.

OR

This is a logical connector. The value of the combined conditions is TRUE even if only one of them is TRUE.

Example

```
IF SW5-ON AND Sale-Item IS ALPHABETIC
      MOVE Sale-Item To Report-Line-1
 ELSE IF Regular-Item IS ALPHABETIC
 AND NOT Regular-Item < = Sale-Item
      MOVE Regular-Item To Report-Line-1
 END-IF.
```

This example illustrates combined conditions. *IF SW5-ON* is a switch-status condition; *Sale-Item IS ALPHABETIC* and *Regular-Item IS ALPHABETIC* are class conditions; and *NOT Regular-Item <= Sale-Item* is a negated relative condition. This example uses the logical connector AND.

If the switch status of SW5 is TRUE and the Sale-Item consists entirely of alphabetic characters, the data in Sale-Item moves to Report-Line-1. If one or both of these conditions is FALSE, Regular-Item is tested to see if it consists entirely of alphabetic characters. If this tests TRUE, Regular-Item is tested to see if it is not less than or equal to Sale-Item. If this is also TRUE, the data in Regular-Item moves to Report-Line-1.

Abbreviated Combined Relation Conditions

Any simple or negated simple relation condition other than the first that appears in a combined conditional statement can be abbreviated.

You can abbreviate a combined conditional statement if the sequence of relation conditions:

- Has no parentheses
- Is combined by a logical connector (AND or OR)
- Contains identical subjects
- Contains identical subjects and relational operators

The sequence can be abbreviated as follows:

- You can omit identical subjects. For example:

  ```
  IF A = B AND = C
  ```

  This is equivalent to `IF A = B AND A = C`.

- You can omit identical subjects and relational operators. For example:

  ```
  IF A = B AND C
  ```

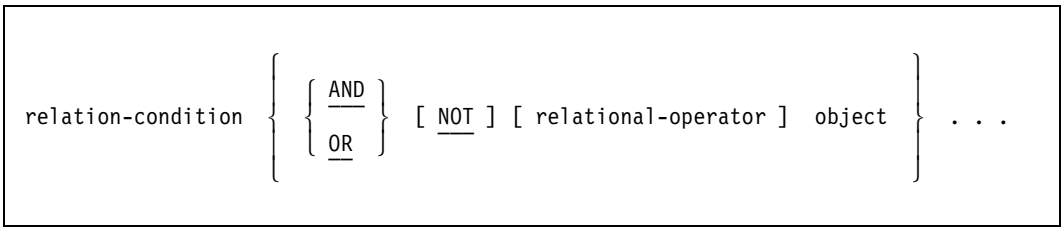
  This is equivalent to `IF A = B AND A = C`.

The use of abbreviated relations in conditional statement sequences that contain parentheses is a Unisys extension. When a condition is abbreviated, the subject and the relational operator are assumed, and are declared immediately before the object for which the assumption was made. This situation occurs even if the expression in which the relational operator, or the subject (or both), is embedded within a parenthetical expression. For example, the statement:

```
IF (A = 1) AND (B-2) AND (C-3) AND (D-4) THEN...
```

is expanded to:

```
IF (((A = 1) AND (A = B-2))) AND
          (A = (C-3))) AND (A = (D-4))) THEN...
```



relation-condition

This causes a comparison of two operands, each of which can be a data item referred to by an identifier, a literal, the value resulting from an arithmetic expression, or an index-name.

AND

This is a logical connector. The truth value is TRUE if both of the joined conditions are TRUE; FALSE if one or both is FALSE.

OR

This is a logical connector. The truth value is TRUE if one or both of the included conditions are TRUE; FALSE if both are FALSE.

NOT

This can be part of a relational operator, if immediately followed by any of the following:

GREATER >
LESS <
EQUAL =
GREATER THAN OR EQUAL TO >=
LESS THAN OR EQUAL TO <=

NOT can also be a logical negator, which would make a negated relation condition.

When used, NOT causes the truth value to be TRUE if the condition is FALSE and FALSE if the condition is TRUE.

relational-operator

This specifies the type of comparison to be made in the relation condition and refers to the following:

GREATER >
LESS <
EQUAL =
GREATER THAN OR EQUAL TO >=
LESS THAN OR EQUAL TO <=

object

This refers to an operand in the comparison test.

Relation Conditions Details

In a sequence of relation conditions, you can use both forms of abbreviation; that is, omission of the subject or omission of the subject and relational-operator.

The effect of using abbreviations is as if the last preceding stated subject were inserted in place of the omitted subject, and the last stated relational-operator were inserted in place of the omitted relational-operator. The result of such implied insertion must comply with the rules of Table 5–7.

The insertion of an omitted subject and the relational-operator ends once a complete simple condition occurs in a complex condition.

Examples

The examples in the following table show abbreviated combined and negated combined relation conditions and their expanded equivalents.

Abbreviated Condition	Expanded Equivalent
a > b AND NOT < c OR d	((a > b AND (a NOT < c)) OR (a NOT < d))
a NOT EQUAL b OR c	(a NOT EQUAL b) OR (a NOT EQUAL c)
NOT a = b OR c	(NOT (a = b)) OR (a = c)
NOT (a GREATER b OR < c)	NOT ((a GREATER b) OR (a < c))
NOT (a NOT > b AND c AND NOT d)	NOT ((((a NOT > b) AND (a NOT > c)) AND (NOT (a NOT > d))))

Precedence in Evaluation of Complex Conditions

Parentheses specify the order in which individual conditions in complex conditions will be evaluated when you want to depart from the implied evaluation precedence.

Conditions in parentheses are evaluated first. Within nested parentheses, evaluation proceeds from the least inclusive condition to the most inclusive condition. An entire complex condition can be considered a nested structure of hierarchical levels with the entire complex condition itself being the most inclusive level. In this context, the evaluation of the conditions is an entire complex condition and proceeds according to the following rule, recursively applied where necessary.

For conditions without parentheses or those that contain parenthesized conditions at the same level of inclusiveness, the evaluation proceeds in the following implied hierarchical order to determine the final truth value:

1. Values for arithmetic expressions
2. Truth values for simple conditions in the following order:
    a. Relation (following the expansion of any abbreviated relation condition)
    b. Class
    c. condition-name
    d. Switch-status
    e. Sign
3. Truth values for negated simple conditions
4. Truth values for combined conditions. The AND logical operators are evaluated before OR logical operators.
5. Truth values for negated combined conditions
6. Truth values of consecutive operations of the same hierarchical level from left to right when the sequence of evaluation is not completely specified by parentheses

Example

```
IF CURRENT-MONTH AND DAY1 = 6 OR 12.
```

For this example, evaluation proceeds in the following order:

1. Truth value of the simple condition-name condition (CURRENT-MONTH is TRUE).
2. Truth value for the relation DAY1 = 6.
3. Truth value for combined conditions using AND (CURRENT-MONTH is TRUE AND DAY1 = 6).
4. Truth value for the relation DAY1 = 12.
5. Truth value for combined conditions using OR (the combined condition is the abbreviated combined relational, subject is DAY1, object is = 12, so DAY1 = 12).

Two possible conditions satisfy this example:

- CURRENT-MONTH is TRUE AND DAY1 = 6.
- DAY1 = 12.

Table Handling

Table handling refers to a way of organizing data items into a table, so that they can be accessed according to their position in the table.

You can create multidimensional variable-length tables, specify ascending or descending keys, and search a dimension of a table for an item that satisfies a condition.

Defining a Table

One way to describe repeating items that make up a table is to use a series of separate data description entries that have the same level-number and that are all subordinate to the same group item, as in the following example:

```
01 Seasons.
     05 Filler PIC X(6) VALUE IS "Spring".
     05 Filler PIC X(6) VALUE IS "Summer".
     05 Filler PIC X(6) VALUE IS "Autumn".
     05 Filler PIC X(6) VALUE IS "Winter".
```

However, this approach has several undesirable effects: it can generate long tables that are cumbersome to document; homogeneity of the table elements is not always apparent; and accessing an individual element of such a table is very difficult.

A better approach is to define a table by including an OCCURS clause in the data description entry of the item to be referenced. The OCCURS clause specifies that a data item is a table element that is to be repeated as many times as stated. The name and description of the data item apply to each repetition.

The following example shows a table defined by the item Mailing-Address. Twenty occurrences of Mailing-Address are specified by the OCCURS clause, and each occurrence consists of a name and an address.

```
01 Table-1.
    02 Mailing-Address OCCURS 20 TIMES.
       03 Name . . .
       03 Address . . .
```

The OCCURS clause enables you to designate either a fixed number of occurrences for a table element or a variable number of occurrences. For more information, refer to "OCCURS Clause" in Section 4.

Table Dimensions

You can define the dimensions of a table by subordinating a table element under multiple group items and including the OCCURS clause with the table element and the group items that contain the element. Theoretically, you can define up to 48 dimensions for any one table. However, due to current hardware limitations, the maximum number of practical dimensions that you can use is 19. This number is derived by using subscripts that range from 1 to 2 with element sizes of 1 byte or 1 hex unit for all dimensions. The practical number of dimensions you can declare decreases with larger subscript ranges or larger element sizes.

In the following example, the table defined by Department is not nested in any other table, so it is a one-dimensional table. The table defined by employee, however, is nested within one other table, Department, and is thus, a two-dimensional table.

```
01 Table-1.
    02 Department OCCURS 10 TIMES.
       03 Employee  OCCURS 50 TIMES.
         04 Name . . .
         04 Address . . .
```

Note that the preceding example has been assigned the name Table-1. You do not need to give a group name to the table unless you want to refer to the complete table as a group item. For example, neither of the one-dimensional tables shown in the following example has a group name:

```
01 Produce.
    02 Lettuce OCCURS 2 TO 5 TIMES DEPENDING ON Lettuce-Count . . .
    02 Cucumber . . .
    02 Apple OCCURS 10 TIMES . . .
```

INDEXED BY Option

The optional INDEXED BY phrase in the OCCURS clause enables you to refer to the subject of the entry (and subordinate entries) by a technique called indexing. Indexing is especially useful for operations such as table searches and the manipulation of specific items.

To use indexing, you assign one or more index-names to an item whose data description entry contains an OCCURS clause. An index-name must be a unique word in the program. The index associated with an index-name acts as a subscript, and its value corresponds to an occurrence number for the item to which the index-name is associated.

No separate entry describes the index associated with an index-name. At object time, the contents of the index correspond to an occurrence number for the table dimension with which the index is associated. In the following example, Apple-1 is an index-name associated with a table of data:

```
01 Produce.
     02 Lettuce . . .
     02 Apple OCCURS 10 TIMES INDEXED BY Apple-1 . . .
     02 Cucumber . . .
```

Initializing Tables

You can set initial values of tables either in the Data Division or through statements in the Procedure Division.

In the Data Division

You can specify the initial values of table elements in the Working-Storage Section of the Data Division as follows:

- The table can be described as a series of separate data description entries all subordinate to the same group item. Each data description entry can specify the value of an element, or part of an element, of the table:

```
WORKING-STORAGE SECTION.
01 Seasons.
     05 Filler PIC X(6) VALUE IS "Spring".
     05 Filler PIC X(6) VALUE IS "Summer".
     05 Filler PIC X(6) VALUE IS "Autumn".
     05 Filler PIC X(6) VALUE IS "Winter".
```

  In defining the record and its elements, any data description clause (USAGE, PICTURE, and so forth) can be used to complete the definition, where required. The previous example uses a picture clause of PIC X(6).

- The hierarchical structure of the table can be shown by a REDEFINES entry and its associated subordinate entries. The subordinate entries are repeated because of OCCURS clauses and must not contain VALUE clauses:

```
WORKING-STORAGE SECTION.
 01 Seasons PIC X(24)
      VALUE IS "SpringSummerAutumnWinter".
 01 Season-Table REDEFINES Seasons.
      02 Season PIC X(6) OCCURS 4 TIMES.
```

- All the dimensions of a table can be initialized by associating the VALUE clause with the description of the entry defining the entire table. The lower level entries will show the hierarchical structure of the table; lower level entries must not contain VALUE clauses:

```
03 Seasons PIC X(28)
     VALUE IS "Spring1Summer2Autumn3Winter4".
03 Season-Table REDEFINES Seasons OCCURS 4 TIMES.
     05 Name PIC X(6).
     05 Number PIC 9.
```

For detailed information on these clauses, refer to “REDEFINES Clause” and “VALUE Clause” in Section 4.

In the Procedure Division

The INITIALIZE statement sets the initial values for an entire table or for specific elements of a table. For detailed information on the syntax of this statement, refer to "INITIALIZE Statement" in Section 6.

If you are using the INDEXED BY option of the OCCURS clause, the initial value of an index at object time is undefined. You must initialize an index before you use it. The following Procedure Division statements can assign an initial value to an index:

- PERFORM statement with the VARYING phrase

  During execution, this statement augments the values referenced by one or more identifiers or index-names in an orderly fashion. For detailed information on the syntax of this statement, refer to Section 7.

- SEARCH statement with the ALL phrase

  This statement performs a binary search of a table and looks for a table element that satisfies the specified condition. It then adjusts the value of the associated index to indicate that table element. For detailed information on the syntax of the SEARCH statement, refer to Section 8.

- SET statement

  This statement assigns a value to an index or to index data items. For detailed information on the syntax of the SET statement, refer to Section 8.

References to Table Items

You can refer to table items by specifying the data-name with the occurrence number. The occurrence number is called a subscript.

Whenever you refer to a table element or a condition-name that is associated with a table element, the reference must indicate which occurrence of the element is intended. This rule applies except for the SEARCH statement.

In a one-dimensional table, the occurrence number of an element table provides complete information for you to access it. For tables of more than one dimension, you must supply an occurrence number for each dimension of the table.

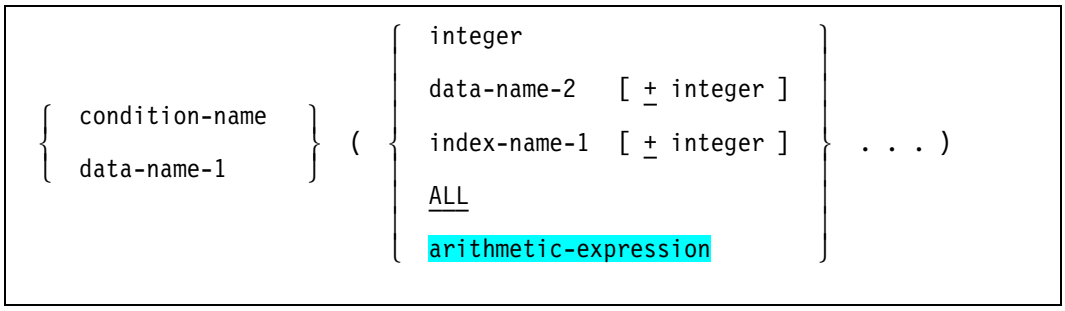
Consider the following example:

```
02 Apple OCCURS 10 TIMES . . .
     03 Granny-Smith . . .
     03 Delicious OCCURS 5 TIMES . . .
```

A reference to the fourth Apple or the fourth Granny-Smith would be complete. However, a reference to the fourth Delicious would be incomplete, because Delicious could be one of five possible occurrences in a two-dimensional table. To reference Delicious, you must define a specific occurrence of it; for example, the fourth Delicious in the fifth Apple.

Subscripting

You specify occurrence numbers by appending one or more subscripts to a condition-name or a data-name.



condition-name

This is a user-defined word that assigns a name to a subset of values. A conditional variable can assume these values.

A condition-name is defined as a level-number 88 entry in the Working-Storage Section of the Data Division and can be associated with a range of values.

data-name-1

This is a user-defined word and can consist of the characters *A* through *Z*, *a* through *z*, *0* through *9*, and the hyphen (-). The hyphen cannot appear as the first or last character.

( )

The left and right parentheses enclose the subscript.

integer
data-name-2
index-name-1

The integer represents the occurrence number. The lowest permissible occurrence number is 1. The highest permissible occurrence number is the maximum number of occurrences of the item as specified in the OCCURS clause.

You can also represent a subscript with a data-name or an index name: data-name-2 must refer to an integer numeric elementary item, and index-name-1 is an index-name associated with a table.

You can mix integers, data-names, and index-names in a single set of subscripts that refer to an individual occurrence in a multidimensional table.

+ integer
– integer

These can follow a data-name for relative subscripting or an index-name for relative indexing. The plus sign (+) or the minus sign (–) and an integer are used as an increment or decrement, respectively.

ALL

ALL can be used as a subscript only for data-names that are used as arguments to functions. The ALL subscript causes the argument to be repeated the number of times specified in the OCCURS clause. You cannot use ALL with a condition-name.

arithmetic-expression

An arithmetic-expression can be used as a subscript. In addition, integers, data-names, and arithmetic-expressions can be mixed in a single set of subscripts that is a reference to an individual occurrence in a multidimensional table. Arithmetic-expressions used in subscripts must be references to integer values.

Details

You write subscripts, which are enclosed in parentheses, immediately following any qualification for the name of the table element. The number of subscripts in such a reference must equal the number of dimensions in the table whose element is being referenced. That is, there must be a subscript for each OCCURS clause in the hierarchy containing the data-name. Moreover, the data-name itself must also have a subscript.

When the table element requires more than one subscript, write the subscripts in order of the outermost to the innermost table. A multi-dimensional table can be thought of as a series of nested tables. The outermost table is the major table; the innermost, the minor table. Therefore, you would write the subscripts from left to right in the order major, intermediate, and minor.

Subscript a reference to an item only if the item is either of the following:

- A table element
- An item or a condition-name in a table element

Example

The following example shows the data-description entries for a three-dimensional table definition.

```
01  CENSUS-TABLE.
    05  CONTINENT-TABLE                    OCCURS 8 TIMES.
        10  CONTINENT-NAME          PIC X(16).
        10  COUNTRY-TABLE                  OCCURS 15 TIMES.
            15  COUNTRY-NAME       PIC X(18).
            15  CITY-TABLE                 OCCURS 20 TIMES.
                20  CITY-NAME      PIC X(10)
                20  CITY-POPULATION  PIC X (12)
```

Related Information

The following table provides references for additional information related to this topic:

For information about . . .	Refer to . . .
The USAGE and OCCURS clauses	Section 4
The INITIALIZE Statement, PERFORM Statement, SEARCH Statement, and SET Statement	Sections 6, 7, and 8
Error handling for subscripts	The BOUNDS compiler control option in Section 15

Subscripts Using Integers or Data-Names

When an integer or data-name represents a subscript, it can refer to items in different tables. Elements in these tables are not required to be of the same size. The same integer or data-name can appear as the only subscript with one item and as one of two or more subscripts with another item.

Subscripts Using Index names

An index-name can refer to only the table with which it is associated through the INDEXED BY phrase of the OCCURS clause.

Relative indexing is an added option that you can use to refer to a table element or to an item in a table element. When the name of a table element is followed by a subscript of the form (index-name + or – integer), the occurrence number required to complete the reference is the same as if the index-name were set up or down by the integer through the SET statement before the reference. The use of relative indexing does not cause the object program to alter the value of the index.

Data-Names and Integers versus Index-Names

The primary difference between subscripting with an integer or data-name and subscripting with an index-name is in the method used to access the desired table entry.

At object time, a subscript is an integer that represents an occurrence in a table, that is, 1 for the first entry, 2 for the second, and so forth.

Since the subscript data-item contains only an occurrence number of the item to be accessed, the program must multiply the occurrence number by the length of a table entry to locate the desired item.

For example, consider a table defined as follows:

```
Y PIC 9(4) COMP OCCURS 10 TIMES INDEXED BY NDX.
```

Assume that a numeric data-item SUB has also been declared for use as a subscript.

To execute the statement *MOVE Y(SUB) TO X*, the program must first multiply the value in SUB by the length of the table element Y. (The multiplication is repeated each time you use SUB to access an item.) This gives the offset value of the desired element from the beginning of the table. Adding the offset value to the beginning address of the table (actually, to an address one element-length before the beginning of the table) gives the location of the item.

Literal subscripts, for example Y(5), are calculated once at compile time, so they involve the same code as nonsubscripted items.

An index-name contains an offset value to a table instead of a simple occurrence number. The statement *SET NDX TO data-name* causes the program to compute the offset value of the element whose position is given by data-name. *SET NDX UP (or DOWN) BY 1* causes the length of one table item to be added to (or subtracted from) the value in NDX.

In the statement *MOVE Y(NDX) TO X*, the location of Y is determined by adding the contents of NDX to the beginning address (actually, an address one element-length before the beginning) of the table. The index does not require the multiplication that is required for the subscript.

The index value is always carried in its final form as an offset value. Multiplication is performed, when applicable, at the time the index is set, not each time it is used.

If you are using indexing, which contains an offset value to a table, and then use a statement that requires an occurrence number, conversion takes place. The offset value will be converted into an occurrence number.

Index Data Items

The value of an index can be made accessible to an object program by storing the value in an index data item. Index data items are memory locations and are described in the program by a data description entry that contains a USAGE IS INDEX clause. The index value is moved to the index data-name by the execution of a SET statement.

Refer to "USAGE Clause" in Section 4 and to "SET Statement" in Section 8 for more information.

Sort and Merge Operations

The COBOL sort function orders the occurrence of records in one or more files. Sort functions are performed according to a set of specified keys that are contained in each record.

The COBOL merge function combines two or more identically ordered files according to specified keys.

Sorting

A sort file is a collection of records to be sorted by a SORT statement.

Sort files often require special processing, such as addition, deletion, creation, alteration, and editing of the individual records in the file. Special processing might be needed before or after the records are reordered by the sort. The COBOL sort function enables you to do this special processing and to specify whether it should occur before or after the sort.

A COBOL program can contain any number of sorts, each with its own input and output procedures. The sort function automatically causes execution of these procedures at the specified point.

In an input procedure, the RELEASE statement creates a sort file. When the input procedure has completed, those records processed by the RELEASE statement compose the sort file. This file is available only to the SORT statement.

Execution of the SORT statement arranges the entire set of records in the sort file according to the keys specified. The sorted records are made available from the sort file through the RETURN statement during execution of the output procedure.

The sort file does not have label procedures that the programmer can control. The rules for blocking and for allocation of internal storage are unique to the SORT statement. The RELEASE and RETURN statements imply nothing about buffer areas, blocks, or reels.

A sort file, then, is an internal file created from the input file by the RELEASE statement, processed by the SORT statement, and then made available to the output file by the RETURN statement. The sort file itself is referred to and accessed only by the SORT statement.

Merging

A merge file is a collection of records to be merged with another input file by a MERGE statement.

Merged files sometimes require special processing, such as addition, deletion, creation, alteration, and editing of the individual records in the file. The COBOL merge function enables you to execute output procedures as the merged output is created.

The merged records from the merge file are made available through the RETURN statement in the output procedure.

The merge file does not have label procedures that the programmer can control. The rules for blocking and for allocation of internal storage are unique to the MERGE statement. The RETURN statement implies nothing about buffer areas, blocks, or reels.

A merge file, then, is an internal file created from input files by combining them (MERGE statement) as the file is made available (RETURN statement) to the output file. The merge file itself is referred to and accessed only by the MERGE statement.

Sort and Merge Constructs

A sort or a merge file is named by a file control entry in the Environment Division and described by a sort-merge file description entry in the Data Division. A sort file is referred to in the Procedure Division by the SORT, RELEASE, and RETURN statements. A merge file is referred to by the MERGE and RETURN statements.

The following list shows the COBOL constructs to use with sort and merge operations.

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.

- Use Format 4 of the FILE-CONTROL paragraph. This file control entry declares the relevant physical attributes of a sort or a merge file.

  Each sort or merge file must be specified in the SELECT clause of the FILE-CONTROL paragraph and must have a sort-merge file description entry in the Data Division of the same program.

  Each sort or merge file described in the Data Division must be specified only once in the FILE-CONTROL paragraph.

  Since the file-name in the SELECT clause represents a sort or a merge file, only the ASSIGN clause can follow the file-name in the FILE-CONTROL paragraph.

  Use the ASSIGN clause to associate the file reference with a storage medium reference.

- Specify the memory area to be shared by the sort or merge files in the SAME RECORD/SORT/SORT-MERGE AREA clause of the I-O-CONTROL paragraph.

  The files referenced in the SAME RECORD/SORT/SORT-MERGE AREA clause are not required to have the same organization or access.

  Each file-name specified in the SAME RECORD/SORT/SORT-MERGE AREA clause must be specified in the FILE-CONTROL paragraph of the same program.

  For detailed information on the syntaxes, uses, and restrictions of these paragraphs, refer to “FILE-CONTROL Paragraph” and “I-O-CONTROL Paragraph” in Section 3.

DATA DIVISION.
FILE SECTION.

- Use sort-merge file description entry, Format 4, in the File Section; see Section 4 for details.

  Each sort or merge file specified in a sort-merge file description entry must also be specified in the SELECT clause of the FILE-CONTROL paragraph of the Environment Division of the same program.

  The sort-merge file description entry (the SD entry) furnishes information on the physical structure and record-names that pertain to a sort or a merge file.

  The FILE SECTION header is followed by a sort-merge file description entry that consists of a level indicator, a file-name, and a series of independent clauses.

  The clauses of an SD entry specify the size and the names of the data records associated with a sort file or a merge file.

- Record description entries are written immediately after the sort-merge file description entry. A record description consists of a set of data description entries that describe the characteristics of a particular record. Each data description entry consists of a level number followed by the data-name or FILLER clause, if specified, followed by a series of independent clauses as required. A record description can have a hierarchical structure. Therefore, the clauses used with an entry can vary considerably, depending upon whether or not the entry is followed by subordinate entries.

  The RECORD clause of the SD entry is the same as the RECORD clause in the FD entry for sequential files.

  The DATA RECORDS clause is the same as the DATA RECORDS clause in the FD entry for sequential files. The DATA RECORDS clause is an obsolete element in Standard COBOL and will be deleted from the next revision of Standard COBOL.

- Refer to "File Section" in Section 4 for detailed information on syntax, usage, and restrictions.

PROCEDURE DIVISION.

- Use the SORT statement to sequentially order a file on a set of specified keys and to make the sort file available to output procedures or an output file.
- Use the RELEASE statement to transfer records to the initial phase of a SORT operation and to write records to a sort file.
- Use the RETURN statement to obtain sorted or merged records from the final phase of a SORT or MERGE operation and to read records from a sort file.
- Use the MERGE statement to combine two or more identically sequenced files on a specified key.

Refer to "RELEASE Statement," "RETURN Statement," and "MERGE Statement" in Section 7 and to "SORT Statement" in Section 8 for detailed information and syntax.

Example

The following example shows the COBOL constructs used in sort and merge operations.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SORTMERGE-EXAMPLE.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
     SELECT DATA-CAPTURE-1      ASSIGN TO DISK.
     SELECT DATA-CAPTURE-2      ASSIGN TO DISK.
     SELECT DATA-CAPTURE-3      ASSIGN TO DISK.
     SELECT MASTER-FILE         ASSIGN TO DISK.
     SELECT SHOW                ASSIGN TO PRINTER.
     SELECT SORT-FILE           ASSIGN TO SORT.
     SELECT MERGE-FILE          ASSIGN TO MERGE.
DATA DIVISION.
FILE SECTION.
FD  DATA-CAPTURE-1.
01  D-RECORD-1.
    03 FILLER            PIC X(180).
FD  DATA-CAPTURE-2.
01  D-RECORD-2.
    03 FILLER            PIC X(180).
FD  DATA-CAPTURE-3.
01  D-RECORD-3.
    03 FILLER            PIC X(180).
FD  MASTER-FILE.
01  M-RECORD.
    03 FILLER            PIC X(180).
FD  SHOW.
01  OUT-RECORD.
    03 FILLER            PIC X(5).
    03 PAYNO             PIC 9(5).
    03 FILLER            PIC X(5).
    03 DEPTNO            PIC X(10).
    03 FILLER            PIC X(107).
SD  SORT-FILE.
01  SORT-RECORD.
    03 FILLER            PIC X(10).
    03 ACC-NO            PIC 9(6).
    03 FILLER            PIC X(10).
    03 QTE               PIC 9(4).
    03 FILLER            PIC X(10).
    03 PRICE             PIC 9(10).
    03 FILLER            PIC X(130).
SD  MERGE-FILE.
01  MERGE-RECORD.
    03 FILLER            PIC X(20).
    03 PAY-NO            PIC 9(5).
    03 FILLER            PIC X(50).
    03 DEPT-NO           PIC X(10).
```

```
    03 FILLER            PIC X(95).
PROCEDURE DIVISION.
BEGIN-SORT.
    SORT SORT-FILE
         ON ASCENDING KEY ACC-NO
         INPUT PROCEDURE IS PROC-1 THRU END-1
         OUTPUT PROCEDURE IS PROC-2 THRU END-2.
    GO TO BEGIN-MERGE.
PROC-1.
    OPEN INPUT DATA-CAPTURE-1.
PROC-1A.
    READ DATA-CAPTURE-1 AT END GO TO END-1.
    RELEASE SORT-RECORD.
    GO TO PROC-1A.
END-1.
    CLOSE DATA-CAPTURE-1.
PROC-2.
    OPEN OUTPUT DATA-CAPTURE-2.
PROC-2A.
    RETURN SORT-FILE AT END GO TO END-2.
    MOVE SORT-RECORD TO D-RECORD-2.
    WRITE D-RECORD-2.
    GO TO PROC-2A.
END-2.
    CLOSE DATA-CAPTURE-2.
BEGIN-MERGE.
    OPEN OUTPUT SHOW.
    MERGE MERGE-FILE ON ASCENDING KEY PAY-NO
         USING MASTER-FILE, DATA-CAPTURE-3
         OUTPUT PROCEDURE IS OUT-1.
OUT-1.
    RETURN MERGE-FILE
         AT END GO TO FINISH-1.
    PERFORM WRITE-PROC.
WRITE-PROC.
    MOVE SPACES TO OUT-RECORD.
    MOVE PAY-NO TO PAYNO.
    MOVE DEPT-NO TO DEPTNO.
    WRITE OUT-RECORD.
FINISH-1.
    CLOSE MERGE-FILE.
    CLOSE SHOW.
    STOP RUN.
```

In the Environment Division, SORT-FILE is declared as a sort file, and MERGE-FILE is declared as a merge file.

SORT-FILE and MERGE-FILE have SD entries in the Data Division.

Data-Capture-1 will be sorted by ACC-NO on an ascending key. The input procedure opens and reads DATA-CAPTURE-1. If the file is not at the end, SORT-RECORD is transferred and written to SORT-FILE. If the file is at the end, DATA-CAPTURE-1 is closed. DATA-CAPTURE-2 is opened output. The next record of SORT-FILE is read. If the file is at the end, then DATA-CAPTURE-2 is closed.

Then the merge begins. The file SHOW is opened output. MASTER-FILE and DATA-CAPTURE-3 are merged into MERGE-FILE. The records in MERGE-FILE are read, and their data is moved to OUT-RECORD. When MERGE-FILE is at end, MERGE-FILE and SHOW are closed.

Section 6
Procedure Division Statements A–H

This section illustrates and explains the syntax of the Procedure Division statements. Statements beginning with the letters A through H are listed in alphabetical order with the following information:

- A brief description of the function of the statement
- A syntax diagram for each format of the statement (if you need information on how to interpret a COBOL syntax diagram, refer to Appendix C)
- A statement of what portion of the syntax, if any, can be used interactively in a Test and Debug System (TADS) session
- An explanation of the elements in the syntax diagram
- Details, rules, and restrictions about the particular statement
- An example of the statement
- References to additional information relevant to the statement

Detailed information about language elements common to many Procedure Division statements, such as user-defined names, literals, and identifiers, is provided in Section 1. Concepts such as arithmetic and conditional expressions, and operations such as table handling, sorting, and merging are described in Section 5.

ACCEPT Statement

The ACCEPT statement makes low-volume data available to a specified data item.

Format	Use
Format 1	This format transfers data from a hardware device to a data item.
Format 2	This format transfers data from date and time registers to a data item.
Format 3	This format returns the number of entries in a storage queue (STOQ) into the entry-data-length field of the specified STOQ parameter block.
Format 4	This format transfers a formatted system date or time to a data item based on the type, convention, and language in effect for the item.

Format 1: Transfer Data from Hardware Device

```
ACCEPT  identifier-1  [ FROM {mnemonic-name-1 } ]
```

Explanation

identifier-1

This is the data item to which data is transferred from the hardware device.

mnemonic-name-1

The mnemonic-name must be specified in the SPECIAL-NAMES paragraph of the Environment Division, and must be associated with the hardware name ODT. If the FROM clause is not specified, the hardware device is assumed to be ODT.

Details

The $ANSI and $ANSICLASS compiler control options control the transfer of data to the receiving item. Table 6–1 explains the effects of this option upon the transfer of data.

Table 6–1. Effect of the $ANSI and $ANSICLASS Compiler Options

When the $ANSI or $ANSICLASS option is . . .	And . . .	Then . . .
Set	The size of the transferred data is less than the size of the receiving data item.	The transferred data is left-justified in the receiving data item, and a "MORE" prompt is displayed on the ODT requesting additional input.
Set	The size of the transferred data is greater than the size of the receiving data item.	The left-most digits are moved into the receiving field and the remainder of the digits are ignored.
Reset	The size of the transferred data is greater than the size of the receiving data item. (Leading zeros are not considered in computing the size of the transfer field.)	The compiled code issues a run-time error and prompts you to re-enter your data.
Reset	The receiving field is alphanumeric or national.	The transferred data is stored aligned to the left and blank-filled.
Reset	The receiving field is numeric.	The transferred data is stored aligned to the right and zero-filled.

Any necessary conversion of data from one form of internal representation to another takes place during data transfer. Control information is removed from national data before the data is transferred into the receiving national data field.

Data transferred to a numeric field is validated by the compiler to prevent you from inadvertently entering a nonnumeric character into a numeric field. Additionally, you cannot enter a number that is too large to fit into the named data item. In either case, an error message appears requesting that you re-enter your data.

Data accepted into an elementary data item of class alphanumeric can contain national characters in external format. In this situation, the control information necessary for external format is retained in the content of the data item.

Examples

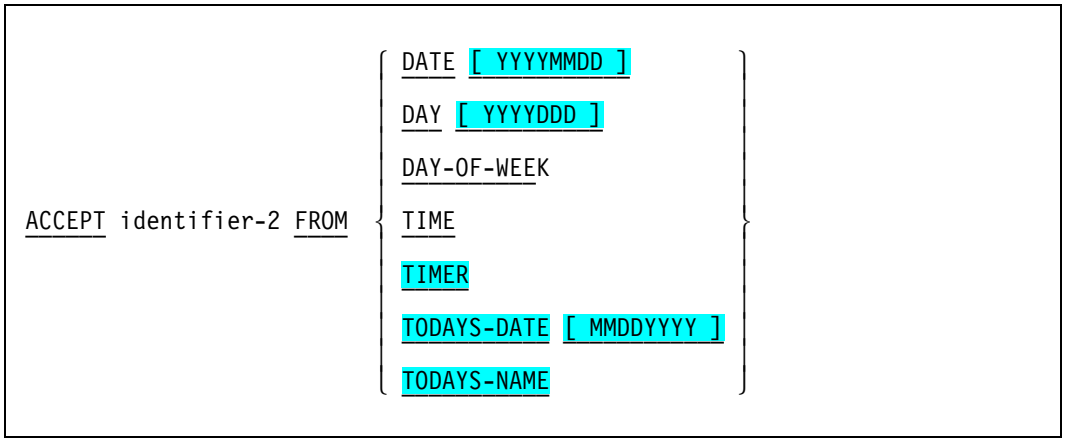
```
ACCEPT keyboard-option
```

In this example, the ACCEPT statement transfers data from the ODT (that is, the default hardware device) to the data item keyboard-option.

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    ODT IS TERMINAL1.
.
.
.
PROCEDURE DIVISION.

PARA-1.
    ACCEPT keyboard-option FROM TERMINAL1.
```

In this example, the ACCEPT statement transfers data from the ODT to the data item keyboard-option. The ODT has been given the mnemonic-name TERMINAL1 in the SPECIAL-NAMES paragraph of the Environment Division.

Format 2: Transfer Data from Date and Time Registers



Explanation

In this format, the ACCEPT statement transfers one of the special registers (date, day, time, and so on) to the data item named by identifier-2. The transfer of data occurs according to the rules of the MOVE statement. For information about these rules, refer to "MOVE Statement" in Section 7. Special registers are conceptual data items that are not declared in a COBOL program. Each register is described in the following list.

identifier-2

This identifier is the user-defined name of the data item.

DATE

This register contains the data elements year, month, and day. If DATE is followed by the qualifier YYYYMMDD, the year is four digits, otherwise the year is two digits.

The sequence of the data elements is from high order to low order (year, month, day). Therefore, July 1, 1988 is expressed as 880701, or, if qualified by YYYYMMDD, July 1, 1988 is expressed as 19880701.

When accessed by a COBOL program, this register behaves as if it had been described in the COBOL program as an unsigned elementary numeric integer data item six digits in length (PIC 9(6) COMP), or, if qualified by YYYYMMDD, a data item eight digits in length (PIC 9(8) COMP).

Data from the DATE register cannot be transferred to a national data item.

DAY

This register contains the data elements year and Julian day (that is, days are numbered consecutively from 001 to 365, or 366 if it is a leap year). If DAY is followed by the qualifier "YYYYDDD", the year is four digits, otherwise the year is two digits. The sequence of the data element codes is from high order to low order (year and day). Therefore, July 1, 1989 is expressed as 89183, or, if qualified by "YYYYDDD", July 1, 1989 is expressed as 1989183.

When accessed by a COBOL program, this register behaves as if it had been described in a COBOL program as an unsigned elementary numeric integer data item five digits in length (PIC 9(5) COMP), or, if qualified by "YYYYDDD", a data item seven digits in length (PIC 9(7) COMP).

DAY-OF-WEEK

This register contains a single data element that represents the day of the week. A value of 1 represents Monday, a value of 2 represents Tuesday, and so on.

When accessed by a COBOL program, this register behaves as an unsigned elementary numeric integer one digit in length (PIC 9(1) COMP).

TIME

This register contains the data elements hours, minutes, seconds, and hundredths of a second. The value of this register is based on elapsed time after midnight on a 24-hour clock; therefore, 2:41 p.m. is expressed as 14410000.

The minimum value of this register is 00000000 (midnight); the maximum value is 23595999 (one one-hundredth of a second before midnight).

If the hardware cannot provide fractional parts of the data elements contained in this register, the value is converted to the closest decimal approximation.

When accessed by a COBOL program, this register behaves as if it had been described in COBOL as an unsigned elementary numeric integer data item eight digits in length (PIC 9(8) COMP).

Data from the TIME register cannot be transferred to a national data item.

TIMER

This register contains the current value of the object computer's interval timer (that is, the number of 2.4-microsecond intervals since midnight).

When accessed by a COBOL program, this register behaves as if it had been described in COBOL as an unsigned elementary numeric integer data item 11 digits in length (PIC 9(11) COMP).

TODAYS-DATE

This register contains the following data elements: month, day, and year. If TODAYS-DATE is followed by the qualifier "MMDDYYYY", the year is four digits, otherwise the year is two digits. Therefore, July 1, 1989 is expressed as 070189, or, if qualified by "MMDDYYYY", July 1, 1989 is expressed as 07011989.

When accessed by a COBOL program, this register behaves as if it had been described in COBOL as an unsigned elementary numeric integer data item six digits in length (PIC 9(6) COMP), or, if qualified by "MMDDYYYY", a data item eight digits in length (PIC 9(8) COMP).

TODAYS-NAME

This register contains the name of the current day of the week.

When accessed by a COBOL program, this register behaves as if it had been described in COBOL as an elementary data item nine alphanumeric characters in length (PIC X(9)). The name is left-justified and space-filled.

Examples

```
ACCEPT date-1 FROM DATE
```

In this example, the ACCEPT statement transfers the content of the DATE register (that is, the current year, month, day) to the data item date-1.

```
ACCEPT time-1 FROM TIME
```

In this example, the ACCEPT statement transfers the content of the TIME register (that is, the current time in hours, minutes, seconds and hundredths of a second) to the data item time-1.

```
ACCEPT name-1 FROM TODAYS-NAME
```

In this example, the ACCEPT statement transfers the content of the TODAYS-NAME register (that is, the name of the current day) to the data item name-1.

Format 3: Transfer Number of Storage Queue Entries

```
ACCEPT identifier-1 MESSAGE COUNT
```

Explanation

This format of the ACCEPT statement transfers the number of messages in the storage queue to the entry-data-length field of the storage queue (STOQ) parameter block specified by identifier-1.

identifier-1

Identifier-1 refers to a 01-level data description entry for a STOQ parameter block.

Details

The STOQ parameter block must be defined as a 01-level data description entry of the following format:

```
01 Identifier-1.
        02  Queue-name              PIC X(6).
        02  Entry-name-length       PIC 9(2) COMP.
        02  Entry-name              PIC X(nn).
        02  Entry-data-length       PIC 9(4) COMP.
        02  Entry-data              PIC X(nnnn).
```

For a complete description of the STOQ function and the STOQ parameter block, refer to "SEND Statement" later in this section.

If an entry-name is specified in the STOQ parameter block, the count returned by this format of the ACCEPT statement is the number of entries in the queue for the specified entry-name or name-group.

If an entry-name is not specified, the count returned is the total number of entries in the queue.

The response is returned as an unsigned integer in the entry-data-length field of the specified STOQ parameter block. A response of zero means that the queue or the designated portion of the queue is empty or cannot be found.

Format 4: Transfer Formatted System Date and Time

```
ACCEPT identifier-1     [FROM    {DATE}]
                        [        {TIME}]
```

Explanation

This format of the ACCEPT statement transfers the formatted system date or time to the data item specified by the identifier-1 using the type, convention, and language in effect for the item. Format 4 is used when the identifier has an associated TYPE clause. If either the convention or language has not been declared for the item, the system determines the convention and language based on a default hierarchy.

identifier-1

This identifier is the user-defined name of the data item specified with the TYPE clause.

Details

The FROM clause is optional and used only for documentation. The specification of either the DATE or the TIME should match the type of the identifier. The DATE specification should be used when the receiving item is of the following types: SHORT-DATE, LONG-DATE, or NUMERIC-DATE. The TIME specification should be used when the item is of either the LONG-TIME type or the NUMERIC-TIME type. If the type of the item and the special register do not match, the compiler issues a warning message, continues the compilation, and assumes the special register is valid for the type declared for the receiving item.

ADD Statement

The ADD statement adds two or more numeric operands together and stores the result.

This statement is partially supported in the TADS environment. Supported syntax is noted in this section.

Format	Use
Format 1	The ADD . . . TO format adds elementary numeric items and/or numeric literals.
Format 2	The ADD . . . TO . . . GIVING format adds elementary numeric items and/or numeric literals, resulting in either an elementary numeric item or an elementary numeric-edited item.
Format 3	The ADD CORRESPONDING format adds the corresponding data items of two group items.

Format 1: ADD . . . TO

```
ADD  { identifier-1 }
     { literal-1    } . . . TO  { identifier-2 [ ROUNDED ] } . . .

[ ON SIZE ERROR imperative-statement-1 ]

[ NOT ON SIZE ERROR imperative-statement-2 ]

[ END-ADD ]
```

TADS Syntax

```
ADD  { identifier-1 }
     { literal-1    } . . . TO  { identifier-2 [ ROUNDED ] } . . .

[ END-ADD ]
```

Explanation

identifier-1
literal-1
identifier-2

In this format, each identifier must refer to an elementary numeric item. Each literal must be a numeric literal.

ROUNDED

This phrase causes the result to be rounded. Refer to "ROUNDED Phrase" in Section 5 for information about the rounding process.

ON SIZE ERROR imperative-statement-1

If a size error condition occurs, imperative-statement-1 will be executed. Refer to "SIZE ERROR Phrase" and "Imperative Statements and Sentences" in Section 5 for more information.

NOT ON SIZE ERROR imperative-statement-2

If a size error does not occur and this phrase is specified, imperative-statement-2 will be executed.

END-ADD

This phrase delimits the scope of the ADD statement.

Details

The values of the operands preceding the word TO are added together, and the sum is stored in a temporary data item. The temporary data item is then added to the value of identifier-2. This process is repeated as many times as required by the statement.

The composite length of the operands cannot exceed 23 decimal digits (the composite length is based on the length of all of the operands in the statement).

The compiler ensures that enough places are carried so as not to lose any significant digits.

Examples

```
ADD key-1 TO key-2
```

In this first example, the elementary numeric item key-1 is added to the elementary numeric item key-2. The result is stored in the data item key-2.

```
ADD key-1, key-2 TO key-3, key-4 ROUNDED END-ADD
```

In this second example, key-1 and key-2 (both elementary numeric items) are added together; the result is stored in a temporary data item. The temporary data item is added to the data item key-3, and the result is stored in the data item key-3. The temporary data item is then added to the data item key-4, and the result is rounded and stored in data item key-4. The END-ADD option terminates the scope of this ADD statement.

Format 2: ADD . . . TO . . . GIVING

```
ADD { identifier-1 } . . . [ TO ] { identifier-2 }
    { literal-1    }              { literal-2    }

GIVING  { identifier-3 [ ROUNDED ] } . . .

[ ON SIZE ERROR imperative-statement-1 ]

[ NOT ON SIZE ERROR imperative-statement-2 ]

[ END-ADD ]
```

TADS Syntax

```
ADD { identifier-1 } . . . [ TO ] { identifier-2 }
    { literal-1    }              { literal-2    }

GIVING  { identifier-3 [ ROUNDED ] } . . .

[ END-ADD ]
```

Explanation

Refer to Format 1 for information on the ROUNDED, ON SIZE ERROR, NOT ON SIZE ERROR, and END-ADD phrases.

identifier-1
identifier-2

Identifier-1 and identifier-2 must be elementary numeric items.

literal-1
literal-2

Each literal must be a numeric literal.

GIVING identifier-3

The values of the operands preceding the word GIVING are added together, and the sum is stored into the data item named by identifier-3. The data item represented by identifier-3 can be an elementary numeric item or an elementary numeric-edited item.

Details

The composite length of the operands in the ADD statement cannot exceed 23 decimal digits (the composite length is based on all of the operands that precede the word GIVING).

The compiler ensures that enough places are carried so as not to lose any significant digits.

Examples

```
ADD key-1 TO key-2 GIVING key-3
```

In this first example, the data items key-1 and key-2 are added, and the result is stored in the data item key-3.

```
ADD key-1, key-2 TO key-3
    GIVING key-4,  key-5 ROUNDED
 END-ADD.
```

In this second example, the data items key-1, key-2, and key-3 are added together, and the result is stored in the data item key-4 and in the data item key-5. The result in key-5 is rounded.

Format 3: ADD CORRESPONDING

```
ADD { CORRESPONDING } identifier-1 TO identifier-2 [ ROUNDED ]
    { CORR          }

[ ON SIZE ERROR imperative-statement-1 ]

[ NOT ON SIZE ERROR imperative-statement-2 ]

[ END-ADD ]
```

TADS Syntax

```
ADD { CORRESPONDING } identifier-1 TO identifier-2 [ ROUNDED ]
    { CORR          }

[ END-ADD ]
```

Explanation

For more information about the CORRESPONDING phrase, refer to "MOVE Statement" in Section 7.

CORRESPONDING
CORR

The CORRESPONDING (or CORR) option enables you to add numeric data items from one group item to data items of the same name within another group item. Only elementary numeric data items can be added with this phrase. Refer to the discussion of the CORRESPONDING phrase under "MOVE Statement" in Section 7 for rules that also apply to the ADD CORRESPONDING phrase.

CORRESPONDING and CORR are equivalent.

identifier-1
identifier-2

In this format, each identifier must refer to a group item.

Data items in the group referred to by identifier-1 are added to and stored in the corresponding data items in the group referred to by identifier-2.

The compiler ensures that enough places are carried so as not to lose any significant digits.

A data item that is subordinate to the data item referred by identifier-1 or identifier-2 and that contains a REDEFINES, RENAMES, OCCURS, or USAGE IS INDEX clause is ignored. In addition, any data item subordinate to such a subordinate data item is also ignored.

A valid group item identifier cannot contain level-number 66, level-number 77, level-number 88, or the USAGE IS INDEX clause.

A valid group item identifier cannot be reference-modified.

Refer to "USAGE Clause," "REDEFINES Clause," "RENAMES Clause," and "OCCURS Clause" in Section 4.

Overlapping Operands

When a sending item and a receiving item share a part of their storage areas and are not defined by the same data description entry, the result of the ADD statement is undefined. The undefined result occurs only when operands share a part, but not all, of their storage areas.

Example

```
DATA DIVISION.
01  group-1.
    05 A PIC 99.
    05 B PIC X(4).
    05 C PIC 9(8).
01  group-2
    05 A PIC 99.
    05 D PIC 99.
    05 B PIC X(4).
    05 E PIC 9(4).
    05 C PIC 9(8).
    05 F PIC 9(8).
     .
     .
     .
    ADD CORR group-1 TO group-2 ROUNDED END-ADD
```

In this example, the data items belonging to the group item group-1 are added to the corresponding data items (A, B, and C) that belong to the group item group-2. The results are rounded.

Refer to "Imperative Statements and Sentences," "ROUNDED Phrase," and "SIZE ERROR Phrase" in Section 5 for more information.

ALLOW Statement

The ALLOW statement reverses the effect of the DISALLOW statement, enabling interrupt procedures to be executed when their associated events are activated (by a CAUSE statement). See the CAUSE statement and the DISALLOW statement for additional information.

```
        { section-name-1  [ ,section-name-2 ]. . . }
ALLOW   {                                          }
        { INTERRUPT                                }
```

section-name-1 [, section-name-2] . . .

This syntax is used to allow access to interrupt procedures that were previously restricted by the DISALLOW section-name statement. Using this syntax during the time that the DISALLOW INTERRUPT statement is in effect causes the interrupt procedures to be queued when their events are activated. The procedures remain queued until an ALLOW INTERRUPT statement is executed.

Section-name indicates the name of the section in the Procedure Division that contains the interrupt procedure to be affected by the ALLOW statement. You can use multiple section names to affect multiple interrupt procedures.

INTERRUPT

The ALLOW INTERRUPT syntax reverses the effect of a previous DISALLOW INTERRUPT statement. Queued interrupt procedures are immediately executed, unless they were queued because of a specific DISALLOW section-name statement. In that case, an ALLOW section-name statement must be issued for those procedures.

Details

You can use the ALLOW statement for interrupt procedures not attached to an event. Note that performing an ATTACH statement for a procedure that has not been specifically restricted by the DISALLOW statement automatically establishes the ALLOW condition for that procedure.

Example

```
ALLOW INTERRUPT.

ALLOW INTERRUPT-PROCEDURE-ONE.
```

ALTER Statement

The ALTER statement modifies a predetermined sequence of operations. This statement is obsolete and will be deleted from the next revision of Standard COBOL.

Refer to "GO TO Statement" in this section for a description of the GO TO statement and the DEPENDING phrase.

```
ALTER  { procedure-name-1 TO [ PROCEED TO ] procedure-name-2 } . . .
```

Explanation

procedure-name-1

Procedure-name-1 refers to the name of a paragraph in the Procedure Division that contains a single sentence consisting of a GO TO statement without the DEPENDING phrase.

procedure-name-2

Procedure-name-2 refers to the name of a paragraph or section in the Procedure Division.

Details

Execution of the ALTER statement modifies the GO TO statement in the paragraph named procedure-name-1, so that a subsequent execution of the GO TO statement transfers control to the procedure named procedure-name-2.

Example

```
Main-paragraph.
.
.
.
    ALTER Search-1 TO PROCEED TO Search-2

Search-1.
    GO TO Unstring-1.
Search-2.
```

In this example, the ALTER statement modifies the GO TO statement in the paragraph named Search-1, so that when the GO TO statement is executed, control is transferred to the paragraph named Search-2.

ATTACH Statement

The ATTACH statement associates an interrupt procedure with an event.

```
ATTACH section-name TO event-identifier.
```

Explanation

section-name

This is the name of the section in the Procedure Division that contains the interrupt procedure with which you want to associate this event.

You can attach multiple interrupt procedures to a single event. When the event is activated, the procedures referenced by the section-names are executed in the reverse order in which they were specified.

event-identifier

This can be one or more of the following:

- The name of a data-item declared with the USAGE IS EVENT phrase. The data-name must be properly qualified and properly subscripted.
- A task attribute of type EVENT. The event task attributes are ACCEPTEVENT and EXCEPTIONEVENT. For details about these task attributes, refer to the *Task Attributes Programming Reference Manual*.
- A file attribute of type EVENT. The event file attributes are CHANGEEVENT, INPUTEVENT, and OUTPUTEVENT. For details about these file attributes, refer to the *File Attributes Programming Reference Manual*.

Details

The ATTACH statement causes an implicit ALLOW condition for specified interrupt procedures that have not been restricted by a previous DISALLOW statement.

The following table explains what happens to interrupt procedures when an event item is activated.

When an EVENT item is activated by a CAUSE statement and . . .	Then . . .	And . . .
The ALLOW INTERRUPT statement was previously used.	The calling program is suspended.	All interrupt procedures attached to that event are executed immediately.
An interrupt procedure attached to that event was previously readied by the ALLOW section-name statement.	The calling program is suspended.	The interrupt procedure is executed immediately.
The DISALLOW INTERRUPT statement was previously used.	The calling program continues executing.	All interrupt procedures attached to that event are queued.
An interrupt procedure attached to that event was previously restricted by the DISALLOW section-name statement.	The calling program continues executing.	The interrupt procedure is queued.

Note: *An interrupt procedure should be attached to only one event at a time. If an interrupt procedure is already attached to an event when the ATTACH statement is executed, the interrupt procedure is automatically detached from the old event and then attached to the new event. Any queued invocations of the interrupt procedure are lost.*

Example

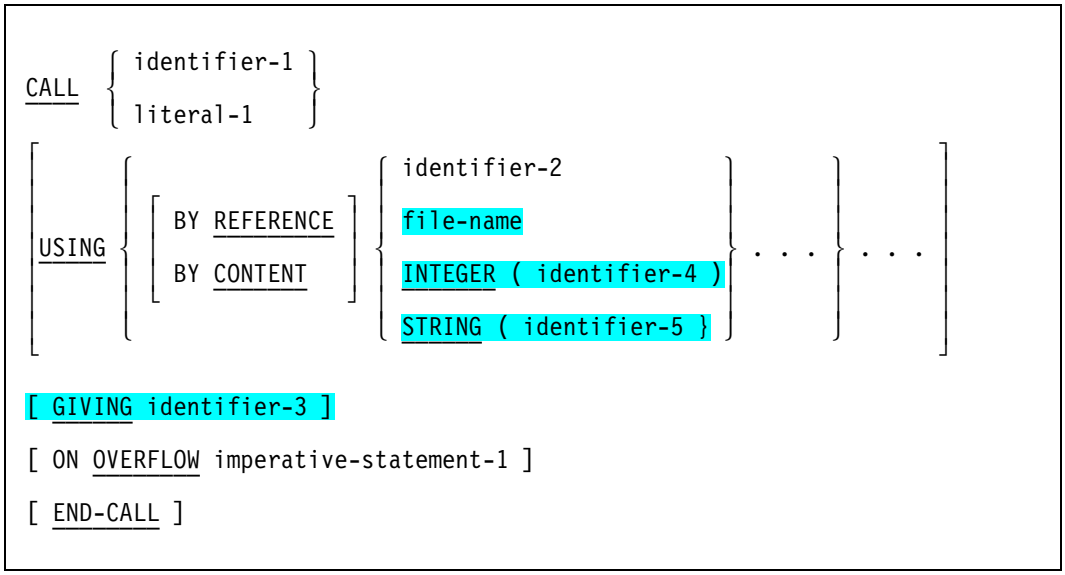
```
ATTACH INTERRUPT-PROCEDURE-ONE TO WS-EVENT77.
```

CALL Statement

The CALL statement transfers control from one object program to another object program in the same run unit.

Format	Use
Format 1	This format provides a CALL statement with an ON OVERFLOW option. This format uses the interprogram communication (IPC) technique, which is described in Section 10.
Format 2	This format provides a CALL statement with an ON EXCEPTION option. This format uses the interprogram communication (IPC) technique, which is described in Section 10.
Format 3	This format provides a CALL statement for invoking an external system procedure or WFL job. This format is partially supported in the TADS environment. Supported syntax is noted in the description of the format.
Format 4	This format provides a CALL statement for binding. The use of Binder with COBOL85 programs is discussed in Appendix E.
Format 5	This format provides a CALL statement for access to entry procedures residing in program libraries. Library concepts and the programmatic components required in programs that call libraries are described in Section 11.
Format 6	This format provides a CALL statement for executing an independently compiled program as a synchronous, dependent task. The concepts of tasking and the components required in programs that perform tasking are described in Section 13.
Format 7	This format provides a CALL statement for transferring control to a portion of code in an externally compiled program bound into the calling program. You can specify a section-name or a user-defined program-name as an entry point.

Format 1: CALL with ON OVERFLOW Option



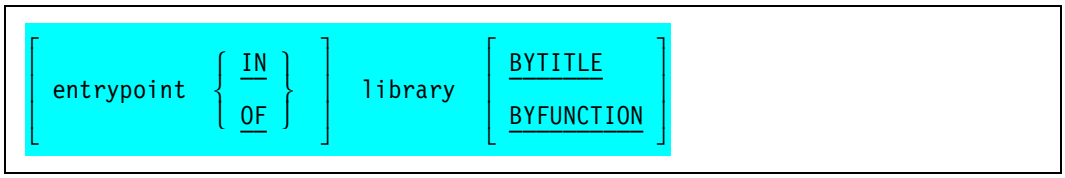
Explanation

identifier-1

This identifier must be defined as an alphanumeric data item whose value is consistent with program-naming conventions. It identifies the name of the called program.

literal-1

This must be a nonnumeric literal that identifies the name of the called program. If you are calling a library entry point, you can specify the called program by using the following syntax:



In this syntax, entrypoint is the program-name specified by the PROGRAM-ID paragraph in the Identification Division, which is exported by the ENTRY PROCEDURE clause in the Program-Library Section. For details about library entrypoints, refer to Table 9-1. Library is the file title of the library if BYTITLE is specified or the function name of the library if BYFUNCTION is specified. If neither BYTITLE nor BYFUNCTION is specified, the library will be called by title. If you choose the BYTITLE option, you can specify the ON <family name> clause in the title.

USING

The USING phrase identifies the individual parameters that can be passed. Parameters can be passed either by reference or by content. Passing by reference is the default.

Long numeric data items are valid in the USING phrase. A long numeric data item is an unsigned numeric DISPLAY or COMPUTATIONAL data item from 24 to 99,999 digits long. Long numeric data items are treated as group items. Data items larger than 23 digits must be unsigned integers.

BY REFERENCE

The BY REFERENCE phrase enables the calling program to pass data to the called program. The values of the passed data may be modified by the called program. If the values of the passed data were modified by the called program, they will be modified in the calling program when control is returned to the calling program.

If the BY REFERENCE phrase is either specified or implied for a parameter, the object program operates as if the corresponding data item in the called program occupies the same storage area as the data item in the calling program.

The data item in the called program and the corresponding data item in the calling program must have the same number of character positions.

Both the BY CONTENT and the BY REFERENCE phrases are transitive across the parameters that follow them until another BY CONTENT or BY REFERENCE phrase is encountered. If neither the BY CONTENT nor the BY REFERENCE phrase is specified prior to the first parameter, the BY REFERENCE phrase is assumed.

BY CONTENT

The BY CONTENT phrase enables the program that contains the CALL statement to pass data to the called program. The original values of the passed data will be restored to the calling program when control is returned to the calling program. This occurs despite any changes the called program might make to the passed data.

The data description of each parameter in the BY CONTENT phrase of the CALL statement must match the data description of the corresponding parameter in the USING phrase of the Procedure Division header.

identifier-2

This is a data item that will be passed to the called program.

Identifier-2 can be an elementary data item or a non-01-level group item declared in the File Section, Working-Storage Section, or the Linkage Section of the calling program. The compiler generates a copy of the data and passes the copy to the called program. If the parameter is passed BY REFERENCE, the data is copied back into the original area on return from the call.

Identifier-2 **cannot be** a function-identifier.

Identifier-2 **can** be a national data item.

To prevent data corruption, identifier-2 cannot be a redefined data item. This rule includes implicit as well as explicit redefinitions. An explicit redefinition occurs when a data item is declared in the File Section with a REDEFINES clause or is subordinate to a data item declared with a REDEFINES clause. An implicit redefinition occurs when the first data item declared in the File Section is followed by subsequent level 01 items. The subsequent level-01 items are considered to be implicit redefinitions of the first item.

COMS headers can be sent as parameters to entry points of libraries and will match to a real array.

file-name

This is a file name of a file to be passed as a parameter. The file must be declared as RECEIVED BY REFERENCE in the file's SELECT clause of the FILE-CONTROL paragraph.

INTEGER (identifier-4)

This declares the parameter to be an integer type parameter. Integer type parameters must be declared with USAGE COMPUTATIONAL.

STRING (identifier-5)

This declares the parameter to be a string type parameter. String type parameters must be declared with USAGE DISPLAY.

GIVING identifier-3

The GIVING phrase is used to provide a data item into which the value of the called function is to be stored. The procedure identified by identifier-1 must be a function that returns a value. Identifier-3 must be a numeric item.

ON OVERFLOW imperative-statement-1

If the program is not present, imperative-statement-1 is executed.

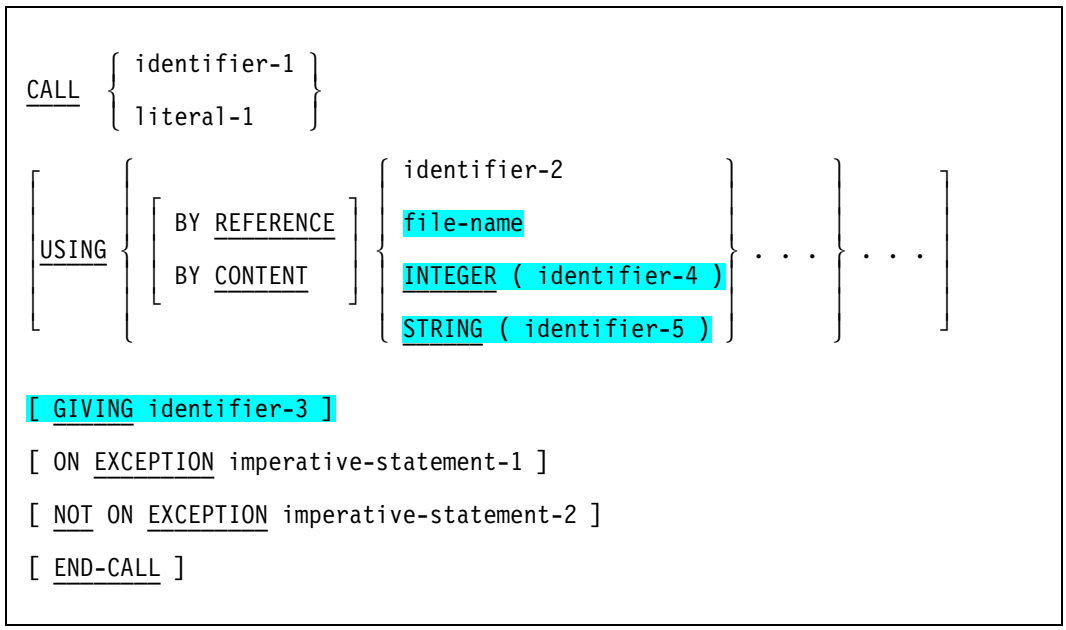
END-CALL

This phrase delimits the scope of the CALL statement.

Details

Details for the CALL with ON OVERFLOW option and the CALL with ON EXCEPTION option appear under the heading “Format 2: CALL with ON EXCEPTION Option” in this section.

Format 2: CALL with ON EXCEPTION Option



Explanation

Refer to Format 1 for descriptions of the syntax elements identifier-1, literal-1, identifier-2, file-name, USING, BY REFERENCE, BY CONTENT, INTEGER (identifier-4), STRING (identifier-5), GIVING, and END-CALL.

ON EXCEPTION imperative-statement-1

If the called program is not present and this phrase is specified, imperative-statement-1 is executed.

NOT ON EXCEPTION imperative-statement-2

If the called program is available and executable as a called program, imperative-statement-2 is executed.

Details

The calling program is the program in which the CALL statement appears. The called program is the object of a CALL statement, combined at execution time with the calling program to produce a run unit.

Literal-1 or the content of the data item referenced by identifier-1 must contain the object name of the called program.

If the program being called in identifier-1 or literal-1 is not a COBOL program, the number of parameters in the formal parameter list of this program must match the number of operands in each USING phrase of the COBOL program. In case of parameter size difference, the COBOL MOVE rules apply.

When a CALL statement is executed, and the program specified by the CALL statement is made available for execution, control is transferred to the called program.

The BY CONTENT phrase of the CALL statement is a method of passing parameters between programs without changing the value in the calling program. Whether the BY CONTENT or the BY REFERENCE phrase is specified in the CALL statement, for the implicit entry procedure interface, the compiler treats the formal parameter as though the BY REFERENCE phrase had been specified. In this case, a copy is made for the BY CONTENT data item and passed by reference. True BY CONTENT applies only to the explicit library interface for level-77 BINARY, DOUBLE, and REAL data items.

Table 6–2 illustrates parameter mapping among COBOL85, ALGOL, Pascal, and COBOL74 programs.

Table 6–2. Parameter Mapping among Languages

Implicit Interface			
COBOL85 Data	**ALGOL Data**	**Pascal Data**	**COBOL74 Data**
BY CONTENT	Reference	Reference	Reference
BY REFERENCE	Reference	Reference	Reference
Explicit Interface			
COBOL85 Data	**ALGOL Data**	**Pascal Data**	**COBOL74 Data**
BY CONTENT (REAL, DOUBLE, BINARY)	Value	Value	No Match

To use the explicit library interface, you must add a LOCAL-STORAGE SECTION and a PROGRAM-LIBRARY SECTION in your program to describe the library and its parameters and attributes. You can pass the parameters from COBOL85 to ALGOL as by value and match the ALGOL VALUE parameter by specifying BY CONTENT in the LOCAL-STORAGE SECTION on the formal description of the COBOL85 parameter; nothing is specified on the corresponding CALL statement.

Example

The following COBOL85 program calls an ALGOL library that is passing a parameter by value.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
77     WS-PARAM1  PIC 9(3) BINARY.
LOCAL-STORAGE SECTION.
LD PROG1.
77      PARAM1  PIC 9(3) BINARY BY CONTENT.
PROGRAM-LIBRARY SECTION.
  LB MYLIB IMPORT
   ATTRIBUTE LIBACCESS IS BYTITLE
        TITLE IS "OBJECT/ALGOL/LIB".
  ENTRY PROCEDURE PROC1 WITH PROG1 USING
    PARAM1.
PROCEDURE DIVISION.
 START-MAIN.
  MOVE 3 TO WS-PARAM1.
  DISPLAY "BEFORE LIB CALL WS-PARAM1=" WS-PARAM1.
  CALL PROC1 USING WS-PARAM1.
  DISPLAY "AFTER LIB CALL WS-PARAM1=" WS-PARAM1.
 END-MAIN.
  STOP RUN.
   And here is the library:
      BEGIN
      PROCEDURE PROC1(I1);
        VALUE I1;
        INTEGER I1;
          BEGIN
          I1:=*+1;
          DISPLAY("IN LIBRARY, PARAMETER CHANGED TO " CAT STRING(I1,*));
          END;
      EXPORT PROC1;
      FREEZE(TEMPORARY);
      END.
```

When the COBOL85 program runs, the displays show that the value of WS-PARAM1 is not changed:

```
RUNNING 9061
9061 DISPLAY:BEFORE LIB CALL WS-PARAM1=003.
9061 DISPLAY:IN LIBRARY, PARAMETER CHANGED TO 4.
9061 DISPLAY:AFTER LIB CALL WS-PARAM1=003.
ET=0.4 PT=0.1 IO=0.1
```

Control and the ON OVERFLOW and ON EXCEPTION Phrases

After control is returned from the called program, the ON EXCEPTION or ON OVERFLOW phrase (if specified) is ignored. Control is transferred to the end of the CALL statement.
After control is returned from the called program and the NOT ON EXCEPTION phrase is specified, control is transferred to imperative-statement-2. Then, execution continues according to the rules for each statement specified in imperative-statement-2. If a procedure-branching or conditional statement that causes explicit transfer of control is executed, control is transferred according to the rules for that statement. Otherwise, upon completion of the execution of imperative-statement-2, control is transferred to the end of the CALL statement.

If the program specified by the CALL statement cannot be made available for execution when it is called, one of the following actions will occur:

- If the ON OVERFLOW or ON EXCEPTION phrase is specified, control is transferred to imperative-statement-1. Execution then continues according to the rules for each statement specified in imperative-statement-1.

  If a procedure branching or conditional statement that causes explicit transfer of control is executed, control is transferred according to the rules for that statement. Otherwise, upon completion of the execution of imperative-statement-1, control is transferred to the end of the CALL statement, and the NOT ON EXCEPTION phrase, if specified, is ignored.
- If the ON OVERFLOW or ON EXCEPTION phrase is not specified in the CALL statement, then imperative-statement-2 in the NOT ON EXCEPTION phrase, if specified, is ignored.

Program-Name Conventions

Two or more programs in a run unit can have the same program-name. If a CALL statement refers to a duplicated program-name, the problem is resolved by the conventions for the scope of names for program-names. Refer to "Conventions for Program-Names" in Section 10 for more information.

For example, when only two programs in the run unit have the same name as that specified in a CALL statement:

- One of those two programs must also be contained directly or indirectly in the program which includes that CALL statement or in the separately compiled program which itself directly or indirectly contains the program which includes that CALL statement.
- The other of those two programs must be a different, separately compiled program.

  The mechanism used in this example is as follows:

- If one of the two programs having the same name as that specified in the CALL statement is directly contained within the program which includes that CALL statement, that program is called.

- If one of the two programs having the same name as that specified in the CALL statement possesses the common attribute and is directly contained within another program which directly or indirectly contains the program which includes the CALL statement, that common program is called unless the calling program is contained within that common program.
- Otherwise, the separately compiled program is called.

Program States

A called program (and each program it directly or indirectly contains) is in its initial state the first time it is called in a run unit or the first time it is called after it has been canceled by a CANCEL statement.

If the called program possesses the initial attribute, it and each program it directly or indirectly contains are placed into an initial state every time the called program is called in a run unit.

On all other entries in the called program, the state of the program (including each program it directly or indirectly contains) remains unchanged from its state when it was last exited.

Files associated with a called program's internal file connectors are not in the open mode when the program is in an initial state. On all other entries into the called program, the states and positions of all such files are the same as when the called program was last exited.

The process of calling a program or exiting from a called program does not alter the status or position of a file associated with any external file connector.

The USING Phrase

The USING phrase is included in the CALL statement only if there is a USING phrase in the Procedure Division header of the called program. In this case, the number of operands in each USING phrase must be the same.

The sequence in which data-names appear in the USING phrase of the CALL statement and in the corresponding USING phrase in the called program's Procedure Division header determines the relationship between the data-names used by the calling and called programs. This relationship is based on position; the first data-name in one USING phrase corresponds to the first data-name in the other, the second to the second, and so forth.

The values of the parameters referenced in the USING phrase of the CALL statement are available to the called program at the time the CALL statement is executed.

CALL Statements in Nested Programs

Called programs can contain CALL statements. However, a called program must not execute a CALL statement that directly or indirectly calls the calling program. As a result, there are no recursive calls.

If a CALL statement is executed within the range of a declarative procedure, that CALL statement cannot directly or indirectly reference any called program to which control has been transferred or that has not completed execution.

Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CALLER.
DATA DIVISION.
.
.
.
WORKING-STORAGE SECTION.
01  COLOR    PIC X(10).
01  SIZE1    PIC 99v99.
01  AMOUNT   PIC 999.
.
.
.

PROCEDURE DIVISION.

PARA-1.
     CALL "CALLED"
     USING BY CONTENT SIZE1
            BY REFERENCE COLOR

     ON EXCEPTION PERFORM EX-1
     NOT ON EXCEPTION
          PERFORM PARA-4.
```

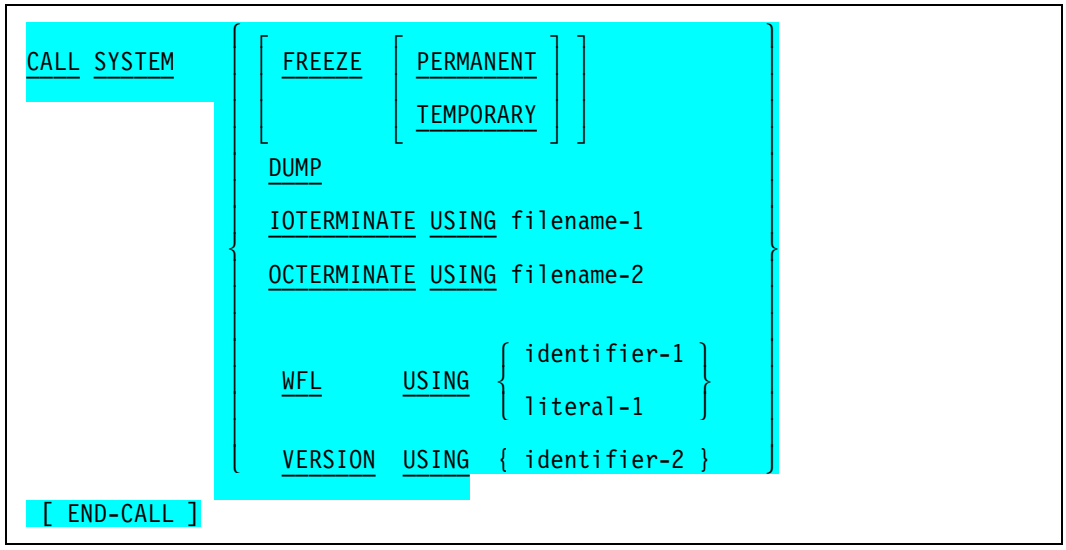
```
IDENTIFICATION DIVISION.
PROGRAM-ID. CALLED.
DATA DIVISION.
.
.
.
 WORKING-STORAGE SECTION.
 .
 .
 .
 LINKAGE SECTION.
 01  HUE      PIC X(10).
 01  MY-SIZE  PIC 99V99.

 PROCEDURE DIVISION
 USING MY-SIZE, HUE.
 PARA-A.
.
 .
 .


      MOVE MY-SIZE TO WS-1.
      MOVE  "RED" TO HUE.
      EXIT PROGRAM.
```

The program on the left (CALLER) is calling the program on the right (CALLED). The identifiers that will be passed, which are SIZE1 and COLOR, are defined in the program that contains the CALL statement. These identifiers correspond to the identifiers MY-SIZE and HUE in the called program.

The values of SIZE1 and COLOR will be passed from CALLER to CALLED, but the value of SIZE1 cannot be modified because it is passed BY CONTENT.

If an exception condition exists, the statements in EX-1 will be executed. The NOT ON EXCEPTION phrase will be ignored. If an exception condition does not exist, the statements in PARA-4 will be executed. The ON EXCEPTION phrase will be ignored.

Format 3: CALL a System Procedure



TADS Syntax

```
CALL SYSTEM DUMP
```

Explanation

This format of the CALL statement invokes a system procedure or a WFL job or retrieves the value of $VERSION. COBOL85 supports calls to the following system procedures:

FREEZE

This calls the library freeze function. The freeze function suspends execution of the library program and makes available all declared entry points into the library.

For more information on libraries, refer to Section 11.

PERMANENT
TEMPORARY

The optional reserved words PERMANENT and TEMPORARY can be used to specify the disposition of the library.

For more information on libraries, refer to Section 11.

DUMP

This calls the dump facility which produces a "snapshot" of the memory area of the program.

IOTERMINATE

This option causes a process to self-terminate after the unsuccessful execution of an I/O statement. The current value of the MCPRESULTVALUE identifier along with the filename specified in this clause are passed to the MCP to produce a meaningful I-DS message. For details about the MCPRESULTVALUE identifier, refer to Section 12.

OCTERMINATE

This option causes a process to self-terminate after the unsuccessful execution of either an OPEN or CLOSE statement. The current value of the MCPRESULTVALUE identifier along with the filename specified in this clause are passed to the MCP to produce a meaningful I-DS message. For details about the MCPRESULTVALUE identifier, refer to Section 12.

WFL

This option initiates an independent task that invokes the WFL job. After initiating the task, the program executes the next statement. The program does not wait for the task executing the WFL job to be completed.

The COBOL compiler does not check the syntax of the WFL job. Thus, errors in the WFL syntax have no effect on the calling program.

The COBOL program does not determine if the CALL action is successful. You can check for successful termination of the WFL compilation and the job itself by using the CANDE *?C* command or the MARC *C* command. The WFL compilation is assigned the name CONTROLCARD. The name assigned to the WFL job is constructed from the BEGIN JOB statement of the WFL job.

For more information about WFL jobs, refer to the *Work Flow Language (WFL) Programming Reference Manual*.

identifier-1

This identifier names a WFL source file that contains a complete WFL job. Identifier-1 must be defined as a 01-level data item that includes the USAGE IS DISPLAY phrase.

literal-1

This literal must be nonnumeric and must specify a complete WFL job.

VERSION

This option causes the $VERSION value to be assigned to a specified receiving area.

Identifier-2

This identifier specifies the receiving area for the value of $VERSION. The identifier must be defined as a data item with the usage of DISPLAY. The format of the identifier is <rrcccpppp>, where rr is the release number, ccc is the cycle number, and pppp is the patch number.

Examples

```
CALL SYSTEM FREEZE PERMANENT.
```

This statement calls the library freeze function and requests that the library be frozen permanently.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
OBJECT-COMPUTER. A15.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WFL-SRC                  PIC X(46)
   VALUE "BEGIN JOB J;DISPLAY ""THIS IS A TEST""; END JOB.".

PROCEDURE DIVISION.
PARA-1.

 CALL SYSTEM WFL USING WFL-SRC.
STOP RUN.
```

This program is written to execute the WFL job referenced by the identifier WFL-SRC. Notice that the identifier WFL-SRC is declared as a 01-level data item and that the complete WFL job follows that declaration.

```
$VERSION 12.345.6789
   IDENTIFICATION DIVISION.
   ENVIRONMENT DIVISION.
   DATA DIVISION.
   WORKING-STORAGE SECTION.
   01  VERSION-VALUE.
       05   REL    PIC 9(2).
       05   CYCLE  PIC 9(3).
       05   PATCH  PIC 9(4).

   PROCEDURE DIVISION.
   PARA-1.

     CALL SYSTEM VERSION USING VERSION-VALUE.
     IF REL = 12 AND CYCLE = 345 AND PATCH = 6789
       DISPLAY "PASSED"
     ELSE
       DISPLAY "FAILED".
   STOP RUN
```

This program calls the version value and displays pass/fail status.

Format 4: CALL for Binding

```
CALL section-name [ USING actual-parameter-list ]
```

Explanation

This format is used to call a procedure in an externally compiled program that will be bound to the calling program.

The actual-parameter-list must consist of a series of data-items, control items, and expressions optionally separated by commas.

Details

In addition to passing arithmetic values, certain kinds of variables can be passed (received) by reference. As a general rule, the usage of the actual parameter must not conflict with the corresponding formal parameter, as specified in the Local-Storage Section.

Table 6–3 shows the formal parameters that can be declared in COBOL85 for bound and host programs, along with the corresponding declarations in ALGOL, and the permissible actual parameters that can be passed.

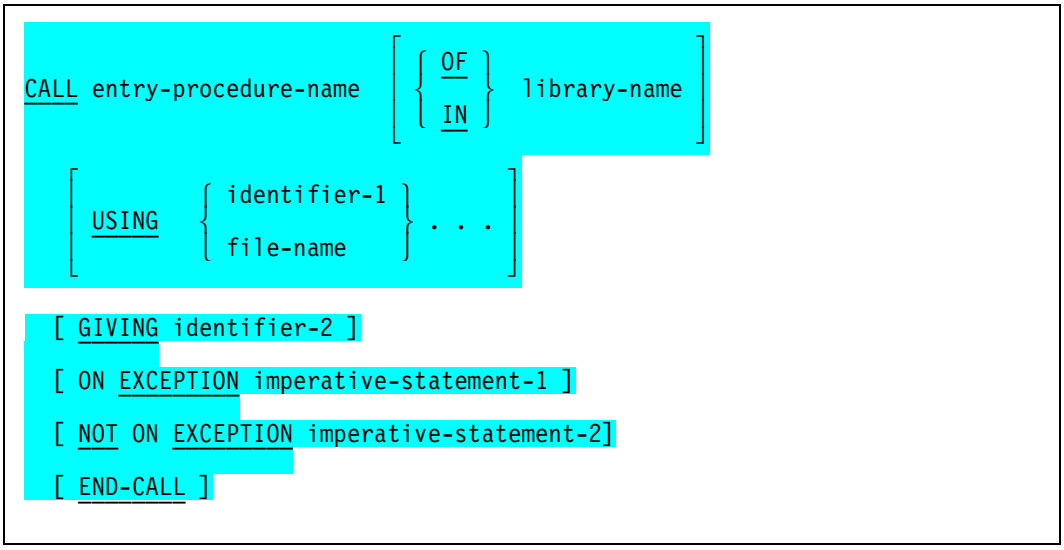
Table 6–3. Formal and Actual Parameters for Bound Procedures

COBOL85 Formal Parameter	ALGOL Formal Parameter	Permissible Actual Parameters
BINARY, 77, 1-11 digits (RECEIVED BY CONTENT)	INTEGER	Arithmetic-expression
REAL, 77 (RECEIVED BY CONTENT)	REAL	Arithmetic-expression
BINARY, 77, 12-23 digits or DOUBLE, 77 (RECEIVED BY CONTENT)	DOUBLE	Arithmetic-expression
BINARY 77, 1-11 digits (RECEIVED BY REFERENCE)	INTEGER	BINARY, 77, 1-11 digits
REAL, 77 (RECEIVED BY REFERENCE)	REAL	REAL, 77
BINARY, 77, 12-23 digits (RECEIVED BY REFERENCE)	DOUBLE	BINARY, 77, 12-23 digits

Table 6–3. Formal and Actual Parameters for Bound Procedures

COBOL85 Formal Parameter	ALGOL Formal Parameter	Permissible Actual Parameters
DOUBLE, 77 (RECEIVED BY REFERENCE)	DOUBLE	DOUBLE, 77
BINARY, 01, 1-11 digits	INTEGER ARRAY	BINARY, 01, 1-11 digits COMP, 01 DISPLAY 01
BINARY, 01, 12-23 digits or DOUBLE, 01	DOUBLE ARRAY	BINARY, 01, 12-23 digits DOUBLE, 01 REAL, 01
REAL, 01	REAL ARRAY	REAL, 01
COMP, 01 INDEX, 01	HEX ARRAY	COMP, 01 DISPLAY, 01 BINARY, 01, 1-11 digits
DISPLAY, 01	EBCDIC ARRAY	COMP, 01 DISPLAY, 01 BINARY, 01, 1-11 digits
FILE	FILE	FILE
TASK, 77 or 01	TASK	TASK, 77 or 01
TASK, 01 group	TASK ARRAY	TASK, 01 group
EVENT or LOCK, 77	EVENT	EVENT or LOCK, 77
EVENT or LOCK, 01 group	EVENT ARRAY	EVENT or LOCK, 01 group
BIT, 77 SYNC RIGHT (RECEIVED BY CONTENT)	BOOLEAN	BIT, 77 Boolean-expression
BIT, 77 SYNC RIGHT (RECEIVED BY REFERENCE)	BOOLEAN	BIT, 77
BIT, 01 SYNC RIGHT	BOOLEAN ARRAY	BIT, 01

Format 5: CALL for Library Entry Procedure



Explanation

This format of the CALL statement transfers program control to a procedure in a library program. Refer to Format 1 for information on the USING clause, the GIVING clause, and the END-CALL clause. Matching of formal and actual parameters can result in coercion of the actual parameter to match the formal parameter description.

entry-procedure-name

This is the name of the procedure to be called in the library program. The entry-procedure-name must have been previously declared in the Program-Library Section of the calling program.

OF library-name
IN library-name

This identifies the library program that contains the called procedure. This phrase is optional, and is used only to differentiate like-named procedures residing in different library programs. The library-name must have been previously declared in the Program-Library Section of the calling program.

ON EXCEPTION imperative-statement-1

If the called library linkage fails or if the program successfully links to the library but some of the entry points do not exist in the library, and this phrase is specified, imperative-statement-1 is executed and the library entry procedure is not called.

If an exception occurs, the LINKLIBRARY-RESULT predefined identifier can be used to find out which type of exception has occurred. Refer to "Linkage Between user Programs and Libraries" in Section 11, "Library Concepts," for more information.

NOT ON EXCEPTION imperative-statement-2

If the called library links successfully and all entry points are in the library, the library entry procedure will be called and imperative-statement-2 is executed.

Details

National-character data-items and national-character literals cannot be passed in the USING phrase of this format of the CALL statement.

Refer to the Local-Storage Section of the Data Division for information on passing parameters in library programs.

Control and the ON EXCEPTION Phrase

If the program cannot link to the library containing the entry procedure specified in the CALL statement, or if the program successfully links to the library but not all entry points exist in the library, one of the following actions occurs:

- If the ON EXCEPTION phrase is specified, control is transferred to imperative-statement-1. Execution then continues according to the rules for each statement specified in imperative-statement-1.

  If a procedure branching or conditional statement that causes explicit transfer of control is executed, control is transferred according to the rules for that statement. Otherwise, upon completion of imperative-statement-1, control is transferred to the end of the CALL statement and the NOT ON EXECPTION phrase, if specified, is ignored.

- If ON EXCEPTION is not specified in the CALL statement, the imperative-statement-2 in the NOT ON EXCEPTION phrase, if specified, is ignored.

Examples

The following examples show an ALGOL library program and three possible COBOL programs that could call the ALGOL library.

ALGOL Library Program

```
BEGIN
     PROCEDURE ENTRYPOINT( A, B);
          STRING A;
          INTEGER B;
          BEGIN
               B := B + 1;
               DISPLAY( A );
          END;

     EXPORT ENTRYPOINT;
     FREEZE( TEMPORARY );
END.
```

This library program takes two parameters, A and B, passed in from the calling program. It adds 1 to B, and then displays A. Control then returns to the calling program.

COBOL74 Library Program

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  A     PIC X(010)      VALUE ALL "*".
01  B     PIC 9(011) COMP VALUE ZERO.
PROCEDURE DIVISION.
P1.
    CALL "ENTRYPOINT OF OBJECT/TEXT/X/STR1"
         USING STRING( A ) INTEGER( B ).
    DISPLAY B.
    STOP RUN.
```

This COBOL74 program calls the ALGOL program, passing the parameters A (filled with "*") and B (filled with zero). Once control returns from the ALGOL program, the COBOL74 program displays the value returned in B and stops.

COBOL85 Implicit Library Program

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  REC.
    02  A     PIC X(010)      VALUE ALL "*".
    02  B     PIC 9(011) COMP VALUE ZERO.
PROCEDURE DIVISION.
P1.
    CALL "ENTRYPOINT OF OBJECT/TEXT/X/STR1"
         USING STRING( A ) BY CONTENT INTEGER( B ).
    DISPLAY B.
    STOP RUN.
```

This COBOL85 program is exactly like its COBOL74 counterpart, except that the data items A and B have been made part of a group item called REC, and the data item B is passed to the library program BY CONTENT. Since B is passed BY CONTENT, the value of B does not change when control is returned to the calling program. Consequently, when B is displayed, it still contains zero.

COBOL85 Explicit Library Program

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  A     PIC X(010)      VALUE ALL "*".
01  B     PIC 9(011) COMP VALUE ZERO.
LOCAL-STORAGE SECTION.
LD  LOCAL1.
01  C     PIC X(010) DISPLAY STRING.
01  D     PIC 9(011) COMP   INTEGER.
PROGRAM-LIBRARY SECTION.
LB  ALGOLLIB IMPORT
    ATTRIBUTE TITLE IS "OBJECT/TEXT/X/STR1".
    ENTRY PROCEDURE ENTRYPOINT WITH LOCAL1 USING C D.
PROCEDURE DIVISION.
P1.
    CALL ENTRYPOINT USING A B.
    DISPLAY B.
    STOP RUN.
```

This is the same program as the two previous programs, except that the library is declared as an explicit library rather than an implicit one.

Format 6: CALL for Initiating a Synchronous, Dependent Process

This format of the CALL statement enables a COBOL85 program to execute a separately compiled program as a synchronous, dependent process.

```
CALL task-variable WITH section-name [ USING actual-parameter-list ].
```

Explanation

task-variable

This specifies the task variable that is to be associated with the process declared in the USE EXTERNAL statement of the specified section-name. The task variable associates a process with its program so that when the program modifies task attribute values, the system knows which process is to be affected. The task variable must be declared as a data item in the Working-Storage section of the Data Division (refer to the USAGE clause in Section 4 for details). For more information about task variables, refer to Section 13.

section-name

This identifies the section in the Procedure Division that contains the name of the object code file that is to be initiated by this CALL statement. You must define the section-name in the Declaratives Section of the Procedure Division and follow the definition with a USE EXTERNAL statement that specifies the name of the object code file.

USING actual-parameter-list

The USING phrase indicates the parameters in the calling program that are to be passed to the called program.

You can include the USING phrase only if a USING phrase occurs in the Procedure Division header of the called program and in the USE statement of the section identified by section-name in the calling program.

The parameters in the USING phrase can be a combination of any 77-level items that reside in the stack or 01-level items. A 77-level item that resides in the stack would be of USAGE BINARY, REAL or DOUBLE. In general, the level number, type, length, and order of items in the USING phrase of the calling and called programs must be identical. However, the items in the following list are interchangeable as parameters, that is, each item can be passed to and received by the other. The lengths of the associated items must be the same, however, or run-time errors might occur.

Interchangeable Group Items

- BINARY
- COMP
- DISPLAY
- DOUBLE
- REAL

Other Interchangeable Items

- DOUBLE items with RECEIVED BY REFERENCE clause
- 77-level BINARY REAL data items

Files to be passed as parameters must have a record description. The record description itself can be passed as a parameter. The USING phrase in the Procedure Division header of the called program must not reference any data item in the File Section of the called program. Both the calling and the called programs can read from and write to the file passed as a parameter in the CALL statement.

Including a task-variable in the USING phrase enables the called program to make references to the calling program.

Variables can be passed by value or by reference. Table 6–4 describes the matching of formal parameters between the COBOL74/85, ALGOL, and COBOL68 languages.

Table 6–4. Parameter Mapping for Tasking Calls

COBOL74/85 Parameter	ALGOL Parameter	COBOL68 Parameter
77-level REAL or BINARY item (single precision)	REAL, INTEGER	77-level COMP or COMP-4 item (single precision)
77-level DOUBLE or BINARY item (double precision)	DOUBLE	77-level COMP or COMP-5 item (double precision)
01-level DISPLAY, COMP, BINARY, REAL, or DOUBLE item	REAL ARRAY[*] INTEGER ARRAY[*] EBCDIC ARRAY[*] HEX ARRAY[*] REAL ARRAY[0] INTEGER ARRAY[0] EBCDIC ARRAY[0] HEX ARRAY[0]	01-level DISPLAY, COMP, OR COMP-2 group item with or without LOWER-BOUNDS
77-level EVENT or LOCK item	EVENT	77-level or 01-level EVENT or LOCK item
77-level or 01-level TASK elementary item	TASK	77-level or 01-level TASK elementary item
01-level EVENT or LOCK group item	EVENT ARRAY	77-level EVENT or LOCK group item
01-level TASK group item	TASK ARRAY	01-level TASK group item
FILE	FILE	FILE
BIT, 77 SYNC RIGHT	BOOLEAN	BIT, 77 Boolean-expression
BIT, 01 SYNC RIGHT	BOOLEAN ARRAY	BIT, 01

Details

When the CALL statement is executed, the calling program is suspended, and the called program is initiated. Upon initiation, the values of any parameters referenced in the USING phrase of the calling program are made available to the called program.

Naming the Program to Be Called

You can specify the name of the program to be called in one of the following ways:

- Put a CHANGE statement before the CALL statement that changes the NAME attribute of the task variable.
- Define a mnemonic-name in the Special-Names paragraph of the Environment Division, and then use it in the USE EXTERNAL statement.
- Use the following steps:
  1. Declare a data item in the Working-Storage Section of the Data Division.
  2. Name the data item in a USE EXTERNAL statement in the Declarative Section of the Procedure Division.
  3. Assign the object code file title to the data item by using a MOVE statement in the Procedure Division.

For program examples that show how to declare the name of the program to be called, refer to Section 13.

How Processor Time Is Shared

Processor control is passed between the calling and the called programs as follows:

When the called program . . .	Then the calling program . . .
Executes an EXIT PROGRAM statement	Resumes execution. The calling program can reinitiate the called program by executing a CONTINUE statement.
Terminates abnormally	Resumes execution beginning with the statement following the CALL statement. The calling program must execute another CALL statement to reinitiate the called program.
Executes a STOP RUN statement	Resumes execution beginning with the statement following the CALL statement. The calling program must execute another CALL statement to reinitiate the called program.

If the calling program is terminated before the called program, a critical block exit error occurs. For details about this type of error and how to prevent it, refer to Section 13.

For program examples that show how control is passed between two programs, refer to Section 13.

Format 7: CALL MODULE

This format of the CALL statement transfers control to a portion of code in an externally compiled program bound into the calling program. You can specify a section-name or a user-defined program-name as an entry point.

```
CALL MODULE [ "section-name" OF ] "program-id"

       { "file-name"  }
FROM   {              }  [ ON "family-name"]
       { MODULEFILE   }
```

Explanation

"section-name"

This name is a user-defined word that you specify in the nondeclarative portion of the Procedure Division in the program being called. Double-byte section names cannot be used in this statement. Refer to "Nondeclarative Procedure Format" in Section 5 for details on declaring a section-name.

"program-id"

This identifier is a user-defined word that you specify in the PROGRAM-ID paragraph in the Identification Division of the program being called.

"file-name"

This name is the file name of the code file that contains the program being called.

"family-name"

This name is the family on which the code file that contains the called program resides. The ON "family-name" phrase is optional if you have specified a default family name with the MODULEFAMILY compiler option. See Section 15, "Compiler Operations" for details.

MODULEFILE

This keyword indicates that the file name specified by the MODULEFILE compiler option is the name of the code file that contains the called program.

Details

You must include the CALLMODULE compiler option in the *called* program to ensure that it contains the necessary structures to make the call possible.

When the calling program enters a section of code in another program, the code in the called program is executed until either a RETURN statement or the end of the program is encountered. You can specify the end of a section with an EXIT MODULE statement, which causes control to be returned to the calling program.

Example

Calling Program TEST/CALLER

```
$ BINDINFO
$ SET LIST CODE
  IDENTIFICATION DIVISION.
  PROGRAM-ID.  HOST.
  DATA DIVISION.
  WORKING-STORAGE SECTION.
  01  CO-ITEM         PIC X(36).
  01  ORIG            PIC X(36).
  01  NEW             PIC X(36).
  PROCEDURE DIVISION.
  FIRST-PA SECTION.
  START-PA.
           DISPLAY "WE ARE STARTING NOW !".
  SECOND-PA SECTION.
  START-PA2.
            MOVE "THIS IS A COMMON DATA ITEM." TO CO-ITEM.
            CALL MODULE "MODA" OF "TEXTMOD"
                 FROM "OBJECT/TEST/CALLED" ON "DISK".
            CALL MODULE "MODB" OF "TEXTMOD"
                 FROM "OBJECT/TEST/CALLED" ON "DISK".
            CALL MODULE "MODC" OF "TEXTMOD"
                 FROM "OBJECT/TEST/CALLED" ON "DISK".
            CALL MODULE "MODD" OF "TEXTMOD"
                 FROM "OBJECT/TEST/CALLED" ON "DISK".
            STOP RUN.
```

Called Program TEST/CALLED

```
$SET LEVEL = 3 CALLMODULE
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  TESTMOD.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  CO-ITEM  PIC X(36) COMMON.
 PROCEDURE DIVISION.
 MODA SECTION.
 START-MODA.
      DISPLAY "WE'VE ARRIVED AT MODA".
      DISPLAY CO-ITEM.
 EXIT-PARA.
      EXIT MODULE.
 MODB SECTION.
 START-MODB.
      DISPLAY "WE'VE ARRIVED AT MODB".
 EXIT-MODB.
      EXIT MODULE.
 MODC SECTION.
 START-MODC.
      DISPLAY "WE'VE ARRIVED AT MODC".
 EXIT-MODC.
      EXIT MODULE.
 MODD SECTION.
 START-MODD.
      DISPLAY "WE'VE ARRIVED AT MODD".
```

Binder File TEST/BIND

```
 HOST IS OBJECT/TEST/CALLER;
```

Explanation

The calling begins with the calling program initiating the called program at the MODA SECTION. After the MODA SECTION in the called program is executed, an EXIT MODULE statement is encountered, and control is returned to the next CALL MODULE statement in the calling program. The calling continues in a similar way until the end of the called program is encountered (after the MODD SECTION). Then control returns to the calling program, which executes a STOP RUN.

Notice that the LEVEL compiler option is set to 3 in the *called* program. A called program must be compiled at level 3 or higher to use the EXIT MODULE statement.

Refer to the *Binder Programming Reference Manual* for information on how to use the Binder file.

CANCEL Statement

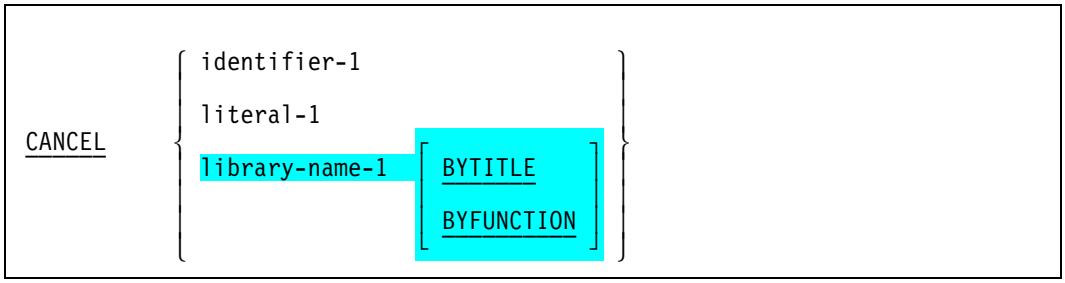
The CANCEL statement breaks the link between the called program and a calling program. The next time the program is called, it will be in its initial state.

For information on program-naming conventions, refer to "User-Defined Words" in Section 1.

Refer to "PROGRAM-ID Paragraph" in Section 2 for information on how a program receives the initial attribute.

For conceptual information on interprogram communication and the use of the CANCEL statement, refer to Section 10.

Refer to "CALL Statement" and "EXIT Statement" in this section for more information.



Explanation

identifier-1

This identifier must reference an alphanumeric data-item whose value is consistent with program-naming conventions.

The content of the data-item referenced by the identifier can identify the program to be canceled.

literal-1

This is a nonnumeric literal that identifies the name of the program to be canceled.

library-name-1

This is the name of the library to be canceled. Cancellation of a library causes the program to be delinked from the library.

Literal-1 must be the file title of the library if the BYTITLE option is specified, or the function name of the library if the BYFUNCTION name is specified. The library name and the option you choose must match those used when the library program was called.

BYTITLE
BYFUNCTION

These options are described as follows:

- BYTITLE indicates that the library was referred to by its file title in the CALL statement of the calling program.
- BYFUNCTION indicates that the library was referred to by its function name in the CALL statement of the calling program.

  The library name and the option you choose must match those used when the library program was called.
- If you do not specify an option, BYTITLE is assumed.

Details

When a program is canceled, the contents of data items in external data records described by that program are not changed.

When Cancellation Occurs

A called program is canceled when any one of the following occurs:

- When it is referred to as the operand of a CANCEL statement
- At the termination of the run unit of which the program is a member
- When an EXIT PROGRAM statement is executed in a called program that has the initial attribute

All programs contained in the program referenced by the CANCEL statement are also canceled.

Explicit and Implicit Cancel Statements

A CANCEL statement can be explicit or implicit. An explicit cancellation occurs when one program cancels another. An implicit cancellation occurs with nested calls. Consider the following example:

1. Program A calls Program B, and Program B calls Program C.
2. Program A contains a CANCEL statement to cancel Program B.
3. This statement, in effect, cancels Program C and then cancels Program B.

Program B is canceled explicitly, because it is directly canceled through a CANCEL statement in Program A. Program C is canceled implicitly because its parent program, Program B, was canceled.

After the execution of an explicit or implicit CANCEL statement, the referenced program does not have a logical relationship to the run unit in which the CANCEL statement appears. If the program referenced by a successfully executed explicit or implicit CANCEL statement in a run unit is then called in that run unit, that program is in its initial state.

No action is taken when an explicit or implicit CANCEL statement is executed naming a program that has not been called into the run unit or that has been called and is presently canceled. Instead, control is transferred to the next executable statement following the explicit CANCEL statement.

During execution of an explicit or implicit CANCEL statement, an implicit CLOSE statement without optional phrases is executed for each file in the open mode that is associated with an internal file connector in the program named in the explicit CANCEL statement. USE procedures associated with these files are not executed.

Rules for Referenced Programs

A program named in a CANCEL statement in another program must be callable by that other program.

A program named in the CANCEL statement must not refer directly or indirectly to any program that has been called and has not yet executed an EXIT PROGRAM statement.

You can establish a logical relationship to a canceled program only by executing a subsequent CALL statement that names the program.

Examples

```
03 Nme PIC X(6) VALUE  "PROG-1".
      .
      .
      .
CANCEL NME.
```

This cancels the called program PROG-1. NME is an identifier, which contains a program-name.

```
CANCEL  "AUDIT1" "AUDIT2".
```

This cancels the called programs AUDIT1 and AUDIT2.

```
CANCEL "AUDIT1", NME, "OBJECT/AUDIT2".
```

This cancels the called programs AUDIT1, PROG-1, and OBJECT/AUDIT2.

CAUSE Statement

The CAUSE statement initiates the specified events.

```
CAUSE [ AND RESET ] event-identifier-1 [ ,event-identifier-2 ] . . .
```

Explanation

[AND RESET]

This phrase causes the specified events to be immediately reset for later use. Using this phrase prevents the interrupt procedure from having to reset the events.

event-identifier-1, event-identifier-2 . . .

The event-identifier can be one or more of the following:

- The name of a data-item declared with the USAGE IS EVENT phrase. The data-name must be properly qualified and properly subscripted.
- A task attribute of type EVENT. The two event task attributes are ACCEPTEVENT and EXCEPTIONEVENT. For details about these task attributes, refer to the *Task Attributes Programming Reference Manual*.
- A file attribute of type EVENT. The three event file attributes are CHANGEEVENT, INPUTEVENT, and OUTPUTEVENT. For details about these files attributes, refer to the *File Attributes Programming Reference Manual*.

Details

When a process is suspended because it encountered a WAIT event-identifier statement, and the CAUSE statement activates that event-identifier, the process resumes execution.

Activating events has the following effect upon interrupt procedures:

When an EVENT item is activated by a CAUSE statement and . . .	Then . . .	And . . .
The ALLOW INTERRUPT statement was previously used.	The calling program is suspended.	All interrupt procedures attached to that event are executed immediately.
An interrupt procedure attached to that event was previously readied by the ALLOW section-name statement.	The calling program is suspended.	The interrupt procedure is executed immediately.
The DISALLOW INTERRUPT statement was previously used.	The calling program continues executing.	All interrupt procedures attached to that event are queued.
An interrupt procedure attached to that event was previously restricted by the DISALLOW section-name statement.	The calling program continues executing.	The interrupt procedure is queued.

Refer to the ALLOW and DISALLOW statements for additional information.

If an event item activated by the CAUSE statement is tested as a conditional expression in an IF statement, the event condition returns the value TRUE.

An event activated by the CAUSE statement remains activated until it is explicitly deactivated by a RESET statement.

Example

```
CAUSE WS-EVENT (3).

CAUSE AND RESET WS-77-EVENT.
```

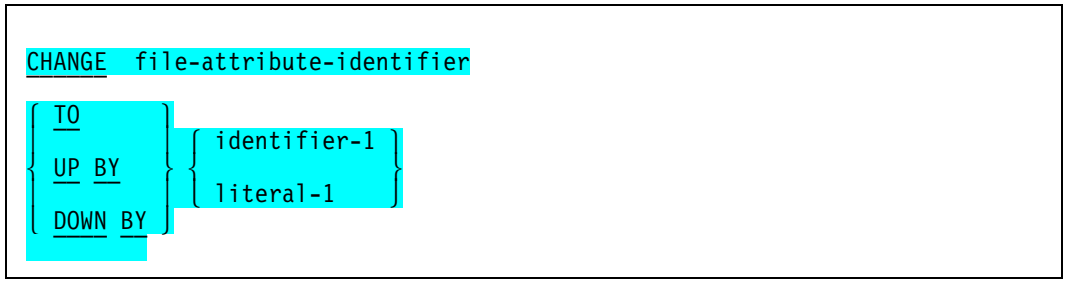
CHANGE Statement

The CHANGE statement enables you to change the value of a file, library, or task attribute.

This statement is fully supported in the TADS environment.

Format	Use
Format 1	This format changes the value of a numeric file attribute.
Format 2	This format changes the value of an alphanumeric file attribute.
Format 3	This format changes the value of a mnemonic file attribute.
Format 4	This format changes the value of a library attribute.
Format 5	This format changes the value of a task attribute.

Format 1: Changing the Value of a Numeric File Attribute

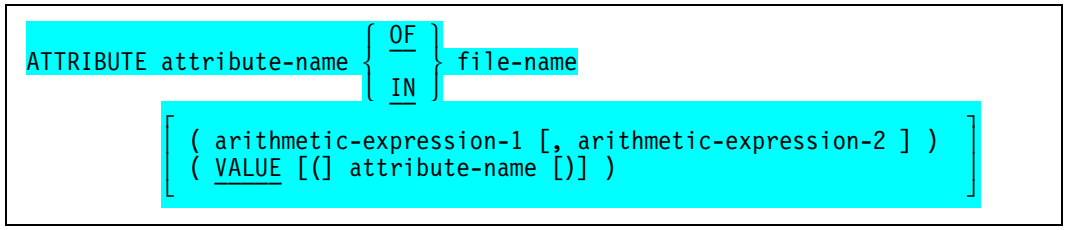


This format is supported in the TADS environment.

Explanation

file-attribute-identifier

This is the syntax that identifies the file attribute whose value you want to change. The syntax of this clause is as follows:



attribute-name

This is the name of one of the file attributes. For a comprehensive list of all the file attributes, refer to the *File Attributes Programming Reference Manual*.

file-name

This is the name of the file whose attribute values you want to change.

arithmetic-expression

If arithmetic-expression-1 is used with a port file, the value of the expression must specify which subfile of the file is affected. A subfile index is required for accessing or changing attributes of a subfile of a port file.

If arithmetic-expression-1 is . . .	Then . . .
Not specified	The attribute of the port is accessed.
Specified and its value is nonzero	The value of the expression specifies a subfile index and causes the attribute of the subfile to be accessed.
Specified and its value is zero	The attributes of all subfiles are accessed.

If an arithmetic expression is used with a disk file, the values of arithmetic-expression-1 and arithmetic-expression-2 must specify the row and copy parameters for the file.

VALUE (attribute-name)

This phrase is valid for use only with the FILEEQUATED attribute.

identifier-1
literal-1

Identifier-1 must be a numeric data item that represents an integer. Literal-1 must be a numeric literal.

UP BY
DOWN BY

These descriptors are for use only with the STATIONLIST file attribute. When UP BY is specified, the current value of the STATIONLIST attribute is increased by the value of identifier-1 or literal-1. When DOWN BY is specified, the value is decreased by the value of identifier-1 or literal-1.

Example

```
CHANGE ATTRIBUTE BLOCKSIZE OF INPUTFILE TO 420.
```

This statement changes the BLOCKSIZE attribute of the file INPUTFILE to the value 420.

Format 2: Changing the Value of an Alphanumeric File Attribute

```
CHANGE  file-attribute-identifier TO  { identifier-1 }
                                      { literal-1    }
```

This format is supported in the TADS environment.

Explanation

file-attribute-identifier

This syntax identifies the file attribute whose value you want to change. The syntax of this identifier is provided with the explanation of Format 1.

identifier-1
literal-1

Identifier-1 must be a nonnumeric, DISPLAY data item that ends with a period (.). Literal-1 must be a nonnumeric literal.

Examples

```
CHANGE ATTRIBUTE TITLE OF MY-FILE TO "MY/FILE".
```

This example shows how to specify a file name for the TITLE file attribute.

```
CHANGE ATTRIBUTE YOURIPADDRESS OF PFILE TO """192.39.0.20"".".
```

This example shows how to specify the IP address when writing a port file program for TCP/IP. Note that the IP address must be enclosed in quotation marks.

Format 3: Changing the Value of a Mnemonic File Attribute

```
CHANGE  file-attribute-identifier TO
        [ VALUE ] [ ( ] mnemonic-attribute-value [ ) ]
```

This format is supported in the TADS environment.

Explanation

file-attribute-identifier

This syntax identifies the file attribute whose value you want to change. The syntax for this identifier is provided in the explanation of Format 1.

mnemonic-attribute-value

This is the mnemonic value that you want to assign to the specified file attribute. If a data-name has the same name as the mnemonic-attribute-value, the value assigned to the attribute is dependent upon whether the optional word VALUE is used. If the word VALUE is present, the attribute is set to the value of the mnemonic. If the word VALUE is omitted, the attribute is set to the current value of the data-name.

Details

The current state of the file might inhibit the changing of certain file attributes. Some file attributes cannot be changed while the file is in open mode. Also, some file attributes cannot be changed until the file is opened. For more information on file attributes, refer to Section 12.

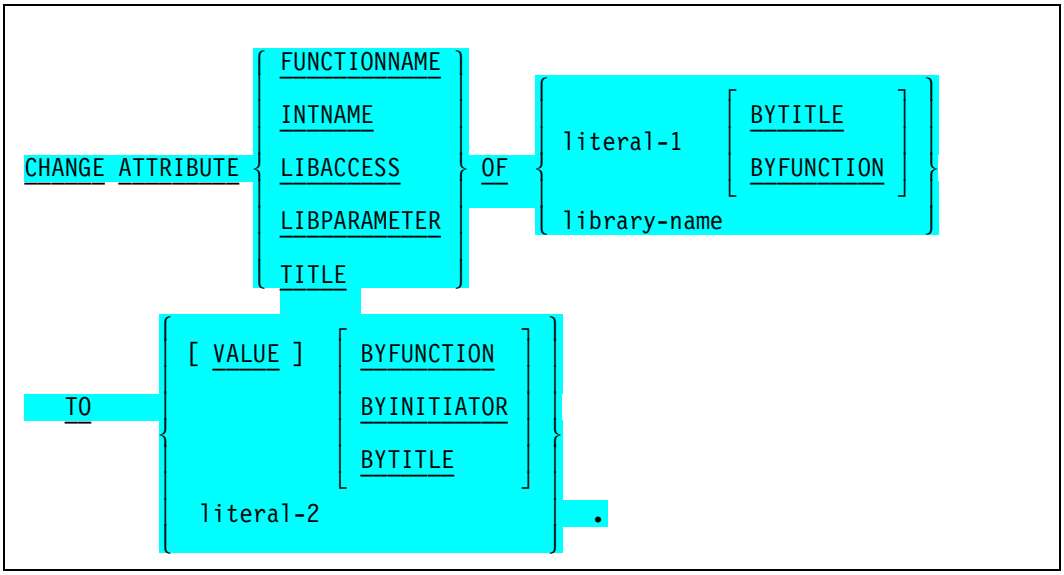
Certain file attributes are used by the compiler to implement various constructs required to declare and access files within the program. Whenever possible, it is preferable to use standard COBOL syntax for setting or declaring a file attribute that is used also by the compiler.

Example

```
CHANGE ATTRIBUTE UNITS OF INPUTFILE TO VALUE WORDS.
```

This changes the UNITS attribute of the file INPUTFILE to the value associated with the mnemonic WORDS.

Format 4: Changing the Value of a Library Attribute



This format is supported in the TADS environment.

Explanation

FUNCTIONNAME
INTNAME
LIBACCESS
LIBPARAMETER
TITLE

These are the library attributes you can change. The FUNCTIONNAME, INTNAME, LIBPARAMETER, and TITLE library attributes are either string or DISPLAY library attributes. The LIBACCESS library attribute is a mnemonic library attribute. For a description of the library attributes, refer to Section 11.

literal-1

This option identifies the library whose attribute is to be changed. If the library is identified by a literal, the library is assumed to be a COBOL74 or implicit library program.

BYTITLE
BYFUNCTION

This indicates whether the CALL statement called the library by its file title or by its function name. BYTITLE indicates that the library was referred to by its file title in the CALL statement. BYFUNCTION indicates that the library was referred to by its function name in the CALL statement. The library name and the option you choose must match those used in the CALL statement.

library-name

If the library is identified by an unquoted library-name, the library is assumed to be an explicit library program. The library-name must have been previously declared in the Program-Library Section of the program containing the CHANGE statement.

BYFUNCTION
BYTITLE
BYINITIATOR

You can assign one of these values to the LIBACCESS library attribute. The LIBACCESS attribute specifies the way in which the library object code file is to be accessed when the library is called.

- If LIBACCESS is equal to BYTITLE, then the TITLE attribute of the library is used to find the object code file. BYTITLE is the default value.
- If LIBACCESS is equal to BYFUNCTION, then the FUNCTIONNAME attribute of the library is used to access the MCP library function table, and the object code file associated with that FUNCTIONNAME is used.
- If LIBACCESS is equal to BYINITIATOR, then the library that initiated the program is the library that is accessed.

For a discussion of library attributes, refer to Section 11.

literal-2

This literal is the value to be assigned to the FUNCTIONNAME, LIBPARAMETERS, or TITLE library attribute.

Examples

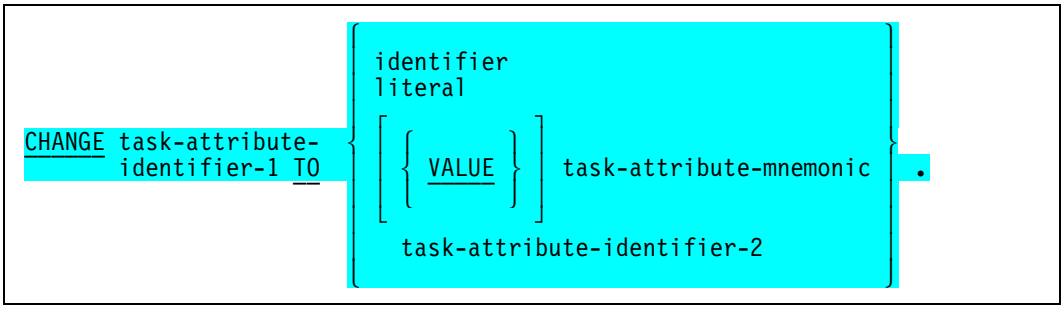
```
CHANGE ATTRIBUTE FUNCTIONNAME OF LIB/TEST/1 TO LIBOPS.
```

This statement changes the system function name of the library titled LIB/TEST/1 to LIBOPS.

```
CHANGE ATTRIBUTE LIBACCESS OF LIB/TEST/1 TO BYFUNCTION.
```

This statement changes the value of the LIBACCESS attribute of the library named LIB/TEST/1 to BYFUNCTION.

Format 5: Changing the Value of a Task Attribute



This format is supported in the TADS environment.

Explanation

task-attribute-identifier-1

This identifies the task attribute whose value you want to change. The syntax for the task attribute identifier is provided with the description of task-attribute-identifier-2.

identifier
literal

This is the value you want to assign to the task attribute.

- If the task attribute requires a numeric value, the identifier must be a numeric data item that represents an integer, or the literal must be a numeric literal.
- If the task attribute requires an alphanumeric value, the identifier must be a nonnumeric DISPLAY data item that ends with a period (.), or the literal must be a nonnumeric literal.

For details about task attributes, see the *Task Attributes Programming Reference Manual*.

task-attribute-mnemonic
VALUE task-attribute-mnemonic

This name is associated with a constant value for an attribute that has a set number of predetermined possible values. If the same name is used for a data-name and a task-attribute-mnemonic, the value assigned to the attribute is determined by the presence of the word VALUE. If the word VALUE is present, the attribute is set to the value of the mnemonic. If the word VALUE is omitted, the attribute is set to the current value of the data-name.

You must use task-attribute-mnemonics to access specific options of the OPTION task attribute. The mnemonics represent specific bits in the OPTION task attribute word. You can access these bits by using Format 3 of the MOVE statement.

Note that attribute mnemonics are not treated as COBOL reserved words. They are reserved only within the context in which they are used and can also be used as data names or procedure names.

task-attribute-identifier-2

This identifier enables you to change the attribute value to the value used by the same attribute of another process. The syntax for the task-attribute-identifier is as follows:

```
ATTRIBUTE attribute-name OF  { task-variable [ (subscript) ] }
                             { MYSELF                        }
                             { MYJOB                         }
                             { ATTRIBUTE attribute name OF ... }
```

attribute-name

This is the name of one of the task attributes. The complete set of task attributes is documented in the *Task Attributes Programming Reference Manual*.

Attribute names are not reserved words. They are reserved only within the context in which they are used and can also be used as data names or procedure names.

task-variable [(subscript)]

This is the task variable that is associated with the process whose task attribute value you want to change. The optional subscript is used to identify a specific task variable when multiple task variables are declared with an OCCURS clause. A maximum of one subscript is permitted. For an example of how a task variable is used with a subscript, refer to "Example of Passing Control Between Two Programs" in Section 13.

A user-declared task variable must be declared as a data item in the Working Storage Section of the Data Division.

MYSELF
MYJOB

These are system-declared task variables. MYSELF refers the process itself. MYJOB refers to the independent process in a group of related dependent processes (the process family).

ATTRIBUTE attribute-name OF . . .

This syntax gives a process access to the task attributes of an associated process. For example, you could specify the parent of TASK-EXAMPLE1 by using the following syntax:

```
ATTRIBUTE NAME OF ATTRIBUTE EXCEPTIONTASK OF TASK-EXAMPLE1
```

Details

You can display the value of any task attribute, except string-type task attributes (attributes whose values are character strings), by using the DISPLAY statement. For string-type task attributes, you must move the attribute into a data area with the MOVE statement, and then display the value with the DISPLAY statement.

Attributes with an implicit numeric class can be used in DISPLAY statements and in place of any identifier in an arithmetic statement, except the receiving-field identifier.

You can determine the mnemonic value of a task attribute by using the task attribute in a conditional expression. For details about conditional expressions, see Section 5.

In general, the types of task attributes and the values that are valid for them are shown in the following table. For detailed information about task attributes and their values, refer to the *Task Attributes Programming Reference Manual*.

For the attribute type of . . .	**The accepted and returned values are . . .**
String	Alphanumeric
Boolean	Numeric (or the value associated with a mnemonic)
Integer	Numeric (or the value associated with a mnemonic)
All other attributes types	Numeric identifier, literal, arithmetic expression, or the value associated with a mnemonic

If the value you assign to a task attribute is not within the permissible range for the specified attribute, an error occurs at the time of compilation or execution.

Examples

```
CHANGE ATTRIBUTE OPTION OF VERSION1/TEST TO TODISK.
```

This first example changes the value of the OPTION attribute of the process named VERSION1/TEST to the value TODISK. (The OPTION attribute affects program dump contents, job summary printing, and backup file handling. For details, refer to the *File Attributes Programming Reference Manual*.)

```
CHANGE ATTRIBUTE BLOCKSIZE OF VERSION1/TEST TO ATTRIBUTE BLOCKSIZE OF
STANDARD/SYS/RUN.
```

This second example changes the value of the BLOCKSIZE attribute of the process named VERSION1/TEST to the value used by the BLOCKSIZE attribute of the process named STANDARD/SYS/RUN.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-TASK         TASK.
01 WS-NAME         PIC X(50).
PROCEDURE DIVISION.
MAIN SECTION.
MAIN-PARA.
    CHANGE ATTRIBUTE NAME OF WS-TASK TO "TEST/RUNFILE/FEB".
    MOVE ATTRIBUTE NAME OF WS-TASK TO WS-NAME.
    DISPLAY WS-NAME.
    STOP RUN.
```

This program example illustrates how you can change the name of a task by using the CHANGE statement, and how you can verify the name change by using the MOVE and DISPLAY statements. The NAME task attribute is a string-type task attribute, which means that its value is a character string. You must use the MOVE and DISPLAY statements to check the value of a string-type task attribute. For all other task attributes, you can use just the DISPLAY statement.

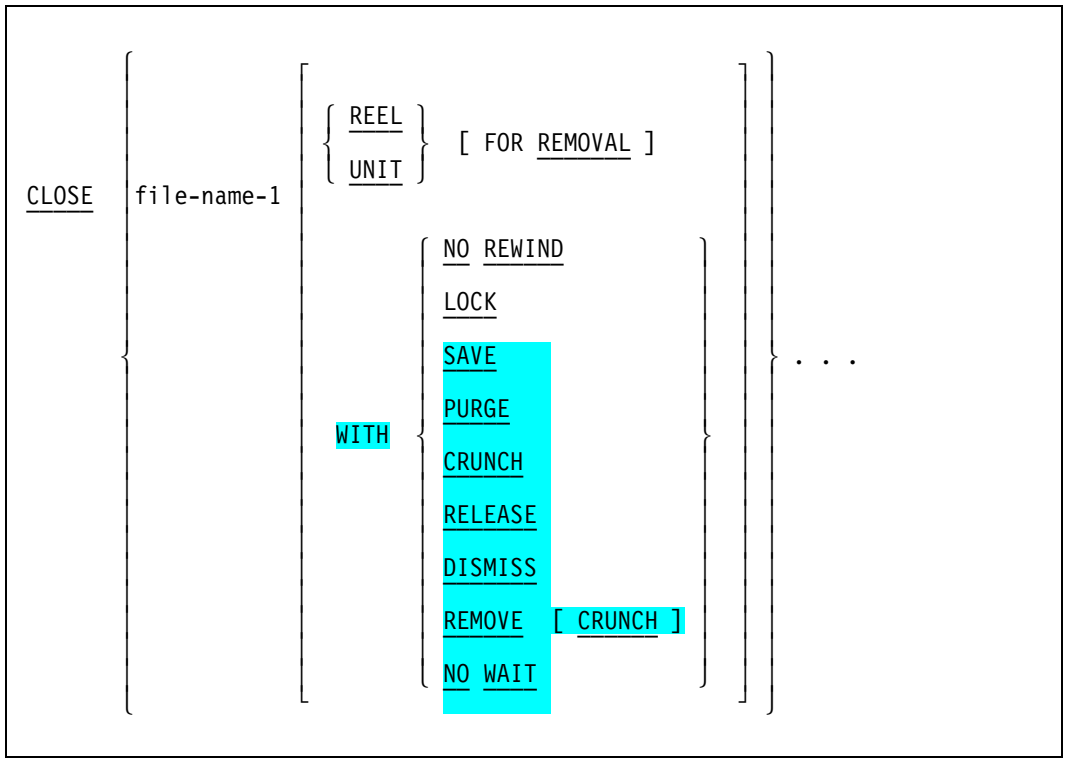
CLOSE Statement

The CLOSE statement ends the processing of a file or a reel/unit of a file. Also, it can specify the disposition of the file and the device to which the file is assigned.

This statement is partially supported in the TADS environment. Applicable exclusions are noted in this section.

Format	Use
Format 1	This format ends the processing of sequential files.
Format 2	This format ends the processing of either relative or indexed files.

Format 1: Sequential I-O



This format is supported in the TADS environment.

Explanation

file-name-1

This name is a user-defined word that specifies the name of the file to be closed.

The specified file must be in an open mode.

Files referenced in the CLOSE statement can have different organizations and access modes.

REEL
UNIT

These are equivalent.

The reel/unit is closed and rewound.

Treatment of sequential mass storage files is logically equivalent to the treatment of a file on tape or a similar sequential medium.

Treatment of a file contained in a multiple-file tape environment is logically equivalent to the treatment of a sequential single-reel/unit file, if the file is contained on one reel.

The REEL or UNIT phrase and the NO REWIND option cannot be specified together in a CLOSE statement.

FOR REMOVAL

This option is used for sequential single-reel/unit files and multi-reel/unit files. The reel/unit is closed, and the system waits for the next reel/unit.

NO REWIND

The file is closed, and the current reel/unit is left in its current position.

The NO REWIND option and the REEL or UNIT phrase cannot be specified together in a CLOSE statement.

LOCK

The logical file is marked as locked, so that it cannot be reopened during the execution of the program. If the file is a mass-storage file, it becomes a permanent file before it is made unavailable. If the file is assigned to tape, the physical unit is made not ready.

SAVE

This disposition is valid only for mass-storage files. The file is made permanent and can be reopened during execution of the program.

PURGE

This disposition is valid only for files assigned to tape or to mass-storage devices.

If the file is assigned at execution time to a tape device, the reel is rewound. If the reel has a write ring, a scratch label is written on it, and the device is released as available to the system.

If the file is a permanent mass-storage file, the file-name is removed from the directory of the system, and the mass-storage area occupied by the file is released as available to the system.

PURGE is generally used for a temporary file that you have used and then want to relinquish to free the space allocated for the file.

RELEASE
DISMISS

DISMISS and RELEASE are synonymous.

This disposition severs the association between the logical file and the physical file. The areas of memory allocated for buffers can be released to the system.

If the device to which the file was assigned can be controlled by the object program, it is released as available to the system.

REMOVE

This disposition is valid only for mass-storage files.

The file is closed, and the association between the logical file and the physical file is severed. The file is made permanent and can be reopened during execution of the program.

CRUNCH

This disposition is valid only for mass-storage files.

The file is made a permanent file. Unused portions of mass-storage areas allocated for the file are released as available to the system. The file cannot subsequently be extended by opening the file with an OPEN EXTEND statement.

NO WAIT

The WITH NO WAIT phrase can be specified only for port files. It is mutually exclusive with all other CLOSE options.

With this option, the program does not wait until the file is closed before resuming execution. Control is returned to the next statement without waiting for the CLOSE operation to be completed.

Details

The execution of the CLOSE statement updates the value of the I-O status associated with the specified file. Refer to Table 3–1 for information on the I-O status codes.

A CLOSE statement can be executed only for a file in open mode. In general, a CLOSE statement changes the FILEUSE attribute of the file to I-O. This change can affect the results of any subsequent access to the RESIDENT, PRESENT, or AVAILABLE attribute of the file. (A CLOSE statement without a specified option retains the file and does not change the FILEUSE attribute of the file.)

End-of-file or reel/unit processing is performed for the file if an optional input file is present. Processing is not performed if an optional input file is not present. In this case, the file position indicator and the current volume pointer are unchanged.

Following the successful execution of a CLOSE statement without the REEL or UNIT phrase, the record area associated with the specified file is no longer available. After the unsuccessful execution of such a CLOSE statement, the record area remains unchanged.

Following the successful execution of a CLOSE statement without the REEL or UNIT phrase, the file is removed from the open mode, and the file is no longer associated with the file connector.

If more than one file-name is specified in a CLOSE statement, the result of executing this CLOSE statement is as if a separate CLOSE statement had been written for each file-name in the same order as specified in the CLOSE statement.

TADS

Any USE procedure is not executed when a CLOSE statement that is compiled and executed in a TADS session fails.

Effect of CLOSE Statements on Different Storage Media

In general, a CLOSE statement changes the FILEUSE attribute of the file to I-O. This change can affect the results of any subsequent access to the RESIDENT, PRESENT, or AVAILABLE attributes of the file. (A CLOSE statement without a specified option retains the file and does not change the FILEUSE attribute of the file.)

A CLOSE statement without file retention also checks the EXCLUSIVE attribute of the file during the CLOSE operation. If this attribute is found to be TRUE, it is set to FALSE during the CLOSE process.

The formats of the CLOSE statements affect various storage media differently. To show the effects of CLOSE statements on various storage media, all files are divided into the following categories:

- Non-reel/unit file

  This is a file whose input or output medium is such that the concepts of rewind, reels, and units have no meaning. This category includes mass-storage files.

  A CLOSE statement executed for a non-reel/unit file can affect the disposition of the device to which it is assigned. The CLOSE statement affects only the disposition of the physical file and its association with the logical file, not the disposition of the physical device.

- Sequential single-reel/unit file

  This is a sequential file that is entirely contained on one reel/unit.

- Sequential multi-reel/unit file

  This is a sequential file that is contained on more than one reel/unit.

Table 6–5 summarizes the results of executing each type of CLOSE statement for each category of file. Definitions of the numeric entries appear in the paragraphs following the table.

Table 6–5. Relationship of File Types and CLOSE Formats

CLOSE Statement Format	Non-Reel or Unit	Sequential Single-Reel or Unit	Sequential Multi-Reel or Unit
CLOSE	3 9	3 7 9	1 2 3 17
CLOSE REEL/UNIT	6 15	6 7 15	6 7
CLOSE REEL/UNIT FOR REMOVAL	6 15	4 6 7	4 6 7
CLOSE WITH NO REWIND	3 8 15	2 8 9	1 2 3 17
CLOSE WITH LOCK	3 5 10 11	3 5 7 10 11	1 3 5 7 10 11
CLOSE WITH SAVE	3 10 13	15	15
CLOSE WITH PURGE	3 12	3 7 12	1 3 7 12
CLOSE WITH RELEASE	3 10 11	3 7 10 11	1 3 7 10 11
CLOSE WITH DISMISS	3 10 11	3 7 10 11	1 3 7 10 11
CLOSE WITH REMOVE	3 10 13	15	15
CLOSE WITH CRUNCH	3 10 14	15	15
CLOSE WITH REMOVE CRUNCH	3 10 13	15	15
CLOSE WITH NO WAIT	16	15	15

The following paragraphs explain the meaning of the numerical values in Table 6-5. In these paragraphs, definitions apply to input, output, and input-output files. Alternate definitions are given where the type of file affects the definition.

1. Previous reels or units are closed.

   Input Files and Input-Output Files:

   All reels or units in the file before the current reel/unit are closed (except for those reels or units controlled by a prior CLOSE REEL or CLOSE UNIT statement). The reels or units in the file following the current one are not processed.

   Output Files:

   All reels or units in the file before the current reel/unit are closed (except for those reels or units controlled by a prior CLOSE REEL or CLOSE UNIT statement).

2. The current reel is not rewound.

   The current reel/unit is left in its current position.

3. The logical file is closed.

   Input Files and Input-Output Files:

   If the file is positioned at its end and label records are specified for the file, the labels are processed according to the system standard label conventions. The results of the CLOSE statement when label records are specified but not present, or when label records are not specified but are present, are unpredictable.

   If the file is positioned at its end and label records are not specified for the file, label processing does not take place. If the file is not positioned at its end, standard closing operations are executed, but there is no end label processing.

   Output Files:

   If label records are specified for the file, the labels are processed according to the system standard label conventions. The results of the CLOSE statement when label records are specified but not present, or when label records are not specified but are present, are unpredictable. If label records are not specified for the file, label processing does not take place.

4. The reel/unit is removed.

   The current reel/unit is rewound, when applicable, and the reel/unit is logically removed from the run unit.

   The reel/unit can be accessed again in its proper order of reels or units within the file. This can occur if a CLOSE statement without the REEL or UNIT phrase is subsequently executed for this file followed by the execution of an OPEN statement for the file.

5. The file is locked.

   The file cannot be opened again during this execution of the run unit.

6. The reel/unit is closed.

   Input Files and Input-Output Files:

   There is no reel/unit swap and the current volume pointer remains unchanged if one of the following conditions exists: if the current reel/unit is the last or only reel/unit for the file; or it the reel is a non-reel or non-unit medium.

   However, if another reel/unit exists for the file, or if a reel/unit swap occurs, the current volume pointer is updated to point to the next reel/unit that exists in the file. The standard beginning reel/unit label procedure is executed. Another reel/unit swap occurs if no data records exist for the current volume.

   Output Files (Reel/Unit Media):

   The standard ending reel/unit label procedure is executed. A reel/unit swap occurs, and the current volume pointer is updated to point to the new reel/unit, and the standard beginning reel/unit label procedure is executed. The next executed WRITE statement to reference that file directs the next logical data record to the next reel/unit of the file.

   Output Files (Nonreel or Nonunit Media):

   Execution of this statement is successful. The file remains in open mode, and no action takes place (except that the value of the I-O status associated with the specified file-name is updated).

7. Rewinding occurs.

   The current reel or a similar device is positioned at its physical beginning.

8. Optional phrases are ignored.

   The CLOSE statement is executed, and optional phrases, if present, are ignored.

9. The file is retained.

   The association between the logical file and the physical file is retained. Subsequent reopening of the file cannot require the operating system to search for the physical file.

10. The file is released.
11. The device is released.
12. The file is purged.
13. The file is saved.
14. The file is crunched.
15. The combination of CLOSE option and file category is illegal.

    If the CLOSE statement specifies the REEL or UNIT phrase, the CLOSE statement has no effect, and the file is not closed.

    If the CLOSE statement does not specify the REEL or UNIT phrase, any optional disposition is ignored, but the file is closed.

16. The WITH NO WAIT phrase can be specified only for port files and is mutually exclusive with all other CLOSE options.
17. The file is reserved.

Port Files

For a port file with an ACTUAL KEY clause, the value of the ACTUAL KEY determines which subfile of the file is to be closed. If the ACTUAL KEY value is nonzero, only the specified subfile is closed. If the ACTUAL KEY value is 0 or if the ACTUAL KEY is not specified, all opened subfiles are closed.

A CLOSE statement with no phrase specified causes the program to wait until the file is closed before resuming execution. This suspension is prevented for port files by specifying the WITH NO WAIT phrase, which causes control to be returned to the next statement without waiting for the CLOSE to be completed.

Examples

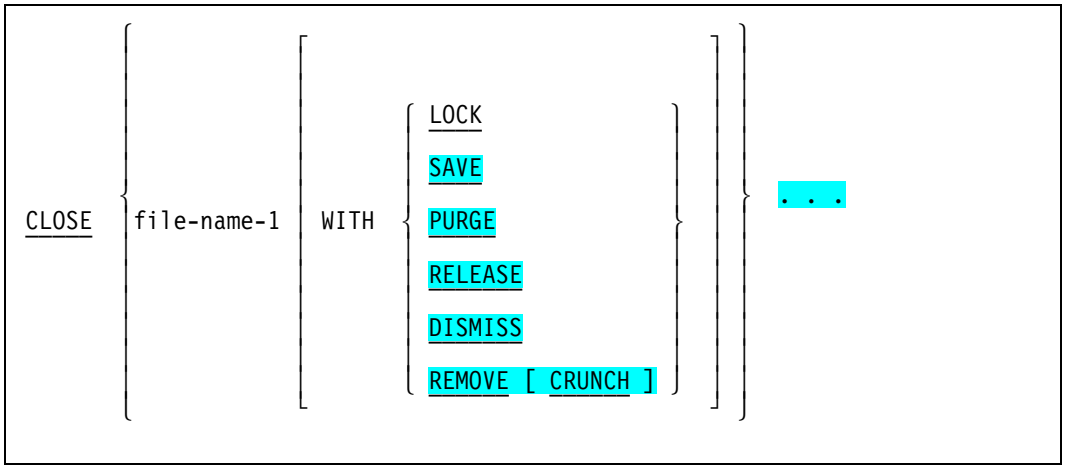
```
CLOSE DSKFIL WITH NO REWIND
```

This ends the processing of the file DSKFIL and the reel or unit is left in its current position.

```
CLOSE MYTAPE REEL FOR REMOVAL
```

This ends the processing of the file MYTAPE, and the reel is closed. The system expects the next volume to this multireel file.

Format 2: Relative and Indexed I-O



This format is supported in the TADS environment.

Explanation

Refer to Format 1 for descriptions of the syntax elements file-name-1 and WITH LOCK, and for other details concerning the CLOSE statement.

For detailed information on file attributes, file organization, and file access modes, refer to Section 11.

Refer to the *File Attributes Programming Reference Manual* for information on file attributes.

Refer to "OPEN Statement" in this section for syntax and detailed information.

Effect of CLOSE Statements on Different Storage Media

Relative and indexed files belong to the category of non-sequential single- or multi-reels or units.

Table 6–6 summarizes the results of executing each type of CLOSE statement for this category of file. Definitions of the numeric entries appear following the table.

Table 6–6. Relationship of CLOSE Formats and Nonsequential Units

CLOSE Statement Format	Nonsequential Single- or Multi-Reel or Unit
CLOSE	1 3
CLOSE WITH LOCK	1 2 4
CLOSE WITH SAVE	1 4 5
CLOSE WITH PURGE	1 6
CLOSE WITH RELEASE	1
CLOSE WITH DISMISS	1
CLOSE WITH REMOVE	1 4 5

The following paragraphs explain the meaning of the numerical values in Table 6-6. In these paragraphs, the definitions apply to input, output, and input-output files. Alternate definitions are given where the file type affects the definition.

1. The file is closed.

   Input Files and Input-Output Files (Sequential Access Mode):

   If the file is positioned at its end and label records are specified for the file, the labels are processed according to the system standard label conventions. The result of the CLOSE statement when label records are specified but not present is unpredictable.

   If the file is positioned at its end and label records are not specified for the file, label processing does not occur.

   If the file is not positioned at its end, the standard closing operations are executed, but there is no end-label processing.

   Input Files and Input-Output Files (Random or Dynamic Access Mode),

   Output Files (Random, Dynamic, or Sequential Access Mode):

   If label records are specified for the file, the labels are processed according to the system standard label conventions. The results of the CLOSE statement when label records are specified but not present, or when label records are not specified but are present, are unpredictable.

2. The file is locked.

   The file is locked and cannot be opened again during the execution of this run unit.

3. The file is retained.

   The association between the logical file and the physical file is retained. Subsequent reopening of the file cannot require the operating system to search for the physical file.

4. The file is released.

   The association between the logical file and the physical file is severed. The areas of memory allocated for buffers can be released to the system.

5. The file is saved.

   The physical file is made permanent. Any existing file with the same name is removed.

6. The file is purged.

   If the file is permanent, the file-name is removed from the directory of the system. Then, the storage area occupied by the file is released as available to the system.

Example

```
CLOSE INXFIL WITH LOCK, INX223 WITH RELEASE
```

This closes and locks INXFIL and then closes INX223. It releases to the system the areas of memory allocated for buffers.

COMPUTE Statement

Format	Use
Format 1	This format assigns to one or more numeric data items the values of an arithmetic expression
Format 2	This format assigns to one or more Boolean data items the values of a Boolean expression

Format 1: Arithmetic Compute

This form of the COMPUTE statement calculates an arithmetic expression and stores the result.

Rules and explanations of the COMPUTE statement and other arithmetic statements are discussed under "Arithmetic Expressions" in Section 5.

For information on rounding, size error conditions, and intermediate data items, refer to "ROUNDED Phrase," "SIZE ERROR Phrase," and "Intermediate Data Item" in Section 5.

Refer to the ADD, DIVIDE, MULTIPLY, and SUBTRACT statements in this section for syntax and detailed information.

This statement is partially supported in the TADS environment. Supported syntax is noted in this section.

```
COMPUTE  { identifier-1 [ ROUNDED ]  } . . .  =  arithmetic-expresion-1
  [ ON SIZE ERROR imperative-statement-1 ]
  [ NOT ON SIZE ERROR imperative-statement-2 ]
  [ END-COMPUTE ]
```

TADS Syntax

```
COMPUTE  { identifier-1 [ ROUNDED ]  } . . .  =  arithmetic-expresion-1
  [ END-COMPUTE ]
```

Explanation

identifier-1

This must refer to either an elementary numeric item or an elementary numeric-edited item.

ROUNDED

This optional phrase rounds the result from the COMPUTE statement to the size required by the data item where the result will be returned. For details about the ROUNDED phrase, refer to Section 5.

arithmetic-expression-1

An arithmetic expression contains combinations of numeric identifiers and numeric literals that are separated by arithmetic operators and parentheses.

An arithmetic expression that consists of a single identifier or literal provides a way to set the value of the data item referenced by identifier-1 equal to the literal or the value of the data item referenced by the single identifier.

ON SIZE ERROR imperative-statement-1

If a size error condition exists, imperative-statement-1 is executed.

NOT ON SIZE ERROR imperative-statement-2

If a size error condition does not exist, imperative-statement-2 is executed.

END-COMPUTE

This delimits the scope of the COMPUTE statement.

Details

The COMPUTE statement enables you to combine arithmetic operations without the restrictions on composite operands and receiving data items imposed by the ADD, SUBTRACT, MULTIPLY, and DIVIDE statements.

If more than one identifier is specified for the result of the operation, the value of the arithmetic expression is calculated and stored as the new value of each of the data items referred to by identifier-1 and so forth.

Notes on Lengths

The length of an intermediate data item is limited to 23 decimal digits. When coding COMPUTE statements, design the statement so that no intermediate fields will be longer than 23 digits.

The ON SIZE ERROR phrase checks for an overflow of significant data only as it is moved into the output field. The phrase does not check for a 23-digit limit overflow in any intermediate data items.

Multiplication results in a field length equal to the sum of the lengths of the operands.

The result of an addition requires a length one digit larger than the largest operand.

Exponentiation results in a field length equal to the length of the base times the power.

Subtraction and division require a result field equal in length to the largest operand.

Examples

```
COMPUTE A = B * C + 4
```

The value of identifier B is multiplied by the value of C, 4 is added to it, and the result is stored in A.

```
COMPUTE A, B ROUNDED = X / 10
     ON SIZE ERROR PERFORM SZ-ERR-PROC
     NOT ON SIZE ERROR PERFORM WRITE-PROC
END-COMPUTE.
```

The value of X is divided by 10, and the result is stored in A and B. The result stored in B is rounded. If a size error condition occurs, the statements in SZ-ERR-PROC are executed. If a size error condition does not exist, the statements in WRITE-PROC are executed.

Format 2: Boolean Compute

```
COMPUTE  { identifier-1 } . . .  =  Boolean-expression [ END-COMPUTE ]
```

Explanation

identifier-1

This must refer to a Boolean data item.

Boolean-expression

A Boolean expression contains combinations of Boolean identifiers and Boolean literals that are separated by Boolean operators and parentheses.

END-COMPUTE

This defines the ending limit of the COMPUTE statement.

Details

The number of Boolean positions in the value resulting from the evaluation of a Boolean-expression is the number of Boolean positions in the largest Boolean item referenced in the expression.

The value resulting from the evaluation of a Boolean expression moves to the data item referenced by the identifier according to the rules of the MOVE statement.

CONTINUE Statement

Format	Use
Format 1	This format indicates that no executable statement is present in the line of code.
Format 2	This format returns control to a synchronous process that has been previously called and exited.

Format 1: Designating an Unexecutable Line of Code

```
CONTINUE
```

Explanation

This form of the CONTINUE statement indicates that there are no executable statements in a line of code.

Details

You can use a CONTINUE statement anywhere you can use a conditional statement or an imperative statement.

The CONTINUE statement is a no-operation statement; it does not affect the execution of the program.

Example

```
IF A > B PERFORM CALC-AB
    IF A > C PERFORM CALC-AC
        IF A > D PERFORM CALC-AD
            IF A >= E PERFORM CALC-AD
            ELSE MOVE E TO E-OUT       CONTINUE
        ELSE MOVE D TO D-OUT           CONTINUE
    ELSE MOVE C TO C-OUT               CONTINUE
ELSE MOVE B TO B-OUT
END-IF.
```

The CONTINUE statements in this example do not affect the outcome of the IF statements. The CONTINUE statements are used for documentation purposes only.

Format 2: Returning to the Called Process

```
CONTINUE task-variable
```

Explanation

This form of the CONTINUE statement reinstates a synchronous, dependent process that was previously initiated by a CALL statement from another program and then exited by an EXIT PROGRAM statement.

task-variable

This task variable is associated with the process that you want to resume execution. This task variable must be the same task variable that was used in a previously executed CALL statement.

Details

If the called process was exited by an EXIT PROGRAM statement, the CONTINUE statement causes the called process to restart at its first executable statement.

If the called process was exited by an EXIT PROGRAM RETURN HERE statement, the CONTINUE statement causes the called process to restart at the statement following the EXIT PROGRAM RETURN HERE statement.

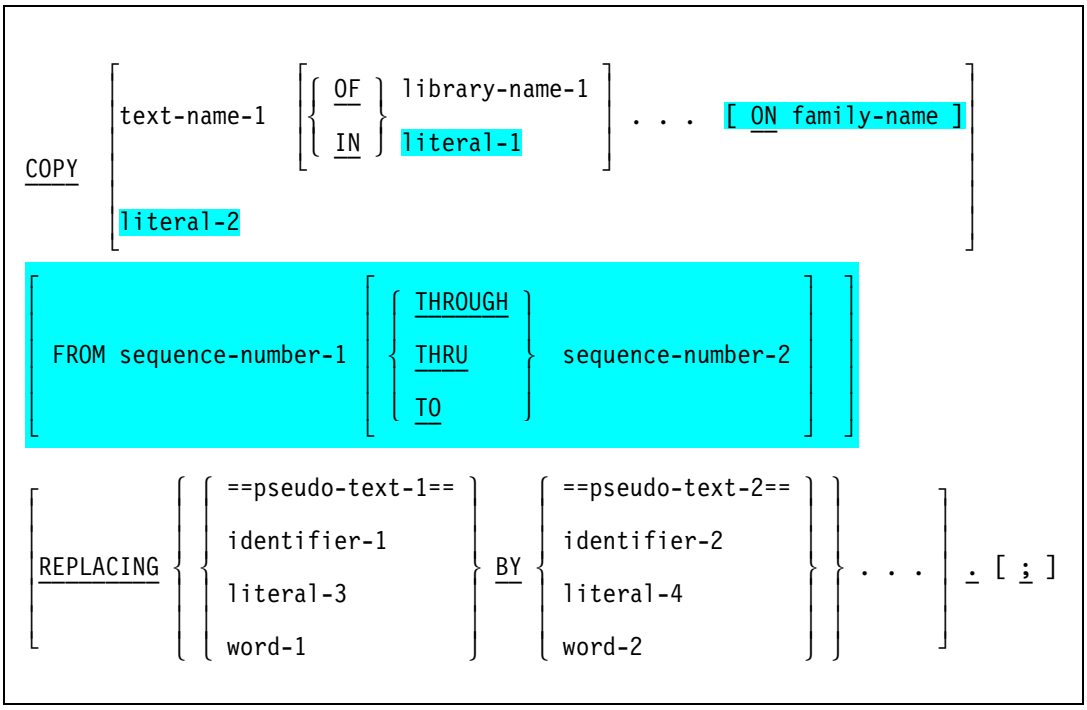
Parameters to be passed between programs are initialized when the CALL statement is executed and do not have to be passed again for successive CONTINUE statements. Data items in the called program also retain their values.

For an example of how the CONTINUE statement is used, see the program examples in Section 13.

COPY Statement

The COPY statement incorporates text from a COBOL library file into the program that contains the COPY statement.

Refer to "REPLACE Statement" in this section for information on replacing source program text.



Explanation

text-name-1

This is the external name of a file in the COBOL library. In the library name LIBRARY/A/B/C, C is the actual file name and, thus, C would be specified for text-name-1. Each text-name must be unique within a COBOL library.

OF
IN
library-name-1
literal-1

This construct specifies the external name of the directory in which the COBOL library file named as text-name-1 resides.

You can specify a multilevel directory by specifying each directory name in reverse order separated by the word OF or IN.

For example, if a library has the file name LIBRARY/A/B/C, LIBRARY/A/B is the directory and C is the file name. The complete specification of the file name and its directories in the COPY statement would be

```
COPY C OF B OF A OF LIBRARY.
```

Literal-1 can have the same contents as library-name-1 and is a nonnumeric literal.

If more than one COBOL library is available during compilation, text-name-1 must be qualified by library-name-1 to identify the COBOL library in which the text associated with text-name-1 resides.

ON family-name

Family-name specifies the name of the family in which the library file resides.

literal-2

This literal allows you to specify the entire title of the copy library file in a single nonnumeric literal rather than specifying the title of the library file in parts. This is an alternative format to the text-name-1 OF library-name-1 ON family-name format.

FROM sequence-number-1

The FROM phrase causes copying to start at the sequence number specified in sequence-number-1. If the THROUGH, THRU, or TO phrase is not specified, copying continues to the end of the file.

THROUGH
THRU
TO
sequence-number-2

THROUGH, THRU, and TO are interchangeable. If this phrase is specified, copying continues until the sequence number specified in sequence-number-2 has been copied.

pseudo-text-1 This is a sequence of text words beginning and ending with two consecutive equal signs (==). Allowable separators within the pseudotext are commas, semicolons, and spaces. A nonnumeric literal can contain quotation marks. Debugging lines and comment lines are permitted within pseudo-text.

Pseudo-text-1 must contain one or more text words.

Pseudo-text-1 must not be null or consist only of commas, semicolons, and spaces.

If pseudo-text-1 is a PICTURE character-string, it must be preceded by the word PICTURE or PIC.

pseudo-text-2

This can contain zero, one, or more text words.

Pseudo-text-2 can be null.

Character-strings within pseudo-text-1 and pseudo-text-2 can be continued. However, both characters of a pseudo-text delimiter must be on the same line.

identifier-1
identifier-2

Each identifier is a syntactically correct combination of a data-name and its qualifiers, subscripts, and reference modifiers.

literal-3
literal-4

These literals can be any numeric, nonnumeric, or national literal.

Note: *See "How the Copy Is Made," later in this subsection (Copy Statement) for more details.*

word-1
word-2

These words can be any single COBOL word, including COBOL reserved words, except COPY.

; (Semicolon)

The semicolon that follows the ending period can be used to control the behavior of compiler control records (CCRs) and the format of listings. This semicolon should always be separated from the ending period of the COPY statement by at least one space.

If a CCR immediately follows a COPY statement, the compiler option changes might occur before the compiler processes the included source information. This situation can be avoided by using the semicolon after the ending period. The semicolon ensures that the compiler processes the included source information before the option changes.

When a compilation listing is produced, a comment immediately following a COPY statement might be printed after the COPY statement but before the information included as a result of the COPY statement. If a semicolon is placed after the ending period, but before the comment entry, the comment is printed after the included source information.

Use the optional semicolon with caution. In some cases, the compiler may recognize the optional semicolon. In other cases, the compiler may prohibit the use semicolon. In the latter cases, the7 semicolon may not produce the desired listing format and may even produce syntax errors. In such cases, use the semicolon as a tool in determining whether errors can be eliminated.

In general, the semicolon can produce undesirable listing formats in the following cases:

- Multiple COPY statements follow each other with no intervening syntax.
- COPY statements have semicolons.
- The last element in the library that is the subject of a COPY statement is a PICTURE string that ends with one or more 9s followed by a period terminating the DATA declaration.

If the last statement of a COBOL85 program is a COPY statement, do not use a semicolon with that statement. The last syntax element of a COBOL85 program must always be a period that terminates the last statement or paragraph-name of the program.

Details

The COPY statement must be preceded by a separator and ended by a period. The use of a separator other than a space immediately before a COPY statement is a Unisys extension to the COBOL language. To comply with language standards, at least one space must be used as a separator immediately before a COPY statement.

COPY is a compiler-directing statement that indicates to the compiler that text will be incorporated into a COBOL source program from another saved program. However, actual inserted text appears only on the compilation listing.

Compiling a source program that contains COPY statements is logically equivalent to processing all COPY statements before processing the resultant source program.

A COPY statement can be specified in a source program wherever a character-string or separator (other than the closing quotation mark) can occur. However, a COPY statement must not occur within a COPY statement.

If the word COPY appears in a comment-entry or in the place where a comment-entry can appear, it is considered part of the comment-entry. In COBOL ANSI-85, the comment-entry is an obsolete entry and will be deleted from the next revision of standard COBOL.

A text word in pseudo-text and in library text can be from 1 through 322 characters long.

Library-name, text-name-1, and family-name can be user-defined words as well as reserved words or unsigned integers.

A COPY library may be an optional file. If, during compilation, the COPY file is not found, the compilation will stop and a message will appear on the ODT. The Optional File (?OF) system command can be used to continue the compilation without the COPY library.

How the Copy Is Made

When a COPY statement is executed, the library text associated with text-name-1 is copied into the source program. Logically, this replaces the entire COPY statement. The replacement begins with the reserved word COPY and ends with the punctuation character period (.).

If the REPLACING phrase is not specified, the library text is copied unchanged.

If the REPLACING phrase is specified, each properly matched occurrence of pseudo-text-1, identifier-1, word-1, and literal-3 in the library text is replaced by the corresponding pseudo-text-2, identifier-2, word-2, or literal-4.

For purposes of matching, identifier-1, word-1, and literal-3 are treated as pseudo-text containing only identifier-1, word-1, or literal-3, respectively.

The comparison to determine text replacement occurs as follows:

1. The first word used for comparison is the leftmost library text word that is not a separator comma or a separator semicolon. Any text word or space that precedes this text word is copied into the source program.

   The entire REPLACING phrase operand that precedes the reserved word BY is compared to an equivalent number of contiguous library text words. The comparison starts with the first text word of the library and the first pseudo-text-1, identifier-1, word-1, or literal-3 that was specified in the REPLACING phrase.

2. Pseudo-text-1, identifier-1, word-1, or literal-3 match the library text if the ordered sequence of text words that forms the replacing phrase operand equals, character for character, the ordered sequence of library text words.

   For matching, each occurrence of a separator comma, semicolon, or space in pseudo-text-1 or in the library text is treated as a single space. Each sequence of one or more space separators is treated as a single space.

3. If a match does not occur, the comparison is repeated with each successive pseudo-text-1, identifier-1, word-1, or literal-3, in the REPLACING phrase until either a match is found or a successive REPLACING phrase operand is not found.

4. When all the REPLACING phrase operands have been compared and a match has not occurred, the leftmost library text word is copied into the source program. The next library text word is then considered the leftmost library text word, and the comparison cycle starts again with the first pseudo-text-1, identifier-1, word-1, or literal-3 specified in the REPLACING phrase.

5. Whenever a match occurs between pseudo-text-1, identifier-1, word-1, or literal-3 and the library text, the corresponding pseudo-text-2, identifier-2, word-2, or literal-4 is placed into the source program.

   The library text word immediately following the rightmost text word that participated in the match is then considered to be the leftmost text word. The comparison cycle starts again with the first pseudo-text-1, identifier-1, word-1, or literal-3 specified in the REPLACING phrase.

6. The comparison operation continues until the rightmost text word in the library text has either participated in a match or has been considered as a leftmost library text word and has participated in a complete comparison cycle.

Comment Lines and Blank Lines

Comment lines or blank lines that occur in the library text and in pseudo-text-1 are ignored for purposes of matching. The sequence of text words in the library text, if any, and in pseudo-text-1 is determined by the rules for reference format. Refer to "Reference Format" in Section 1 for more information.

Comment lines or blank lines that appear in pseudo-text-2 are copied into the resultant program.

Comment lines or blank lines that appear in library text are copied into the resultant source program and are unchanged with the following exception: a comment line or a blank line in library text is not copied if it appears within the sequence of text words that matches pseudo-text-1.

Debugging Lines

Debugging lines are permitted within library text and pseudo-text. Text words within a debugging line participate in the matching rules as if the "D" did not appear in the indicator area. A debugging line is specified within pseudo-text if the debugging line begins in the source program after the opening pseudo-text-delimiter but before the matching closing pseudo-text-delimiter.

Continuation of Lines and Additional Lines

Each text word that is copied from the library but not replaced is copied so it will start in the same area of the line in the resultant program as it begins in the line within the library. This is true except in the following situation. A text word that is copied from the library begins in area A but follows another text word that also begins in area A of the same line. If a preceding text word in the line is replaced by text of greater length, the following text word begins in area B if it cannot begin in area A.

When either the $ANSI or $ANSICLASS compiler control option is set, the first text word of pseudo-text-2 begins in the same area of the resultant program as it appears in pseudo-text-2. Otherwise, the first text word of pseudo-text-2 begins in the same area of the resultant program as the leftmost library text word that participated in the match would appear if it had not been replaced.

Each text word in pseudo-text-2 that will be placed into the resultant program begins in the same area of the resultant program as it appears in pseudo-text-2. Each identifier-2, literal-2, and word-2 that will be placed into the resultant program begins in the same area of the resultant program as the leftmost library text word that participated in the match would appear if it had not been replaced.

Library text must conform to the rules for COBOL reference format. Refer to "Reference Format" in Section 1 for more information.

If additional lines are introduced into the source program as a result of a COPY statement, each text word introduced appears on a debugging line if the COPY statement begins on a debugging line or if the text word being introduced appears on a debugging line in library text.

When a text word specified in the BY phrase is introduced, it appears on a debugging line if the first library text word being replaced is specified on a debugging line. Except in the preceding cases, only those text words that are specified on debugging lines where the debugging line is within pseudo-text-2 appear on debugging lines in the resultant program.

If a literal is specified as literal-4 or within pseudo-text-2 or if library text exceeds a single line without continuation to another line in the resultant program and the literal is not being placed on a debugging line, additional continuation lines are introduced that contain the remainder of the literal.

If replacement requires that the continued literal be continued on a debugging line, an error will result.

For compilation, text words after replacement are placed in the source program according to the rules for reference format. When copying text words of pseudo-text-2 into the source program, additional spaces can be introduced only between text words where there already exists a space (including the assumed space between source lines).

If additional lines are introduced into the source program as a result of processing COPY statements, the indicator area of the introduced line will contain the same character as the line on which the text being replaced begins. However, if that line contains a hyphen, the introduced line will contain a space. If a literal is continued onto an introduced line that is not a debugging line, a hyphen is placed in the indicator area.

Syntax Checking

The syntactic correctness of the library text cannot be independently determined. Except for COPY and REPLACE statements, the syntactic correctness of the entire COBOL source program cannot be determined until all COPY and REPLACE statements have been completely processed.

When COPY . . . REPLACING a library file, if the library file contains a REPLACE statement, character strings within the pseudo-text of the REPLACE statement will not be affected.

If replacement of a PICTURE string is required, then the word PICTURE (or PIC) must precede the string to be replaced within pseudo-text-1.

In the event that an occurrence of a particular pseudo-text-1, identifier-1, literal-3, or word-1 is not found in the library file, a warning is issued. The warning shows the item that was not replaced.

Examples

```
COPY STANDARD-RECOVERY-ROUTINE OF ERRORS ON USER1
    FROM 200 THRU 1000
    REPLACING  "Item-1" BY "Pay-Field".
```

This copies lines 200 to 1000 of the text STANDARD-RECOVERY-ROUTINE from the library known as ERRORS, replacing the literal “Item-1” with another literal “Pay-Field”.

```
COPY FILE-DESCRIPTION REPLACING F-REC BY NEW-REC,
        ==VALUE OF FILENAME IS  "NEWFIL"==
     BY ==VALUE OF BLOCKSIZE IS 200==.
```

This copies all of the text associated with FILE-DESCRIPTION and inserts it into the source file at the point where this COPY statement appears. Whenever the name F-REC is encountered, it is replaced by the new name called NEW-REC. Also, the pseudo-text string VALUE OF FILENAME IS “NEWFIL” is replaced by a new pseudo-text string VALUE OF BLOCKSIZE IS 200.

DEALLOCATE Statement

The DEALLOCATE statement deallocates the storage of record areas.

```
DEALLOCATE  record name
```

The record–name must be a 01-level item.

A record-name described without a RECORD AREA clause does not need to be specified in a DEALLOCATE statement since normal the normal system overlay releases the area of memory used.

The record–name specified must not have a usage of EVENT, LOCK, CONTROL–POINT, TASK, or BINARY.

DELETE Statement

The DELETE statement removes a logical record from a relative file or an indexed file.

Refer to the discussion of the RECORD CONTAINS clause under "File Section" in Section 4 for more information.

For information on open mode, refer to "OPEN Statement" in Section 7. Also refer to "READ Statement" in Section 7 and "USE Statement" in Section 8 for detailed information and syntax.

For information on file organization and file access modes, refer to Section 12.

This statement is partially supported in the TADS environment. Applicable exclusions are noted in this section.

```
DELETE file-name-1 RECORD
  [ INVALID KEY imperative-statement-1 ]
  [ NOT INVALID KEY imperative-statement-2 ]
  [ END-DELETE ]
```

This format is supported in the TADS environment.

Explanation

file-name-1

The file referred to by file-name-1 must be a mass storage file and must be open in the I/O mode when the DELETE statement is executed. The I/O mode is used for retrieving and updating records.

INVALID KEY imperative-statement-1

If the file does not have the record indicated by the key, imperative-statement-1 will be executed.

The INVALID KEY or NOT INVALID KEY phrases can be used if file-name-1 refers to a file in random or dynamic access mode. These phrases cannot be used if file-name-1 refers to a file in sequential access mode.

The INVALID KEY phrase must be specified for a DELETE statement if the file referred to in the DELETE statement does not contain a USE AFTER STANDARD EXCEPTION procedure.

NOT INVALID KEY imperative-statement-2

If the file has the record indicated by the key, the record is deleted and imperative-statement-2 is executed.

END-DELETE This phrase delimits the scope of the DELETE statement.

Details

After the successful execution of a DELETE statement, the identified record is logically removed from the file and cannot be accessed.

For a file in the sequential access mode, the last input-output statement executed for file-name-1, before the execution of the DELETE statement, must have been a successfully executed READ statement. The disk or disk pack logically removes from the file the record that was accessed by that READ statement.

For a relative file in random or dynamic access mode, the disk or disk pack logically removes from the file the record identified by the relative key data item associated with file-name-1.

For an indexed file in random or dynamic access mode, the disk or disk pack logically removes from the file the record identified by the prime record key data item associated with file-name-1.

The invalid key condition occurs if the relative or indexed file does not contain the record specified by the key.

Execution of the DELETE statement updates the value of the I/O status associated with file-name-1.

The execution of a DELETE statement does not affect the content of the record area or the content of the data item referenced by the data-name specified in the DEPENDING-ON phrase of the RECORD clause associated with file-name-1. (The RECORD clause is part of the file description entry for file-name-1.)

The file position indicator is not affected by the execution of a DELETE statement.

TADS

Any USE procedure is not executed when a DELETE statement that is compiled and executed in a TADS session fails.

Example

```
ENVIRONMENT DIVISION.
.
.
.
FILE-CONTROL.
     SELECT MSTFIL ASSIGN TO DISK
     ORGANIZATION IS INDEXED
     ACCESS MODE IS DYNAMIC
     RECORD KEY IS SOC-SEC-NO
.
.
.
PROCEDURE DIVISION.
.
.
.
     READ MSTFIL RECORD INTO IN-RECORD
          KEY IS SOC-SEC-NO
          INVALID KEY PERFORM INVALID-PROC.
     DELETE MSTFIL RECORD
          INVALID KEY PERFORM NO-REC-PROC
          NOT INVALID KEY PERFORM PROC-4
     END-DELETE.
```

This program first reads the indexed file MSTFIL. If the key is invalid, the statements in INVALID-PROC are executed. If the record is found, then the record identified by the prime record key data item SOC-SEC-NO is logically deleted from the file, and the statements in PROC-4 are executed.

If the record is not found, an invalid key condition exists, and the statements in NO-REC-PROC are executed.

DETACH Statement

The DETACH statement dissociates a procedure from a task variable or an event.

Format	Use
Format 1	This format dissociates one or more processes from their corresponding task variables. This statement dissociates only those task variables that have been implicitly attached in CALL, PROCESS, or RUN statements.
Format 2	This format dissociates interrupt procedures from an event.

Format 1: Detaching from a Task Variable

```
DETACH task-variable-1 [ ,task-variable-2 ] . . .
```

Explanation

task-variable-1
[task-variable-2] . . .

These are the task variables from which you want to dissociate the processes.

Details

For the DETACH statement to be valid, the task variable must have been previously attached to the process by the execution of a CALL, PROCESS, or RUN statement.

The successful execution of the DETACH statement terminates the processes associated with the specified task variables by setting the STATUS attributes of the processes to TERMINATED. Once the processes are terminated, they are no longer associated with the task variables.

Note that the program that contained the DETACH statement continues to execute asynchronously while the detachment is performed. Thus, before using a detached task variable in a subsequent CALL, PROCESS, or RUN statement, you should verify that the STATUS attribute of the process to which it was attached has a value of TERMINATED. You can check the value of an attribute by using the attribute in an IF statement. The IF statement is described later in this section.

Example

```
DETACH INTERRUPT-PROCEDURE-ONE
```

Format 2: Detaching from an Event

```
DETACH section-name-1 [ ,section-name-2 ] . . .
```

section-name-1 [, section-name-2] . . .

This is the name of one or more sections that contain the processes that you want to dissociate from the event. The section-name must be declared in the Declaratives Section with a USE AS INTERRUPT clause.

Details

Execution of a DETACH section-name statement severs the association of an interrupt procedure with its currently attached event. Executions of the interrupt procedure which might have been queued at the time of the detachment do not occur.

Note that performing the DETACH statement for an interrupt procedure that is not attached to an event does not cause an error.

The DETACH statement has no effect on the allowed or disallowed condition of an interrupt procedure.

For additional information, refer to the ALLOW, ATTACH, CAUSE, and DISALLOW statements described in this section.

Example

```
DETACH INTERRUPT-PROCEDURE-ONE
```

DISALLOW Statement

The DISALLOW statement prevents an interrupt procedure from being executed when its associated event is activated by a CAUSE statement.

```
DISALLOW { section-name-1 [ ,section-name-2 ] . . . }
         { INTERRUPT                                 }
```

Explanation

section-name-1 [, section-name-2] . . .

This syntax causes specific interrupt procedures to be queued when their attached events are activated. Subsequent execution of an ALLOW section-name statement causes the queued procedures to be executed immediately, unless the general DISALLOW INTERRUPT statement is in effect. In this case, the procedures remain queued until the general ALLOW INTERRUPT statement is executed.

Section-name indicates the section in the Procedure Division that contains the specific interrupt procedure that you want to prevent from executing. You can use multiple section names to specify multiple interrupt procedures.

INTERRUPT

The DISALLOW INTERRUPT statement causes all interrupt procedures to be queued when their attached events are activated. Subsequent execution of an ALLOW INTERRUPT statement causes the queued procedures to be executed immediately.

Note that a procedure restricted by a DISALLOW section-name statement remains queued until an ALLOW section-name statement is executed for it.

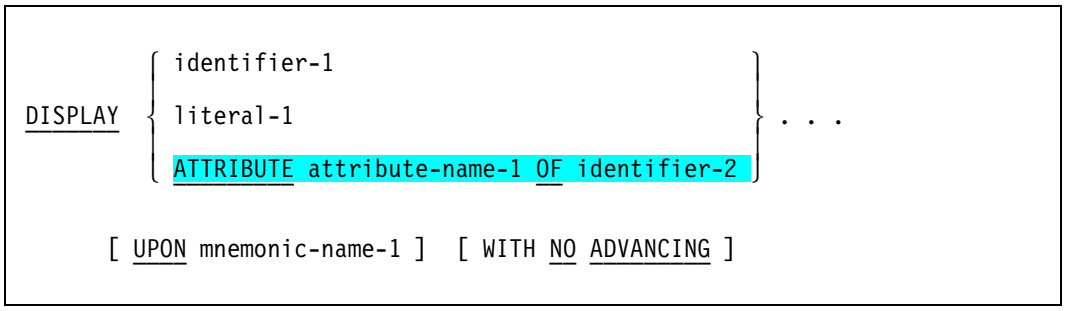
Example

```
DISALLOW INTERRUPT.

DISALLOW INTERRUPT-PROCEDURE-ONE.
```

DISPLAY Statement

The DISPLAY statement causes low-volume data to be transferred to an operator display terminal (ODT).

The TADS environment fully supports the DISPLAY statement. Additionally, it provides the ITEMS, GROUP ITEMS, ELEMENTARY ITEMS, GROUPS, HEX, EBCDIC, and DECIMAL clauses.



Explanation

identifier-1

This identifier references a data-item to be displayed. Identifier-1 cannot reference a long numeric data item.

If identifier-1 is a table, the number of subscripts must equal the number required by the table item. In a COBOL85 TADS session, the subscripts are optional.

literal-1

Literal-1 specifies a literal to be displayed.

If the literal is a figurative constant, only a single occurrence of the figurative constant is displayed. The figurative constant [ALL] literal is allowed. If [ALL] literal is used, the ALL is ignored and the literal is displayed once.

The literal cannot be a long numeric literal.

attribute-name-1
identifier-2

Attribute-name-1 is a file, library, or task attribute. Identifier-2 must be the proper type to match attribute-name-1. For example, if attribute-name-1 is a file attribute, then identifier-2 must be a file.

UPON mnemonic-name-1

The data-item mnemonic-name-1 is the name of a mnemonic associated with a hardware device defined in the SPECIAL-NAMES paragraph of the Environment Division. Mnemonic-name-1 must be associated with the hardware-name ODT.

If the UPON phrase is not used, the device used is the ODT.

WITH NO ADVANCING

This optional phrase does not reset the hardware device to the next line or change it in any other way following the display of the last operand.

If the hardware device can be positioned to a specific character position, the device will remain at that character position immediately following the last character of the last operand displayed.

If you do not specify the WITH NO ADVANCING phrase, then the positioning of the hardware device will be reset to the leftmost position of the next line of the device after the last operand has been transferred.

Details

The DISPLAY statement transfers the content of each operand to the hardware device in the order listed. Any conversion of data required between a literal or the data item referred to by the identifier and the hardware device is defined by the implementor. If the hardware device is capable of receiving both national characters and alphanumeric characters, figurative constants, except for the ALL national literal, are displayed as alphanumeric characters.

The data item is transferred if the hardware device can receive data of the same size as the data item being transferred. The maximum size of a data transfer to the ODT is 430 characters.

If the hardware device cannot receive data of the same size as the data item being transferred, then one of the following applies:

- If the size of the data item being transferred exceeds the size that the hardware device can receive in a single transfer, the data beginning with the leftmost character is stored and aligned to the left in the receiving hardware device. Any remaining data is then transferred in like fashion, until all data is transferred. This process appears as if multiple executions of the DISPLAY statement have occurred.
- If the size of the data item that the hardware device can receive exceeds the size of the data being transferred, the transferred data is stored and aligned to the left in the receiving hardware device.

When a DISPLAY statement contains more than one operand, the size of the sending item is the sum of the sizes associated with the operands. The values of the operands are transferred in the sequence in which the operands are encountered without modifying the positioning of the hardware device between the successive operands.

Data within the current file record can be displayed from COBOL85 TADS. The TITLE, KIND, and OPEN attributes are displayed automatically for files. If the OPEN attribute is TRUE, then the STATE and NEXTRECORD attributes are also displayed automatically. In addition, a user can request the display of other file attributes.

Examples

```
DISPLAY OCCUPATION
```

This displays the contents of the data item OCCUPATION on the ODT.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. DISPLAY-VERB.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
    ODT IS MODT.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  OCCUPATION              PIC X(20).
PROCEDURE DIVISION.
DISPLAY-PARA.
    MOVE "THIS IS A TEST CASE"TO OCCUPATION.
    DISPLAY OCCUPATION.
    DISPLAY "PROCEDURE-1 COMPLETED" UPON MODT WITH NO ADVANCING.
    DISPLAY "PROCEDURE-2 COMPLETED".
    STOP RUN.
```

This example displays the contents of the data-item OCCUPATION and, after advancing one line, the literal “Procedure-1 completed”, and, without advancing to another line, the literal “Procedure-2 completed”.

DIVIDE Statement

The DIVIDE statement divides a numeric operand into another operand and stores the quotient and the remainder.

This statement is partially supported in the TADS environment. Supported syntax is noted in this section.

Format	Use
Format 1	The DIVIDE . . . INTO format enables you to divide a numeric operand into another numeric operand and to store the result in the second operand.
Format 2	The DIVIDE . . . INTO . . . GIVING format enables you to divide a numeric operand into another numeric operand and to specify a place to store the result.
Format 3	The DIVIDE . . . BY . . . GIVING format enables you to divide a numeric operand by another numeric operand and to specify a place to store the result.
Format 4	The DIVIDE . . . INTO . . . GIVING . . . REMAINDER format enables you to divide a numeric operand into another numeric operand and to specify a place to store the result and the remainder.
Format 5	The DIVIDE . . . BY . . . GIVING . . . REMAINDER format enables you to divide a numeric operand by another numeric operand and to specify a place to store the result and the remainder.

Format 1: DIVIDE . . . INTO

```
DIVIDE { identifier-1 } INTO identifier-2  [ ROUNDED ]
       { literal-1    }

    [ [ , ]  identifier-3  [ ROUNDED ]  ] . . .
    [ ON SIZE ERROR imperative-statement-1 ]
    [ NOT ON SIZE ERROR imperative-statement-2 ]
    [ END-DIVIDE ]
```

TADS Syntax

```
DIVIDE { identifier-1 } INTO identifier-2  [ ROUNDED ]
       { literal-1    }

    [ [ , ]  identifier-3  [ ROUNDED ]  ] . . .
    [ END-DIVIDE ]
```

Explanation

identifier-1
literal-1
identifier-2
identifier-3

Each identifier must refer to an elementary numeric item.

Literal-1 must be a numeric literal.

During execution, the value of the data item referenced by identifier-1 or literal-1 is stored in a temporary data item. The value in this temporary data item is then divided into the value of the data item referenced by identifier-2.

The value of the dividend (that is, the value of the data item referenced by identifier-2) is replaced by this quotient. Similarly, the temporary data item is divided into each successive occurrence of identifier-3, and so on.

The composite value of operands is a hypothetical data item that results from the superimposition of all receiving data items of a given statement aligned on their decimal points. The composite must not contain more than 23 digits.

INTO

This keyword specifies that identifier-1 or literal-1 will be divided into identifier-2.

ROUNDED

This option increases the absolute value of the quotient by adding 1 to the quotient's low-order digit. This occurs whenever the absolute value of the most significant digit of the excess is greater than or equal to 5.

Refer to "ROUNDED Phrase" in Section 5 for more information.

ON SIZE ERROR
imperative-statement-1 If a size error condition occurs and this phrase is specified, imperative-statement-1 is executed.

Refer to "SIZE ERROR Phrase" in Section 5 for more information.

NOT ON SIZE ERROR
imperative-statement-2

If a size error condition does not occur and this phrase is specified, imperative-statement-2 is executed.

END-DIVIDE

This phrase delimits the scope of the DIVIDE statement.

Example

```
DIVIDE .25 INTO Sale-Item ROUNDED
   ON SIZE ERROR PERFORM Para-6
   NOT ON SIZE ERROR PERFORM Write-Para
END-DIVIDE.
```

The literal .25 is divided into the value of Sale-Item. Then, the quotient is rounded, if necessary, and stored in Sale-Item. If a size error condition occurs, the statements in Para-6 are executed. If a size error condition does not occur, the statements in Write-Para are executed.

Format 2: DIVIDE . . . INTO . . . GIVING

```
DIVIDE { identifier-1 } INTO { identifier-2 }
       { literal-1    }      { literal-2    }

  GIVING identifier-3  [ ROUNDED ]

    [ [ , ]  identifier-4 [ ROUNDED ]  ] . . .

    [ ON SIZE ERROR imperative-statement-1 ]

    [ NOT ON SIZE ERROR imperative-statement-2 ]

    [ END-DIVIDE ]
```

TADS Syntax

```
DIVIDE { identifier-1 } INTO { identifier-2 }
       { literal-1    }      { literal-2    }

  GIVING identifier-3  [ ROUNDED ]

    [ [ , ]  identifier-4 [ ROUNDED ]  ] . . .

    [ END-DIVIDE ]
```

Explanation

Refer to Format 1 for descriptions of the ON SIZE ERROR and NOT ON SIZE ERROR phrases and the syntax elements INTO and END-DIVIDE.

identifier-1
literal-1
identifier-2
literal-2

Each identifier must refer to an elementary numeric item.

Each literal must be a numeric literal.

The value of the data item referenced by identifier-1 or literal-1 is divided into the value of the data item referenced by identifier-2 or literal-2. The result is stored in each data item referenced by identifier-3, identifier-4, and so on.

GIVING identifier-3 identifier-4

The GIVING phrase allows the quotient to be stored in the data item referenced by identifier-3 and, if present, in the data items referenced by identifier-4 and so on. Each identifier must refer to either an elementary numeric item or an elementary numeric-edited item.

ROUNDED

This option increases the absolute value of the quotient, which will be stored in identifier-3, by adding 1 to its low-order digit. This occurs whenever the absolute value of the most significant digit of the excess is greater than or equal to 5.

Refer to "ROUNDED Phrase" in Section 5 for more information.

Example

```
DIVIDE Discount INTO Item GIVING Sale-Price ROUNDED
    ON SIZE ERROR PERFORM Err-Proc
    NOT ON SIZE ERROR PERFORM Write-Report
END-DIVIDE.
```

This example divides the value of the identifier Discount into the value of the identifier Item. The result is rounded, if necessary, and stored in the identifier Sale-Price. If a size error condition occurs, the statements in Err-Proc are executed. If a size error condition does not occur, the statements in Write-Report are executed.

Format 3: DIVIDE . . . BY . . . GIVING

```
DIVIDE { identifier-1 } BY { identifier-2 }
       { literal-1    }    { literal-2    }
  GIVING identifier-3  [ ROUNDED ]
    [ [ , ]  identifier-4 [ ROUNDED ]  ] . . .
    [ ON SIZE ERROR imperative-statement-1 ]
    [ NOT ON SIZE ERROR imperative-statement-2 ]
    [ END-DIVIDE ]
```

TADS Syntax

```
DIVIDE { identifier-1 } BY { identifier-2 }
       { literal-1    }    { literal-2    }
  GIVING identifier-3  [ ROUNDED ]
    [ [ , ]  identifier-4 [ ROUNDED ]  ] . . .
    [ END-DIVIDE ]
```

Explanation

Refer to Format 1 for a description of the ON SIZE ERROR and NOT ON SIZE ERROR phrases and the syntax element END-DIVIDE. Refer to Format 2 for a description of the GIVING and ROUNDED phrases.

identifier-1
literal-1
identifier-2
literal-2

Each identifier must refer to an elementary numeric item. Each literal must be a numeric literal.

The value of the data item referenced by identifier-1 or literal-1 is divided by the value of the data item referenced by identifier-2 or literal-2. The result is stored in each data item referenced by identifier-3, identifier-4, and so forth.

BY

This keyword indicates that identifier-1 or literal-1 will be divided by identifier-2 or literal-2.

GIVING identifier-3
identifier-4

The GIVING phrase allows the quotient to be stored in the data item referenced by identifier-3 and, if present, in the data items referenced by identifier-4 and so on. Each identifier must refer to either an elementary numeric item or an elementary numeric-edited item.

Examples

```
DIVIDE Salary BY 52 GIVING Weekly-Salary ROUNDED
    ON SIZE ERROR GO TO SIZE-ERR
    NOT ON SIZE ERROR MOVE Weekly-Salary TO Print-Out
END-DIVIDE.
```

This divides the value of identifier Salary by the numeric literal 52. The result is rounded, if necessary, and stored in Weekly-Salary. If a size error condition occurs, control passes to the statements in SIZE-ERR. If a size error condition does not occur, the value of Weekly-Salary is moved to the data item Print-Out.

Format 4: DIVIDE . . . INTO . . . GIVING . . . REMAINDER

```
DIVIDE { identifier-1 } INTO { identifier-2 }
       { literal-1    }      { literal-2    }
  GIVING identifier-3  [ ROUNDED ]
  REMAINDER identifier-4
    [ ON SIZE ERROR imperative-statement-1 ]
    [ NOT ON SIZE ERROR imperative-statement-2 ]
    [ END-DIVIDE ]
```

TADS Syntax

```
DIVIDE { identifier-1 } INTO { identifier-2 }
       { literal-1    }      { literal-2    }
  GIVING identifier-3  [ ROUNDED ]
  REMAINDER identifier-4
    [ END-DIVIDE ]
```

Explanation

Refer to Format 1 for a description of the ON SIZE ERROR and NOT ON SIZE ERROR phrases and the syntax elements INTO and END-DIVIDE.

Refer to Format 2 for a description of identifier-1, literal-1, identifier-2, literal-2, identifier-3, and the GIVING and ROUNDED phrases.

REMAINDER
identifier-4

The REMAINDER phrase stores the remainder of a division operation in a data item.

Identifier-4 will contain the value of the remainder and must refer to either an elementary numeric item or an elementary numeric-edited item.

If identifier-4 is subscripted, the subscript is evaluated after the quotient is stored in identifier-3 and immediately before the remainder is stored in the data item referenced by identifier-4.

Defining the Remainder

The remainder in COBOL is defined as the result of subtracting the product of the quotient (identifier-3) and the divisor from the dividend. If identifier-3 is defined as a numeric-edited item, the quotient used to calculate the remainder is an intermediate field that contains the unedited quotient.If the ROUNDED phrase is specified in formats 4 and 5, the quotient used to calculate the remainder is an intermediate field that contains the quotient of the DIVIDE statement. The quotient in the intermediate field is truncated rather than rounded. This intermediate field is defined as a numeric field that contains the same number of digits, the same decimal point location, and the same presence or absence of a sign as the quotient (identifier-3).

Using the ON SIZE ERROR Phrase

When the ON SIZE ERROR phrase is used, the following rules apply:

- If the size error occurs on the quotient, the remainder calculation is not valid. The contents of the data items referenced by both identifier-3 and identifier-4 remain unchanged.
- If the size error occurs on the remainder, the content of the data item referenced by identifier-4 remains unchanged.

You must recognize which situation has actually occurred. Refer to "SIZE ERROR Phrase" in Section 5 for more information.

Example

```
DIVIDE Item-1 INTO Item-2 GIVING Item-3 ROUNDED REMAINDER Item-4
    ON SIZE ERROR DISPLAY  "Size error"
    NOT ON SIZE ERROR MOVE Item-3 TO Item-5 MOVE Item-4 TO Item-6
END-DIVIDE.
```

In this example, Item-1 is divided into Item-2. The quotient and the remainder are calculated. If necessary, the quotient is rounded and then stored in Item-3. The remainder is stored in Item-4. If a size error condition occurs, the literal "Size error" is displayed on the ODT. If a size error condition does not occur, the quotient (Item-3) is moved to Item-5, and the remainder (Item-4) is moved to Item-6.

Format 5: DIVIDE . . . BY . . . GIVING . . . REMAINDER

```
DIVIDE { identifier-1 } BY { identifier-2 }
       { literal-1    }    { literal-2    }

  GIVING identifier-3  [ ROUNDED ]

  REMAINDER identifier-4

    [ ON SIZE ERROR imperative-statement-1 ]

    [ NOT ON SIZE ERROR imperative-statement-2 ]

    [ END-DIVIDE ]
```

TADS Syntax

```
DIVIDE { identifier-1 } BY { identifier-2 }
       { literal-1    }    { literal-2    }

  GIVING identifier-3  [ ROUNDED ]

  REMAINDER identifier-4
    [ END-DIVIDE ]
```

Explanation

Refer to Format 1 for descriptions of the ON SIZE ERROR and NOT ON SIZE ERROR phrases and the syntax element END-DIVIDE.

Refer to Format 2 for a description of the GIVING and ROUNDED phrases.

Refer to Format 3 for a description of the BY phrase.

Refer to Format 4 for a description of the REMAINDER phrase and for the paragraphs labeled "Defining the Remainder" and using the "ON SIZE ERROR Phrase."

For a detailed discussion of arithmetic expressions, refer to Section 5. Also refer to "Intermediate Data Item," "ROUNDED Phrase," and "SIZE ERROR Phrase" in Section 5 for additional information.

Example

```
DIVIDE Item-1 BY Item-2 GIVING Item-3 ROUNDED REMAINDER Item-4
    ON SIZE ERROR DISPLAY  "Size error"
    NOT ON SIZE ERROR MOVE Item-3 TO Item-5 MOVE Item-4 TO Item-6
END-DIVIDE.
```

In this example, Item-1 is divided by Item-2. The quotient and remainder are calculated. If necessary, the quotient is rounded and then stored in Item-3. The remainder is stored in Item-4. If a size error condition occurs, the literal "Size error" is displayed on the ODT. If a size error condition does not occur, the quotient (Item-3) is moved to Item-5, and the remainder (Item-4) is moved to Item-6.

EVALUATE Statement

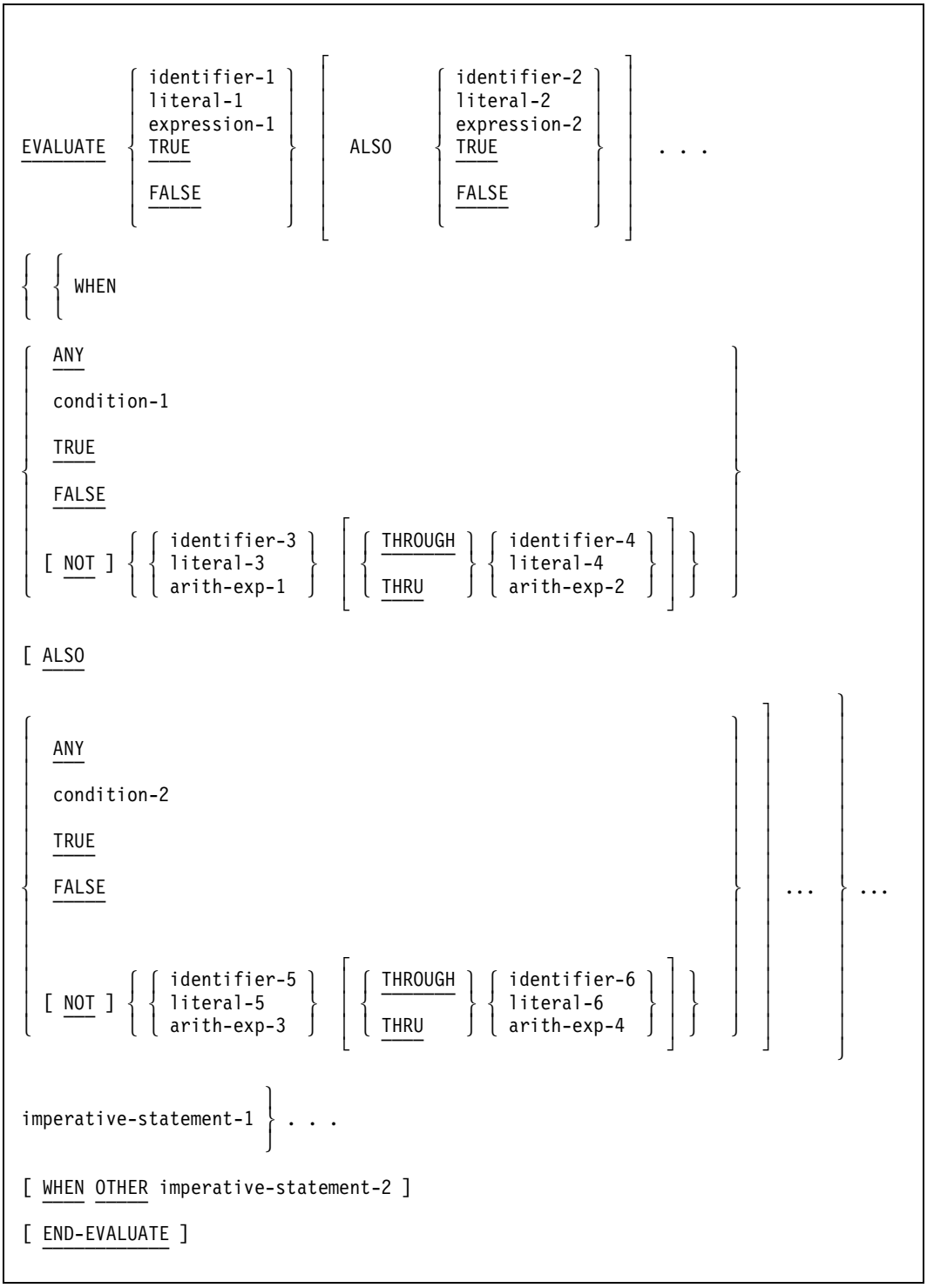
The EVALUATE statement causes multiple conditions to be evaluated: it tests one or many subjects against corresponding multiple objects. Subsequent action of the object program depends on the results of the evaluations.

The operands or the words TRUE and FALSE that appear before the first WHEN phrase of the EVALUATE statement are referred to individually as selection subjects and collectively as the set of selection subjects.

The operands or the words TRUE, FALSE, and ANY that appear in a WHEN phrase are referred to individually as selection objects and collectively as the set of selection objects.

For conceptual information about the different statements, sentences, and expressions, and how they are evaluated, refer to "Types of Statements and Sentences," "Arithmetic Expressions," and "Conditional Expressions" in Section 5.

Refer to "IF Statement" in this section for information on evaluating a condition.


EVALUATE
identifier-1
literal-1
expression-1
TRUE
FALSE
ALSO
identifier-2
literal-2
expression-2
TRUE
FALSE
. . .
WHEN
ANY
condition-1
TRUE
FALSE
[NOT]
identifier-3
literal-3
arith-exp-1
THROUGH
THRU
identifier-4
literal-4
arith-exp-2
[ALSO
ANY
condition-2
TRUE
FALSE
[NOT]
identifier-5
literal-5
arith-exp-3
THROUGH
THRU
identifier-6
literal-6
arith-exp-4
... ...
imperative-statement-1 . . .
[WHEN OTHER imperative-statement-2]
[END-EVALUATE]

Explanation

identifier-1
identifier-2

These identifiers are selection subjects and are user-defined words that refer to a data item.

literal-1
literal-2

These numeric, nonnumeric, or national literals are selection subjects.

expression-1
expression-2

These selection subjects can be arithmetic or conditional expressions. Refer to "Arithmetic Expressions" and "Conditional Expressions" in Section 5 for a description of these expressions.

TRUE

This reserved word is a conditional constant.

If TRUE appears before the WHEN phrase, it is a selection subject. If TRUE appears in a WHEN phrase, it is a selection object. The truth value of "TRUE" is assigned to those items specified with the word TRUE.

FALSE

This reserved word is a conditional constant.

If FALSE appears before the WHEN phrase, it is a selection subject. If FALSE appears in a WHEN phrase, it is a selection object. The truth value of "FALSE" is assigned to those items specified with the word FALSE.

ALSO

If this keyword appears before the WHEN phrase, it separates selection subjects from each other.

If ALSO appears in a WHEN phrase, it separates selection objects from each other.

The selection subjects or selection objects connected by ALSO form a selection set.

WHEN

Each WHEN phrase contains the selection objects. These objects are evaluated and compared with the selection subjects.

The number of selection objects within each set of selection objects must match the number of selection subjects.

When multiple WHEN phrases are used, each WHEN phrase, except the last, is ended by the beginning of the next WHEN phrase. The final WHEN phrase can be ended by a period or with the END-EVALUATE phrase.

If all the selection objects and the corresponding subjects match, imperative-statement-1 is executed. If they do not match, the next WHEN phrase is evaluated. Otherwise, control passes to the next executable statement.

ANY

This reserved word can correspond to a selection subject of any type.

condition-1
condition-2

These conditional expressions are selection objects.

NOT

This option causes the selection object preceded by the keyword NOT to match the selection subject if the value or range of values are different from those specified by the selection object.

For example, the NOT in the statement EVALUATE A WHEN NOT 14 MOVE A TO B causes A to be moved to B if A is any number other than 14.

identifier-3 through identifier-6
literal-3 through literal-6
arithmetic-expression-1 through arithmetic-expression-4

These elements are selection objects. The identifiers are user-defined words that refer to data items. The literals are numeric, nonnumeric, or national. For information on valid arithmetic expressions, refer to "Arithmetic Expressions" in Section 5.

THROUGH
THRU

These keywords can be used interchangeably. They connect two operands that represent a range of values.

The operands connected by a THROUGH or THRU phrase form a single selection object and must be of the same class.

imperative-statement-1

This imperative statement will be executed if the selection object matches the selection subject. After this statement is executed, control passes to the next executable statement after the EVALUATE statement.

WHEN OTHER
imperative-statement-2

This additional WHEN phrase provides a WHEN phrase that will be executed if none of the selection objects specified in the other WHEN phrases match the selection subjects.

END-EVALUATE

This phrase delimits the scope of the EVALUATE statement.

Correspondence between Selection Objects and Subjects

The selection object and selection subject must be of the same category and capable of matching.

Within a set of selection objects, each selection object must correspond to the selection subject that has the same position within the set of selection subjects. The following rules apply:

- Identifiers, literals, or arithmetic expressions that appear in a selection object must be valid operands for comparison to the corresponding operand in the set of selection subjects.
- Condition-1 or condition-2 and the words TRUE or FALSE, when used as selection objects, must correspond to either a conditional expression or the words TRUE or FALSE in the set of selection subjects.
- The word ANY can correspond to a selection subject of any type.

How Values Are Determined for Selection Subjects and Objects

The execution of the EVALUATE statement operates as if each selection subject and selection object were evaluated and assigned a numeric or nonnumeric value, a range of numeric or nonnumeric values, or a truth value. These values are determined as shown in the following table:

If the selection subject is specified by . . .	And the selection object is specified by . . .	Then the assigned value is . . .
Identifier-1 or identifier-2	Identifier-3 or identifier-5 (without the NOT or THROUGH phrase)	The value and class of the data item referenced by the identifier.
Literal-1 or literal-2	Literal-3 or literal-5 (without the NOT or THROUGH phrase)	The value and class of the specified literal. If literal-3 or literal-5 is the figurative constant ZERO, the literal is assigned the class of the corresponding selection subject.
An arithmetic expression for expression-1 or expression-2	Arithmetic-expression-1 or arithmetic-expression-3 (without either the NOT or the THROUGH phrases)	A numeric value according to the rules for evaluating an arithmetic expression. Refer to "Arithmetic Expressions" in Section 5 for more information.
A conditional expression for expression-1 or expression-2	Condition-1 or condition-2	A truth value that adheres to the rules for evaluating conditional expressions. Refer to "Conditional Expressions" in Section 5 for more information.
The words TRUE or FALSE	The words TRUE or FALSE	A truth value (a value of TRUE for TRUE and FALSE for FALSE).
The word ANY		Not evaluated further.
The THROUGH phrase (without the NOT phrase)		A range of values that includes all permissible values of the selection subject that are greater than or equal to the first operand and less than or equal to the second operand.

If the selection subject is specified by . . .	And the selection object is specified by . . .	Then the assigned value is . . .
	The NOT phrase	The set of all permissible values of the selection subject not equal to the value, or not included in the range of values, that would have been assigned to the item had the NOT phrase been left unspecified.

Comparison of Values

Execution of the EVALUATE statement proceeds as if the values assigned to the selection subjects and selection objects were compared to determine if any WHEN phrase satisfies the set of selection subjects. This comparison proceeds as follows:

1. Each selection object within the set of selection objects for the first WHEN phrase is compared to the selection subject that has the same ordinal position in the set of selection subjects. One of the conditions shown in the following table must be satisfied for the comparison to result in a match.

Item Being Compared	How the Item Satisfies the Comparison
Numeric or nonnumeric values	If one value, or a range of values, of the selection object equals the value of the selection subject
Truth values	If the items are assigned identical truth values
ANY	Always satisfies a comparison, regardless of the value of the selection subject

2. If the above comparison is satisfied for every selection object within the set of compared selection objects, the WHEN phrase that contains the set of selection objects satisfies the set of selection subjects.
3. If the above comparison is not satisfied for one or more selection objects within the set of compared selection objects, that set of selection objects does not satisfy the set of selection subjects.
4. This procedure is repeated for subsequent sets of selection objects, in the order of their appearance in the source program, until either a WHEN phrase satisfies the set of selection subjects or all sets of selection objects have been compared.

Completion of the EVALUATE Statement

After the comparison described in the preceding step is completed, execution of the EVALUATE statement proceeds as follows:

- If a WHEN phrase is selected, execution continues with the first imperative-statement-1 following the selected WHEN phrase. The use of multiple WHEN phrases with an imperative-statement is treated as a set of consecutive OR conditions.
- If a WHEN phrase is selected and a WHEN OTHER phrase is specified, exection continues with imperative-statement-2.
- The scope of execution of the EVALUATE statement is terminated when execution reaches either the end of imperative-statement-1 of the selected WHEN phrase or the end of imperative-statement-2, or when no WHEN phrase is selected and no WHEN OTHER phrase is specified.

Examples

```
EVALUATE Days-Overdue
  WHEN 0 THROUGH 30 PERFORM Standard-Bill
  WHEN 31 THROUGH 60 PERFORM Notice1
  WHEN 61 THROUGH 90 PERFORM Notice2
  WHEN OTHER PERFORM Collections-Report
END-EVALUATE.
```

This example is explained in the following table:

If the identifier Days-Overdue has a value that is . . .	Then the statements in . . . are executed.
0 to 30	Standard-Bill
31 to 60	Notice1
61 to 90	Notice2
Different from the other entries in this table	Collections-Report

```
EVALUATE TRUE ALSO Employee-Only =  "Y"
  WHEN Medical-Option = 1 ALSO TRUE MOVE Name TO Report-1-Name
  WHEN Medical-Option = 2 ALSO TRUE MOVE Name TO Report-2-Name
  WHEN Medical-Option = 3 ALSO TRUE MOVE Name TO Report-3-Name
END-EVALUATE.
```

This example is explained in the following table.

If the Medical-Option has a value of . . .	And Employee-Only has a value of . . .	Then the value of Name is moved to . . .
1	Y	Report-1-Name
2	Y	Report-2-Name
3	Y	Report-3-Name

The following examples produce the same result.

```
EVALUATE Medical-Option ALSO Employee-Only = "Y"
  WHEN 1 ALSO TRUE MOVE Name TO Report-1-Name
  WHEN 2 ALSO TRUE MOVE Name TO Report-2-Name
  WHEN 3 ALSO TRUE MOVE Name TO Report-3-Name
END-EVALUATE.

EVALUATE TRUE
  WHEN Medical-Option = 1 AND Employee-Only = "Y" MOVE Name TO Report-1-Name
  WHEN Medical-Option = 2 AND Employee-Only = "Y" MOVE Name TO Report-2-Name
  WHEN Medical-Option = 3 AND Employee-Only = "Y" MOVE Name TO Report-3-Name
END-EVALUATE.
```

These examples are explained in the following table.

If the Medical-Option has a value of . . .	And Employee-Only has a value of . . .	Then the value of Name is moved to . . .
1	Y	Report-1-Name
2	Y	Report-2-Name
3	Y	Report-3-Name

The following example illustrates the use of multiple WHEN phrases:

```
     EVALUATE WS-FIELD
*    Multiple WHEN phrases for one imperative-statement are ORed
*    to form the selection object.
     WHEN 1
     WHEN 2
*        True when WS-FIELD = 1 or 2.
         DISPLAY "VALUE IS 2"
     WHEN 3
         DISPLAY "VALUE IS 3"
     END-EVALUATE.

*    This produces the same result as the preceding EVALUATE.
     EVALUATE TRUE
     WHEN WS-FIELD = 1 OR 2
         DISPLAY "VALUE IS 2"
     WHEN WS-FIELD = 3
         DISPLAY "VALUE IS 3"
     END-EVALUATE.
```

EXIT Statement

Format	Use
Format 1	This format indicates the logical end of a series of sections or paragraphs referenced by a PERFORM statement.
Format 2	This format exits a program that was called by Format 1 or Format 2 of the CALL statement.
Format 3	This format exits a bound procedure that was called by Format 4 of the CALL statement.
Format 4	This format exits a task that was initiated by Format 6 of the CALL statement.
Format 5	This format exits a bound procedure that was called by a CALL MODULE statement (Format 6 of the CALL statement).
Format 6	This format provides a way to bypass the remainder of a PERFORM statement range.

Format 1: EXIT from an Out-of-Line PERFORM

```
EXIT
```

Explanation

EXIT

The EXIT statement must appear in a sentence by itself. It must be the only sentence in the paragraph.

Details

An EXIT statement assigns a procedure-name to a given point in a program. The EXIT statement has no other effect on the compilation or execution of the program.

Example

```
Main-Para.
     .
     .
     .
    PERFORM Read-Para THRU Exit-Para
        UNTIL In-Record =  "NO".
     .
     .
     .
Read-Para.
    READ INFILE AT END MOVE  "NO" TO In-Record
         GO TO Total-Print-Para.
    GO TO Exit-Para.

Total-Print-Para.
     .
     .
     .
Exit-Para.
    EXIT.
```

The EXIT statement in this example concludes a series of paragraphs indicated by the PERFORM statement.

Format 2: EXIT from a Called Program (ANSI IPC)

```
EXIT PROGRAM
```

Details

The EXIT PROGRAM statement is not required to be in a separate paragraph as is the Format 1 EXIT statement.

The effect of the EXIT program statement on the called program depends on whether the IS INITIAL PROGRAM clause is present in the PROGRAM-ID paragraph of the called program. This clause declares that the program and any programs it contains are placed in their initial state each time they are called.

- If the IS INITIAL PROGRAM clause is present, the EXIT PROGRAM statement is equivalent to a CANCEL statement for the called program.
- If the IS INITIAL PROGRAM clause is absent, the EXIT PROGRAM statement causes execution to continue with the next executable statement following the CALL statement in the calling program.

  When control is passed between the calling and called programs, it is possible for the contents of shared data items and shared data files to change.

  The EXIT PROGRAM statement closes all PERFORM statements in the called program. If a PERFORM procedure is interrupted and the implicit return instruction at the end of that procedure has not been executed, the EXIT PROGRAM statement cancels that implicit return.

  If the EXIT PROGRAM statement is in a program that is not under the control of a calling program, the statement has no effect and the program continues execution.

Restrictions

The following restrictions apply to the EXIT PROGRAM statement; it must

- Be the last statement in a consecutive sequence of imperative statements.
- Not appear in a declarative procedure in which the GLOBAL phrase is specified.

Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PROGA.
    .
    .
    .
PROCEDURE DIVISION.

    CALL "PROGB" USING A, C.
    .
    .
    .
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PROGB.
     .
     .
     .
LINKAGE SECTION.
01 Employee-Data . . .
01 Salary . . .
     .
     .
     .
PROCEDURE DIVISION USING
     Employee-Data, Salary.
     .
     .
     .
     EXIT PROGRAM.
```

In this example, program PROGA calls program PROGB. The statements in PROGB are executed. The EXIT PROGRAM statement causes execution to continue with the next executable statement following the CALL statement in PROGA.

Format 3: EXIT from a Bound Procedure

```
EXIT PROCEDURE
```

Details

The EXIT PROCEDURE statement should be used only for procedures compiled at lexical level 3 or higher. If the procedure has been processed or called as a coroutine when the EXIT PROCEDURE statement is encountered, the process goes to end-of-task (EOT). If the procedure has been called as a procedure, a normal procedure exit occurs back to the statement that follows the procedure invocation in the calling program.

An implicit EXIT PROCEDURE statement is compiled for all procedures compiled at level 3 or higher. The EXIT PROCEDURE statement need not be used when it would be the final statement in the procedure. Refer to "CALL Statement," "CANCEL Statement," and "PERFORM Statement" in this section for syntax and detailed information.

Format 4: EXIT from a Called Program (Tasking)

```
EXIT PROGRAM  [ RETURN HERE ]  .
```

Details

Use this statement in the called program to return to the calling program. (The calling program originally initiated the called program by using Format 6 of the CALL statement.)

When either an EXIT PROGRAM or EXIT PROGRAM RETURN HERE statement is reached in the called program, control is returned to the statement following the CALL statement in the calling program. Afterward, control is passed between the two programs as shown in the following table.

If the calling program issues a subsequent CONTINUE statement and the called program was previously exited by an . . .	Then control returns to . . .
EXIT PROGRAM statement	The first logically executable statement in the called program
EXIT PROGRAM RETURN HERE statement	The statement immediately following the EXIT PROGRAM RETURN HERE statement

Note that the contents of data items and data files shared between the calling and called programs might change between successive executions of the CONTINUE statement.

Note also that the EXIT PROGRAM statement closes all PERFORM statements in the called program. If a PERFORM procedure is interrupted and the implicit return instruction at the end of that procedure has not been executed, the EXIT PROGRAM statement cancels that implicit return.

An EXIT PROGRAM RETURN HERE statement cannot appear in a bound procedure.

For an example of how the EXIT PROGRAM statement is used, refer to the tasking examples in Section 13.

Format 5: EXIT MODULE

```
EXIT MODULE
```

Details

An EXIT MODUDE statement returns control from a called program to the calling program. The EXIT MODULE statement must appear in a sentence by itself and must be the only sentence in the paragraph. Refer to the "CALL MODULE Statement" in this section and to the *Binder Programming Reference Manual*.

If a program has not been initiated by a CALL MODULE statement and an EXIT MODULE statement is encountered, the program is not exited, and execution begins with the next statement of the program.

The compiler generates a warning message if an EXIT MODULE statement is found in a program that does not have the CALLMODULE CCI set. See Section 15 for a description of the CALLMODULE compiler option.

Format 6: EXIT from a PERFORM Statement

```
EXIT PERFORM
```

Details

The EXIT PERFORM statement provides a way to bypass the remainder of a PERFORM statement range.

If the program is under the control of an in-line or an out-of-line PERFORM statement when the EXIT PERFORM statement is encountered, any remaining statements in the PERFORM statement range are bypassed. This will terminate format 1 PERFORM statements. All other PERFORM statement formats terminate only when the specified terminating conditions are met. If an EXIT PERFORM statement is executed when no PERFORM statement is active, control passes to the next statement.

Examples

Note: *In the following out-of-line PERFORM and in-line PERFORM examples, the EXIT PERFORM statement bypasses the ADD statement. The program displays "WS-COUNT = 0".*

Exit from basic out-of-line PERFORM:

```
IDENTIFICATION DIVISION.
ENVIRONMENT    DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  WS-COUNT  PIC 9(2) VALUE 0.

PROCEDURE DIVISION.
MAIN.
   PERFORM P1.
   DISPLAY "WS-COUNT = " WS-COUNT.
   STOP RUN.
P1.
   IF (WS-COUNT = 0)
       EXIT PERFORM
   ELSE
       CONTINUE
   END-IF.
   ADD 1 TO WS-COUNT.
```

Exit from basic in-line PERFORM:

```
IDENTIFICATION DIVISION.
ENVIRONMENT    DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  WS-COUNT  PIC 9(2) VALUE 0.

PROCEDURE DIVISION.
MAIN.
   PERFORM
     IF (WS-COUNT = 0)
         EXIT PERFORM
     ELSE
         CONTINUE
     END-IF
     ADD 1 TO WS-COUNT
   END-PERFORM.
   DISPLAY "WS-COUNT = " WS-COUNT.
   STOP RUN.
```

Note: *In the following out-of-line PERFORM ... UNTIL and in-line PERFORM ... UNTIL examples, The PERFORM statement range is executed five times. The EXIT PERFORM statement bypasses the DISPLAY statement in the PERFORM statement range. The program display "WS-COUNT = 1" and "COUNT = 5".*

Exit from out-of-line PERFORM ... UNTIL:

```
IDENTIFICATION DIVISION.
ENVIRONMENT    DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  WS-COUNT  PIC 9(2) VALUE 0.

PROCEDURE DIVISION.
MAIN.
   PERFORM P1 UNTIL WS-COUNT = 5
   DISPLAY "COUNT = " WS-COUNT.
   STOP RUN.
P1.
   ADD 1 TO WS-COUNT.
   IF (WS-COUNT > 1)
       EXIT PERFORM
   ELSE
       CONTINUE
   END-IF.
   DISPLAY "WS-COUNT = " WS-COUNT.
```

Exit from in-line PERFORM ... UNTIL:

```
IDENTIFICATION DIVISION.
ENVIRONMENT    DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  WS-COUNT  PIC 9(2) VALUE 0.

PROCEDURE DIVISION.
MAIN.
   PERFORM UNTIL WS-COUNT = 5
     ADD 1 TO WS-COUNT
     IF (WS-COUNT > 1)
         EXIT PERFORM
     ELSE
         CONTINUE
     END-IF
     DISPLAY "WS-COUNT = " WS-COUNT
   END-PERFORM.
   DISPLAY "COUNT = " WS-COUNT.
   STOP RUN.
```

Note: *In the following example, the EXIT PERFORM statement is executed when no PERFORM statement is active, so control passes to the DISPLAY statement. The program displays "WS-COUNT = 1".*

EXIT PERFORM when no PERFORM statement is active:

```
IDENTIFICATION DIVISION.
ENVIRONMENT    DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  WS-COUNT  PIC 9(2) VALUE 0.

PROCEDURE DIVISION.
MAIN.
   IF WS-COUNT = 0
      GO TO P1.
   STOP RUN.
P1.
   ADD 1 TO WS-COUNT.
   EXIT PERFORM.
   DISPLAY "WS-COUNT = " WS-COUNT.
```

GO TO Statement

The GO TO statement transfers control unconditionally from one procedure to another.

Format	Use
Format 1	The GO TO format transfers control from one part of the Procedure Division to another.
Format 2	The GO TO . . . DEPENDING ON format transfers control from one part of the Procedure Division to another depending on the value of a specified integer identifier.

Format 1: GO TO

```
GO TO  [ procedure-name-1 ]
```

Explanation

GO TO

If procedure-name-1 is specified, control is transferred to procedure-name-1.

If procedure-name-1 is not specified, the following rules apply:

- The GO TO statement can appear only in a single statement paragraph (that is, a paragraph that consists of a paragraph header followed by the GO TO statement).
- An ALTER statement that refers to this GO TO statement must be executed before the execution of this GO TO statement. (The ALTER statement directs the GO TO statement to a destination.)

Control is not returned to the statement following the GO TO statement.

procedure-name-1

This user-defined word names a paragraph or section in the Procedure Division. It consists of a paragraph-name (which can be qualified) or a section-name.

Procedure-name-1 as an option is an obsolete element that will be deleted from the next revision of Standard COBOL. Procedure-name-1 will become a mandatory part of the syntax.

Details

If a Format 1 GO TO statement appears in a consecutive sequence of imperative statements in a sentence, it must appear as the last statement in that sequence.

Format 2: GO TO . . . DEPENDING ON

```
GO TO  [ procedure-name-1 [ [  , ] procedure-name-2 ] ] . . .
  DEPENDING ON identifier-1
```

Explanation

procedure-name-1
procedure-name-2

These user-defined words name a paragraph or section in the Procedure Division. A procedure-name consists of a paragraph-name (which can be qualified) or a section-name.

DEPENDING ON

This phrase causes the transfer of control to depend on the value of identifier-1 being 1, 2, . . . , n. The exact procedure-name is selected by the value of identifier-1. If the value of identifier-1 is anything other than the positive or unsigned integers 1, 2, . . . , n, then no transfer occurs and control passes to the next statement in the normal sequence for execution.

identifier-1

This element must reference an elementary, numeric data item that is an integer. Identifier-1 cannot reference a long numeric data item.

Refer to "ALTER Statement" in this section and "PERFORM Statement" in Section 7 for syntax and detailed information.

Examples

```
WORKING-STORAGE SECTION.
 01  POINTER-ID PIC 99  VALUE 01.
 .
 .
 .
 PROCEDURE DIVISION.
 .
 .
 .
 Check-Para.
        IF POINTER-ID IS GREATER THAN 59 MOVE 01 TO POINTER-ID
          GO TO UNSTRING-Para.
        MOVE SPACES TO OUT-RECORD.
 .
 .
 .
 UNSTRING-Para.
```

In this example, if the value of POINTER-ID is greater than 59, 01 is moved to POINTER-ID, and control is transferred to UNSTRING-Para.

```
WORKING-STORAGE SECTION.
77 I   PIC 9.
     .
     .
     .
PROCEDURE DIVISION.
P1.
     .
     .
     .
P2.
     .
     .
     .
P3.
    GO TO P1, P2 DEPENDING ON I.
```

In this example, control will be transferred from the GO TO statement in P3 depending on the value in I. If I equals 1, then P1 is executed. If I equals 2, P2 is executed. If I equals anything but 1 or 2, the program proceeds to the statements that follow the GO TO statement in the program.

Section 7
Procedure Division Statements I–R

This section illustrates and explains the syntax of the Procedure Division statements. Statements beginning with the letters I through R are listed in alphabetical order with the following information:

- A brief description of the function of the statement
- A syntax diagram for each format of the statement (if you need information on how to interpret a COBOL syntax diagram, refer to Appendix C).
- A statement of what portion of the syntax, if any, can be used interactively in a Test and Debug System (TADS) session
- An explanation of the elements in the syntax diagram
- Details, rules, and restrictions about the particular statement
- An example of the statement
- References to additional information relevant to the statement

Detailed information about language elements common to many Procedure Division statements, such as user-defined names, literals, and identifiers is provided in Section 1. Concepts such as arithmetic and conditional expressions, and operations such as table handling, sorting, and merging are described in Section 5.

IF Statement

The IF statement evaluates a condition. Subsequent action of the object program depends on whether the value of the condition is TRUE or FALSE.

See "Conditional Expressions" in Section 5 for conceptual information on the different types of conditions. See "EVALUATE Statement" in this section for more information.

```
                          ⎧  ⎧ statement-1   ⎫       ⎫
IF condition-1 THEN       ⎨  ⎨              ⎬ . . .  ⎬
                          ⎩  ⎩ NEXT SENTENCE ⎭       ⎭

⎧ ELSE { statement-2 } . . . [ END-IF ]  ⎫
⎪                                        ⎪
⎨ ELSE NEXT SENTENCE                     ⎬
⎪                                        ⎪
⎩ END-IF                                 ⎭
```

Explanation

condition-1

This element specifies a test condition. The object program selects between alternate paths of control depending on the truth-value of the condition. A condition can be simple or complex. Refer to "Conditional Expressions" in Section 5 for detailed information.

THEN statement-1 . . .

If the condition is TRUE, statement-1 is executed. If the condition is FALSE, statement-1 is ignored.

Statement-1 can be either an imperative statement or a conditional statement optionally preceded by an imperative statement.

Statement-1 can contain an IF statement. In this case, the IF statement is said to be nested.

NEXT SENTENCE

This phrase is for documentation purposes only. If condition-1 is TRUE, NEXT SENTENCE indicates that the next sentence (the sentence following the IF statement) will be executed.

ELSE statement-2 . . .
END-IF

If condition-1 is false, statement-2 is executed.

Statement-2 can contain an IF statement. In this case, the IF statement is said to be nested.

The END-IF phrase delimits the scope of an IF statement at the same level of nesting.

ELSE NEXT SENTENCE

This phrase is for documentation purposes only. If condition-1 is false, NEXT SENTENCE indicates that the sentence following the IF statement will be executed.

This phrase can be omitted if it immediately precedes the terminal period of the sentence.

END-IF

This delimits the scope of the IF statement.

If the END-IF phrase is specified, the NEXT SENTENCE phrase must not be specified.

How the IF Statement Is Evaluated

When an IF statement is executed, transfers of control occur according to the conditions described in the following tables.

If condition-1 is TRUE and . . .	Then . . .
One or more statements are specified with the THEN phrase	Control is transferred to the first statement specified with the THEN phrase. Execution continues according to the rules for that statement. If statement-1 is a conditional statement or a procedure-branching statement that explicitly transfers control, then control is transferred according to the rules for that statement. After the statement or statements associated with the THEN phrase are executed, control passes to the end of the IF statement. Note that any statements specified with the ELSE phrase are ignored.
The NEXT SENTENCE phrase is specified (instead of statement-1)	Control passes to the next executable sentence. Any statements specified with the ELSE phrase are ignored.

If condition-1 is FALSE and . . .	Then . . .
One or more statements are specified with the ELSE phrase.	Statement-1 or its surrogate NEXT SENTENCE is ignored, and control is transferred to the first statement specified in the ELSE phrase. Execution continues according to the rules for each statement. If statement-2 is a conditional statement or a procedure-branching statement that explicitly transfers control, then control is transferred according to the rules for that statement. After the statement or statements associated with the ELSE phrase are executed, control passes to the end of the IF statement.
The ELSE phrase is not specified	Statement-1 is ignored and control passes to the next executable sentence.

Nested IF Statements

IF statements within IF statements can be considered as paired IF, ELSE, and END-IF combinations, proceeding from left to right. Each ELSE or END-IF encountered corresponds to the immediately preceding IF that has not been paired with an ELSE or END-IF, as follows:

```
IF A > B PERFORM A-PROC
    IF B < C PERFORM C-PROC
    ELSE PERFORM B-PROC
    END-IF
ELSE PERFORM MAIN-ROUTINE
END-IF.
```

The first IF (IF A > B) is paired with the last ELSE (ELSE PERFORM MAIN-ROUTINE) and the last END-IF. The second IF (IF B < C) is paired with the next encountered ELSE (ELSE PERFORM B-PROC) and END-IF.

Terminating an IF Statement

The scope of an IF statement can be terminated by one of the following:

- An END-IF phrase at the same level of nesting
- A separator period
- If nested, by an ELSE phrase associated with an IF statement at a higher level of nesting

Examples

```
IF hours-worked > 40
    SUBTRACT 40 FROM hours-worked GIVING overtime-hours
    MULTIPLY .5 BY hourly-rate GIVING overtime-pay ROUNDED
    ADD overtime-pay TO gross-pay
ELSE
   PERFORM Standard-Pay-Routine
END-IF.
```

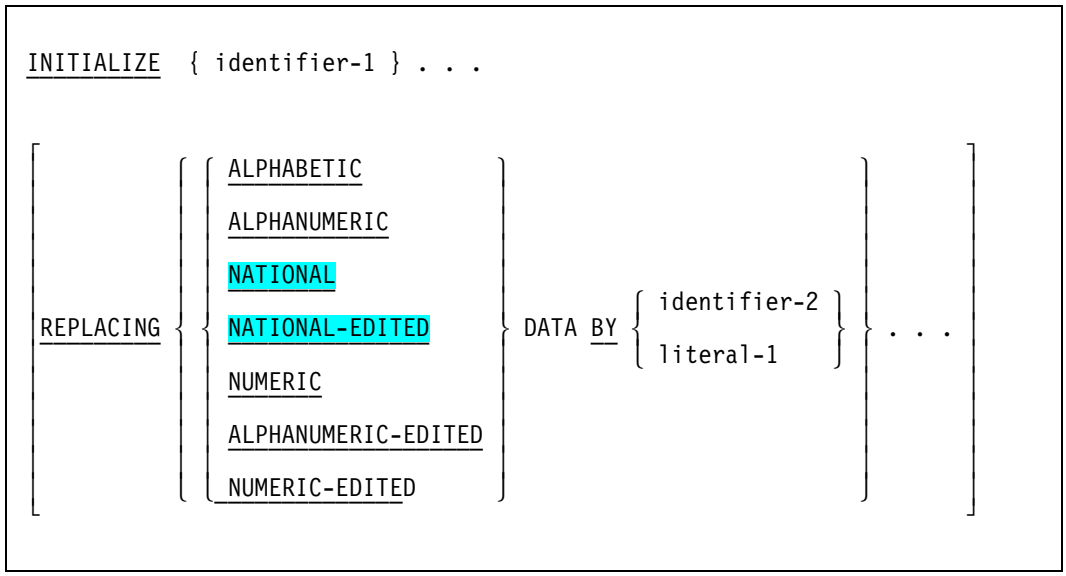
If the value of hours-worked is greater than 40, the calculations for overtime pay are made. If the value of hours-worked is equal to or less than 40, the statements in Standard-Pay-Routine are executed.

```
IF SW1-ON
    PERFORM Inspect-Proc
ELSE IF SW5-ON
          PERFORM Search-Proc
     ELSE NEXT SENTENCE.
```

If SW1 is set, then the statements in Inspect-Proc are executed. If SW1 is not set and SW5 is set, the statements in Search-Proc are executed. If SW1 is not set and SW5 is not set, control passes to the next executable sentence.

INITIALIZE Statement

The INITIALIZE statement enables you to set selected types of data fields to predetermined values (for example: numeric data to zeros, or alphanumeric data to spaces). A group of elementary data items can have its initial value set with one statement.



Explanation

identifier-1

The data item referred to by identifier-1 represents the receiving area. Identifier-1 can represent an elementary or group item.

The following restrictions apply to identifier-1:

- An index data item cannot appear as an operand of an INITIALIZE statement. If identifier-1 is a long numeric data item, then one of the following conditions must be true:
  - Identifier-2 must be a long numeric data item of the same size and usage.
  - Literal-1 must be either an appropriate figurative constant, 0 (zero), or a numeric literal that contains the same number of digits as the long numeric data item.
- The description of the data item referred to by identifier-1, or any items subordinate to identifier-1, cannot contain the DEPENDING phrase of the OCCURS clause. In addition, the data description entry for the data item referred to by identifier-1 cannot contain a RENAMES clause.
- Any item that is subordinate to the data item referred to by identifier-1 and that contains the REDEFINES clause, or any item that is subordinate to such an item, is excluded from the initialization operation. However, the data item referred to by identifier-1 can have a REDEFINES clause, or be subordinate to a data item with a REDEFINES clause.

REPLACING . . .

The data category you specify in the REPLACING phrase must be a permissible category for the data item. You cannot repeat the same data category in the same REPLACING phrase.

If you do not specify a REPLACING phrase, the following values are assumed for the categories of data items:

Data items of category . . .	**Are set to. . .**
Alphabetic	Spaces
Alphanumeric	Spaces
Alphanumeric-edited	Spaces
National	National spaces
National-edited	National spaces
Numeric	Zeros
Numeric-edited	Zeros

In all cases, the INITIALIZE statement operates as if each affected data item is the receiving area in an elementary MOVE statement with the indicated source literal (that is, spaces or zeros).

BY
identifier-2
literal-1

The data item referred to by identifier-2 or the string in literal-1 represents the sending area.

Details

All operations are performed as if a series of MOVE statements (each with an elementary item as its receiving field) had been written, subject to the following rules:

- If identifier-1 references a group item, any elementary item within that group item is initialized only if it belongs to the data category specified in the REPLACING phrase.
- If identifier-1 references an elementary item, that item is initialized only if it belongs to the data category specified in the REPLACING phrase.

The data item referred to by identifier-2 or literal-1 acts as the sending operand in an implicit MOVE statement (refer to the "MOVE Statement" in this section for more information). The data item referred to by identifier-1 is set to the indicated value in the order (left to right) of the appearance of identifier-1. Within this sequence, where identifier-1 references a group item, affected elementary items are initialized in the sequence of their definitions within the group. However, index data items and filler data items are not affected by an INITIALIZE statement.

If the data item referred to by identifier-1 occupies the same storage area as the data item referred to by identifier-2, the result of the execution of the INITIALIZE statement is undefined, even if the data items are defined by the same data description entry.

Refer to "REDEFINES Clause," "OCCURS Clause," and "RENAMES Clause" in Section 4 for syntax and detailed information.

Refer to "MOVE Statement" in this section for information on the MOVE rules.

Examples

```
DATA DIVISION.
WORKING-STORAGE SECTION.
77  Temp-1  PIC X(3).
       .
       .
       .
 PROCEDURE DIVISION.
    INITIALIZE  Temp-1 REPLACING ALPHANUMERIC DATA BY  "ABC".
```

This statement moves the literal "ABC" to the data item Temp-1.

```
DATA DIVISION.
01  Group-1
    05  Name    PIC A(10).
    05  Address PIC X(20).
    05  Age     PIC 99.
    05  Salary  PIC $99999V99.

PROCEDURE DIVISION.
    INITIALIZE Group-1.
```

The INITIALIZE statement in this example initializes the elementary data items in Group-1 with zeros or spaces, depending on their definition. Name and Address are initialized with spaces; Age and Salary are initialized with zeros.

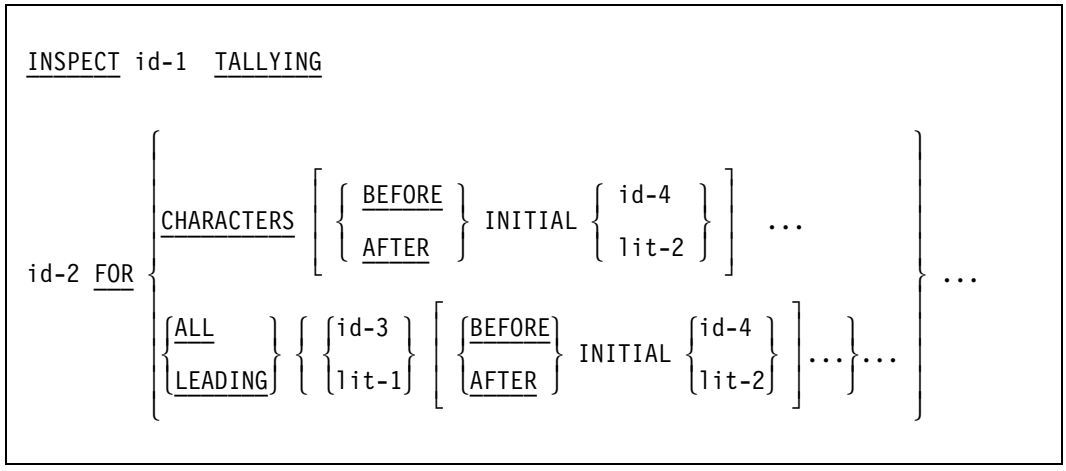
INSPECT Statement

The INSPECT statement can tally, replace, or tally and replace occurrences of single characters or groups of characters in a data item.

Refer to "STRING Statement" and "UNSTRING Statement" in this section for information on other statements that enable you to manipulate data.

Format	Use
Format 1	The INSPECT . . . TALLYING format tallies single characters or groups of characters.
Format 2	The INSPECT . . . REPLACING format replaces single characters or groups of characters.
Format 3	The INSPECT . . . TALLYING and REPLACING format tallies and replaces single characters or groups of characters; this format combines Formats 1 and 2.
Format 4	The INSPECT . . . CONVERTING format replaces single characters as if multiple Format 2 statements had been written.

Format 1: INSPECT . . . TALLYING



Explanation

id-1

Id-1 references the data item that you want to inspect.

Id-1 must reference either a group item or any category of elementary item described (either implicitly or explicitly) as USAGE IS DISPLAY or USAGE IS NATIONAL.

Identifier-1 can reference a long numeric data item.

The INSPECT statement treats id-1 as shown in the following table.

If the data item is . . .	Then the INSPECT statement treats the content of the data item as . . .
Alphanumeric	A character string.
_ Alphanumeric-edited _ Numeric-edited _ Unsigned numeric	Though it had been redefined as alphanumeric, and as though the INSPECT statement had been written to reference the redefined data item.
National	A national character string.
National-edited	Though it had been redefined as national, and as though the INSPECT statement had been written to reference the redefined data item.
Signed numeric	Though it had been moved to an unsigned numeric data item of the same length, and as though the INSPECT statement had been written to reference the redefined data item. The original value of the sign is retained upon completion of the INSPECT statement.

id-2

Id-2 designates the data item in which the tally count is to be accumulated. The data item referenced by identifier-2 must be an elementary numeric item. The data item cannot be a long numeric data item. Note that the data item is not initialized by the execution of the INSPECT statement.

id-3
id-4

Id-3 and id-4 must reference either a group item or any category of elementary item, described (either implicitly or explicitly) as USAGE IS DISPLAY or USAGE IS NATIONAL. The usage for id-3 and id-4 must be the same as the usage for id-1.

The INSPECT statement treats id-3 and id-4 in the same way as id-1. Refer to the description of id-1.

When the CHARACTERS phrase is used, id-4 and lit-2 must be one character in length.

lit-1
lit-2

Lit-1 and lit-2 must be nonnumeric literals if id-1 is described as USAGE IS DISPLAY, or national literals if id-1 is described as USAGE IS NATIONAL.

When the CHARACTERS phrase is used, lit-2 and id-4 must be one character in length.

CHARACTERS

If you specify CHARACTERS, the content of the data item referred to by id-2 is incremented by one for each character within the data item referred to by id-1.

ALL
LEADING

These adjectives apply to each lit-1 or id-3 that follows them, until the next ALL or LEADING phrase.

If you specify ALL, the content of id-2 is incremented by one for each occurrence of lit-1 (or the data item referred to by id-3) found within the data item referred to by id-1.

If you specify LEADING, the content of id-2 is incremented by one for each contiguous occurrence of lit-1 (or the data item referred to by id-3) found within the data item referred to by id-1, provided that the leftmost occurrence is at the point where comparison began in the first comparison cycle in which lit-1 or id-3 was eligible to participate (refer to the heading "The Comparison Cycle" under this statement).

BEFORE
AFTER

You can specify only one BEFORE and one AFTER phrase for any one ALL, LEADING, or CHARACTERS phrase. Both BEFORE and AFTER can appear in the same INSPECT statement. See "The Comparison Cycle" for details of how these keywords function.

The Process of Inspection

Inspection, which includes the comparison cycle, the establishment of boundaries for the BEFORE and AFTER phrases, and the mechanism for tallying, begins at the leftmost character position of the data item referred to by id-1, regardless of its class. Inspection proceeds from left to right to the rightmost character position.

Function-identifiers and identifiers with subscripts are evaluated only once as the first operation in the execution of the INSPECT statement.

If the data items referred to by id-1, id-3, or id-4 occupy the same storage area as the data item referred to by id-2, the result of the execution of the INSPECT statement is undefined, even if the identifiers are defined by the same data description entry.

During inspection of the content of the data item referred to by id-1, each properly matched occurrence of lit-1 (or the data item referred to by id-3) is tallied, and the tally is stored in the data item referred to by id-2.

The Comparison Cycle

The operands of the TALLYING phrase are considered in the order they are specified in the INSPECT statement from left to right. The first lit-1 is compared to an equal number of contiguous characters, starting with the leftmost character position in the data item referred to by id-1. Lit-1 matches that portion of the content of the data item referred to by id-1 if, and only if, they are equal, character for character.

Note: *In this discussion, any reference to lit-1 applies to id-3. Any reference to lit-2 applies to id-4.*

If a match does not occur, the comparison continues with the next lit-1. This process repeats until there is no next lit-1.

The next cycle begins with the character position in the data item referred to by id-1 immediately to the right of the leftmost character position considered in the last comparison cycle.

Whenever a match occurs, the content of the data item referred to by id-2 is incremented as described earlier, and the character position to the right of the rightmost character position considered in the comparison becomes the leftmost character position of a new comparison cycle.

The comparison cycles continue until the rightmost character position of the data item referred to by id-1 has participated in a match or has been considered as the leftmost character position. When this occurs, inspection is terminated.

How the BEFORE and AFTER Phrases Affect the Comparison Cycle

The BEFORE and AFTER phrases affect the comparison cycle as follows:

- If you do not specify a BEFORE or AFTER phrase, the entire data item referred to by id-1 is involved in the inspection.
- If you specify the BEFORE phrase, the comparison cycle includes only that portion of the data item referred to by id-1 from its leftmost character up to, but not including, the first occurrence of the data item referred to by lit-2. The position of this first occurrence is determined before the first comparison cycle.
- If the data item referred to by lit-2 is not found in the content of the data item referred to by id-1, the INSPECT statement executes as if you did not specify a BEFORE phrase.
- If you specify the AFTER phrase, the comparison cycle includes only that portion of the data item referred to by id-1 between the character position immediately to the right of the rightmost character position of the first occurrence of lit-2 and the rightmost character position of the data item referred to by id-1. The position of this first occurrence is determined before the first comparison cycle begins.
- If the data item referred to by lit-2 is not found in the content of the data item referred to by id-1, there is no inspection for the corresponding lit-1.

Examples

```
INSPECT Word TALLYING count1
FOR LEADING  "L"BEFORE INITIAL "A".
```

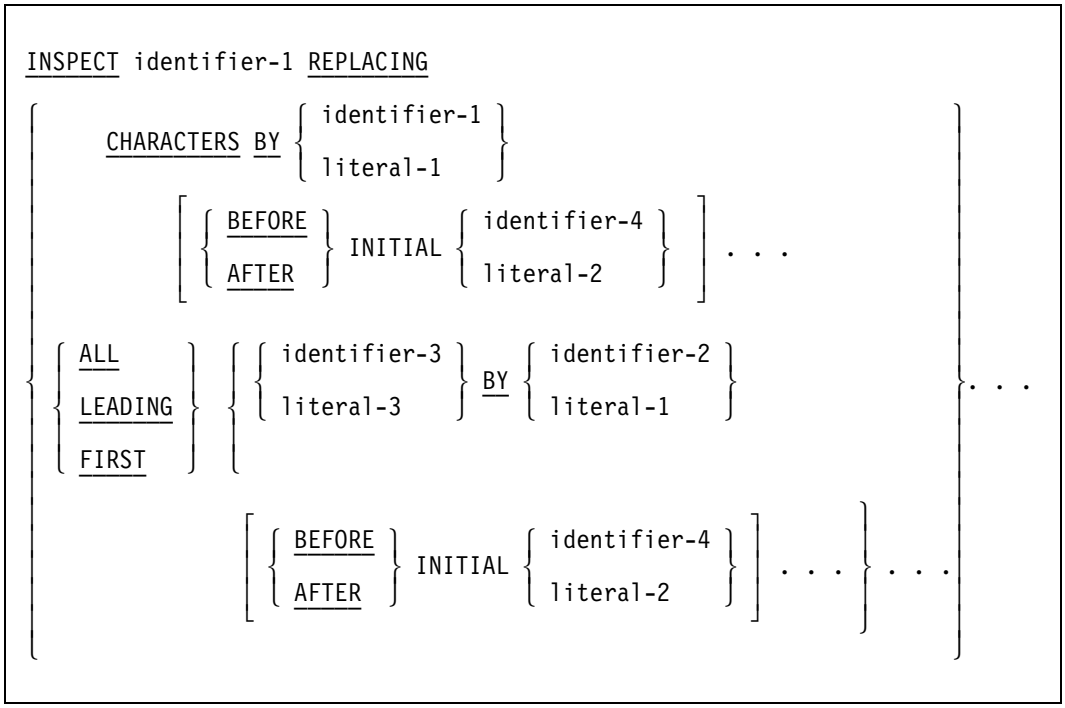
If the data item Word contained the value LARGE, the tally, which is stored in the data item Count1, would be 1.

If the data item Word contained the value ANALYST, the tally would be 0.

```
INSPECT Word TALLYING count1
FOR CHARACTERS AFTER INITIAL "J".
```

If the data item Word contained the value ADJECTIVE, the tally, which is stored in the data item Count1, would be 6.

Format 2: INSPECT . . . REPLACING



Explanation

identifier-1
identifier-2
identifier-3
identifier-4

Identifier-1 must refer to either a group item or any category of elementary item described (either implicitly or explicitly) as USAGE IS DISPLAY or USAGE IS NATIONAL. Identifier-1 can reference a long numeric data item.

Identifier-2 through identifier-n must refer to an elementary alphanumeric (alphabetic) item, an elementary numeric item, or an elementary national item.

When identifier-1 through identifier-n are alphanumeric (alphabetic) or numeric, the data items they reference must be described (implicitly or explicitly) as USAGE IS DISPLAY.

When identifier-1 through identifier-n are national, the data items they reference must be described (implicitly or explicitly) as USAGE IS NATIONAL.

The size of the data item referred to by identifier-3 must be equal to the size of the data item referred to by identifier-2.

When you specify the CHARACTERS BY phrase, the data item referred to by identifier-2 and identifier-4 must be one character in length.

In Format 2 of the INSPECT statement, the following rules apply to the data item referenced by identifiers:

If the data item is . . .	Then the INSPECT statement treats the content of the data item as . . .
Alphanumeric	A character-string.
_ Alphanumeric-edited _ Numeric-edited _ Unsigned numeric	Though it is redefined as alphanumeric and as though the INSPECT statement is written to refer to the redefined data item.
National	A national character string.
National-edited	Though it is redefined as national, and as though the INSPECT statement is written to reference the redefined data item.
Signed numeric	Though it is moved to an unsigned numeric data item of the same length and as though the INSPECT statement is written to refer to the redefined data item.

literal-1
literal-2
literal-3

These literals must be nonnumeric if identifier-1 is described as USAGE IS DISPLAY. These literals must be national if identifier-1 is described as USAGE IS NATIONAL. These literals cannot be any figurative constant that begins with the word ALL. If any literal is national, all the literals must be national. When you specify the CHARACTERS BY phrase, literal-1 and literal-2 must be one character in length.

If literal-2 is a figurative constant, it is implicitly a one-character data item.

The size of literal-3 must be equal to the size of literal-1.

REPLACING

The REPLACING phrase enables you to replace a single character or groups of characters in a data item.

CHARACTERS BY

If you specify the CHARACTERS BY phrase, each character in the data item referred to by identifier-1 is replaced by literal-1 or the data item referred to by identifier-2.

Literal-1 (or the data item referred to by identifier-2) and literal-2 (or the data item referred to by identifier-4) must be one character in length.

ALL
LEADING
FIRST

These apply to each literal-3 or identifier-3 that follows them, until the next ALL, LEADING, or FIRST phrase.

If you specify ALL, each occurrence of literal-3 that is matched in the content of the data item referred to by identifier-1 is replaced by literal-1.

If you specify LEADING, each contiguous occurrence of literal-3 that is matched in the content of the data item referred to by identifier-1 is replaced by literal-1, provided that the leftmost occurrence is at the point where the comparison began in the first comparison cycle in which literal-3 was eligible to participate (refer to "The Comparison Cycle" in the discussion of the Format 1 INSPECT statement).

If you specify FIRST, the leftmost occurrence of literal-3 that is matched within the content of the data item referred to by identifier-1 is replaced by literal-1.

Note: *In this discussion, any reference to literal-1 applies to identifier-2. Any reference to literal-2 applies to identifier-4.*

BEFORE
AFTER

You can specify only one BEFORE and one AFTER phrase for any one ALL, LEADING, FIRST, or CHARACTERS phrase. However, both BEFORE and AFTER can be used in the same INSPECT statement. See "The Comparison Cycle" in the discussion of the Format 1 INSPECT statement for an explanation of how the BEFORE and AFTER phrases operate.

Details

Inspection, which includes the comparison cycle, the establishment of boundaries for the BEFORE and AFTER phrases, and the mechanism for replacing, begins at the leftmost character position of the data item referred to by identifier-1, regardless of its class. Inspection proceeds from left to right to the rightmost character position.

If an identifier is subscripted or is a function-identifier, the subscript or function-identifier is evaluated once, as the first operation in the execution of the INSPECT statement.

Examples

```
INSPECT Word REPLACING CHARACTERS BY "B"BEFORE INITIAL "R".
```

If the data item Word contained the value ARXAX, the resulting value of Word after replacement would be BRXAX.

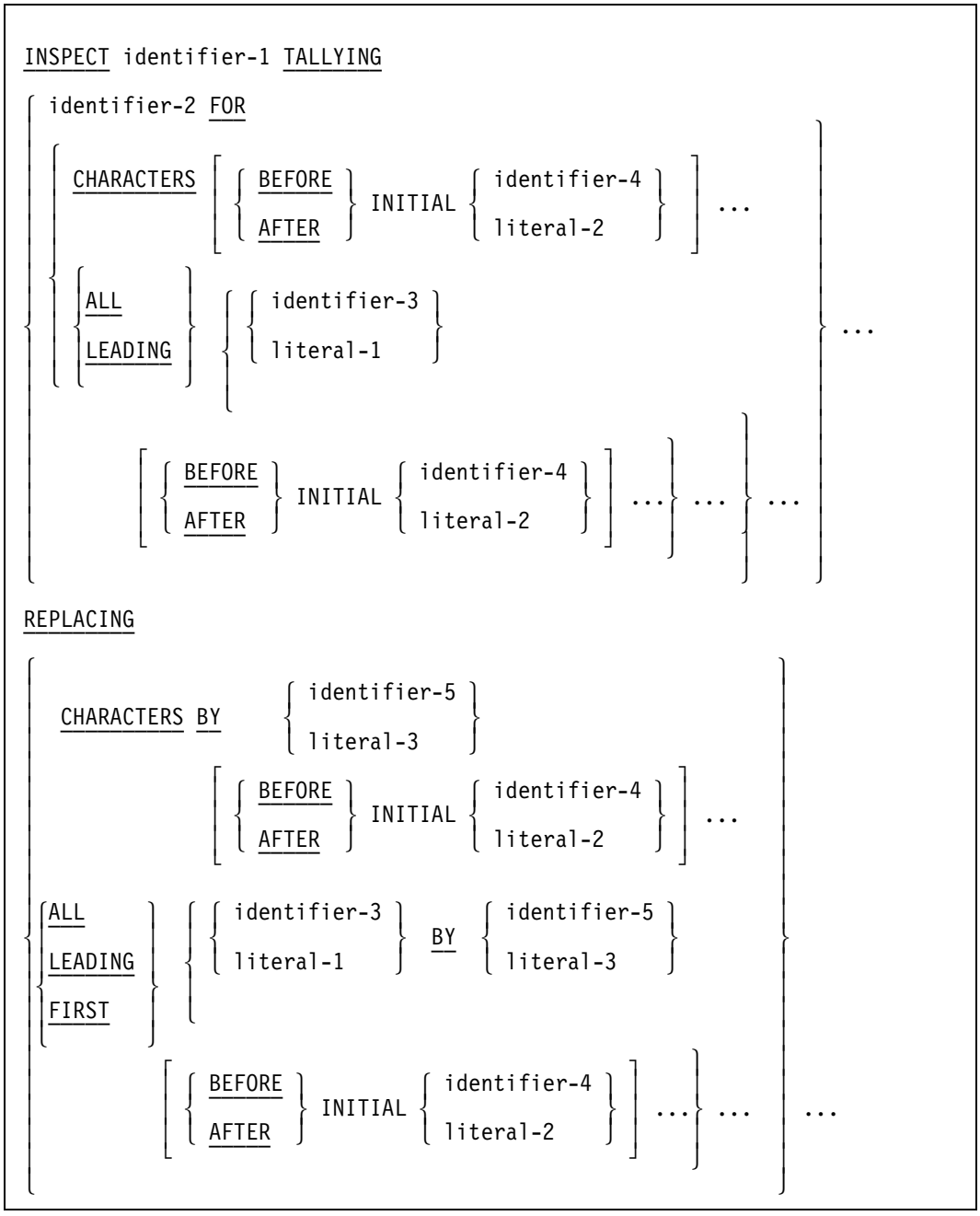
```
INSPECT Word REPLACING ALL "ABC"BY "XYZ".
```

If the data item Word contained the value BBEABCABABBCABEE, the resulting value of Word after replacement would be BBEXYZABABBCABEE.

```
INSPECT Word REPLACING
   ALL "AB"BY "XY",
       "D"BY  "X",
      "BC"BY "VW",
   LEADING "EF"BY "TU",
   FIRST "G"BY "R",
   FIRST "G"BY "P",
   CHARACTERS BY "Z".
```

If the data item Word contains the value EFABDBCGABEFGG, the resulting value of Word after replacement is TUXYXVWRXYZZPZ.

Format 3: INSPECT . . . TALLYING and REPLACING



Explanation

This format of the INSPECT statement is interpreted and executed as though two successive INSPECT statements specify the same identifier-1: one statement is a Format 1 statement (TALLYING), and the other statement is a Format 2 statement (REPLACING). Refer to the descriptions of Formats 1 and 2.

Subscripting associated with any identifier in the Format 2 part of this statement is evaluated only once before executing the Format 1 statement.

Refer to Formats 1 and 2 for descriptions of the syntax elements in Format 3.

Examples

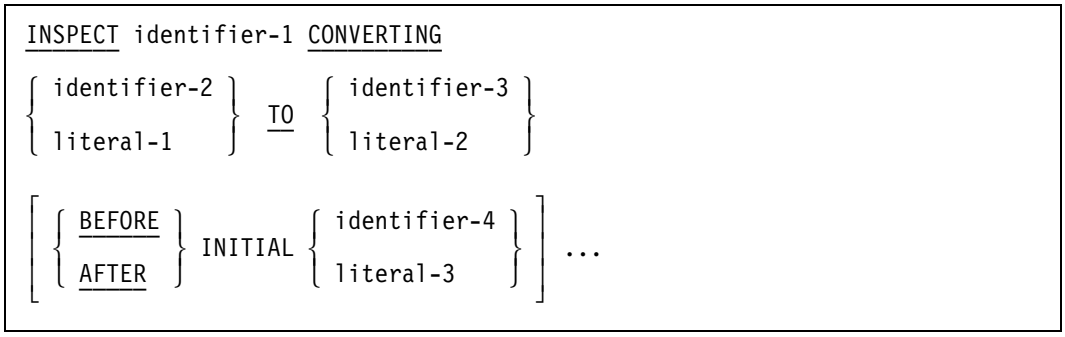
```
INSPECT Word TALLYING Count1 FOR ALL "L",
REPLACING ALL "A"BY "E"AFTER INITIAL "L".
```

This example of a Format 3 INSPECT statement is equivalent to the following two statements:

```
INSPECT Word TALLYING Count1 FOR ALL "L".

INSPECT Word REPLACING ALL "A"BY "E"AFTER INITIAL "L".
```

If the data item Word contained the value CALLAR, the data item Count1 would contain 2, and the value of Word after replacement would be CALLER.

Format 4: INSPECT. . . CONVERTING



Explanation

identifier-1

The same rules for an identifier-1 in a Format 1 or Format 2 statement apply to this identifier.

CONVERTING

With the CONVERTING phrase, you can replace single characters as if you had written a Format 2 (REPLACING) statement with a series of ALL phrases, one ALL phrase for each character in literal-1.

identifier-2
identifier-3

The size of identifier-2 must be equal to the size of identifier-3. The same character cannot appear more than once in the data item referred to by identifier-2.

When you use a figurative constant as literal-3, the data item referred to by identifier-2 (or literal-1) must be the same size as the figurative constant.

identifier-4

The same rules for an identifier-4 in a Format 1 or Format 2 statement apply to this identifier.

literal-1
literal-2
literal-3

The size of literal-1 must be equal to the size of literal-2. The same character cannot appear more than once in literal-1.

When you use a figurative constant as literal-3, literal-1 (or the data item referred to by identifier-2) must be the same size as the figurative constant.

Details

The first character in the data item referred to by identifier-2 (or literal-1) is replaced by the first character in the data item referred to by identifier-3 (or literal-2) wherever that character occurs in the data item referred to by identifier-1.

If identifier-2, identifier-3, or identifier-4 occupies the same storage space as identifier-1, the result of the execution of the INSPECT statement is undefined, even if the identifiers are defined by the same data description entry.

The BEFORE and AFTER phrases are discussed in "The Comparison Cycle" in the discussion of Format 1.

Subscripting associated with any identifier is evaluated only once, as the first operation in the execution of the INSPECT statement.

Example

The following two INSPECT statements are equivalent:

```
INSPECT Word CONVERTING  "ABCD"TO  "XYZX"AFTER QUOTE
BEFORE  "#".

INSPECT Item REPLACING
ALL  "A"BY  "X"AFTER QUOTE BEFORE  "#".
ALL  "B"BY  "Y"AFTER QUOTE BEFORE  "#".
ALL  "C"BY  "Z"AFTER QUOTE BEFORE  "#".
ALL  "D"BY  "X"AFTER QUOTE BEFORE  "#".
```

If the data item Word contained `AC"AEBDFBCD#AB"D`, the resulting value of data item after the INSPECT statement would be `AC"XEYXFYZX#AB"D`.

LOCK Statement

The LOCK statement enables a process to lock a common data storage area so that other related processes cannot access it.

```
LOCK  { event-identifier }          [ AT LOCKED { statement-1    } ]
      { lock-identifier  }          [           { NEXT SENTENCE } ]

[ END-LOCK ] .
```

Explanation

event-identifier
lock-identifier

The event-identifier can be one or more of the following:

- The name of a data-item declared with the USAGE IS EVENT phrase. The data-name must be properly qualified and properly subscripted.
- A task attribute of type EVENT. The two event task attributes are ACCEPTEVENT and EXCEPTIONEVENT. For details about these task attributes, refer to the *Task Attributes Programming Reference Manual*.
- A file attribute of type EVENT. The three event file attributes are CHANGEEVENT, INPUTEVENT, and OUTPUTEVENT. For details about these file attributes, refer to the *File Attributes Programming Reference Manual*.

The lock-identifier is the data-name of the storage area declared as a data item with the USAGE IS LOCK clause. (For details, see "USAGE Clause" in Section 4).

AT LOCKED statement-1
AT LOCKED NEXT SENTENCE

This syntax enables the process to test the storage area represented by the event or lock identifier to see if it is locked. If the area is locked when the LOCK statement is executed, control passes either to the statement specified with the AT LOCKED phrase or to the next sentence after the LOCK statement.

END-LOCK

If multiple LOCK statements are nested in the same block, you must use the END-LOCK phrase to signify the end of each LOCK statement.

Details

If you do not use the AT LOCKED phrase, the system continues to try to lock the storage area until it is successful. This might cause a time delay if the process has to wait until another process unlocks the storage area.

Example

```
LOCK WS-EVENT (3).

LOCK WS-77-EVENT AT LOCKED GO TO ERROR1.
```

LOCKRECORD Statement

The LOCKRECORD statement enables a process to lock a record in a file so that other processes cannot access it. The LOCKRECORD statement has the following format:

```
LOCKRECORD file-name

[ON EXCEPTION imperative-statement-1]

[NOT ON EXCEPTION imperative-statement-2]

[END-LOCKRECORD]
```

Explanation

file-name

This user-defined word is the name of the file that contains the record to be locked. You specify the record to be locked with the ACTUAL KEY clause in the File Control Entry of the Environment Division.

The file you specify for locking must be an open file that resides on the local host and has a KIND of DISK. In addition the file must have

- Sequential organization
- Random access mode
- The BUFFERSHARING file attribute value declared as SHARED or EXCLUSIVELYSHARED

ON EXCEPTION imperative-statement-1

This clause specifies an alternate statement to be performed if the LOCKRECORD statement is not successful.

NOT ON EXCEPTION

This clause specifies a statement to be performed after the record is successfully locked.

Details

The successful execution of the LOCKRECORD statement locks the record specified by the value contained in the data item referenced by the ACTUAL KEY clause in the File Control Entry of the Environment Division. The record remains locked until one of the following actions occurs:

- An UNLOCKRECORD statement is executed.
- The file is closed.
- The job is terminated.

If an existing locked record blocks the request, the system-wide default time-limit FILELOCKTLIMIT is used to time out the request. Deadlocks are not detected.

Failure of the LOCKRECORD Statement

The LOCKRECORD statement can fail for any of the following reasons:

- The specified file does not exist.
- The specified file does not support locking (see the requirements for the file described with the explanation of the file-name syntax).
- The specified file is not open.
- The specified file is not open for write operations.
- The specified record key has an invalid or inconsistent value.
- An existing locked record blocks the request and the resulting waiting period timed out.
- The number of locked records would exceed the system limit if the lock request were successfully executed.

Security and Integrity Issues

The BUFFERSHARING attribute and record locking protect a file during concurrent access only if all users of the file declare the file as shared (set the BUFFERSHARING attribute to a value other than NONE) and then lock the record before using it.

If you open a file without setting a value for BUFFERSHARING, the value defaults to NONE, for no sharing. Other users can open the same file with BUFFERSHARING set to SHARED. Likewise, a user can open a file declared as SHARED while the same physical file is not declared as SHARED. In both situations, the users who opened the file as SHARED are not protected from operations performed by users who do not declare the file as SHARED.

Locking a record prevents other users only from *locking* the same record. Other users can still read and write a locked record. If the MUSTLOCK compiler option is TRUE, the current user must lock a record before executing a WRITE statement.

The typical procedure for updating a record in a shared file is as follows:

1. Lock the desired record.
2. Read, modify, and write the record.
3. Unlock the record.

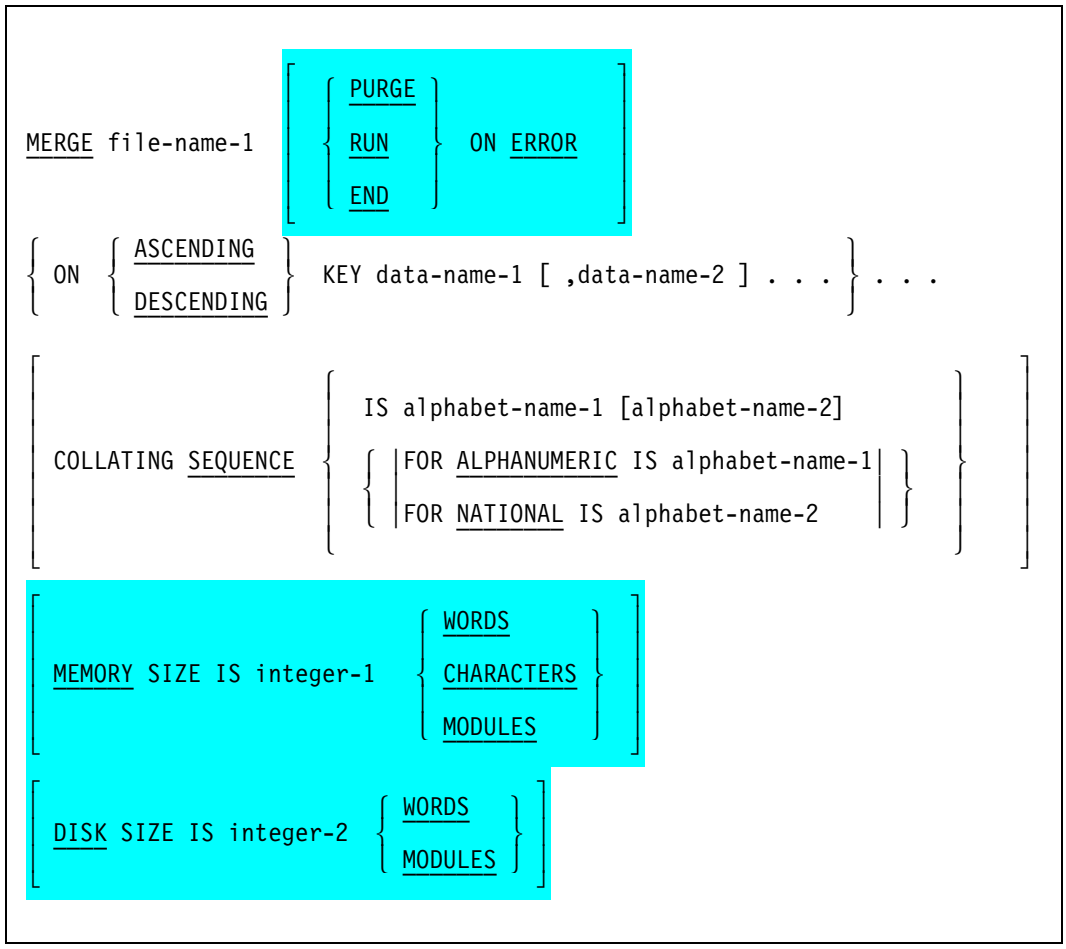
Related Information

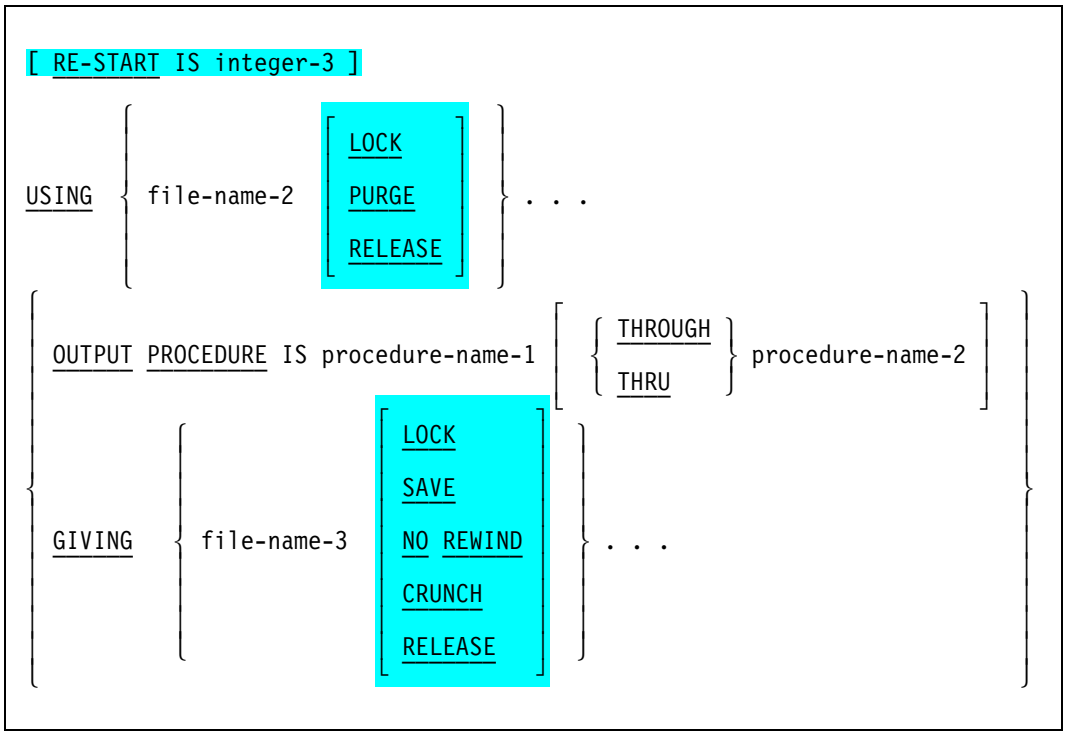
The following table provides references to information related to this topic:

For information about . . .	Refer to . . .
The BUFFERSHARING file attribute	The File Attributes Programming Reference Manual.
Unlocking a file	The UNLOCKRECORD statement.
I/O status codes resulting from error with locking and unlocking files	Table 3-6.

MERGE Statement

The MERGE statement combines two or more identically sequenced files on a set of specified keys. The merged records then become available, in merged order, to an output procedure or to an output file. A MERGE statement can appear anywhere in the Procedure Division, except in the declaratives portion. The syntax for the MERGE statement is displayed on the following two pages.




[RE-START IS integer-3]
USING
file-name-2
LOCK
PURGE
RELEASE
. . .
OUTPUT PROCEDURE IS procedure-name-1
THROUGH
THRU
procedure-name-2
GIVING
file-name-3
LOCK
SAVE
NO REWIND
CRUNCH
RELEASE
. . .

Explanation

file-name-1

This refers to the merge file, which is an internal file.

File-name-1 must be described in a sort-merge file description entry in the Data Division.

The size of the records contained in the file referred to by file-name-1 must not be larger than the largest record described for file-name-3.

No more than one file-name from a multiple-file reel can appear in the MERGE statement.

File-names cannot be repeated within the MERGE statement.

file-name-2
file-name-3

File-name-2, which may repeat, refers to the input file or files, which contain the records to be merged.

File-name-3 refers to the output file.

These file names must be described in a file description entry in the Data Division, not in a sort-merge file description entry.

The size of the records contained in the files referred to by file-name-2 cannot be larger than the largest record defined for file-name-1.

No more than one file-name from a multiple-file reel can appear in the MERGE statement.

File-names cannot be repeated within the MERGE statement.

No two files, except those in the GIVING clause, can be specified in the same SAME/RECORD/SORT/SORT-MERGE AREA clause.

If the records in the files referred to by file-name-2 are not ordered by an ASCENDING or DESCENDING KEY phrase, the results of the MERGE statement are undefined.

If the file referred to by file-name-3 is a relative file, the content of the relative KEY data item after execution of the MERGE statement will indicate the last record returned to the file.

data-name-1
data-name-2

These data-names are KEY data-names and are subject to the following rules:

- The data items identified by these KEY data-names must be described in records associated with file-name-1.
- KEY data-names can be qualified.
- The data items identified by these KEY data-names cannot be variable-length items or long numeric data items.

If file-name-1 has more than one record description, the data items identified by these KEY data-names can all be described within one of the record descriptions or in any combination of record descriptions. It is not necessary to redescribe the KEY data-names in each record description.

None of the data items identified by KEY data-names can be described by an entry that either contains an OCCURS clause or is subordinate to an entry which contains an OCCURS clause.

If file-name-3 references an index file, the first specification of data-name-1 must be associated with an ASCENDING phrase, and the data item referred to by that data-name-1 must occupy the same character positions in its record as the data item associated with the prime record key for that file.

The KEY data-names are listed from left to right in the MERGE statement in order of decreasing significance, without regard to how they are divided into KEY phrases; that is, data-name-1 is the major key, data-name-2 is the next most significant key, and so on.

When, according to the rules for the comparison of operands in a relation condition, the contents of all the KEY data items of one data record are equal to the contents of the corresponding KEY data items of one or more other data records, the order of return of these records follows the order of the associated input files as specified in the MERGE statement. Therefore, all records associated with one input file are returned before the return of records from another input file.

ON ERROR

The ON ERROR options enable you to have control over irrecoverable parity errors when input/output procedures are not present in a program.

PURGE causes all records in a block that contains an irrecoverable parity error to be dropped; processing is continued after a message displayed on the ODT gives the relative position of the bad block in the file.

RUN causes the bad block to be used by the program and provides the same message as defined for PURGE.

END causes a program termination; this is the default.

ASCENDING DESCENDING

The ASCENDING and DESCENDING phrases have the following effects:

- If you specify the ASCENDING phrase, the merged sequence will be from the lowest value of the contents of the data items identified by the KEY data-names to the highest value, according to the rules for comparison of operands in a relation condition.
- If you specify the DESCENDING phrase, the merged sequence will be from the highest value of the contents of the data items identified by the KEY data-names to the lowest value, according to the rules for comparison of operands in a relation condition.

COLLATING SEQUENCE

Alphabet-name-1 references an alphabet that defines an alphanumeric collating sequence.

Alphabet-name-2 references an alphabet that defines a national collating sequence.

The alphanumeric collating sequence that applies to the comparison of key data items for class alphabetic and class alphanumeric, and the national collating sequence that applies to the comparison of key data items of class national, are determined separately at the beginning of the execution of the MERGE statement in the following order of precedence:

1. The collating sequence is established by the COLLATING SEQUENCE phrase, if specified, in this MERGE statement.

   The collating sequence associated with alphabet-name-1 applies to key data items of class alphabetic and alphanumeric; the collating sequence associated with alphabet-name-2 applies to key data items of class national.

2. The collating sequences are established as the program collating sequences.

MEMORY SIZE IS integer-1

MEMORY SIZE is a guideline for allocating MERGE memory area, and it takes precedence over the same clause in the OBJECT-COMPUTER paragraph. It can be allocated as MODULES, WORDS, or CHARACTERS. If MEMORY SIZE is not specified, either in the OBJECT-COMPUTER paragraph or in the MERGE statement, a default value of 12,000 words is assumed.

DISK SIZE IS integer-2

DISK SIZE is a guideline for allocating MERGE disk area, and it takes precedence over the same clause in the OBJECT-COMPUTER paragraph. It can be allocated as WORDS or MODULES. If DISK SIZE is not specified, either in the OBJECT-COMPUTER paragraph or in the MERGE statement, a default value of 900,000 words is assumed. One module of disk is equivalent to 1.8 million words of disk.

RE-START IS integer-3

The RE-START specification enables the sort intrinsic to resume processing at the most recent checkpoint after discontinuation of a program during the merge. The program restores and maintains variables, files, and everything that is necessary for the program to continue from the point of interruption.

The restart capability is implemented only for disk merges and sorts.

Select the type of RE-START action to be performed by choosing one of the following values for integer-3:

0	No restart capability.
1	Restart previous sort. The prior uncompleted sort must have been capable of a restart.
2	Allow restartable sort.
4 or 6	Allow a restartable sort and enable extensive error recovery from I/O errors.
9	Restart previous sort if all input has been received. The prior uncompleted sort must have been capable of a restart.
10	Allow restartable sort after all input is received.
12 or 14	Options 4 and 10.

Refer to the MERGE section in the *System Software Utilities Operations Reference Manual* for more details on the RE-START capability of MERGE.

USING

You can specify up to eight file-names in the USING phrase.

OUTPUT PROCEDURE

Procedure-name-1 represents the name of an output procedure.

The OUTPUT PROCEDURE phrase must consist of one or more paragraphs or sections that appear in a source program and do not form a part of any other procedure.

To make merged records available for processing, the output procedure must include the execution of at least one RETURN statement. Control cannot be passed to the output procedure except when a related SORT or MERGE statement is being executed. The output procedure can consist of any procedures needed to select, modify, or copy the records that are being returned, one at a time in merged order, from file-name-1.

Restrictions

The restrictions on the procedural statements in the output procedure are as follows:

- The output procedure cannot contain any transfers of control to points outside the output procedure; ALTER, GO TO, and PERFORM statements in the output procedure are not permitted to refer to procedure-names outside the output procedure. Statements that cause an implied transfer of control to declaratives are allowed.
- The output procedures cannot contain any SORT, MERGE, or RELEASE statements.
- The remainder of the Procedure Division cannot contain any transfers of control to points inside the output procedures; ALTER, GO TO, and PERFORM statements in the remainder of the Procedure Division are not permitted to refer to procedure-names within the output procedures.

If you specify an output procedure, control passes to it during execution of the MERGE statement. The compiler inserts a return mechanism at the end of the last paragraph or section in the output procedure. When control passes the last statement in the output procedure, the return mechanism provides for termination of the merge, and then passes control to the next executable statement after the MERGE statement. Before entering the output procedure, the merge procedure reaches a point at which it can select the next record in merged order when requested. The RETURN statements in the output procedure are the requests for the next record.

During execution of the output procedure, no statement can be executed that manipulates the file referred to by or accesses the record area associated with file-name-2.

THRU
THROUGHThe keywords THRU and THROUGH are interchangeable.

GIVINGIf you specify the GIVING phrase, all the merged records are automatically written on file-name-3 as the implied output procedure for the MERGE statement. At the start of execution of the MERGE statement, the file referred to by file-name-3 cannot be in the open mode. You can specify up to eight file names in the GIVING phrase.

LOCK
PURGE
RELEASE
SAVE
NO REWIND
CRUNCH

These options enable you to specify the type of close procedure to use on a file.

You can specify the LOCK, PURGE, and RELEASE options for file-name-2 (input files specified by the USING phrase).

You can specify SAVE, LOCK, NO REWIND, CRUNCH, and RELEASE options for file-name-3 (output file specified by the GIVING phrase).

For a description of these options, refer to "CLOSE Statement" in this section.

Details

The MERGE statement will merge all records contained in file-name-2. The files referenced in the MERGE statement cannot be open at the time the MERGE statement is executed. These files are automatically opened and closed by the merge operation with all implicit functions performed, such as the execution of any associated USE procedures. The terminating function for each file is performed as if a CLOSE statement, without optional phrases, had been executed.

If a record in the file referred to by file-name-2 (the file to merge) has fewer character positions than the record length of the file referred to by file-name-1 (the base file), then the record from file-name-2 is space-filled on the right, beginning with the first character position after the last character in the record, when the record is released to the file referred to by file-name-1.

During the execution of any USE AFTER EXCEPTION procedure implicitly invoked by the MERGE statement, no statement can be executed that manipulates the file referenced by, or accesses the record area associated with, file-name-2 or file-name-3.

If the OUTPUT PROCEDURE clause is used, the GIVING clause cannot be used. If the GIVING clause is used, the OUTPUT PROCEDURE clause cannot be used.

Refer to "SAME Clause" under "Input-Output Control Entry Format 3: Sort-Merge" in Section 3.

Refer to "File Description Entry" in Section 4 for information on how to describe a merge file.

Refer to "Sort and Merge Operations" in Section 5 for conceptual information on sort and merge operations.

Refer to "CLOSE Statement" in Section 6 and "SORT Statement" in Section 8 for a description of close options and sort operations, respectively.

Example

```
MERGE File-abc ON ASCENDING KEY Name, Number1
   USING File-def, File-ghi, File-jkl
   OUTPUT PROCEDURE IS Routine-1 THRU Routine-8.
```

The MERGE statement in this example merges the three files File-def, File-ghi, File-jkl into File-abc using the output procedures Routine-1 through Routine-8. After execution of the output procedures, control of the program will pass to the next executable statement after this sentence.

For another example of the MERGE statement, refer to "Example" under "Sort and Merge Operations" in Section 5.

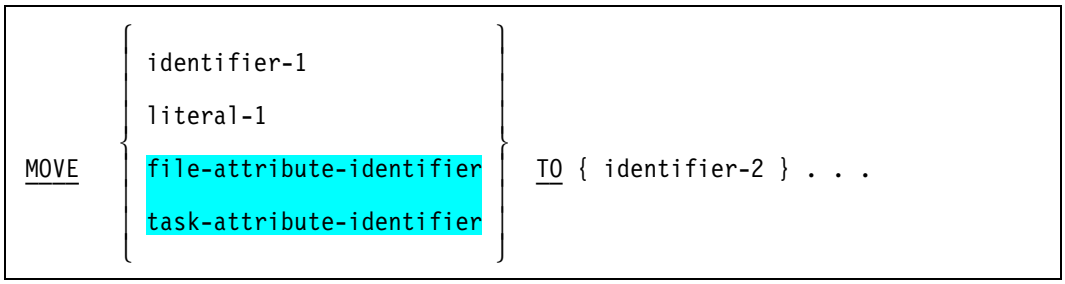
MOVE Statement

The MOVE statement transfers data from one data area to one or more data areas.

This statement is fully supported in the TADS environment.

Format	Use
Format 1	This format transfers data to one or more data areas.
Format 2	The MOVE CORRESPONDING format transfers selected items in identifier-1 to selected items in identifier-2. This format transfers items having the same name as one in the receiving field to that corresponding field.
Format 3	This format transfers selected bit ranges between two BINARY data items.

Format 1: MOVE Data



This format is supported in the TADS environment.

Explanation

In this format, the item before the word TO represents the *sending* area. The item after the word TO represents the *receiving* area.

identifier-1
literal-1
identifier-2

Literal-1 or the data item referred to by identifier-1 (the sending field) represents the data that is to be moved to the data item referred to by identifier-2 (the receiving field).

Literal-1 can be a long numeric literal.

Identifier-1 and identifier-2 can reference long numeric data items. If both identifiers reference long numeric data items, the data items must be of the same size and usage.

If identifier-2 references a long numeric data item, then literal-1 must be the same size.

You can move an appropriate figurative constant or the value 0 (zero) to a long numeric data item.

file-attribute-identifier
task-attribute-identifier

These identifiers represent the attribute value that you want to move to the data area defined by identifier-2. You can then use the data item in a number of Procedure Division statements to monitor or query the attribute value. For more information about file attributes, refer to Section 10 of this manual and the *File Attributes Programming Reference Manual*. For details about task attributes, refer to Section 11 of this manual and the *Task Attributes Programming Reference Manual*.

Details

The MOVE statement transfers data according to the rules of editing as described under the heading "Editing Rules" under "PICTURE Clause" in Section 4.

How the MOVE Statement Is Evaluated

If identifier-1 has a subscript or a reference modifier or is a function identifier, the subscript, reference modifier, or function-identifier is evaluated only once. The evaluation occurs immediately before data is moved to the first of the receiving operands. The rules that apply to identifier-2 apply to the other receiving areas as well.

An indexed data item must not appear as an operand of a MOVE statement.

Any length or subscripting associated with identifier-2 is evaluated immediately before the data is moved to the respective data item.

The evaluation of the length of identifier-1 or identifier-2 can be affected by the DEPENDING ON phrase of the OCCURS clause. Refer to "OCCURS Clause" in Section 4 for more information.

The following MOVE statement yields the same result as the three subsequent MOVE statements:

```
MOVE a (b) TO b, c (b)

MOVE a (b) TO temp
MOVE temp TO b
MOVE temp TO c (b)
```

In this case, temp is an intermediate data item provided by the compiler. Refer to "Intermediate Data Item" in Section 5 for more information.

Categories of Elementary Data Items

An elementary move is any move in which the receiving operand is an elementary item and the sending operand is either a literal or an elementary item.

Elementary items must belong to one of the following categories:

- Alphabetic
- Alphanumeric
- Alphanumeric-edited
- Boolean
- National
- National-edited
- Numeric
- Numeric-edited

Refer to "PICTURE Clause" in Section 4 for explanations of each category.

Table 7–1 shows the categories in which literals, figurative constants, and intrinsic functions belong.

Table 7–1. Categories of Elementary Data Items

Elementary Items	Category
Numeric literals	Numeric
Nonnumeric literals	Alphanumeric
National literals	National
Boolean literals	Boolean
Figurative constant ZERO (ZEROS, ZEROES)	Numeric, when moved to a numeric or a numeric-edited item National, when moved to a national or national-edited item Alphanumeric, in all other cases
Figurative constant SPACE (SPACES)	National, when moved to a national or national-edited item Alphabetic, in all other cases
Figurative constants when moved to a national or national-edited item	National
Figurative constants in general	Alphanumeric
Intrinsic Functions	Alphanumeric or numeric, depending upon the definition of the function (see Section 7 for details)

Valid MOVE Actions

Table 7–2 summarizes valid MOVE actions between categories of data items.

Table 7–2. Valid MOVE Actions

Category of Sending Data Item	Category of Receiving Data Item				
	Alphabetic	Alphanumeric or Alphanumeric -Edited	Boolean	National or National-edited	Numeric Integer or Numeric Noninteger or Numeric-Edited
Alphabetic	Yes	Yes	No	No	No
Alphanumeric	Yes	Yes	Yes	No	Yes
Alphanumeric-Edited	Yes	Yes	No	No	No
Boolean	No	Yes	Yes	No	No
National	No	No	Yes	Yes	No
National-edited	No	No	No	Yes	No
Numeric Integer	No	Yes	No	No	Yes
Numeric Noninteger	No	No	No	No	Yes
Numeric-Edited	No	Yes	No	No	Yes

Note: *If the CCSVERSION clause is specified in the program, the national or national-edited data items are represented internally as contiguous 8-bit characters in the national character set. In this case, MOVE operations between the following categories are permitted: alphabetic, alphanumeric, alphanumeric-edited, national, and national-edited.*

Alphanumeric and Alphanumeric-Edited Moves

When the receiving item is alphanumeric-edited or alphanumeric, alignment and any necessary space-filling takes place according to the rules presented under "Standard Alignment Rules" in this section.

If the usage of the sending operand is different from that of the receiving operand, conversion of the sending operand to the internal representation of the receiving operand takes place.

In addition, note the following results of specific MOVE operations when the receiving field is alphanumeric or alphanumeric-edited:

When the receiving operand is alphanumeric or alphanumeric-edited, and the sending operand is . . .	Then . . .
Signed numeric	The operational sign is not moved. If the operational sign occupies a separate character position, the character is not moved. In this case, the size of the sending operand is considered to be one less than its actual size in terms of standard data format characters. Refer to "SIGN Clause" in Section 4 for more information.
Numeric-edited	De-editing does not take place.
Numeric	All digit positions specified with the PICTURE symbol P have the value of zero and are counted in determining the size of the sending operand.

National and National-Edited Moves

If the receiving item is national or national-edited, the following rules apply:

- Alignment and any necessary space-filling takes place according to the standard alignment rules. A discussion of the standard alignment rules is included at the end of the discussion of the MOVE statement.
- The sending operand must be described as national or national-edited.

Numeric and Numeric-Edited Moves

When the receiving item is numeric or numeric-edited alignment is by decimal point. Any necessary zero-filling takes place according to the standard alignment rules, except where zeros are replaced because of editing requirements. For more information, refer to the paragraphs headed "Standard Alignment Rules" under "Format 2: MOVE CORRESPONDING" in this section.

When the receiving item is signed numeric, the sign of the sending operand is placed in the receiving item. Conversion of the representation of the sign takes place as necessary. Refer to "SIGN Clause" in Section 4 for more information.

If the receiving item is unsigned numeric, the absolute value of the sending operand is moved, and no operational sign is generated for the receiving item.

When the sending operand is REAL or DOUBLE, and the receiving item is DISPLAY, COMP, or BINARY, precision can be lost if the sending operand represents a value that the machine must approximate. For more information, refer to USAGE IS REAL and USAGE IS DOUBLE of Data Description Entry Format 1 in Section 4.

In addition, note the following results of specific MOVE operations when the receiving field is numeric or numeric-edited:

When the receiving operand is numeric or numeric-edited, and the sending operand is . . .	Then . . .
Numeric-edited	De-editing is applied to establish the unedited numeric value of the operand (which can be signed). The unedited numeric value is moved to the receiving field.
Unsigned	A positive sign is generated for the receiving item.
Alphanumeric	Data is moved as if the sending operand were described as an unsigned numeric integer. If the alphanumeric item contains characters other than the digits 0 through 9, the result in the receiving field is unpredictable. When the figurative constants HIGH-VALUE, LOW-VALUE, or ALL "literal" are moved to data items of usage COMP or DISPLAY and the COMPATIBILITY suboption FIGCONST is set, the specified values are copied so that the entire field is filled into the data area, including any decimal places. No conversion of the data to valid numeric values occurs. Moves to COMPUTATIONAL fields strip the zone before the move operation occurs. A move of a HIGH-VALUES or LOW-VALUES figurative constant to an item defined as usage COMP will result in a syntax error when the $COMPATIBILITY sub-option FIGCONST is reset. A move of a HIGH-VALUE, LOW-VALUE, QUOTE, nonnumeric literal, ALL nonnumeric literal, symbolic character, or ALL symbolic character to an item defined as usage CONTROL-POINT, DOUBLE, EVENT, INDEX, LOCK, REAL, or TASK results in a syntax error.

Alphabetic Moves

If the receiving item is alphabetic, justification and any necessary space filling takes place according to the standard alignment rules, which are described near the end of the discussion of the MOVE statement.

Unless other data-item categories are specified in the MOVE statement, all moves are treated as alphanumeric-to-alphanumeric elementary moves. However, data is not converted from one internal representation to another. In such a move, the receiving area will be filled without consideration for the individual elementary or group items contained within either the sending or receiving area. Refer to "OCCURS Clause" in Section 4 for a description of the exceptions.

Examples

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PROCESS-TASK-CALLED.
DATA DIVISION.
WORKING-STORAGE SECTION.
77  MY-NAME                                PIC X(45).
PROCEDURE DIVISION.
MAIN SECTION.
MAIN-PARA.
    MOVE ATTRIBUTE NAME OF MYSELF TO MY-NAME.
    DISPLAY MY-NAME.
    STOP RUN.
```

The preceding example shows how to use the MOVE and DISPLAY statements to display the current value of the NAME task attribute for the process associated with the system-declared task variable, MYSELF. (MYSELF refers to the process itself.)

```
01 A PIC X(30).
01 B PIC $99.99.
MOVE B TO A.
```

Because the receiving data item is alphanumeric, the value of B is left-justified in the receiving character positions, with space-fill to the right. Because the sending field is numeric-edited, de-editing does not occur.

The following is an example of a MOVE statement that transfers data into a numeric-edited receiving item:

```
01 A PIC $99.99.
01 B PIC 99V99.
MOVE B TO A.
```

Because the receiving data item is numeric-edited, the value of B is aligned by decimal point in the receiving character positions, with zero-fill or truncation at either end, as required.

The following is an example of a MOVE statement with de-editing:

```
01 A PIC $99.99.
01 B PIC 99V99.
MOVE A TO B.
```

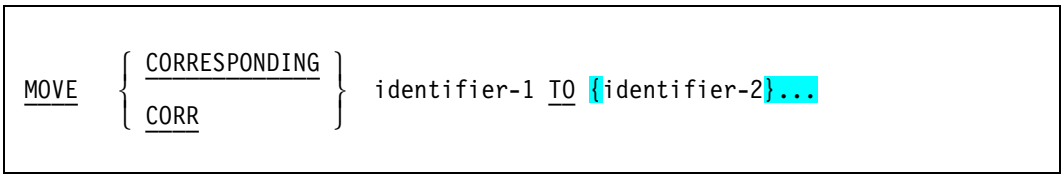
If A contains the value $19.87, B will contain the value 1987 after the MOVE.

The following is an example of a MOVE statement with the DEPENDING ON phrase in the OCCURS clause:

```
77 D      PIC 99
01 GRP.
   02 TBL PIC X(10) OCCURS 10 TO 20 TIMES
                DEPENDING ON D.
01 GRP-1  PIC X(300).
MOVE GRP TO GRP-1.
```

The result of the MOVE statement depends on the value of D.

Format 2: MOVE CORRESPONDING



This format is supported in the TADS environment.

Explanation

CORRESPONDING
CORR

When you specify MOVE CORR, the results are the same as if you had referred to each pair of corresponding identifiers in separate MOVE statements. CORRESPONDING and CORR are synonymous.

The paragraphs headed "CORRESPONDING Phrase" in this section explain the rules that govern a Format 2 MOVE statement.

identifier-1
identifier-2

All identifiers used with the CORRESPONDING phrase must be group items.

Details

CORRESPONDING Phrase

If the CORRESPONDING phrase is used, selected items in identifier-1 are moved to selected items in identifier-2, according to the rules in the following paragraphs. In these paragraphs, the terms D1 and D2 represent identifiers that refer to group items. A pair of data items, one from D1 and one from D2, correspond if the following conditions exist:

- Corresponding data items in D1 and D2 are not designated by the keyword FILLER, and have the same data-name and the same qualifiers up to, but not including, D1 and D2.
- At least one of the data items is an elementary data item, and the resulting move is valid according to the rules for the MOVE statement.
- The description of D1 and D2 does not contain level-number 66, 77, or 88, or the USAGE IS INDEX clause.
- A data item is ignored if it is subordinate to D1 or D2 and contains a REDEFINES, RENAMES, OCCURS, or USAGE IS INDEX clause. Also ignored are those data items subordinate to the data item that contains the REDEFINES, OCCURS, or USAGE IS INDEX clause.
- D1 and D2 cannot be reference modified.
- The name of each data item that satisfies the conditions in the previous paragraphs must be unique after application of the implied qualifiers.

Standard Alignment Rules

The standard rules for positioning data in an elementary item depend on the category of the receiving item. The following table describes how sending data is aligned after it is moved to the receiving data field. Note that these rules are modified if the JUSTIFIED clause is specified for the receiving item. (Refer to the paragraphs headed "JUSTIFIED Clause" under "Data Description Entry Format 1" in Section 4 for more information.)

If the receiving data item is . . .	Then the moved data is aligned . . .
Numeric	By decimal point. Zero-fill or truncation occurs on either end, as required. When an assumed decimal point is not explicitly specified, the data item is treated as if it has an assumed decimal point immediately following its rightmost digit and is aligned as stated in the preceding phrase.
Numeric-edited	By decimal point Zero-fill or truncation occurs at either end, as required. This rule is true except where editing requirements cause replacement of the leading zeros.

If the receiving data item is . . .	Then the moved data is aligned . . .
Alphanumeric (other than a numeric-edited data item), alphanumeric-edited, or alphabetic	At the leftmost character position in the data item. Space-fill or truncation to the right can occur, as required
National or national-edited	At the leftmost character position in the data item. Space-fill with national space characters or truncation to the right can occur as required.
Valid MOVE actions between categories of data items	Table 7–2

Related Information

The following table provides references for more information related to this statement:

For information about . . .	Refer to . . .
The alignment of receiving items	"JUSTIFIED Clause" in Section 4
Declaring long numeric data items	"PICTURE Clause" in Section 4.
Filling the receiving area in a MOVE action	"OCCURS Clause" in Section 4
The categories of elementary data items	"PICTURE Clause" in Section 4
Signed data items	"SIGN Clause" in Section 4
Intermediate data items	"Intermediate Data Item" in Section 5

Example

```
03 GRP-1                03 GRP-2
   05 A                    05 Z
   05 C                    05 A
   05 D                    05 D
   05 X                    05 E
                           05 X

PROCEDURE DIVISION.
     MOVE CORR GRP-1 TO GRP-2
```

In this example, items A, D, and X from GRP-1 are moved to the corresponding (A, D, and X) items in GRP-2. Each data item moved must be an elementary item at the same level.

Format 3: MOVE Selected Bits

```
MOVE  identifier-1  TO  identifier-2

 [  { literal-1                  }
    { arithmetic-expression-1    }

    : { literal-2                } : { literal-3                } ]
      { arithmetic-expression-2  }   { arithmetic-expression-3  }
```

This format is supported in the TADS environment.

Explanation

identifier-1
identifier-2

The data item referred to by identifier-1 represents the sending area. The data item referred to by identifier-2 represents the receiving area. Both data items must be single-precision BINARY data items (declared as USAGE IS BINARY) or single-precision REAL data items (declared as REAL). Both data items must have a size of 11 digits or less).

literal-1
arithmetic-expression-1

This represents the location in identifier-1 at which the transfer begins. This is referred to as the source bit location.

literal-2
arithmetic-expression-2

This represents the location in identifier-2 at which the transfer begins. This is referred to as the destination bit location.

literal-3
arithmetic-expression-3

This represents the number of bits to be transferred.

Details

Starting with the bit value in the source bit position, data is moved from identifier-1 to the destination bit position of identifier-2. Succeeding bits are transferred until the number of bits specified have been transferred.

The bit positions in a single-precision BINARY data item are numbered from left to right, with the leftmost bit position assigned the number 47, and the rightmost bit position assigned the number 0. Therefore, only values ranging from 0 to 47 are valid for source and destination bit positions.

Examples

```
MOVE A-AND-B-BOTH TO A-ONLY [39:19:20].
MOVE A-AND-B-BOTH TO B-ONLY [19:19:20].
```

These examples unpack a BINARY data item that contains two 20-bit fields.

```
MOVE B-ONLY TO A-AND-B-BOTH.
MOVE A-ONLY TO A-AND-B-BOTH [19:39:20].
```

These examples repack the fields unpacked in the previous example.

MULTIPLY Statement

The MULTIPLY statement multiplies numeric data items and stores the result.

The composite length of the operands in successive MULTIPLY operations is based on a hypothetical data item resulting from the superimposition of all receiving data items of a given statement on their decimal points. This length cannot exceed 23 decimal digits.

This statement is partially supported in the TADS environment. Supported syntax is noted in this section.

The MULTIPLY statement has two formats:

Format	Use
Format 1	This format multiplies elementary numeric items.
Format 2	This format multiplies elementary numeric items. The operands of the GIVING phrase must be either elementary numeric items or numeric-edited items.

Format 1: MULTIPLY

```
MULTIPLY   { identifier-1 }  BY  { identifier-2  [ ROUNDED ] } . . .
           { literal-1    }

    [ ON SIZE ERROR imperative-statement-1 ]

    [ NOT ON SIZE ERROR imperative-statement-2 ]

    [ END-MULTIPLY ]
```

TADS Syntax

```
MULTIPLY   { identifier-1 }  BY  { identifier-2  [ ROUNDED ] } . . .
           { literal-1    }

    [ END-MULTIPLY ]
```

Explanation

identifier-1
identifier-2
literal-1

In this format, each identifier must refer to an elementary numeric item. Each literal must be a numeric literal.

ROUNDED

The word ROUNDED causes the value of the result to be rounded. Refer to "ROUNDED Phrase" in Section 5 for more information.

ON SIZE ERROR imperative-statement-1
NOT ON SIZE ERROR imperative-statement-2

The options ON SIZE ERROR and NOT ON SIZE ERROR enable you to specify an action to be taken if an error in the size of the result is or is not encountered. Refer to "SIZE ERROR Phrase" in Section 5 for more information.

END-MULTIPLY

This phrase delimits the scope of the MULTIPLY statement.

Details

The value of the operand that precedes the word BY is stored in a temporary data item. The value of this data item is multiplied by the value of identifier-2. The product of the multiplication replaces the value of identifier-2. The temporary data item is multiplied by each successive occurrence of identifier-2 in the left-to-right order in which identifier-2 is specified.

Example

```
MULTIPLY A BY B ON SIZE ERROR PERFORM ERROR-PARA.
```

In this example, the value of A is multiplied by the value of B. If the result creates an ON SIZE ERROR, the compiler will execute the ERROR-PARA and the value of B remains unchanged.

Format 2: MULTIPLY . . . GIVING

```
MULTIPLY   { identifier-1 }   BY   { identifier-2 }
           { literal-1    }        { literal-2    }

GIVING { identifier-3 [ ROUNDED ] } . . .

    [ ON SIZE ERROR imperative-statement-1 ]

    [ NOT ON SIZE ERROR imperative-statement-2 ]

    [ END-MULTIPLY ]
```

TADS Syntax

```
MULTIPLY   { identifier-1 }   BY   { identifier-2 }
           { literal-1    }        { literal-2    }

GIVING { identifier-3 [ ROUNDED ] } . . .

    [ END-MULTIPLY ]
```

Explanation

Refer to Format 1 for descriptions of the following phrases: ROUNDED, ON SIZE ERROR, NOT ON SIZE ERROR, and END-MULTIPLY.

identifier-1
identifier-2
identifier-3
literal-1

Each identifier preceding the word GIVING must refer to an elementary numeric item. Each identifier following the word GIVING must refer to either an elementary numeric item or an elementary numeric-edited item. Each literal must be a numeric literal.

GIVING

Using the GIVING phrase in this format enables you to multiply data items and to store the result in a data item referred to by identifier-3.

Details

The values of the operands that precede the word GIVING are multiplied. The result is stored in the data items referred to by each identifier-3.

Related Information

The following table provides references for additional information related to this statement:

For information about . . .	Refer to . . .
The ROUNDED phrase	Section 5
The ON SIZE ERROR and NOT ON SIZE ERROR options	The SIZE ERROR Phrase in Section 5
The MULTIPLY statement and rules regarding this statement	The following headings in Section 5: "Arithmetic Expressions" "Allowed Combinations of Elements" "General Rules for Arithmetic Statements" "Multiple Results in Arithmetic Statements" "Conditional Statements and Sentences" "Statement Scope Terminators" "Delimited Scope Statements"

Examples

```
MULTIPLY A BY B GIVING C ROUNDED.
```

In this example, the value of A is multiplied by the value of B and the result is stored in C. The value of C is rounded.

Results of COBOL ANSI-74 and COBOL ANSI-85 are the same with overlapping fields and the same data description. The following examples show COBOL ANSI-74 and COBOL ANSI-85 programs with overlapping operands.

COBOL ANSI-74 Program

```
IDENTIFICATION DIVISION.
PROGRAM-ID. LEESTEST.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-MISC-VALUES.
   03  C PIC 9(03).
   03  CB REDEFINES C.
       05 FILLER PIC X(01).
       05 D      PIC 9(02).
   03  A PIC 9(03) VALUE 000.
   03  B REDEFINES A
         PIC 9(03).
PROCEDURE DIVISION.
MOVE-IT.
   MOVE 5 TO D, A.
   MOVE 4 TO C.
   MULTIPLY C BY D GIVING C.
   DISPLAY C.
   MULTIPLY A BY B GIVING A.
   DISPLAY A.
   STOP RUN.
```

Results

```
RUN
#RUNNING 7203
#7203 DISPLAY:016   (C)
#7203 DISPLAY:025   (A)
#ET
```

COBOL ANSI-85 Program

```
IDENTIFICATION DIVISION.
PROGRAM-ID. LEESTEST.

DATA DIVISION.
WORKING-STORAGE SECTION.
01 WS-MISC-VALUES.
   03  C PIC 9(03).
   03  CB REDEFINES C.
       05 FILLER PIC X(01).
       05 D      PIC 9(02).
   03  A PIC 9(03) VALUE 000.
   03  B REDEFINES A
         PIC 9(03).
PROCEDURE DIVISION.
MOVE-IT.
   MOVE 4 TO D, B.
   MOVE 5 TO C.
   MULTIPLY C BY D GIVING D.
   DISPLAY D.
   MULTIPLY A BY B GIVING B.
   DISPLAY B.
   STOP RUN.
```

Results

```
RUN
#RUNNING 7217
#7217 DISPLAY:25   (D)
#7217 DISPLAY:016  (B)
#ET
```

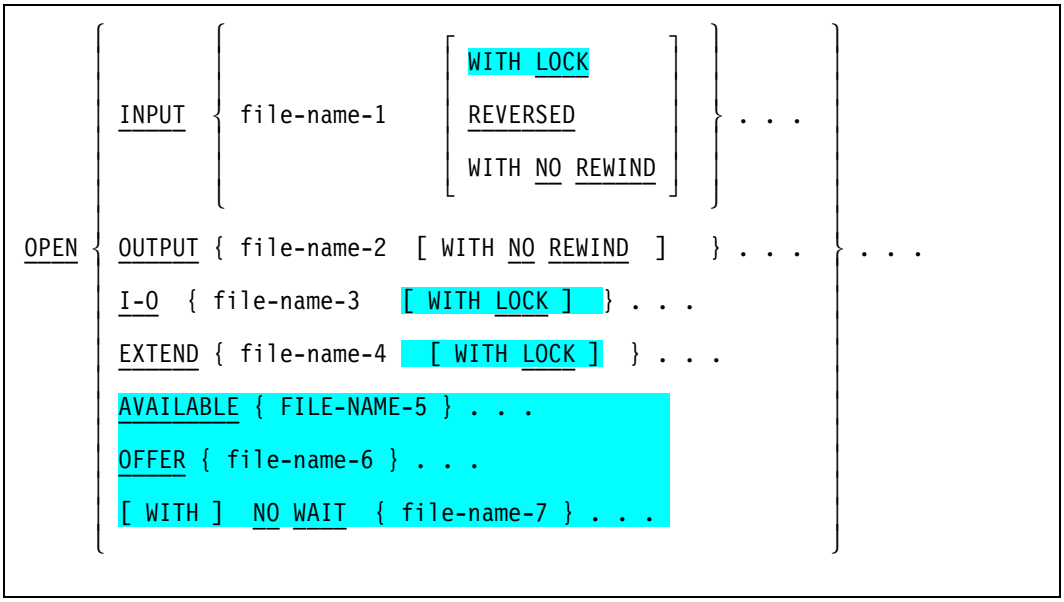
OPEN Statement

The OPEN statement initiates the processing of files. It also performs checking and/or writing of labels and other input-output operations.

There are three types of file organization in COBOL: sequential, relative, and indexed. There are three types of file access in COBOL: sequential, random, and dynamic. The files referenced in the OPEN statement need not all have the same organization or access. If you specify more than one file-name in an OPEN statement, the result is the same as if you had specified multiple OPEN statements.

For information on file attributes, file organization, and file access modes, refer to Section 10. Also refer to "CLOSE Statement" in Section 6 and "MERGE Statement" and "READ Statement" in this section.

This statement is partially supported in the TADS environment. Applicable exclusions are noted in this section.



This format is supported in the TADS environment.

Explanation

The REVERSED, NO REWIND, and EXTEND options apply only to sequential files.

file-name-1
file-name-2
file-name-3
file-name-4

The file description entry for files must be equivalent to that used when the file was created.

If you specify more than one file-name in an OPEN statement, the result is the same as if you had written separate OPEN statements for each file.

The minimum and maximum record sizes for a file are established at the time the file is created and cannot be subsequently changed.

file-name-5
file-name-6
file-name-7

These file-names must be names of port files. File-name-1 through file-name-4 cannot be names of port files.

WITH LOCK

This phrase applies to mass-storage files only and is ignored if applied to other types of files.

OPEN WITH LOCK on a mass-storage file denies the use of that file to all other programs in the mix. When you execute an OPEN WITH LOCK, the following occurs:

- If the specified file is already in the open mode, the program is suspended, and waits for exclusive availability of the file.
- If the specified file is not currently in an open mode, the file is opened.

NO REWIND
REVERSED

These options can be used only with the following:

- Sequential reel/unit files with a single reel/unit
- Sequential files that are wholly contained in a single reel of tape within a multiple-file tape environment

These phrases will be ignored if they do not apply to the storage medium on which the file resides.

If the medium on which the file resides permits rewinding, the following rules apply:

- If you do not specify REVERSED, EXTEND, or NO REWIND, execution of the OPEN statement causes the file to be positioned at its beginning.
- If you specify NO REWIND, execution of the OPEN statement does not cause the file to be repositioned; that is, the file must already be positioned at the beginning before the execution of the OPEN statement.
- If you specify REVERSED, execution of the OPEN statement positions the file at its end.

If you specify REVERSED, the last record of the file is the first available record.

INPUT

For sequential or relative files being opened with the INPUT phrase, the OPEN statement sets the current record pointer to the first record currently existing within the file. If the file does not contain any records, the current record pointer is set such that the next executed READ statement for the file will result in an AT END condition.

When you open a sequential or relative file with the INPUT phrase, the file position indicator is set to 1.

When you open an indexed file with the INPUT phrase, the file position indicator is set to the characters that have the lowest ordinal position in the collating sequence associated with the file, and the prime record key is established as the key of reference.

If you open an optional file with the INPUT phrase and the file is unavailable, the OPEN statement sets the file position indicator to indicate that an optional input file is not present.

OUTPUT

Upon successful execution of an OPEN statement with the OUTPUT phrase specified, a file is created. At that time, the associated file does not contain any data records.

When you open a sequential file with the OUTPUT option, all files on the associated multiple-file reel whose position numbers are less than the position number of the file being opened must already exist on the reel. In addition, there cannot be a file with a position number greater than the position number of the file being opened.**I-O**

For files being opened with the I-O phrase, the OPEN statement sets the current record pointer to the first record currently existing within the file. If the file does not contain any records, the current record pointer is set such that the next executed READ statement for the file will result in an AT END condition.

The OPEN statement with the I-O phrase must reference a file that supports the input and output operations allowed for that file organization (sequential, relative, or indexed). The execution of the OPEN statement with the I-O phrase places the referenced file in the open mode for both input and output.

Multiple files located on disk pack or tape are allowed for sequential I-O.

The I-O phrase can be used only for mass-storage files and port files. The I-O phrase permits the opening of a mass-storage file for both input and output operations. Because this phrase implies the existence of the file, it cannot be used if the mass-storage file is being initially created.

When you open a sequential or relative file with the INPUT phrase, the file position indicator is set to 1.

When you open an indexed file with the I-O phrase, the file position indicator is set to the characters that have the lowest ordinal position in the collating sequence associated with the file, and the prime record key is established as the key of reference.

The execution of the OPEN statement causes the value of the I-O status associated with the file-name to be updated.

For an optional file that is unavailable, the successful execution of an OPEN statement with an I-O phrase creates the file as if the following statements had been executed:

```
OPEN OUTPUT file-name.
CLOSE file-name.
```

EXTEND

This option enables you to write additional records to the end of a sequential file.

The EXTEND option can be used only with the following:

- Sequential reel/unit files with one reel/unit
- Files for which the LINAGE clause has not been specified

This option requires file-name-4 to be a previously created file (that is, already in the disk or pack directory, or on tape).

When you specify the EXTEND option, execution of the OPEN statement positions the file immediately after the last logical record for that file (that is, the last record written in the file). Subsequent WRITE statements that reference the file will add records to the file as though the file had been opened with the OUTPUT phrase.

For an optional file that is unavailable, the successful execution of an OPEN statement with an EXTEND phrase creates the file as if the following statements had been executed:

```
OPEN OUTPUT file-name.
CLOSE file-name.
```

AVAILABLE

When the available phrase is specified for nonport files and the file cannot be opened, the system reports the reason for the failure without suspending the program or requiring operator intervention. For more information, refer to the discussion of the AVAILABLE file attribute in the *File Attributes Programming Reference Manual* .

For information about using the AVAILABLE phrase with port files, see the paragraphs on port files under the following "Details" heading.

OFFER

The OFFER phrase can be specified for port files only. See the paragraphs on port files under the following "Details" heading.

WITH NO WAIT

The WITH NO WAIT phrase can be specified for port files only.

Details

The successful execution of an OPEN statement determines the availability of the file and results in the file being in an open mode. The successful execution of an OPEN statement associates the file with the file-name. Table 7–3 shows the result of an OPEN statement on available and unavailable files.

The execution of an OPEN statement does not affect either the content or availability of the file's record area. Execution of the OPEN statement does not obtain or release the first data record.

When a given file is not in an open mode, statements that reference the file, either explicitly or implicitly, cannot be executed, except for an OPEN statement, or for a MERGE or SORT statement with the USING or GIVING phrases.

Table 7–3. Result of OPEN Statement

File Disposition	File Available	File Unavailable
INPUT	Normal open	Open is unsuccessful.
INPUT (optional file)	Normal open	Normal open; the first read causes an AT END or INVALID KEY condition.
I-O	Normal open	Open is unsuccessful.
I-O (optional file)	Normal open	Open causes the file to be created.
OUTPUT	Normal open; the file contains no records	Open causes the file to be created.

Table 7–3. Result of OPEN Statement

File Disposition	File Available	File Unavailable
EXTEND	Normal open	Open is unsuccessful.
EXTEND (optional file)	Normal open	Open causes the file to be created.

An OPEN statement must be successfully executed before the execution of any of the permissible input-output statements. In Tables 7–4 and 7–5, an X indicates that the specified statement, used in the access mode shown in the leftmost column, can be used with the file organization and open mode shown at the top of the column.

A file can be opened with the INPUT, OUTPUT, I-O, and EXTEND phrases in the same program. Following the initial execution of an OPEN statement, each subsequent OPEN statement execution for the same file must be preceded by the execution of a CLOSE statement without a REEL, UNIT, or LOCK phrase.

During execution of an OPEN statement, file attribute conflicts result in an unsuccessful open operation on the file.

Treatment of a sequential file contained in a multiple-file tape environment is logically equivalent to the treatment of a sequential file contained in a single-file tape environment. Whenever a set of sequential files resides on a multiple-file reel, and one file of the set is referenced in an OPEN statement, the following rules apply:

- Not more than one sequential file of the set can be in the open mode at any one time.
- You can open sequential files in the input mode in any order.

Table 7–4. Permissible Statements—Sequential Files

Statement	Open Mode			
	Input	Output	I-O	Extend
READ	X		X	
WRITE		X		X
REWRITE			X	

Table 7–5. Permissible Statements—Relative and Indexed Files

File Access Mode	Open Mode				
	Statement	Input	Output	I-O	Extend
Sequential	READ	X		X	
	WRITE		X		X
	REWRITE			X	
	START	X		X	
	DELETE			X	
Random	READ	X		X	
	WRITE		X	X	
	REWRITE			X	
	START				
	DELETE			X	
Dynamic	READ	X		X	
	WRITE		X	X	
	REWRITE			X	
	START	X		X	
	DELETE			X	

If you designate an input file with the OPTIONAL phrase in its SELECT clause, and the file is not present at the time the OPEN statement is executed, the operator is notified of this fact. At this time, the file can be loaded, or the operator can enter the system command OF. If the operator uses the OF command, the first READ statement for this file causes an AT END or INVALID KEY condition to occur. (Refer to the *System Commands Operations Reference Manual* for information on the OF command.)

If you specify label records for the file, the beginning labels are processed as follows:

- If you specify the INPUT phrase, the execution of the OPEN statement causes the labels to be checked.
- If you specify the OUTPUT phrase, the execution of the OPEN statement causes the labels to be written.
- If you specify the EXTEND phrase and the LABEL RECORDS clause indicates that label records are present, the execution of the OPEN statement includes the following steps:
    1. The beginning file labels are processed only in the case of a single reel/unit file.
    2. The beginning reel/unit labels on the last existing reel/unit are processed as though the file were being opened with the INPUT phrase.
    3. The existing ending file labels are processed as though the file were being opened with the INPUT phrase. These labels are then deleted.
    4. Processing then proceeds as though the file had been opened with the OUTPUT phrase.

If you do not specify label records, the operator can intervene to equate the file to one with labels, in which case the label records are ignored.

TADS: Any USE procedure is not executed when a DELETE statement that is compiled and executed in a TADS session fails.

Port Files

The logical communication path between two port files is established by the operating system, provided that the connection descriptions of the two files match.

If an ACTUAL KEY is specified, its value determines which subfile of the file is to be opened. If the ACTUAL KEY value is 0 or if it is not specified, the entire port file is opened. If the ACTUAL KEY value is nonzero, only the specified subfile is opened.

AVAILABLE

The AVAILABLE phrase for a port file specifies that the subfile is opened if it matches a port subfile that has already been offered. If a match does not occur, the port file is not opened and is no longer considered for subsequent matching. A failure to match is considered an error in the open procedure. It causes the program to abort if there is no FILE STATUS or ERROR PROCEDURE declared for the file.

OFFER

The OFFER phrase specifies that the port subfile can be offered for matching to another process and that control returns immediately to the next statement without waiting for a match to occur. OPEN OFFER implies a pending open procedure. This state is not an error, provided that an I/O error has not occurred and the status key value is 00 (although the file has not been opened).

WITH NO WAIT

A READ statement for a port file normally causes the program to wait until a message is available. This suspension can be prevented by using the NO WAIT phrase. For OPEN WITH NO WAIT with AVAILABLEONLY = FALSE, a pending open procedure is not an error, and a status key value of 00 is returned with the subfile unopened. For normal OPEN with AVAILABLEONLY = TRUE, failure to open is an error. If an error is returned by the OPEN operation, control is transferred to the applicable USE procedure. If no USE procedure is specified, the program is terminated.

If FILE STATUS is declared for the subfile, the status key is updated to 00 if the OPEN statement is executed correctly. A status key of 81 is returned if an error is encountered during the execution of the OPEN statement.

Example

```
OPEN INPUT File-A NO REWIND, INPUT File-B.
```

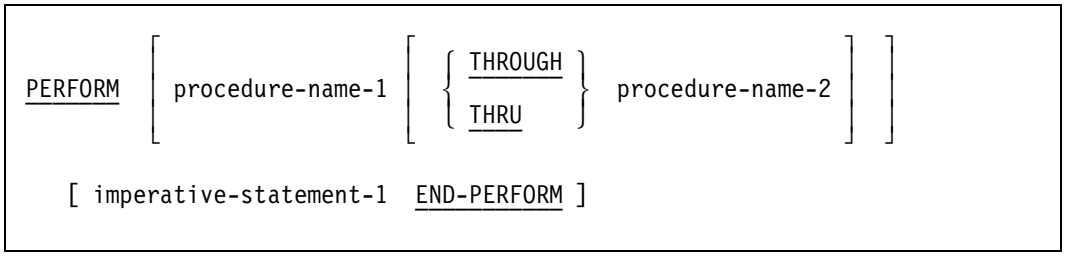
This statement opens two files; File-A must be a sequential file.

PERFORM Statement

The PERFORM statement transfers control explicitly to one or more procedures and returns control implicitly whenever execution of the specified procedure is complete. The PERFORM statement also controls execution of one or more imperative statements that are in the scope of that PERFORM statement.

Format	Use
Format 1	This format is for basic PERFORM statements.
Format 2	The PERFORM . . . TIMES format enables you to PERFORM procedures a specified number of times.
Format 3	The PERFORM . . . UNTIL format enables you to PERFORM procedures until a specified condition is TRUE.
Format 4	The PERFORM . . . VARYING format enables you to PERFORM procedures that augment the values referred to by identifiers or index-names in an orderly fashion.

Format 1: Basic PERFORM



Explanation

procedure-name-1
procedure-name-2

These elements are the names of the procedures to be performed. Together, they represent the beginning and ending of a range of procedures to be performed. If you specify procedure-name-1, do not specify the imperative-statement-1 END-PERFORM phrase.

THROUGH
THRU

These words are interchangeable and connect two procedures that represent the range of the PERFORM statement.

imperative-statement-1 END-PERFORM

The syntax element imperative-statement-1 and the END-PERFORM phrase are required for in-line PERFORM statements. If you specify imperative-statement-1 END-PERFORM, do not specify procedure-name-1.

The END-PERFORM phrase delimits the scope of an in-line PERFORM statement.

Details

When you specify procedure-name-1, the PERFORM statement is known as an out-of-line PERFORM statement. When you omit procedure-name-1, the PERFORM statement is known as an in-line PERFORM statement.

If you use an in-line PERFORM statement, you must specify both imperative-statement-1 and the END-PERFORM phrase.

If you use an out-of-line PERFORM statement, do not specify either imperative-statement-1 or the END-PERFORM phrase.

In Format 1, the specified set of statements is executed once. Then, control passes to the end of the PERFORM statement.

The specified set of statements is defined as follows:

- For an out-of-line PERFORM statement, the set is composed of the statements contained in the range of procedure-name-1 (through procedure-name-2, if specified).
- For an in-line PERFORM statement, the set is composed of the statements contained in the PERFORM statement itself.

When you specify both procedure-name-1 and procedure-name-2, and either one is the name of a procedure in the declaratives portion of the Procedure Division, both must be procedure-names in the same declarative section. For a definition of declarative procedures and a description of format requirements, refer to Section 5.

Examples

```
PARA-3.
    PERFORM PROCESS-PARA.
        .
        .
        .
PROCESS-PARA.
    MOVE SPACES TO PRINT-LINE.
```

In this first example, control passes to the procedure PROCESS-PARA. When the last statement in PROCESS-PARA is executed, control passes to the end of the PERFORM statement.

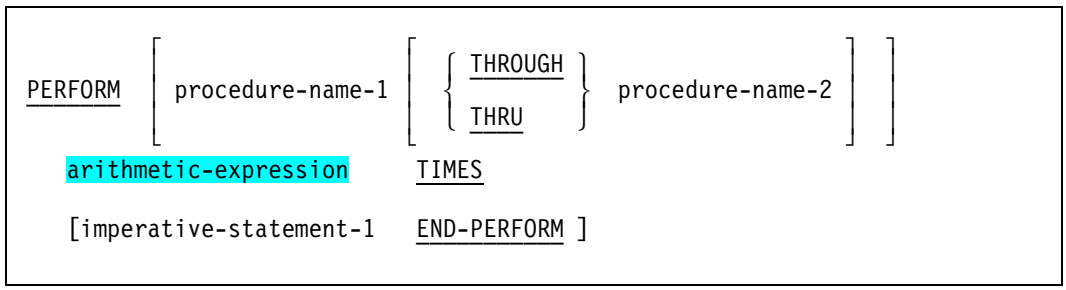
```
PARA-3.
    PERFORM FACT THRU END-1.
        .
        .
        .
FACT.
        .
        .
        .
SEARCH-PARA.
        .
        .
        .
END-1.
```

In this second example, control is transferred to the range of procedures referred to by FACT THRU END-1. When the last statement in the range (FACT THRU END-1) is executed, control passes to the end of the PERFORM statement.

```
PERFORM ADD 1 TO COUNTER
   IF COUNTER = 10
   MOVE 0 TO COUNTER
   DISPLAY  "LIMIT EXCEEDED"
   END-IF
END-PERFORM.
```

In this third example, there is no transfer of control. After the PERFORM statement is executed, control passes to the end of the PERFORM statement. This basic in-line PERFORM could be enhanced later with a TIMES or an UNTIL phrase (refer to Format 2 and Format 3 in this section).

Format 2: PERFORM . . . TIMES



Explanation

Refer to Format 1 for descriptions of the syntax elements procedure-name-1, procedure-name-2, THROUGH, THRU, imperative-statement-1, and the END-PERFORM phrase.

Note that you cannot specify both procedure-name-1 and imperative-statement-1.

arithmetic-expression

This element represents the number of times that a particular set of statements is performed. The result of the expression is integerized and truncated as necessary.

TIMES

In the PERFORM...TIMES statement, a particular set of statements is performed a specified number of times. The number of executions is indicated by the initial value of arithmetic-expression.

Details

When the PERFORM statement is executed, if the value of the arithmetic-expression referred to by arithmetic-expression is equal to zero or is negative, control is passed to the end of the PERFORM statement. After the specified set of statements is executed the specified number of times, control is passed to the end of that PERFORM statement.

While the PERFORM statement is executing, other statements can refer to identifier-1. However, the statements in the scope of the PERFORM cannot alter the number of times that the specified set of statements is executed.

The following statement is not allowed in a nested program where the name used for section/paragraph is declared previously anywhere in the source.

```
PERFORM <section/paragraph> (<arithmetic expression>) TIMES
```

In other words, if the name specified in a PERFORM statement is followed by a left parenthesis, is a global subscripted data item, and is also used as a local section or paragraph that has not yet been recognized, then the name specified in the PERFORM statement must be designated to the global subscripted data item. For example:

```
In the main source

     ...
  01 AA GLOBAL.
     03 BB OCCURS 10 TIMES.
       05 CC  PIC 9(10).

In a nested program

     ...
     PERFORM CC ( I + 1 ) TIMES ...

     CC.
        DISPLAY "In paragraph CC".

then the name CC in the PERFORM statement refers to the global subscripted
data item CC.
```

Examples

```
PERFORM 10 TIMES
ADD CST-LIVING-INC TO TOTAL-PAY
END-PERFORM.
```

This first example shows an in-line PERFORM statement. In this program, the procedures for the imperative statement "ADD CST-LIVING-INC TO TOTAL-PAY" are performed 10 times. Then, control is passed to the END-PERFORM phrase, which is a required element for an in-line PERFORM.

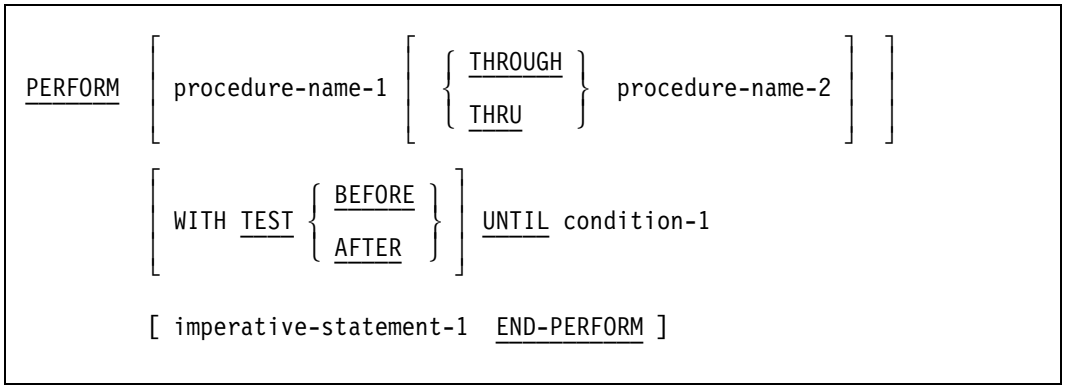
```
PARA-3.
    PERFORM FACT 3 TIMES.
        .
        .
        .
FACT.
        .
        .
        .
PARA-4.
```

In this second example, control passes to FACT. After all statements in FACT are performed three times, control passes to the end of the PERFORM statement.

```
PARA-3.
    PERFORM READS THRU WRITES 4 TIMES.
        .
        .
        .
READS.
        .
        .
        .
PARA-4.
        .
        .
        .
WRITES.
        .
        .
        .
PARA-5.
```

In this third example, control is transferred to the range of procedures referred to by READS THRU WRITES. After the procedure range (READS THRU WRITES) is executed four times, control passes to the end of the PERFORM statement.

Format 3: PERFORM . . . UNTIL



Explanation

Refer to Format 1 for descriptions of the syntax elements procedure-name-1, procedure-name-2, THROUGH, THRU, imperative-statement-1, and the END-PERFORM phrase.

Note that you cannot specify both procedure-name-1 and imperative-statement-1.

WITH TEST BEFORE
WITH TEST AFTER

These phrases enable you to specify whether the condition is tested before or after the specified set of statements is executed.

UNTIL condition-1

This phrase enables you to specify a condition that you want to test. Condition-1 is a conditional expression. Refer to "Conditional Expressions" in Section 5 for more information.

Details

In the PERFORM . . . UNTIL format, the specified set of statements is performed until the condition specified by the UNTIL phrase is true. When this condition is true, control passes to the end of the PERFORM statement.

If the condition is true when the PERFORM statement is entered, and the TEST BEFORE phrase is specified or implied, control is not transferred to procedure-name-1. Instead, control passes to the end of the PERFORM statement.

If the TEST AFTER phrase is specified, the PERFORM statement functions as if the TEST BEFORE phrase were specified, except that the condition is tested after the specified set of statements has been executed.

If you specify neither the TEST BEFORE nor the TEST AFTER phrase, the TEST BEFORE phrase is assumed.

Any subscripting or reference modification that is associated with the operands specified in condition-1 is evaluated each time the condition is tested.

Examples

```
PARA-3.
    PERFORM PROCESS-PARA THRU PROCESS-EXIT UNTIL A = B.
        .
        .
        .
PROCESS-PARA.
        .
        .
        .
PARA-7.
        .
        .
        .
PARA-8.
        .
        .
        .
PROCESS-EXIT.
```

In this first example, control passes to the range of statements from the beginning of PROCESS-PARA through the last statement in PROCESS-EXIT. This range of statements is executed iteratively until the condition A equal to B is true.

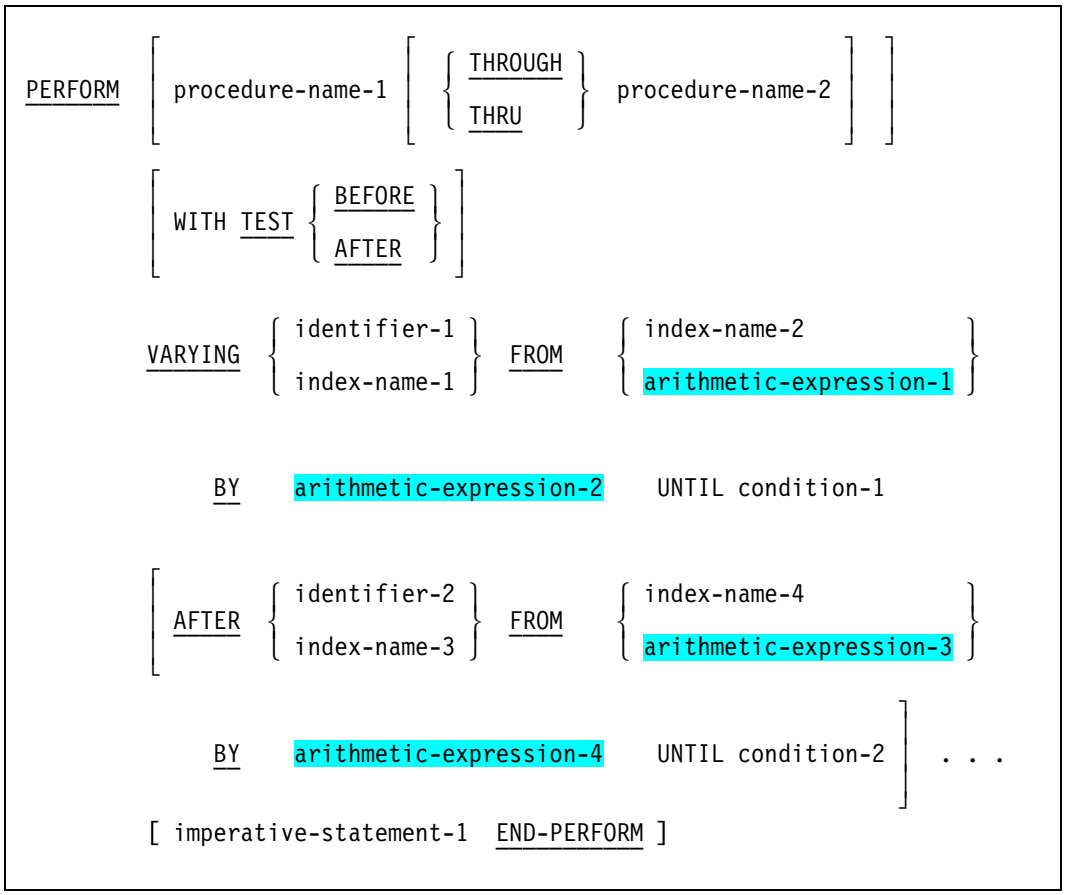
```
SECTION-02.
PARA-4.
    PERFORM SECTION-04 UNTIL I GREATER THAN 10.
        .
        .
        .
SECTION-04.
PARA-9.
```

Because SECTION-04 is a section and can include several paragraphs, the range of the PERFORM statement includes all statements from the first statement in the first paragraph through the final statement in the last paragraph. These statements are performed until the value of I is greater than 10. Then, control passes to the end of the PERFORM statement.

```
PARA-3.
    PERFORM FACT WITH TEST AFTER UNTIL CONDITION-1 =  "TRUE".
        .
        .
        .
FACT.
```

In this second example, condition-1 will be tested after the specified set of statements is executed. If condition-1 is true, control passes to the end of the PERFORM statement. If condition-1 is false, the specified set of statements is executed again. Then, the condition is tested.

Format 4: PERFORM . . . VARYING



Explanation

Refer to Format 1 for descriptions of the syntax elements procedure-name-1, procedure-name-2, THROUGH, THRU, imperative-statement-1, and the END-PERFORM phrase.

Note that you cannot specify both procedure-name-1 and imperative-statement-1.

WITH TEST BEFORE
WITH TEST AFTER

These phrases enable you to specify whether the condition is tested before or after the specified set of statements is executed.

If you specify neither phrase, the TEST BEFORE phrase is assumed.

VARYING
identifier-2
index-name-1

This phrase enables you to vary a data item referred to by an identifier or an index-name. Index-name is a user-defined word that names an index associated with a table.

FROM
index-name-2
arithmetic-expression-1

The FROM phrase establishes the starting value of identifier-1 or index-name-1 that is varied. If arithmetic-expression-1 is used and the data item referred to by identifier-1 is an integer or index-name-1 is used, the result of arithmetic-expression-1 will be integerized and truncated.

BY
arithmetic-expression-2

The BY phrase determines the amount by which identifier-1 or index-name-1 is to be augmented between iterations of the PERFORM range.

UNTIL condition-1

This phrase establishes a condition that, when met, terminates the VARYING . . . FROM . . . BY operation.

AFTER
identifier-2
index-name-3

This option enables you to use a maximum of six nested PERFORM loops in each PERFORM statement. If you omit procedure-name-1, do not specify the AFTER phrase.

FROM
arithmetic-expression-3

This FROM phrase establishes the starting value of identifier-2 or index-name-3 that is to be varied in the nested PERFORM loop. If arithmetic-expression-3 is used and the data item referred to by identifier-2 is integer or index-name-3 is used, the result of arithmetic-expression-3 will be integerized and truncated.

BY
arithmetic-expression-4

The BY phrase determines the amount by which identifier-2 or index-name-3 is to be augmented between iterations of the PERFORM range.

UNTIL condition-2

This phrase establishes a condition that, when met, terminates the AFTER . . . FROM . . . BY operation.

Rules for Identifiers

Each identifier represents a numeric elementary item that is described in the Data Division.

If identifier-2 is subscripted, the subscripts are evaluated each time the content of the data item referred to by the identifier is set or augmented.

Rules for Arithmetic Expressions

The results of arithmetic expressions referred to by arithmetic-expression-2 and arithmetic-expression-4 cannot have a value of zero. If arithmetic-expression-1, arithmetic-expression-2, arithmetic-expression-3, or arithmetic-expression-4 contain subscripted data items, the subscripts are evaluated each time the associated data item is referred to in the expression.

Rules for Index-Names

If you specify index-name-1 or index-name-3, the value of the associated index at the beginning of the PERFORM statement is set to index-name-2, which must not be greater than the number of occurrences. Subsequent augmentation, as described later in this section, of index-name-1 or index-name-3 must not result in the associated index being set to a value outside the range of the table associated with index-name-1 or index-name-3. This restriction applies until the completion of the PERFORM statement; at that time, the index associated with index-name-1 can contain a value that is outside the range of the associated table by one increment or decrement value.

If you specify index-name-2 or index-name-4, the value of the data item referred to by identifier-2 or identifier-5, at the beginning of the PERFORM statement, must be equal to an occurrence number of an element in a table associated with index-name-2 or index-name-4.

If you specify an index-name in the VARYING or AFTER phrase, then the arithmetic expression in the associated FROM or BY phrases must result in a positive integer.

If you specify an index-name in the FROM phrase, then

- The identifier in the associated VARYING or AFTER phrase must refer to an integer data item.
- The arithmetic expression in the associated BY phrase must result in an integer.

Rules for Condition-Names

Condition-1, condition-2, and so forth, can be any conditional expression. For a description of conditional expressions, refer to "Conditional Expressions" in Section 5.

Any subscripting or reference modification associated with the operands specified in condition-1 and condition-2 is evaluated each time the condition is tested.

Action of Various PERFORM Statements

Representations of the actions of several types of Format 4 PERFORM statements appear on the following pages.

In the following discussions, each reference to identifiers as the object of the VARYING, AFTER, and FROM (current value) phrases also refers to index-names.

TEST BEFORE with One Identifier

If the TEST BEFORE phrase is specified or implied, and the data item associated with one identifier is varied, the following actions occur in order:

1. The content of the data item referred to by identifier-2 is set to literal-1 or to the current value of the data item referred to by identifier-3 at the time the PERFORM statement was initially executed.
2. If the condition of the UNTIL phrase is FALSE, the specified set of statements is executed once. The value of the data item referred to by identifier-2 is augmented by the specified increment or decrement value (literal-2 or the value of the data item referred to by identifier-4), and condition-1 is evaluated again.
3. When condition-1 is TRUE, control is transferred to the end of the PERFORM statement.
4. If condition-1 is TRUE at the beginning of execution of the PERFORM statement, control is transferred to the end of the PERFORM statement.

Figure 7–1 illustrates the TEST BEFORE phrase with one identifier varied.

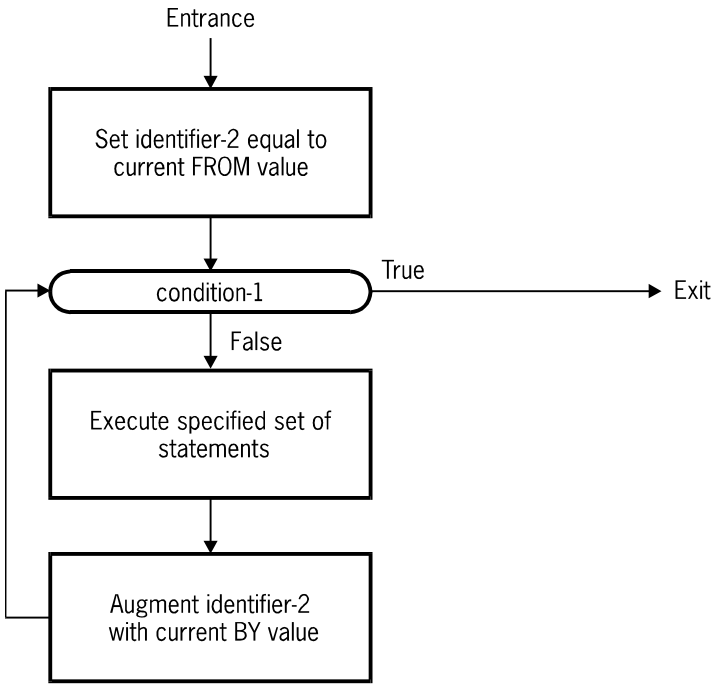


Figure 7–1. TEST BEFORE with One Identifier Varied

TEST BEFORE with Two Identifiers

If the TEST BEFORE phrase is specified or implied, and the data items associated with two identifiers are varied, the following actions occur in order:

1. The content of the data item referred to by identifier-2 is set to literal-1 or to the current value of the data item referred to by identifier-3.
2. The content of the data item referred to by identifier-5 is set to literal-3 or to the current value of the data item referred to by identifier-6.
3. After the contents of the data items of these identifiers have been set, condition-1 is evaluated.
    a. If condition-1 is TRUE, control is transferred to the end of the PERFORM statement.
    b. If condition-1 is false, condition-2 is evaluated.
    c. If condition-2 is false, the specified set of statements is executed once. The content of the data item referred to by identifier-5 is augmented either by literal-4 or by the content of the data item referred to by identifier-7. Then, condition-2 is evaluated again.
    d. The evaluation and augmentation cycle continues until condition-2 is true. When condition-2 is true, the content of the data item referred to by identifier-2 is augmented by literal-2 or by the content of the data item referred to by identifier-4. The content of the data item referred to by identifier-5 is set to literal-3 or to the current value of the data item referred to by identifier-6. Then, condition-1 is reevaluated.
4. The PERFORM statement is completed if condition-1 is true; the cycle continues until condition-1 is true.
5. When the PERFORM statement ends, the data item referred to by identifier-5 contains literal-3 or the current value of the data item referred to by identifier-6. The data item referred to by identifier-2 contains a value that exceeds the last-used setting by one increment or decrement value, unless condition-1 was true when the PERFORM statement was entered. In that case, the data item referred to by identifier-2 contains literal-1 or the current value of the data item referred to by identifier-3.

Figure 7–2 illustrates the TEST BEFORE phrase with two identifiers varied.

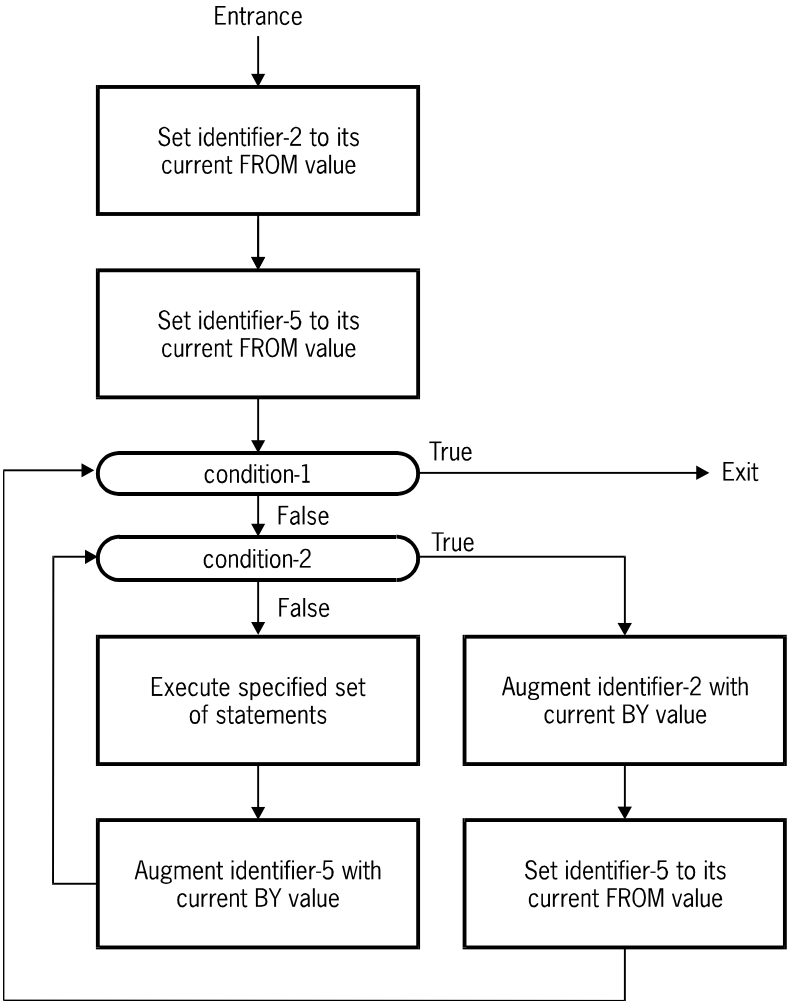


Figure 7–2. TEST BEFORE with Two Identifiers Varied

TEST AFTER with One Identifier

If the TEST AFTER phrase is specified and the data item associated with one identifier is varied, the following actions occur in order:

1. The content of the data item referred to by identifier-2 is set either to literal-1 or to the value of the data item associated with identifier-3 at the time the PERFORM statement is executed.
2. The specified set of statements is executed once, and condition-1 of the UNTIL phrase is tested.
   a. If condition-1 is false, the value of the data item referred to by identifier-2 is augmented by the specified increment or decrement value (literal-2 or the value of the data item referred to by identifier-4), and the specified set of statements is executed again.
   b. When condition-1 is true, control is transferred to the end of the PERFORM statement.

Figure 7–3 illustrates the TEST AFTER phrase with one identifier varied

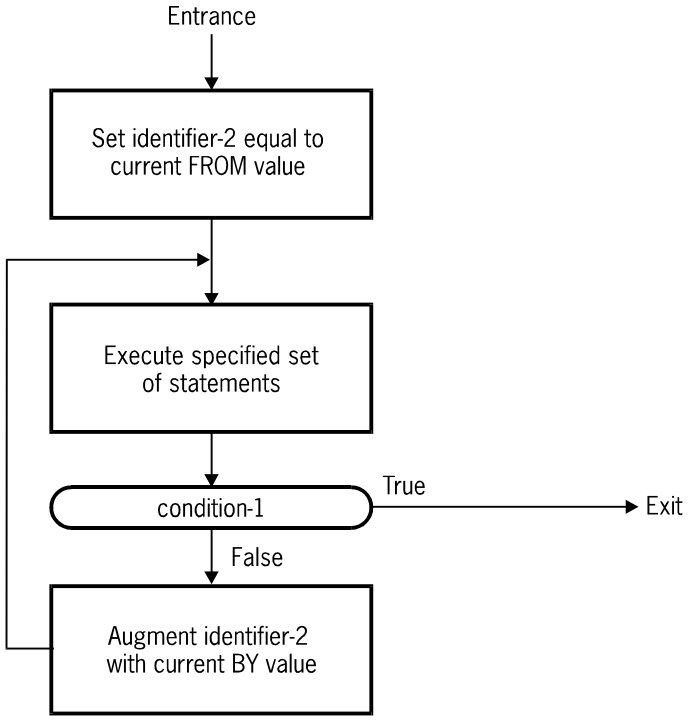


Figure 7–3. TEST AFTER Phrase with One Identifier Varied

TEST AFTER with Two Identifiers

If the TEST AFTER phrase is specified and the data items associated with two identifiers are varied, the following actions occur in order:

1. The content of the data item referred to by identifier-2 is set to literal-1 or to the current value of the data item referred to by identifier-3.
2. The content of the data item referred to by identifier-5 is set to literal-3 or to the current value of the data item referred to by identifier-6. Then, the specified set of statements is executed.
3. Condition-2 is evaluated.
    a. If condition-2 is false, the content of the data item referred to by identifier-5 is augmented by literal-4 or by the content of the data item referred to by identifier-7. Then, the specified set of statements is executed again.
    b. When condition-2 is true, condition-1 is evaluated.
    c. If condition-1 is false, the content of the data item referred to by identifier-2 is augmented by literal-2 or by the content of the data item referred to by identifier-4. The content of the data item referred to by identifier-5 is set to literal-3 or to the current value of the data item referred to by identifier-6. Then, the specified set of statements is executed.
    d. When condition-1 is true, control is transferred to the end of the PERFORM statement.

Figure 7–4 illustrates the TEST AFTER phrase with two identifiers varied.

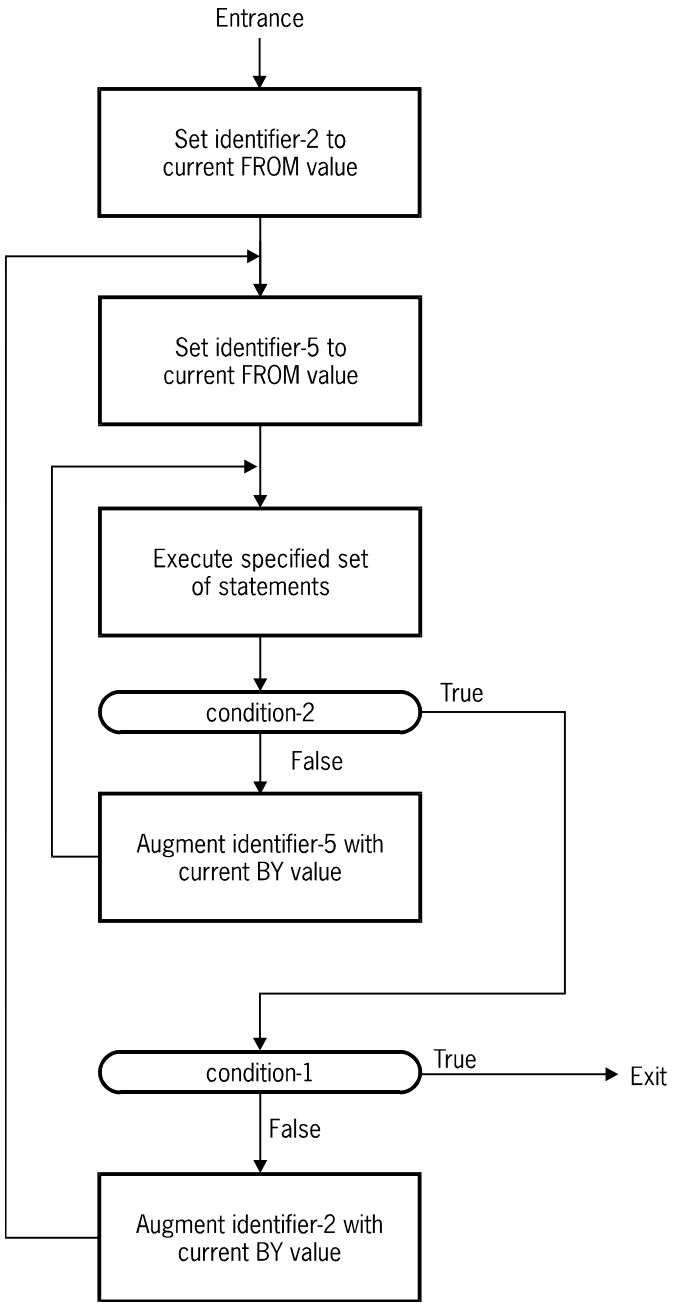


Figure 7–4. TEST AFTER Phrase with Two Identifiers Varied

When the PERFORM statement ends, each data item that was varied by an AFTER or VARYING phrase contains the same value it contained at the end of the most recent execution of the specified set of statements.

How Changes in Variables Affect the PERFORM Statement

When the specified set of statements associated with the PERFORM statement is executed, changes to any of the following variables will affect the operation of the PERFORM statement:

- The VARYING variable (the data item referred to by identifier-2 and index-name-1)
- The BY variable (the data item referred to by identifier-4 or identifier-7)
- The AFTER variable (the data item referred to by identifier-5 and index-name-3)
- The FROM variable (the data item referred to by identifier-3, identifier-6, index-name-2 or index-name-4)

When the data items associated with two identifiers are varied, the data item referred to by identifier-5 goes through a complete cycle (FROM, BY, UNTIL) each time the content of the data item associated with identifier-2 is varied.

Varying the contents of three or more data items is similar to varying the contents of two data items. The data item being varied by each AFTER phrase goes through a complete cycle each time the data item being varied by the preceding AFTER or VARYING phrase is augmented.

Example

```
01 TOTALS.
   03 DIV OCCURS 3 TIMES INDEXED BY DIV-CODE.
      05 DEPT OCCURS 15 TIMES INDEXED BY DEPT-CODE.
         07 COST-CENTER PIC S9(10) COMP OCCURS 30 TIMES
            INDEXED BY COST-CENTER-CODE.
PROCEDURE DIVISION.
      .
      .
      .

   PERFORM PRINT-ROUTINE VARYING DIV-CODE FROM 1 BY 1
       UNTIL DIV-CODE > 3
   AFTER DEPT-CODE FROM 1 BY 1
       UNTIL DEPT-CODE GREATER THAN 15
   AFTER COST-CENTER-CODE FROM 1 BY 1
       UNTIL COST-CENTER-CODE GREATER THAN 30.
      .
      .
      .

PRINT-ROUTINE.
      .
      .
      .

   MOVE COST-CENTER (DIV-CODE, DEPT-CODE,
                COST-CENTER-CODE) TO PRINT-LINES.
   WRITE PRINT-LINES BEFORE 2.
```

In the example, the PERFORM VARYING statement is used to print division, department, and cost-center names on a report. Division, department, and cost-center are indexed by DIV-CODE, DEPT-CODE, and COST-CENTER-CODE, respectively. The two AFTER phrases are used to vary the cost-center code and the department code 15 times and 30 times, respectively.

Rules for All Formats of the PERFORM Statement

Unless specifically qualified by the word in-line or out-of-line, all rules that apply to the out-of-line PERFORM statement also apply to the in-line PERFORM statement. An in-line PERFORM statement can achieve the same result as an out-of-line PERFORM statement, except that the statements contained in the in-line PERFORM statement are executed in place of the statements in the range procedure-name-1 (through procedure-name-2, if specified).

When the PERFORM statement is executed, control is transferred to the first statement of the specified set of statements (except as indicated in the rules for formats 2, 3, and 4). This transfer of control occurs only once for each execution of a PERFORM statement. When an explicit transfer of control to the specified set of statements does take place, an implicit transfer of control to the end of the PERFORM statement is established as indicated in the following table:

If . . .	**And . . .**	**Then . . .**
Procedure-name-1 is a paragraph-name.	You have not specified procedure-name-2.	Control returns to the next executable statement after the last statement of procedure-name-1.
Procedure-name-1 is a section-name.	You have not specified procedure-name-2.	Control returns to the next executable statement after the last statement of the last paragraph in procedure-name-1.
You specify procedure-name-2.	It is a paragraph-name.	Control returns to the next executable statement after the last statement of the last paragraph of procedure-name-2.
You specify procedure-name-2.	It is a section-name.	Control returns to the next executable statement after the last statement of the last paragraph in procedure-name-2.
You specify an in-line PERFORM statement.		An execution of the PERFORM statement is completed after the last statement contained within it has been executed.

A relationship between procedure-name-1 and procedure-name-2 is not necessary. This is true except when a consecutive sequence of operations will be executed from the procedure named by procedure-name-1 through the procedure named by procedure-name-2. In particular, GO TO and PERFORM statements can occur between procedure-name-1 and the end of procedure-name-2. If there are two or more logical paths to the return point, then procedure-name-2 can be the name of a paragraph that consists of the EXIT statement, to which all of the logical paths must lead.

If control passes to the specified set of statements by means other than a PERFORM statement, control will pass through the last statement of the set to the next executable statement as if no PERFORM statement referred to the set.

Range of a PERFORM Statement

The range of a PERFORM statement consists, logically, of all statements that are executed as a result of executing the PERFORM statement until the implicit transfer of control to the end of the PERFORM statement. Also, the range includes all statements that are executed as a result of a transfer of control by CALL, EXIT, GO TO, and other PERFORM statements, within the range of the PERFORM statement; and all statements in declarative procedures that are executed as a result of the execution of statements in the range of the PERFORM statement. The statements in the range of a PERFORM statement need not appear consecutively in the source program.

Statements executed as a result of a transfer of control that was caused by the execution of an EXIT PROGRAM statement are not considered part of the range of the PERFORM statement when the EXIT PROGRAM statement is

- Specified in the same program in which the PERFORM statement is specified
- Is within the range of the PERFORM statement

Procedure-name-1 and procedure-name-2 must not name sections or paragraphs in any other program in the run unit, whether or not the other program contains or is contained in the program that includes the PERFORM statement. In the run unit, statements in other programs can be obeyed only as a result of executing a PERFORM statement, if the range of that PERFORM statement includes CALL and EXIT PROGRAM statements. The CALL statement and the EXIT statement are described earlier in this section.

If the range of a PERFORM statement includes another PERFORM statement, the sequence of procedures associated with the included PERFORM statement must be either totally included in, or totally excluded from, the logical sequence referred to by the first PERFORM statement. Thus, an active PERFORM statement whose execution point begins within the range of another active PERFORM statement must not allow control to pass to the end of the other active PERFORM statement. Moreover, two or more such active PERFORM statements cannot have a common exit.

Figure 7–5 shows examples of valid PERFORM structures.

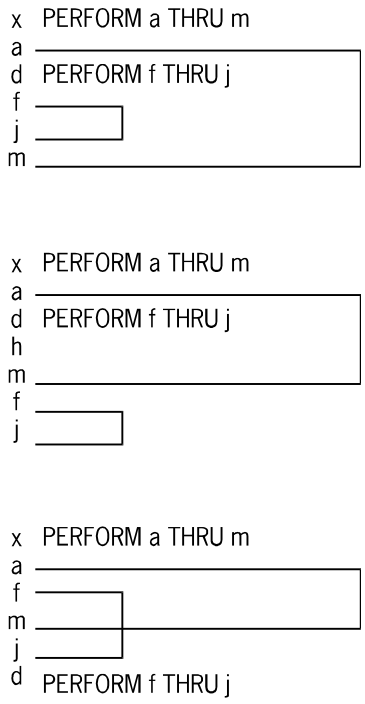


Figure 7–5. Valid PERFORM Structures

For more information about conditional expressions, refer to "Conditional Expressions" in Section 5.

Refer to "CALL Statement" and "EXIT Statement" in Section 6 for more information about CALL or EXIT PROGRAM statements within a PERFORM statement.

PROCESS Statement

The PROCESS statement enables a program to execute a separately compiled program as an asynchronous, dependent process.

The format for this statement is as follows:

```
PROCESS task-variable WITH section-name [ USING actual-parameter-list ].
```

Explanation

task-variable

This specifies the task variable that is to be associated with the process declared in the section identified by section-name. The task variable must be declared as a data item in the Working-Storage Section of the Data Division. For more information about task variables, refer to the USAGE clause in Section 4 and to Section 11.

section-name

This identifies the section in the Procedure Division that contains the name of the object code file that is to be initiated by the PROCESS statement. You must define the section-name in the Declaratives Section of the Procedure Division followed by a USE EXTERNAL statement that specifies the name of the object code file.

USING actual-parameter-list

The USING phrase indicates the parameters in the calling program that are to be passed between both programs. Include the USING phrase only if a USING phrase exists in the Procedure Division header of the called program and in the USE statement of the section identified by section-name in the calling program.

The parameters in the USING phrase can be any combination of 77-level or 01-level or greater data items. In general, the level number, type, length, and order of items in the USING phrase of the calling and called programs must be identical. However, the items in the following list are interchangeable as parameters. That is, each item can be passed to and received by the other. The lengths of the associated items must be the same, however, or run-time errors might occur.

Interchangeable Group Items

- BINARY
- COMP
- DISPLAY
- DOUBLE
- REAL

Other Interchangeable Items

- DOUBLE items with RECEIVED BY REFERENCE clause
- 77-level BINARY REAL data items

Files to be passed as parameters must have a record description. The record description itself can be passed as a parameter. The USING phrase in the Procedure Division header of the called program must not reference any data item in the File Section of the called program. Both the calling and the called programs can read and write to the file passed as a parameter in the CALL statement.

Including a task-variable in the USING phrase enables the called program to make references to the calling program.

Variables can be passed by reference (default) or by value. Table 7–4, which accompanies Format 6 of the CALL statement, describes the matching of formal parameters between the COBOL74/85, ALGOL, and COBOL68 languages.

Details

The process initiated by the PROCESS statement is asynchronous, so it executes simultaneously with the program that initiated it. The initiated process is also dependent, so its existence relies on the continued execution of the process that initiated it. If the initiating process terminates before the dependent process terminates, a critical block exit occurs. For information on how to prevent critical block exit errors, refer to Section 11.

Naming the Program to Be Initiated

You can specify the name of the program to be initiated by the PROCESS statement in one of the following ways:

- Precede the PROCESS statement by a CHANGE statement that changes the NAME attribute of the task variable.
- Define a mnemonic-name in the Special-Names paragraph of the Environment Division, and then use it in the USE EXTERNAL statement.
- Use the following steps:
  - Declare a data item in the Working-Storage section of the Data Division.
  - Name the data item in a USE EXTERNAL statement in the Declarative Section of the Procedure Division.
  - Assign the object code file title to the data item by using a MOVE statement in the Procedure Division.

For program examples that show how to name the program to be initiated, refer to Section 11.

Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CALL-TASK-CALLER.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
     "OBJECT/CALLED"IS TASK-ID.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 DEP-TASK            TASK.
PROCEDURE DIVISION.
DECLARATIVES.
CALL-A-TASK SECTION.
     USE EXTERNAL TASK-ID AS PROCEDURE.
END DECLARATIVES.
MAIN SECTION.
MAIN-PARA.
     PROCESS DEP-TASK   WITH CALL-A-TASK.
    STOP RUN.
```

READ Statement

The READ statement enables you to access records from various kinds of files. For files in sequential access mode, the READ statement makes available the next logical record from a file. For files in random access mode, the READ statement makes available a specific record from a mass storage file.

For an explanation of the three types of file organization (sequential, relative, and indexed) and the three file access modes (sequential, random, and dynamic), refer to Section 10.

If the logical records of a file are described by more than one record description, the records share the same record area in storage. This sharing implicitly redefines the record area. The contents of any data items that are outside the range of the current data record are undefined when the READ statement is executed.

This statement is partially supported in the TADS environment. Applicable exclusions are noted in this section.

Format	Use
Format 1	This format reads the next logical record in a sequential file or any file in sequential access mode.
Format 2	This format is for relative files in random access mode or for files in dynamic access mode when records will be retrieved randomly.
Format 3	This format is for indexed files in random access mode or for files in dynamic access mode when records will be retrieved randomly.

Format 1: Files in Sequential Access Mode

```
READ file-name-1 [ NEXT ] RECORD [WITH NO WAIT ] [ INTO identifier-1 ]
    [ AT END imperative-statement-1 ]
    [ NOT AT END imperative-statement-2 ]
    [ END-READ ]
```

This format is supported in the TADS environment.

Explanation

file-name-1

This user-defined word is the name of the file you want to read.

NEXT

In sequential access mode, the NEXT phrase is optional and does not affect the execution of the READ statement. The NEXT phrase must be specified for files in dynamic access mode when records are retrieved sequentially.

RECORD

This optional word makes the program more specific.

WITH NO WAIT

This phrase is valid only for port files. When this phrase is specified, the program does not wait for a logical record to become available.

INTO identifier-1

This option enables you to move the current record from the record area to the data item specified by identifier-1. Identifier-1 can reference a long numeric data item.

AT END imperative-statement-1

This option enables you to specify an action to be taken if the AT END condition occurs. If there is no next logical record and the AT END phrase is specified, imperative-statement-1 will be executed.

NOT AT END imperative-statement-2

This option enables you to specify an action to be taken if the NOT AT END condition occurs. If the file is not AT END and the NOT AT END phrase is specified, imperative-statement-2 will be executed.

END-READ

This phrase delimits the scope of the READ statement.

Details

Format 1 is for all files in sequential access mode whether the organization of the file is sequential, relative, or indexed.

- A sequential file consists of records organized in a series, one after the other.
- A relative file consists of records defined by their ordinal position in the file.

- An indexed file is a two-part file that consists of an index or key file and a data file that contains the actual data records.

The imperative-statement-1 can be the NEXT SENTENCE phrase. The NEXT SENTENCE phrase is valid only for sequential files.

Refer to the paragraphs headed "Rules for Record Selection" in this section for information on file position indicators and comparisons of records in sequential access mode.

The method used to overlap access time with processing time does not change the use of the READ statement. A record is available to the object program before the execution of imperative-statement-2, if specified, or before the execution of any statement following the READ statement, if imperative-statement-2 is not specified.

If, during the execution of the READ statement, the end of a reel or a unit is recognized, or a reel or a unit does not contain a next logical record, and the logical end of the file has not been reached, the following operations are executed:

- The standard ending reel or unit label procedure is executed.
- A reel or a unit swap occurs: the current volume pointer is updated to point to the next reel or unit existing for the file.
- The standard beginning reel or unit label procedure is executed.

Rules for Relative and Indexed Files (Format 1)

When dynamic access mode is specified for a relative or an indexed file, execution of a Format 1 READ statement with the NEXT phrase retrieves the next logical record from the file.

If file-name-1 is a relative file and the RELATIVE KEY phrase is specified in the file control entry of the Input-Output Section for that file, execution of a Format 1 READ statement moves the relative record number of the record made available to the relative key data item, according to the rules for the MOVE statement. The MOVE statement is described earlier in this section.

When sequential access mode is specified for an indexed file, records that have the same value in an alternate record key (which is the key of reference) are made available in one of the following ways:

- In the same order in which they are released by the execution of WRITE statements
- By the execution of REWRITE statements that create such duplicate values

If you specify dynamic access mode for an indexed file, data-name-1 or the prime record key can also be used for retrievals by subsequent executions of Format 1 READ statements until another key of reference is established for the file.

Format 2: Sequential and Relative Files in Random Access Mode

```
READ file-name-1  RECORD  [ WITH NO WAIT ]  [ INTO identifier-1 ]
    [ INVALID KEY imperative-statement-3 ]
    [ NOT INVALID KEY imperative-statement-4 ]
    [ END-READ ]
```

This format is supported in the TADS environment.

Explanation

Refer to Format 1 for descriptions of syntax elements file-name-1, WITH NO WAIT, INTO identifier-1, and END-READ.

INVALID KEY imperative-statement-3

The INVALID KEY option enables you to specify an action to be taken when the key is invalid.

For more information about the invalid key condition, refer to the "Details" section which follows.

You must include the INVALID KEY option if there is no USE AFTER STANDARD EXCEPTION statement specified for file-name-1.

NOT INVALID KEY imperative-statement-4

The NOT INVALID KEY option enables you to specify an action to be taken when the key is valid.

Details

A sequential file consists of records organized in a series, one after the other.

A relative file consists of records defined by their ordinal position in the file.

The execution of a Format 2 READ statement:

- Sets the file position indicator to the value contained in the data item referred to by the ACTUAL KEY phrase for sequential files or the RELATIVE KEY phrase for relative files. (These phrases are located in the file control entry for the file in the Input-Output Section.)

  For more information on file position indicators and comparisons of records for relative files, refer to "Rules for Record Selection" later in this section.

- Makes the record whose sequential or relative record number equals the file position indicator available in the record area associated with file-name-1.

If the contents of the ACTUAL KEY or RELATIVE KEY data item are less than 1 or are greater than the ordinal number of the last record written to the file, the INVALID KEY condition exists and the READ statement is unsuccessful. For more information about this condition, refer to "Rules for Exception Processing of the READ Statement" later in this section.

If a READ statement is unsuccessful, the content of the associated record area is undefined and the file position indicator is set to indicate that a valid next record has not been established.

Format 3: Indexed Files in Random Access Mode

```
READ file-name-1 RECORD  [ INTO identifier-1 ]
    [ KEY IS data-name-1 ]
    [ INVALID KEY imperative-statement-3 ]
    [ NOT INVALID KEY imperative-statement-4 ]
    [ END-READ ]
```

This format is supported in the TADS environment.

Explanation

Refer to Format 1 for descriptions of syntax elements file-name-1, INTO identifier-1, and END-READ.

Refer to Format 2 for descriptions of the INVALID KEY and NOT INVALID KEY phrases.

KEY IS data-name-1

This phrase establishes data-name-1 as the key of reference for this retrieval. Data-name-1 must be the name of a data item specified as a record key associated with file-name-1. Data-name-1 can be qualified.

Data-name-1 cannot reference a long numeric data item.

Details

An indexed file consists of two parts, an index or "key file" and a data file that contains the actual data records. The key determines the position of each record in the file. This type of file organization provides multiple paths to a given record.

Refer to the paragraphs headed "Rules for Record Selection" in this section for information on file position indicators and comparisons of records for indexed files.

If you specify the KEY phrase in a Format 3 READ statement, data-name-1 becomes the key of reference for this retrieval.

If you do not specify the KEY phrase in a Format 3 READ statement, the prime record key becomes the key of reference for this statement.

The execution of a Format 3 READ statement sets the file position indicator to the value of the key of reference. The first record that matches the value of the key of reference is selected. If there is an alternate key with duplicate values, the record selected is the first record of a sequence of duplicates that was released by the execution of WRITE or REWRITE statements. The selected record is made available to the record area that is associated with file-name-1. If a record is not selected, the invalid key condition exists and execution of the READ statement is unsuccessful.

If the contents of the data item used as the key are less than 1 or are greater than the ordinal number of the last record written to the file, the INVALID KEY condition exists. For more information about this condition, refer to "Rules for Exception Processing of the READ Statement" later in this section.

If a READ statement is unsuccessful, the associated record area is undefined, the key of reference is undefined for indexed files, and the file position indicator is set to indicate that a valid next record has not been established.

Using Port Files (Format 1 or Format 2)

A READ statement causes the program to wait until a logical record is available. The possibility of this suspension is prevented for port files by specifying the WITH NO WAIT phrase. A status key value of 94 indicates that no logical record was available for the read.

If an ACTUAL KEY is declared for a port file, your program is responsible for updating the ACTUAL KEY with an appropriate subfile index. If the ACTUAL KEY is nonzero, a READ statement from the specified subfile is performed. If the ACTUAL KEY is 0 (zero), a nonselective read is performed and the ACTUAL KEY is updated to indicate the subfile index of the subfile that was read.

For a nonselective read, the first logical record to arrive at a subfile in the port file is returned as the data for the READ statement. The subfile to be read is determined by the operating system, and no specific selection algorithm is guaranteed. However, no subfile is read continuously at the expense of the other subfiles.

If no ACTUAL KEY is declared for the port file, the file must contain only a single subfile, and that subfile is read.

Rules for All Formats of the READ Statement

The storage area associated with the data item referred to by identifier-1 and the record area associated with the file referred to by file-name-1 must not be the same storage area.

The file referred to by file-name-1 must be open in the INPUT or I-O mode when the READ statement is executed.

The execution of the READ statement updates the value of the I-O status that is associated with file-name-1. I-O status codes are described in Section 3.

The INTO phrase can be specified in a READ statement when one of the following conditions is true:

- If only one record description is subordinate to the file description entry
- If all record-names associated with file-name-1 and the data item referred to by identifier-1 describe a group item or an elementary alphanumeric item

The result of the execution of a READ statement with the INTO phrase is the same as the result of the following statements:

- The execution of the same READ statement without the INTO phrase.
- An implied MOVE, in which the current record is moved from the record area to the area specified by identifier-1, according to the rules for the MOVE statement without the CORRESPONDING phrase.

  In this operation, the size of the current record is determined by rules specified in the RECORD clause. If the file description entry contains a RECORD IS VARYING clause, the implied move is a group move. The implied move does not occur if the execution of the READ statement was unsuccessful. Any subscripting associated with identifier-1 is evaluated after the record has been read and immediately before it is moved to the data item. The record is available in both the record area and the data item referred to by identifier-1.

  Refer to "MOVE Statement" in this section for an explanation of MOVE actions without the CORRESPONDING phrase.

If the number of character positions in the record that is read is less than the minimum size specified by the record description entries for file-name-1, the portion of the record area to the right of the last valid character is undefined. If the number of character positions in the record that is read is greater than the maximum size specified by the record description entries for file-name-1, the record is truncated on the right to the maximum size. In either of these cases, execution of the READ statement is successful and an I-O status is set to indicate that a record length conflict has occurred. I-O status codes are described in Section 3.

TADS: Any USE procedure is not executed when a READ statement that is compiled and executed in a TADS session fails.

Using the READ Statement with Sequential Files

Records in sequential files are compared according to their record number. The following rules apply:

- If the file position indicator was established by an OPEN statement, the record selected is the first in the file whose record number is greater than or equal to the file position indicator.
- If the file position indicator was established by a READ statement, the record selected is the first in the file whose record number is greater than the file position indicator.
- If a record is available, the file position indicator points to the record number of that record.

Using the READ Statement with Relative Files

Records in relative files are compared according to their relative key number. The following rules apply:

- If the file position indicator was established by an OPEN or a START statement, the record selected is the first in the file whose relative record number is greater than or equal to the file position indicator.
- If the file position indicator was established by a READ statement, the record selected is the first in the file whose relative record number is greater than the file position indicator.
- If a record is found that satisfies the preceding rules, it is made available in the record area that is associated with file-name-1. The record is available unless you specify the RELATIVE KEY phrase for file-name-1, and the number of significant digits in the relative record number of the selected record is larger than the size of the relative key data item. In this case, the I-O status is set to a value of 14. Then, execution proceeds as described in the paragraphs headed "Rules for Exception Processing of the READ Statement" in this section.
- If a record is available, the file position indicator points to the relative record number of that record.

Using the READ Statement with Indexed Files

Records in indexed files are selected according to the value of the current key of reference. Comparisons are made according to the collating sequence of the file. The following rules apply:

- If the file position indicator was established by an OPEN or a START statement, the record selected is the first in the file whose key value is greater than or equal to the file position indicator.
- If the file position indicator was established by a READ statement, and the current key of reference does not allow duplicate values, the record selected is the first in the file whose key value is greater than the file position indicator.
- If the file position indicator was established by a READ statement, and the current key of reference does allow duplicate values, the record is selected in one of the following ways:
  - The first record in the file whose key value is equal to the file position indicator and whose logical position in the set of duplicates is immediately after the record that was made available by that READ statement is selected.
  - The first record in the file whose key value is greater than the file position indicator is selected.
- If a record is available, the file position indicator points to the value of the current key of reference for that record.

Using the READ Statement with Relative and Indexed Files

You must use Format 2 or Format 3 for files in random access mode, or for files in dynamic access mode when records will be retrieved randomly.

You must specify the INVALID KEY phrase or the AT END phrase if you do not specify an applicable USE AFTER STANDARD EXCEPTION procedure for file-name-1.

The method used to overlap access time with processing time does not change the use of the READ statement. A record is available to the object program before the execution of imperative-statement-2 or imperative-statement-4, if specified, or before the execution of any statement following the READ statement, if neither imperative-statement-2 nor imperative-statement-4 is specified.

Rules for Record Selection

The setting of the file position indicator at the start of the execution of a READ statement determines which record will be made available. The following rules apply:

- If the file position indicator indicates that a valid next record has not been established, execution of the READ statement is unsuccessful.
- If the file position indicator indicates that an optional input file is not present, execution proceeds as described in the paragraphs headed "Rules for Exception Processing of the READ Statement" in this section.

If a record is found that satisfies these rules, it is made available in the record area that is associated with file-name-1.

If a record is not found that satisfies these rules, the file position indicator is set to indicate that a next logical record does not exist. Execution proceeds as described in the paragraphs headed "Rules for Exception Processing of the READ Statement" in this section.

Rules for Exception Processing of the READ Statement

If the file position indicator shows that a next logical record does not exist, an optional input file is not present, or the number of significant digits in the relative record number is larger than the size of the relative key data item, the following rules apply:

- A value of 24 is placed into the I-O status that is associated with file-name-1 to indicate the AT END condition.
- If the AT END phrase is specified in the statement causing the AT END condition, control passes to imperative-statement-1 in the AT END phrase. Any USE AFTER STANDARD EXCEPTION procedure that is associated with file-name-1 is not executed.
- If the AT END phrase is not specified, a USE AFTER STANDARD EXCEPTION procedure must be associated with file-name-1, and that procedure is executed. After that procedure is executed, control returns to the next executable statement after the READ statement.

If an AT END or an invalid key condition does not occur during the execution of a READ statement, the AT END phrase or the INVALID KEY phrase is ignored, if specified. The following actions occur:

- The file position indicator is set, and the I-O status associated with file-name-1 is updated.
- If an exception condition exists that is not an AT END or an invalid key condition, control is transferred (according to the rules of the USE statement) after the execution of any USE AFTER STANDARD EXCEPTION procedure that is associated with file-name-1. The USE statement is described later in this section.
- If an exception condition does not exist, the record is made available in the record area, and any implicit move resulting from the presence of an INTO phrase is executed. Control passes to the end of the READ statement or to imperative-statement-2, if specified. Execution continues according to the rules for each statement specified in imperative-statement-2. If a procedure branching or a conditional statement that explicitly transfers control is executed, control is transferred according to the rules for that statement. Otherwise, upon completion of the execution of imperative-statement-2, control passes to the end of the READ statement.

If the AT END condition occurs, execution of the READ statement is unsuccessful. If a READ statement is unsuccessful, the associated record area is undefined, the key of reference is undefined for indexed files, and the I-O status is set to indicate that a valid next record has not been established.

TADS: Any USE procedure is not executed when a READ statement that is compiled and executed in a TADS session fails.

READ Statement Examples

The following paragraphs contain program examples and brief descriptions of how the READ statement in each example is used.

```
FILE-CONTROL.
     SELECT EMP-FILE ASSIGN TO DISK
     ORGANIZATION IS INDEXED
     ACCESS MODE IS DYNAMIC
     RECORD KEY IS ACC-NO
     ALTERNATE RECORD KEY IS NAME WITH DUPLICATES.
DATA DIVISION.
FILE SECTION.
FD EMP-FILE.
01 EMP-NUMBER.
   03  NAME         PIC X(10).
   03  ACC-NO       PIC X(6).
   03  BALANCE      PIC 9(6).
PROCEDURE DIVISION.
BEGIN.
   OPEN I-O EMP-FILE.
   MOVE "010000"TO ACC-NO.
   START EMP-FILE KEY IS NOT LESS THAN ACC-NO
         INVALID KEY PERFORM EDIT-KEY-TROUBLE.
   READ EMP-FILE NEXT AT END PERFORM ERR-PARA.
     .
     .
     .
```

EMP-FILE is an indexed file in dynamic access mode that is opened I/O. The value "010000" is moved to ACC-NO. The first record with a key value not less than "010000" is found, and the pointer is moved to that record. The INVALID KEY phrase is required. When an invalid key condition exists, the EDIT-KEY-TROUBLE procedure is performed. The file is processed beginning with the next record. Refer to "START Statement" in this section for more information about its use with an indexed file.

```
READ EMP-FILE KEY IS NAME
         INVALID KEY PERFORM ERROR1.
```

In this first example, EMP-FILE is an indexed file in random access mode. A key other than the primary key is specified, which affects the order of the record delivery. A value must be placed in NAME before a READ can occur. If an invalid key condition occurs, ERROR1 is performed.

```
FILE-CONTROL.
    SELECT EMP-FILE ASSIGN TO DISK
    ORGANIZATION IS SEQUENTIAL
    ACCESS MODE IS SEQUENTIAL.
DATA DIVISION.
FILE SECTION.
FD EMP-FILE.
01 EMP-NUMBER.
   03  NAME         PIC X(10).
   03  ACC-NO       PIC X(6).
   03  BALANCE      PIC 9(6).
WORKING-STORAGE SECTION.
01 WK-AREA          PIC X(22).
PROCEDURE DIVISION.
BEGIN-PARA.
    OPEN INPUT EMP-FILE.
    READ EMP-FILE INTO WK-AREA
        AT END PERFORM ERR-PARA.
```

In this second example, EMP-FILE is a sequential file. EMP-FILE is read into WK-AREA, which is an identifier that refers to a data item in the working storage area. If an AT END condition occurs, ERR-PARA is performed.

```
FILE-CONTROL.
    SELECT EMP-FILE ASSIGN TO DISK
    ORGANIZATION IS RELATIVE
    ACCESS MODE IS RANDOM
    RELATIVE KEY IS REC-KEY.
DATA DIVISION.
FILE SECTION.
FD EMP-FILE.
01 EMP-NUMBER.
   03  NAME         PIC X(10).
   03  ACC-NO       PIC X(6).
   03  BALANCE      PIC 9(6).
WORKING-STORAGE SECTION.
01 REC-KEY          PIC 999.
01 WK-AREA          PIC X(22).
PROCEDURE DIVISION.
BEGIN-PARA.
    OPEN I-O EMP-FILE.
    MOVE 12 TO REC-KEY.
    READ EMP-FILE INVALID KEY
        PERFORM ERR-PARA.
```

In this third example, EMP-FILE is a relative file in random access mode. Before this file is read, a value pointing to the ordinal position of the record is moved to the key. The INVALID KEY phrase is required.

RECEIVE Statement

The RECEIVE statement enables a program to obtain data from another program in the same multiprogramming mix or from a storage queue.

Format	Use
Format 1	This format is used to receive data in a synchronous way from another program that is active in the same multiprogramming mix. Format 1 uses the CRCR (core-to-core) capabilities of the MCP. For an overview of CRCR functionality, refer to Format 1 of the SEND Statement and to the Task Management Programming Guide.
Format 2	This format is used to receive data from a storage queue in an asynchronous way. Format 2 uses the STOQUE (STOQ) capability of the MCP. For an overview of the STOQ functionality, refer to Format 2 of the SEND Statement and to the Task Management Programming Guide.

Format 1: Receive Data Synchronously

```
(CRCR)    [ identifier-1 ]
RECEIVE   {              }   INTO identifier-2
          [ literal-1    ]

[ ON EXCEPTION imperative-statement ]

[ NOT ON EXCEPTION imperative-statement ]

[ END-RECEIVE ].
```

Explanation

identifer-1
literal-1

This must be a nonnumeric data item that specifies the name of the program that is to send the data. The program name must be a file title that contains a maximum of 256 characters. It is not necessary to terminate the file title with a period. If no usercode is specified, the usercode of the receiving program is used. If the "ON <family name>" clause is used in the file title, it is ignored by the system in the comparison.

The program sending the data must be present in the mix.

If the value of identifier-1 or literal-1 is all blanks (called *GLOBAL FILL*), the receiving program can receive data from any program in the mix that issues a CRCR SEND statement.

identifier-2

This identifier must reference either an alphanumeric data item or a long numeric data item in the receiving program where the transferred data is to be stored. If the size of the sending and receiving fields is unequal, the smaller size is used, and the data is truncated or filled with blanks, whichever is necessary.

The variable declared in the sending process can be of a different type than the variable declared in the receiving process. However, the system does not perform any data translation. The data received is a bit-image of the data that was sent.

ON EXCEPTION imperative-statement

This clause provides an alternate statement for the receiving program to perform if the sending program is not ready (an *exception condition*.) If this clause is not used and the sending program is not ready, the receiving program is suspended until the sending program is ready.

NOT ON EXCEPTION imperative-statement

This clause provides a statement for the receiving program to perform after the data transfer has successfully occurred.

Details

This format of the RECEIVE statement uses the CRCR (core-to-core) capabilities of the MCP. CRCR is a synchronous communication method that enables a program to send data to or receive data from another program that is present in the mix. Both programs must be ready to communicate for the data transfer to occur.

If the program designated to receive the data does not execute a RECEIVE statement, the sending program is suspended until the RECEIVE statement is executed. For details about the CRCR functionality, refer to Format 1 of the SEND statement and to the *Task Management Programming Guide*.

Format 2: Receive Data Asynchronously (STOQ)

```
RECEIVE FROM { TOP    } identifier-1
             { BOTTOM }

[ ON EXCEPTION imperative-statement-1 ]

[ NOT ON EXCEPTION imperative-statement-2 ]

[ END-RECEIVE ].
```

Explanation

TOP
BOTTOM

This determines whether the received data is to be retrieved from the top of the queue or from the bottom of the queue.

identifier-1

This identifier must refer to a 01-level data-description-entry for a storage queue (STOQ) parameter block.

ON EXCEPTION imperative-statement

This clause provides an alternate instruction to be performed if the queue is empty or no individual entry satisfies the specified name (*exception conditions*.) If this clause is not used and an exception condition exists, the receiving program is suspended until the requested item is placed in the queue.

NOT ON EXCEPTION imperative-statement

This clause provides an instruction for the receiving program to perform after the data transfer has occurred.

Details

To understand how the RECEIVE statement retrieves data from a queue, it is necessary to know the structure of a storage queue (STOQ) parameter block. The STOQ parameter block is a 01-level data description entry of the following format:

```
01 Identifier-1.
       02  Queue-name              PIC X(6).
       02  Entry-name-length       PIC 9(2) COMP.
       02  Entry-name              PIC X(nn).
       02  Entry-data-length       PIC 9(4) COMP.
       02  Entry-data              PIC X(nnnn).
```

For a detailed description of the STOQ parameter block and an overview of STOQ functionality, refer to Format 2 of the SEND Statement and to the *Task Management Programming Guide*.

Operation of the RECEIVE Statement

The RECEIVE statement causes the user program to receive data from the queue named in the STOQ parameter block identified by identifier-1. If an entry-name is also specified in the parameter block, the data is received from the item or items (sub-queue) within the queue that match the entry-name. The data is placed in the entry-data field.

When the request is complete, execution resumes with either the statement included with the NOT ON EXCEPTION clause, if specified, or with the next statement in the program.

Size of Received Data

The size of the data is returned by the system and reflects the actual length of the data returned.

If the data in the queue is . . .	Then . . .
Longer than the value in the entry-data-length field	Only the first entry-data-length characters are received. The entry-data-length field is adjusted to show the actual number of characters of the data in the queue.
Shorter than the value in the entry-data-length field	The field is adjusted to show the actual number of characters in the data.

Entry-Name

More than one item in the storage queue can have the same name; the entry name need not be unique. Also, the name given to an item when sent by the SEND verb can be longer than the name specified in the entry-name for a RECEIVE request. In either situation, the queue is searched for an item whose name matches the entry-name in the first entry-name-length of characters, as follows:

- If you specify RECEIVE FROM TOP, the first entry from the top of the queue that meets the selection criteria is chosen.
- If you specify RECEIVE FROM BOTTOM, the first entry from the bottom of the queue that meets the selection criteria is chosen.

Exception Conditions

If the queue is empty or if no individual entry satisfies the specified name, the ON EXCEPTION condition exists. In that case,

- If you included the ON EXCEPTION clause, the imperative-statement is executed.
- If you did not include the ON EXCEPTION clause, the program is suspended until the requested item is placed in the queue.

RELEASE Statement

The RELEASE statement transfers records to the initial phase of a sort operation and writes records to a sort file or a merge file.

```
RELEASE  record-name-1  [ FROM identifier-1 ]
```

Explanation

record-name-1

Record-name must be the name of a logical record in a sort/merge file description entry. It can be qualified.

FROM identifier-1

This option moves the contents of the data item referred to by identifier-1 to record-name-1.

Details

You can use a RELEASE statement only within the range of an input procedure. The input procedure must be associated with a SORT statement for the file-name whose sort/merge file description entry contains record-name-1.

Record-name-1 and identifier-1 cannot refer to the same storage area.

The execution of a RELEASE statement releases the record referred to by record-name-1 to the initial phase of a sort operation. That is, a RELEASE statement writes a record into the sort/merge file.

Once you release a logical record by executing the RELEASE statement, the record is no longer available in the record area. This is true unless the sort/merge file-name that is associated with record-name-1 is specified in a SAME RECORD AREA clause. In this case, the logical record is also available as a record of other files referenced in that SAME RECORD AREA clause.

If you use the FROM phrase when you execute a RELEASE statement, the result is the same as the execution of the following two statements, in order:

1. MOVE identifier-1 TO record-name-1
2. RELEASE record-name-1

In these statements, the contents of identifier-1 are moved to record-name-1. Then, the contents of record-name-1 are released to the sort file. Movement occurs according to the rules specified for the MOVE statement. Refer to "MOVE Rules" under "MOVE Statement" in this section for more information about MOVE actions.

After the execution of the RELEASE statement, the information in the area referred to by identifier-1 is available. The information in the area referred to by record-name-1 is not available, except as specified by the SAME RECORD AREA clause.

If you do not use the FROM phrase, you must use MOVE statements to move data into the sort/merge file record area.

Related Information

The following table provides references for additional information related to this statement:

For information about . . .	Refer to . . .
Sort/merge file descriptions	"File Description Entry" in Section 4
Sort/merge file operations	"Sort and Merge Operations," "Sorting," "Merging," and "Sort and Merge Constructs" in Section 5
RELEASE and RETURN actions (examples)	"RETURN Statement" in this section
Relationship of RELEASE statement to SORT statement	"SORT Statement" in Section 8

REPLACE Statement

The REPLACE statement replaces source program text.

Format	Use
Format 1	This format starts REPLACE operations.
Format 2	This format discontinues REPLACE operations.

Format 1: Start REPLACE Operations

```
REPLACE { ==pseudo-text-1==  BY  ==pseudo-text-2== }  . . .
```

Explanation

REPLACE . . . BY

This syntax specifies the text of the source program to be replaced and describes the text that is to replace it. Each matched occurrence of pseudo-text-1 in the source program is replaced by the corresponding pseudo-text-2.

pseudo-text-1
pseudo-text-2

Pseudotext is a sequence of text words, comment lines, or a separator space in a source program. Double equal signs (=) serve as delimiters. For more information on pseudotext, refer to Section 1.

Pseudo-text-1 must contain one or more text words and must not consist entirely of commas (,) or semicolons (;). Pseudo-text-2 can contain zero, one, or more text words.

Character-strings in pseudo-text-1 and pseudo-text-2 can be continued.

Details

Each occurrence of a REPLACE statement is in effect from the point at which it is specified to the next occurrence of the REPLACE statement or to the end of the separately compiled program.

REPLACE statements in a source program are processed after COPY statements. Refer in Section 6 to "COPY Statement," which copies text from a library program into the program that contains the COPY statement.

Rules

A REPLACE statement can occur in the source program anywhere a character-string can occur. The statement must be preceded by a period (.) unless it is the first statement in a separately compiled program.

A REPLACE statement must be terminated by a separator period.

If the word REPLACE appears in a comment-entry or in the area where a comment-entry can appear, it is recognized as part of the comment-entry.

The text produced by a REPLACE statement must not contain another REPLACE statement.

A text word in pseudotext can be from 1 through 322 characters long.

Text Replacement Comparisons

During text replacement operations, the following comparisons occur:

1. The text words in pseudo-text-1 are compared to an equivalent number of contiguous source program text words. The comparison begins with the leftmost source program text word and the first word in pseudo-text-1.
2. If the ordered sequence of text words that forms pseudo-text-1 is equal, character for character, to the ordered sequence of source program text words, a match of the source program text occurs. During comparisons, each occurrence of a separator comma, semicolon, or space in pseudo-text-1 or in the source program text is recognized as a single space. Each sequence of one or more space separators is recognized as a single space.
    a. If a match does not occur, each new comparison begins with each successive occurrence of pseudo-text-1, until either a match is found or a successive occurrence of pseudo-text-1 does not exist.
    b. When all occurrences of pseudo-text-1 have been compared and a match has not occurred, the next source program text word becomes the leftmost source program text word. Then, the comparison cycle begins again with the first occurrence of pseudo-text-1.
    c. When a match occurs between pseudo-text-1 and the source program text, the corresponding pseudo-text-2 replaces the matched text in the source program. The source program text word immediately following the rightmost text word that participated in the match becomes the leftmost source program text word. Then, the comparison cycle begins again with the first occurrence of pseudo-text-1.
3. The comparison cycles continue until the rightmost text word in the source program text that is in the scope of the REPLACE statement either has participated in a match, or has become the leftmost source program text word and has participated in a complete comparison cycle.

Comment Lines

Comment lines or blank lines that appear in the source program text and in pseudo-text-1 are ignored during comparisons. The rules for reference format determine the sequence of text words in the source program text and in pseudo-text-1. For a description of reference format, see Section 1.

If a comment line or a blank line appears in the sequence of text words that match pseudo-text-1, it is not copied into the resultant source program text. Comment lines or blank lines in pseudo-text-2 are inserted, unchanged, in the resultant program whenever pseudo-text-2 is inserted in the source program by a REPLACE statement.

Debugging Lines

Pseudotext permits debugging lines, which are identified by a *D* in the indicator area. Text words within a debugging line participate in the matching rules as if the *D* were not present in the indicator area.

Except for COPY and REPLACE statements, the syntactic accuracy of the source program text cannot be determined until all COPY and REPLACE statements have been processed completely.

Text Words

Text words that are inserted in the source program by a REPLACE statement are inserted in the source program according to the rules for reference format.

When text words of pseudo-text-2 are inserted in the source program, additional spaces can be introduced only between text words where a space already exists (including the assumed space between source lines).

Additional Lines

If additional lines are introduced in the source program by REPLACE statements, the indicator area of the introduced lines contains the same character as the line on which the text being replaced begins. However, if that line contains a hyphen (-), the introduced line contains a space.

Literals

If the length of any literal in pseudo-text-2 is too long to fit on a single line in the resultant program, and the literal is not on a debugging line, additional continuation lines are introduced for the remainder of the literal. If replacement requires that the literal be continued on a debugging line, the program is in error.

Format 2: Discontinue REPLACE Operations

```
REPLACE OFF
```

Explanation

REPLACE OFF

This syntax discontinues any text replacement currently in effect.

Details

A REPLACE statement can be stopped in one of two ways:

- If you specify a new REPLACE . . . BY statement
- If you specify REPLACE OFF

Rules

A REPLACE statement can occur in the source program anywhere a character-string can occur. The statement must be preceded by a separator period unless it is the first statement in a separately compiled program.

A REPLACE statement must be terminated by a separator period.

If the word REPLACE appears in a comment-entry or in the area where a comment-entry can appear, it is recognized as part of the comment-entry. (In COBOL ANSI-85, comment-entry is an obsolete element and will be deleted from the next revision of standard COBOL.)

Related Information

The following table provides references for additional information related to this statement:

For information about . . .	Refer to . . .
Sort and merge file operations	"Sort and Merge Operations," "Sorting," "Merging," and "Sort and Merge Constructs" in Section 5
The relationship of the RETURN statement to the MERGE and SORT statements	"MERGE Statement" in this section and "SORT Statement" in Section 8
Valid and invalid move actions	"MOVE Statement" in this section

Examples

```
REPLACE ==PICTURE-68== BY ==REC-1==
        ==CMP== BY ==COMP==.
   02 PICTURE-68    PIC 99 CMP.
REPLACE OFF.
```

In this example, REC-1 and COMP replace PICTURE-68 and CMP. REPLACE OFF cancels the REPLACE statement. You can also cancel one REPLACE statement by introducing another REPLACE statement.

The REPLACE statement is especially useful for resolving conflicts between the new COBOL ANSI-85 reserved words and user-defined words in COBOL ANSI-74 or COBOL ANSI-68 programs. For example, the following REPLACE statement changes several user-defined words that are acceptable in COBOL ANSI-68 and COBOL ANSI-74 (but not in COBOL ANSI-85) into user-defined words acceptable in COBOL ANSI-85:

```
REPLACE ==WITH DEBUGGING MODE== BY == ==
        ==ORDER==               BY ==SEQUENCE==
        ==TEST==                BY ==TEST-CASE==
        ==CONVERTING==          BY ==CONVERTING-TO==
        ==CONTENT==             BY ==CONTENT-OF==
        ==PURGE==               BY ==REMOVE==.
```

In this example, several user-defined words replace new reserved words. To remove text from a source program, insert two equal signs, a space, and two more equal signs (== ==) following the word BY. In this example, WITH DEBUGGING MODE is removed from the program because the DEBUG module is an obsolete element in COBOL ANSI-85 and will be deleted from the next revision of standard COBOL.

RESET Statement

The RESET statement turns off specified events.

```
RESET event-identifier-1 [ ,event-identifier-2 ] . . .
```

Explanation

event-identifier-1
event-identifier-2 . . .

The event-identifier can be one or more of the following:

- The name of a data-item declared with the USAGE IS EVENT phrase. The data-name must be properly qualified and properly subscripted.
- A task attribute of type EVENT. The two event task attributes are ACCEPTEVENT and EXCEPTIONEVENT. For details about these task attributes, refer to the *Task Attributes Programming Reference Manual*.
- A file attribute of type EVENT. The three event file attributes are CHANGEEVENT, INPUTEVENT, and OUTPUTEVENT. For details about these file attributes, refer to the *File Attributes Programming Reference Manual*.

Example

```
RESET WS-01-EVENT.
```

RETURN Statement

The RETURN statement obtains sorted or merged records from the final phase of a SORT or MERGE operation.

```
RETURN  file-name-1  RECORD  [ INTO identifier-1 ]
    AT END imperative-statement-1
     [ NOT AT END imperative-statement-2 ]
     [ END-RETURN ]
```

Explanation

file-name-1

This user-defined word is the name of a sort or merge file. File-name-1 must be described in a sort-merge file description entry in the Data Division.

INTO identifier-1

This option returns records to a record area and to a data item referred to by identifier-1. The record is available in both the record area and the data item referred to by identifier-1.

AT END imperative-statement-1

This option enables you to include an imperative statement that specifies an action to be taken if the AT END condition occurs. If the file is at the end (there is no next logical record) and the AT END phrase is specified, imperative-statement-1 will be executed.

A possible value for imperative-statement-1 is the literal "NEXT SENTENCE." This phrase is valid only for sequential files. If an end-of-file condition occurs, this phrase causes control to be passed to the next executable statement.

NOT AT END imperative-statement-2

This option enables you to include an imperative statement that specifies an action to be taken if the NOT AT END phrase is specified. If the file is not at the end and the NOT AT END phrase is specified, imperative-statement-2 is executed.

END-RETURN

This phrase delimits the scope of the RETURN statement.

Details

The storage area associated with identifier-1 and the record area associated with file-name-1 cannot be the same storage area.

You can use a RETURN statement only within the range of an output procedure that is associated with a SORT or a MERGE statement for file-name-1.

When the logical records in a file are described by more than one record description, these records share the same storage area. This sharing is the same as an implicit redefinition of the area. Note that the contents of any data items that are outside the range of the current data record are undefined when the execution of the RETURN statement is complete.

The execution of the RETURN statement transfers the next existing record in the file referred to by file-name-1 (as determined by the keys listed in the SORT or MERGE statement) to the record area that is associated with file-name-1.

Function of the RETURN Statement

When the RETURN statement is executed and . . .	Then . . .	And . . .
A next logical record does not exist in the file referred to by file-name-1.	The "at end" condition exists.	Control passes to imperative-statement-1 of the AT END phrase. Execution continues according to the rules for each statement specified in imperative-statement-1.
Imperative-statement-1 is a procedure-branching or conditional statement that explicitly transfers control.	The statement is executed.	Control is transferred according to the rules for that statement.
Imperative-statement-1 is not a statement that explicitly transfers control.	The statement is executed.	Control passes to the end of the RETURN statement and the NOT AT END phrase is ignored, if specified.
The "at end" condition occurs.	Execution of the RETURN statement is unsuccessful. In this case, the contents of the record area that is associated with file-name-1 are undefined. After the execution of imperative-statement-1 in the AT END phrase, a RETURN statement cannot be executed as part of the current output procedure.	
An "at end" condition does not occur.	The record is made available and any implicit move resulting from the presence of an INTO phrase is executed.	Control passes to imperative-statement-2, if specified.

Specifying the INTO Phrase

The INTO phrase can be specified in a RETURN statement when either of the following conditions is true:

- Only one record description is subordinate to the sort-merge file description entry.
- All record-names associated with file-name-1 and the data item referred to by identifier-1 describe a group item or an elementary alphanumeric item.

The result of executing a RETURN statement with an INTO phrase is the same as the application of the following two rules, in order:

1. The execution of the same RETURN statement without the INTO phrase.
2. An implied MOVE statement, in which the current record is moved from the record area to the area specified by identifier-1, according to the rules for the MOVE statement without the CORRESPONDING phrase. In this operation, the size of the current record is determined by rules specified for the RECORD clause. If the file description entry contains a RECORD IS VARYING clause, the implied move is a group move. The implied MOVE statement is not performed if the execution of the RETURN statement was unsuccessful. Any subscripting associated with identifier-1 is evaluated after the record has been read and immediately before it is moved to the data item.

Example

```
FILE CONTROL.
    SELECT FILE-1                ASSIGN TO DISK.
    SELECT FILE-2                ASSIGN TO DISK.
    SELECT SRT-FIL               ASSIGN TO SORT.
DATA DIVISION.
FILE SECTION.
FD   FILE-1.
01   F1REC.
    03 FILLER           PIC X(180).
FD   FILE-2.
01   F2REC.
    03 FILLER           PIC X(180).
SD   SRT-FIL.
01   SREC.
    03 FILLER           PIC X(10).
    03 ACC-NO           PIC 9(6).
    03 FILLER           PIC X(10).
    03 BAL-DUE          PIC 9(20).
PROCEDURE DIVISION.
BEGIN.
    SORT SRT-FIL
        ON ASCENDING KEY ACC-NO
        INPUT PROCEDURE IS PROC-1 THRU END-1
        OUTPUT PROCEDURE IS PROC-2 THRU END-2.
            .
            .
```

```
            .
PROC-1.
    OPEN INPUT FILE-1.
PROC-1A.
    READ FILE-1 AT END GO TO END-1.
            .
            .
            .
    RELEASE SREC FROM F1REC.
    GO TO PROC-1A.
END-1.
    CLOSE FILE-1.
PROC-2.
    OPEN OUTPUT FILE-2.
    RETURN SRT-FIL INTO F2REC AT END GO TO END-2.
    WRITE F2REC.
            .
            .
            .
END-2.
    CLOSE FILE-2.
```

The input procedure occurs first. FILE-1 is opened and read. Then, its record is released from F1REC to SORT. After the file is sorted, the output procedure begins. FILE-2 is opened, and the sorted records are returned to F2REC and written to the disk file.

REWRITE Statement

The REWRITE statement logically replaces a record that exists in a mass-storage file.

This statement is partially supported in a TADS environment. Applicable exclusions are noted in this section.

Format	Use
Format 1	This format is for sequential files.
Format 2	This format is for relative and indexed files.

Format 1: Sequential Files

```
REWRITE  record-name-1  [ SYNCHRONIZED ]
             [ FROM identifier-1 ]  [ END-REWRITE ]
```

This format is supported in the TADS environment.

Explanation

record-name-1

This user-defined word names a logical record in the File Section of the Data Division. It can be qualified.

SYNCHRONIZED

This option enables you to override the synchronization specified by the file attribute for a specific output record.

Synchronization means that output must be written to the physical file before the program initiating the output can resume execution, thereby ensuring synchronization between logical and physical files. Synchronization of all output records can be designated with the SYNCHRONIZE file attribute. Synchronization is available for use by tape files and disk files with sequential organization only, and is not available for use by port files.

A periodic synchronous REWRITE statement that follows one or more asynchronous REWRITE statements can be used as a checkpoint to ensure that all outstanding records are written to the file before the program continues execution.

FROM identifier-1

This option enables you to move data from the data item referred to by identifier-1 into a record, and then to rewrite the record. If identifier-1 is a function-identifier, it must reference an alphanumeric function. If identifier-1 is not a function-identifier, it cannot reference the same storage area as record-name-1.

END-REWRITE

This phrase delimits the scope of the REWRITE statement.

Details

A successfully executed READ statement must have been the last input-output statement executed for the associated file before the execution of the REWRITE statement. The disk or disk pack logically replaces the record that was accessed by the READ statement.

If the number of character positions specified in the record referred to by record-name-1 is not equal to the number of character positions in the record being replaced, then

- The execution of the REWRITE statement is unsuccessful.
- The record is not updated.
- The content of the record area is unaffected.
- The I-O status of the file associated with record-name-1 is set to 44. Refer to "I-O Status Codes" in Section 3 for more information.

Refer to the paragraphs headed “Rules for All File Organizations” in this section for more information on Format 1.

Example

```
READ MST-FIL AT END PERFORM END-LOGIC.
REWRITE MST-FIL-REC FROM DATA-AREA.
```

In this example, MST-FIL-REC is obtained from DATA-AREA (another area of storage) and is rewritten.

Format 2: Relative and Indexed Files

```
REWRITE  record-name-1 [ SYNCHRONIZED ] [ FROM identifier-1 ]
     [ INVALID KEY imperative-statement-1 ]
     [ NOT INVALID KEY imperative-statement-2 ]
     [ END-REWRITE ]
```

This format is supported in the TADS environment.

Explanation

Refer to Format 1 for descriptions of the syntax elements record-name-1, FROM identifier-1, and END-REWRITE.

SYNCHRONIZED

This option enables you to override the synchronization specified by the file attribute for a specific output record.

Synchronization means that output must be written to the physical file before the program initiating the output can resume execution, thereby ensuring synchronization between logical and physical files. Synchronization of all output records can be designated with the SYNCHRONIZE file attribute. Synchronization is available for use by tape files and disk files with sequential organization only, and is not available for use by port files.

A periodic synchronous REWRITE statement that follows one or more asynchronous REWRITE statements can be used as a checkpoint to ensure that all outstanding records are written to the file before the program continues execution.

INVALID KEY imperative-statement-1

This phrase enables you to specify an action to be taken when the key is invalid.

NOT INVALID KEY imperative-statement-2

This phrase enables you to specify an action to be taken when the key is valid.

Details

Rules for All File Organizations

Record-name-1 and identifier-1 must not refer to the same storage area.

The file referred to by the file-name that is associated with record-name-1 must be a mass-storage file and must be open in I-O mode at the time the REWRITE statement is executed. Refer to "OPEN Statement" in this section for more information on opening a file in the I-O mode.

Execution of the REWRITE statement does not affect the contents or accessibility of the record area.

Execution of a REWRITE statement with the FROM phrase is the same as the execution of the statement "MOVE identifier-1 TO record-name-1" (according to the MOVE statement rules), followed by the execution of the same REWRITE statement without the FROM phrase. Refer to "MOVE Statement" in this section for more information on the MOVE rules.

After execution of the REWRITE statement is complete, information in the area referred to by identifier-1 is available. However, information in the area referred to by record-name-1 is not available except as specified in the SAME RECORD AREA clause.

The file position indicator is not affected by the execution of a REWRITE statement.

Execution of the REWRITE statement updates the value of the I-O status of the file-name associated with record-name-1. Also, execution of the REWRITE statement releases a logical record to the operating system. Refer to "I-O Status Codes" in Section 3 for more information on I-O status.

TADS: Any USE procedure is not executed when a REWRITE statement that is compiled and executed in a TADS session fails.

Specifying the INVALID KEY and NOT VALID KEY Phrases

Transfer of control after the successful or unsuccessful execution of the REWRITE operation depends on the presence or absence of the INVALID KEY and NOT INVALID KEY phrases in the REWRITE statement.

For relative files in the random or dynamic access mode, you must specify the INVALID KEY phrase in the REWRITE statement. However, for relative files in sequential access mode, do not specify the INVALID KEY and the NOT INVALID KEY phrases in a REWRITE statement.

For relative and indexed files, if you do not specify an applicable USE AFTER STANDARD EXCEPTION procedure for the associated file-name, you must specify the INVALID KEY and the NOT INVALID KEY phrases.

TADS: Any USE procedure is not executed when a REWRITE statement that is compiled and executed in a TADS session fails.

Invalid Key Condition (Indexed Files)

For an indexed file, the invalid key condition exists under any of the following conditions:

- When the file is open in the sequential access mode, and the value of the prime record key of the record to be replaced is not equal to the value of the prime record key of the last record read from the file
- When the file is open in the dynamic or random access mode, and the value of the prime record key of the record to be replaced is not equal to the value of the prime record key of any record existing in the file
- When the value of an alternate record key of the record to be replaced, for which duplicates are not allowed, equals the value of the corresponding data item of a record already in the file

Results of Invalid Key Condition

The invalid key condition has the following effects on relative and indexed files:

- The execution of the REWRITE statement is unsuccessful.
- The record is not updated.
- The content of the record area is unaffected.
- The I-O status of the file associated with record-name-1 is set to a value that indicates that an invalid key condition has occurred. Refer to "I-O Status Codes" in Section 3 for more information. Status codes 21, 22, and 23 can indicate an invalid key condition.

Rules for Record Length

For fixed-length records, the number of character positions in the record referred to by record-name-1 must be equal to the number of character positions in the record being replaced.

For variable-length records, the number of character positions in the record referred to by record-name-1 must not be larger than the largest or smaller than the smallest number of character positions specified in the RECORD IS VARYING clause for the file-name that is associated with record-name-1.

If the number of character positions is larger or smaller than the number allowed, then

- The execution of the REWRITE statement is unsuccessful.
- The record is not updated.
- The content of the record area is unaffected.
- The I-O status of the file associated with record-name-1 is set to 44. Refer to "I-O Status Codes" in Section 3 for more information.

Record Replacement by the REWRITE Statement

For a relative file in random or dynamic access mode, the disk or disk pack logically replaces the record specified by the content of the relative key data of the file-name that is associated with record-name-1. If the file does not contain the record specified by the key, the invalid key condition exists.

For an indexed file in the sequential access mode, the record to be replaced is specified by the value of the prime record key. When REWRITE is executed, the value of the prime record key of the record to be replaced must equal the value of the prime record key of the last record read from this file.

For an indexed file in random or dynamic access mode, the record to be replaced is specified by the prime record key.

Effect of REWRITE on Indexed Files with Alternate Record Keys

Execution of the REWRITE statement for an indexed file that has an alternate record key occurs in one of the following ways:

- When the value of a specific alternate record key is not changed and when that key is the key of reference, the order of retrieval is unchanged.
- When the value of a specific alternate record key is changed, the subsequent order of retrieval of that record can be changed when that alternate record key is the key of reference. When duplicate key values are permitted, the record is logically positioned last within the set of duplicate records that contains the same alternate record key value as the alternate record key value that was inserted in the record.

Example

```
REWRITE RANDM-FIL-REC INVALID KEY PERFORM ERR-PARA.
```

In this example, RANDM-FIL-REC is rewritten. If the invalid key condition exists, ERR-PARA is performed.

RUN Statement

The RUN statement enables a program to initiate another program as an asynchronous, independent process.

```
RUN task-variable WITH section-name
    [ USING arithmetic-expression-1 [ ,arithmetic-expression-2 ] . . . ].
```

Explanation

task-variable

This specifies the task variable that is to be associated with the program specified in the section identified by section-name. The task variable must be declared as a data item in the Working-Storage section of the Data Division. For more information about task variables, refer to the USAGE clause in Section 4 and to Section 11.

section-name

This identifies the section in the Procedure Division that contains the name of the object code file that is to be initiated by this RUN statement. You must define the section-name in the Declaratives Section of the Procedure Division followed by a USE EXTERNAL statement that specifies the name of the object code file.

USING arithmetic-expression

Only parameters with arithmetic values can be referenced in the USING phrase. The formal parameters to which the values of the arithmetic expressions are passed must be described as single-precision or double-precision 77-level data items and must have a RECEIVED BY CONTENT clause.

To ensure that the passed value has the same precision as the corresponding formal parameter, the compiler truncates double-precision values to single precision and extends single-precision values to double precision. All values are passed with a scale of 0, regardless of the scale of the corresponding formal parameter. All values can be passed as normalized values.

Details

The RUN statement initiates a program as an asynchronous, independent process. An asynchronous process executes simultaneously with the program that initiated it. An independent process does not share the resources of the initiating program; thus, it can continue to execute if the initiating program is terminated.

Naming the Program to Be Initiated

You can specify the name of the program to be executed by the RUN statement in one of the following ways:

- Precede the CALL statement with a CHANGE statement that changes the NAME attribute of the task variable before the program is called.
- Define a mnemonic-name in the Special-Names paragraph of the Environment Division, and then use it in the USE EXTERNAL statement.
- Use the following steps:
  - Declare a data item in the Working-Storage section of the Data Division.
  - Name the data item in a USE EXTERNAL statement in the Declarative Section of the Procedure Division.
  - Assign the object code file title to the data item by using a MOVE statement in the Procedure Division.

For program examples that show how to name the program to be initiated, refer to Section 11.

Example

Following is an example:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. RUN-TASK-CALLER-WPARM.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
77 IND-TASK                  TASK.
01 WS-PROGID.
     05 WS-PID-1              PIC X(11) VALUE "OBJECT/".
     05 WS-PID2               PIC X(03) VALUE "C85".
     05 WS-PID3               PIC X(18) VALUE "/RUN/CALLED/WPARM.".
77 WS-77-BINARY              PIC 9(11) BINARY.
77 WS-77-REAL                REAL.
77 WS-77-BINARY-DBL          PIC 9(23) BINARY.
77 WS-77-DOUBLE              DOUBLE.
LOCAL-STORAGE SECTION.
LD HOW-PARAMS-PASSED.
77 LS-77-BINARY-CON          PIC 9(11) BINARY CONTENT.
77 LS-77-REAL-CON            REAL RECEIVED BY CONTENT.
77 LS-77-BINARY-DBL-CON      PIC 9(23) BINARY CONTENT.
```

```
77 LS-77-DOUBLE-CON         DOUBLE RECEIVED BY CONTENT.
PROCEDURE DIVISION.
DECLARATIVES.
RUN-A-PROCESS SECTION.
     USE EXTERNAL AS PROCEDURE WITH HOW-PARAMS-PASSED USING
         LS-77-BINARY-CON      LS-77-REAL-CON     LS-77-BINARY-DBL-CON
         LS-77-DOUBLE-CON.
END DECLARATIVES.
MAIN SECTION.
MAIN-PARA.
    CHANGE ATTRIBUTE NAME OF IND-TASK TO WS-PROGID.
    RUN IND-TASK WITH RUN-A-PROCESS USING
        WS-77-BINARY  WS-77-REAL  WS-77-BINARY-DBL  WS-77-DOUBLE.
    STOP RUN.
C85/RUN/CALLED/WPARM:
 IDENTIFICATION DIVISION.
 PROGRAM-ID. C85-RUN-CALLED-WPARM.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 LINKAGE SECTION.
 77 WS-1      PIC 9(11) BINARY.
 77 WS-2      REAL.
 77 WS-3      PIC 9(23) BINARY.
 77 WS-4      DOUBLE.
*
 PROCEDURE DIVISION USING
  WS-1, WS-2, WS-3, WS-4,
 MAIN SECTION.
 MAIN-PARAG.
   DISPLAY "THIS IS THE CALLED PROGRAM".
 EXIT PROGRAM.
```

Section 8
Procedure Division Statements S–Z

This section illustrates and explains the syntax of the Procedure Division statements. Statements beginning with the letters S through Z are listed in alphabetical order with the following information:

- A brief description of the function of the statement
- A syntax diagram for each format of the statement (if you need information on how to interpret a COBOL syntax diagram, refer to Appendix C).
- A statement of what portion of the syntax, if any, can be used interactively in a Test and Debug System (TADS) session
- An explanation of the elements in the syntax diagram
- Details, rules, and restrictions about the particular statement
- An example of the statement
- References to additional information relevant to the statement

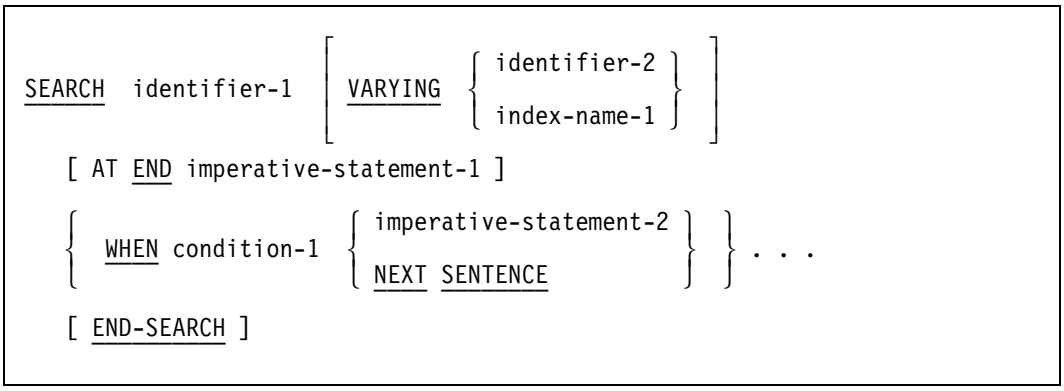
Detailed information about language elements common to many Procedure Division statements, such as user-defined names, literals, and identifiers is provided in Section 1. Concepts such as arithmetic and conditional expressions, and operations such as table handling, sorting, and merging are described in Section 5.

SEARCH Statement

The SEARCH statement searches a table for an element that satisfies a specified condition, and adjusts the value of the associated index to point to that table element. Use the SEARCH statement only for indexed tables; do not use it for subscripted tables.

Format	Use
Format 1	This format performs a serial search on an unordered table. An unordered table is not arranged in a particular order.
Format 2	This format performs a binary search on an ordered table. An ordered table is arranged in ascending or descending order.

Format 1: SEARCH . . . VARYING (Serial Search)



Explanation

identifier-1

This user-defined data item can be neither subscripted nor reference-modified. However, its description must contain an OCCURS clause that includes an INDEXED BY phrase. Refer to "OCCURS Clause" in Section 4 for more information.

VARYING

This phrase enables you to increment an index-name and the associated index for a table you are searching.

identifier-2

This element must refer to a data item that is described in one of the following ways:

- A data item that is declared in the Data Division with the USAGE IS INDEX clause
- A numeric elementary data item without any positions to the right of the assumed decimal point

This element cannot be subscripted by the first (or only) index-name specified in the INDEXED BY phrase of the OCCURS clause that is associated with identifier-1.

index-name-1

This user-defined word names an index associated with a specific table.

AT END imperative-statement-1

This option enables you to include an imperative statement that specifies an action to be taken if the at end condition occurs. If there are no more elements in the table and you specify the AT END phrase, then imperative-statement-1 is executed.

WHEN condition-1

This phrase indicates what condition must be met for the search of the table to be terminated. Condition-1 can be any conditional expression. Refer to "Conditional Expressions" in Section 5 for more information.

imperative-statement-2

This option enables you to include an imperative statement that specifies an action to be taken when all conditions in the WHEN phrase are satisfied.

NEXT SENTENCE

This phrase transfers control to the next executable sentence when all conditions in the WHEN phrase are satisfied.

END-SEARCH

This phrase delimits the scope of the SEARCH statement. If the END-SEARCH phrase is specified, the NEXT SENTENCE phrase must not be specified.

Details

A serial search begins with the current index setting. The object program inspects each table entry from the current index setting to the end of the table until it finds a match. If you want to search the entire table, the value of the current index setting must be equal to one. If the table is large, a serial search is very time consuming.

You can terminate the scope of the SEARCH statement by including:

- An END-SEARCH phrase at the same level of nesting
- A separator period
- An ELSE or END-IF phrase that is associated with a previous IF statement

The following paragraphs describe how specific phrases in the SEARCH statement syntax affect the operation of the SEARCH statement.

Identifier-1

The number of occurrences of identifier-1, the last of which is the highest permissible occurrence, is discussed under "OCCURS Clause" in Section 4. The following table indicates how the value of identifier-1 affects the SEARCH statement.

If the index-name associated with identifier-1 contains a value that corresponds to an occurrence number that is . . .	**Then the SEARCH statement . . .**
Greater than the highest permissible occurrence number for identifier-1	Stops immediately.
Not greater than the highest permissible occurrence number for identifier-1	Evaluates the conditions in the order that they are written, makes use of index settings (wherever specified), and determines the occurrence of items to be tested.

VARYING Phrase

The following paragraphs explain the effect of the VARYING phrase on the SEARCH statement.

- If you do not specify the VARYING phrase, the index-name used for the search is the first (or only) index-name specified in the INDEXED BY phrase of the OCCURS clause that is associated with identifier-1. Any other index-names for identifier-1 remain unchanged.
- If you specify the VARYING index-name-1 phrase, and if index-name-1 appears in the INDEXED BY phrase in the OCCURS clause referred to by identifier-1, that index-name is used for this search. If this is not the case, or if you specify the VARYING identifier-2 phrase, the first (or only) index-name given in the INDEXED BY phrase in the OCCURS clause referred to by identifier-1 is used for the search.
- If you specify the VARYING index-name-1 phrase, and if index-name-1 appears in the INDEXED BY phrase in the OCCURS clause referred to by another table entry, the occurrence number represented by index-name-1 is incremented at the same time and by the same amount as the occurrence number represented by the index-name associated with identifier-1.
- If you specify the VARYING identifier-2 phrase, and identifier-2 is an index data item, then the data item referred to by identifier-2 is incremented at the same time and by the same amount as the index associated with identifier-1. If identifier-2 is not an index data item, the data item referred to by identifier-2 is incremented by the value one (1) at the same time as the index referred to by the index-name associated with identifier-1.

AT END Phrase

If you specify the AT END phrase, imperative-statement-1 is executed. If you do not specify the AT END phrase, control passes to the end of the SEARCH statement.

Imperative-Statement-1, Imperative-Statement-2

After the execution of an imperative-statement-1, or an imperative-statement-2 that does not terminate with a GO TO statement, control passes to the end of the SEARCH statement. Refer to "GO TO Statement" in this section for more information.

WHEN Phrase

If none of the conditions are satisfied, the index-name for identifier-1 is incremented to refer to the next occurrence. The process is then repeated using the new index-name settings, unless the new value of the index-name setting for identifier-1 corresponds to a table element outside the permissible range of occurrence values. In this case, the search stops immediately.

If one of the conditions is satisfied upon evaluation, the search stops immediately. Control passes to the imperative statement associated with that condition, if present. However, if the NEXT SENTENCE phrase is associated with that condition, control passes to the next executable sentence. The index-name remains set at the occurrence that caused the condition to be satisfied.

If any of the conditions specified in the WHEN phrase cannot be satisfied for any setting of the index within the permitted range, control passes to imperative-statement-1 if the AT END phrase is specified. If the AT END phrase is not specified, control passes to the end of the SEARCH statement. In either case, the final setting of the index cannot be predicted.

If all conditions can be satisfied, the index indicates an occurrence that permits the conditions to be satisfied. In this case, control passes to imperative-statement-2, if specified, or to the next executable sentence, if the NEXT SENTENCE phrase is specified.

Figure 8–1 represents the action of a Format 1 SEARCH statement that contains two WHEN phrases.

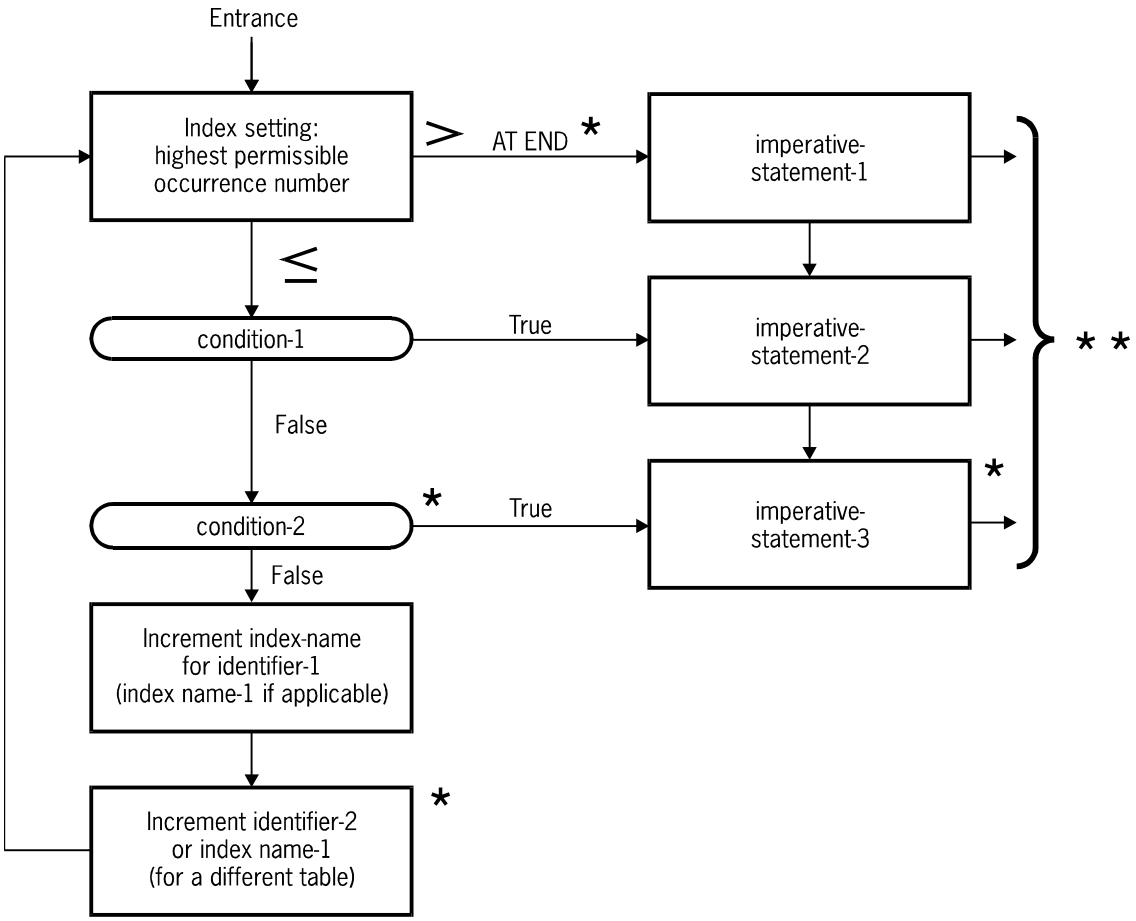


Figure 8–1. Format 1 SEARCH Statement with Two WHEN Phrases

* These operations are options included only when specified in the SEARCH statement.

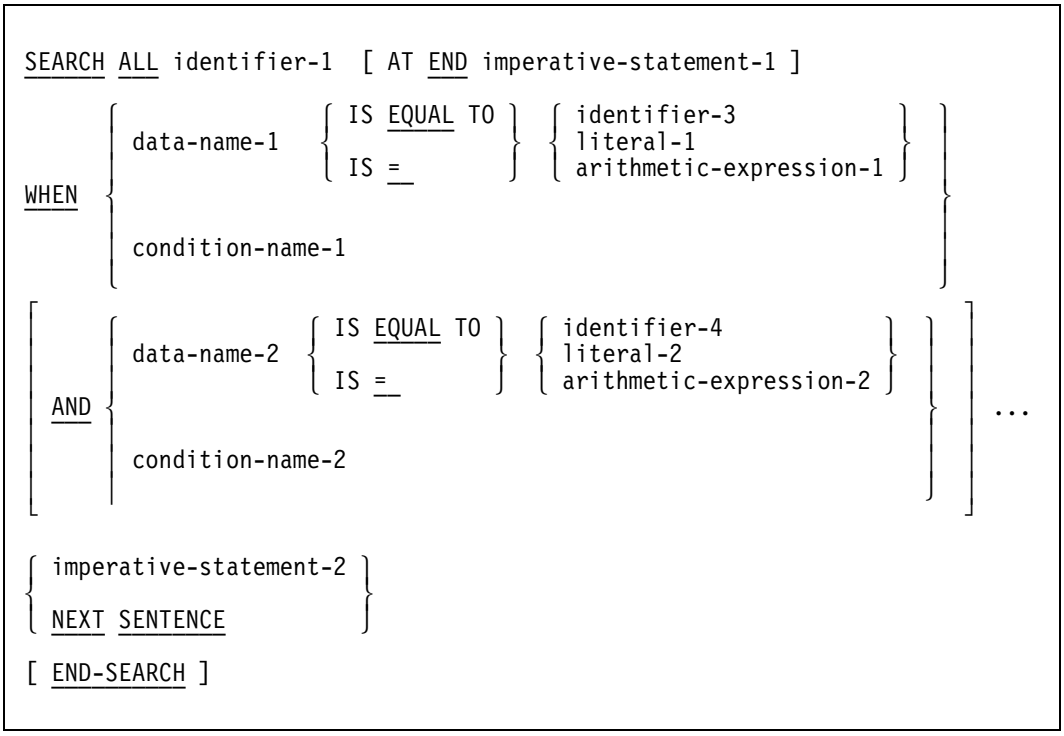
** Each of these steps transfers control to the end of the SEARCH statement, unless the imperative statement ends with a GO TO statement.

Example

```
05 TBL OCCURS 14 TIMES INDEXED BY J.
   SEARCH TBL VARYING ALT-INX AT END PERFORM END-PARA
      WHEN TBL(J) = "A"PERFORM ACCT-PARA
      WHEN TBL(J) = "C"PERFORM COST-PARA
      WHEN TBL(J) = "D"PERFORM DELIVERY-PARA
   END-SEARCH.
```

In this example, J points to the table element where a serial search begins. The entire table (from the current index setting) is searched for matches to the three WHEN conditions. If a match occurs, the imperative statement is performed and control passes to the END-SEARCH phrase. If a match does not occur, control passes to the AT END phrase. Because VARYING is specified, the occurrence number represented by ALT-INX is incremented at the same time and by the same amount as the occurrence number represented by J.

Format 2: SEARCH ALL (Binary Search)



Explanation

Refer to Format 1 for descriptions of the syntax elements AT END imperative-statement-1, imperative-statement-2, NEXT SENTENCE, and END-SEARCH.

SEARCH ALL

These words indicate that you are searching an entire ordered table. The OCCURS clause, which defines the table, must specify the ASCENDING or DESCENDING phrase.

identifier-1

This user-defined data item can be neither subscripted nor reference-modified. However, its description must contain an OCCURS clause that includes an INDEXED BY phrase. Identifier-1 must also contain the KEY IS phrase in its OCCURS clause. Refer to "OCCURS Clause" in Section 4 for more information.

WHEN
AND

These phrases provide the structure for searches that include matches of data-names or condition-names with identifiers, literals, and arithmetic expressions. This format permits only one WHEN phrase. Refer to "Arithmetic Expressions" and "Conditional Expressions" in Section 5 for more information.

data-name-1
data-name-2

The data-name associated with a condition-name must appear in the KEY IS phrase in the OCCURS clause referred to by identifier-1.

These elements can be qualified.

These must be subscripted by the first index-name associated with identifier-1 along with other subscripts, as required. They must be referred to in the KEY IS phrase in the OCCURS clause referred to by identifier-1.

IS EQUAL TO
IS =

These phrases are interchangeable. They test for the equality of a data-name or a condition-name and an identifier, a literal, or an arithmetic expression.

condition-name-1
condition-name-2

Each condition-name must have only a single value.

identifier-3
identifier-4

These identifiers, or the identifiers specified in arithmetic-expression-1 or arithmetic-expression-2, must not be

- Referred to in the KEY IS phrase in the OCCURS clause referred to by identifier-1
- Subscripted by the first index-name associated with identifier-1

literal-1
literal-2

These literals can have numeric or nonnumeric values.

arithmetic-expression-1
arithmetic-expression-2

Identifiers and literals that appear in arithmetic expressions must represent either numeric elementary items or numeric literals on which arithmetic operations can be performed. Refer to "Arithmetic Expressions" in Section 5 for more information.

Details

When you refer to either

- A data-name in the KEY IS phrase in the OCCURS clause that is referred to by identifier-1, or
- A condition-name that is associated with a data-name in the KEY IS phrase in the OCCURS clause referred to by identifier-1

you must refer to all preceding data-names in the KEY IS phrase of the OCCURS clause that are referred to by identifier-1 or their associated condition-names.

How A Binary Search Is Performed

In a binary search, the table elements must be in either ascending or descending order. When you specify the ASCENDING phrase, table elements are searched from the lowest to the highest value. When you specify the DESCENDING phrase, table elements are searched from the highest to the lowest value. The search occurs as follows:

1. The object program compares the item being searched for to the item in the middle of the table.
   a. If a match occurs, the search is completed.
   b. If a match does not occur, the object program determines whether the item being searched for is in the first or second half of the table. If the item being searched for is in the first half of the table, only that portion of the table is searched.
2. Next, the object program finds the item in the middle of the first half of the table.
   a. If a match occurs, the search is completed.
   b. If a match does not occur, the object program determines whether the item being searched for is in the first or second portion of this half of the table. If the item being searched for is in the first portion of this half of the table, only that part of the table is searched.
3. The object program continues to narrow down the items in the table that it compares to the item being searched for until one of the following occurs:
   a. A match occurs.
   b. The entire table has been searched and a match does not occur.

Operation of the SEARCH ALL Statement

In a Format 2 SEARCH statement, the results of the SEARCH ALL operation are predictable only when both of the following conditions are met:

- The data in the table are ordered as described in the KEY IS phrase of the OCCURS clause referred to by identifier-1.
- The contents of the key or keys referred to by the WHEN phrase are sufficient to identify a unique table element.

SEARCH ALL begins a binary search. In a binary search, the initial setting of the index-name for identifier-1 is ignored and its setting is varied during the search operation. This setting is restricted so that it never contains either of the following values:

- A value that exceeds the value that corresponds to the last element of the table
- A value that is less than the value that corresponds to the first element of the table

The length of the table is discussed under "OCCURS Clause" in Section 4.

In a binary search, the index-name used for the search is the first (or only) index-name specified in the INDEXED BY phrase of the OCCURS clause that is associated with identifier-1. Any other index-names for identifier-1 remain unchanged.

Example

```
05 TBL OCCURS 20 TIMES ASCENDING KEY IS KEY-1 INDEXED BY J.
   10 KEY-1        PIC X.
   10 FLD          PIC X(15).
SEARCH ALL TBL AT END PERFORM END-PARA
   WHEN KEY-1(J) = "X"
   DISPLAY FLD(J).
```

In this example, the entire table is searched regardless of which table element J points to. SEARCH ALL automatically initiates a binary search.

SEEK Statement

The SEEK statement repositions a mass-storage file for subsequent sequential access.

```
SEEK file-name RECORD.
```

This format is supported in the TADS environment.

Explanation

file-name

This name must identify a mass-storage file of sequential organization. The ACTUAL KEY clause must be specified in the FILE-CONTROL paragraph for file-name.

Details

The SEEK statement uses the value of the data item declared in the ACTUAL KEY clause for the file as the record number at which the file is to be repositioned. The next input/output (I/O) operation accesses the record associated with the record number used for the SEEK statement.

If the value of ACTUAL KEY item is less than or equal to zero, the file is repositioned to the first record of the file.

The ACTUAL KEY clause cannot reference a long numeric data item.

Execution of a SEEK statement does not cause the contents of the STATUS KEY data item to be updated, and cannot cause a USE routine to be executed.

SEND Statement

The SEND statement enables a program to send data to another program in the same multiprogramming mix or to a storage queue.

Format	Use
Format 1	This format is used to send data in a synchronous way to a program that is active in the same multiprogramming mix. Format 1 uses the CRCR (core-to-core) capabilities of the MCP.
Format 2	This format is used to send data in an asynchronous way to a storage queue. The sending program need not be present in the mix at the same time as the receiving program executes the RECEIVE statement. Format 2 uses the STOQUE (STOQ) capability of the MCP.

Format 1: Send Data Synchronously (CRCR)

```
SEND { identifier-1 }   FROM identifier-2
     { literal-1    }

[ ON EXCEPTION imperative-statement]

[ NOT ON EXCEPTION imperative-statement]

[ END-SEND ].
```

Explanation

identifier-1
literal-1

This is a nonnumeric data item that specifies the name of the receiving program. The value must be a file title that contains from 1 to 256 characters. It is not necessary to terminate the file title with a period (.).

If no usercode is specified, the usercode of the sending program is used. If the ON <family name> clause is used in the file title, it is ignored by the system in the comparison.

The receiving program must be present in the multiprogramming mix.

identifier-2

This field is referred to as the *sending field*. This identifier must reference either an alphanumeric data item or a long numeric data item contained in the sending program.

The size of identifier-2 is limited only by the amount of memory required by the sending and receiving programs. If the size of the sending and receiving fields is not equal, the smaller size is used. The data is truncated if necessary.

ON EXCEPTION imperative-statement

This clause provides an alternate statement to be performed if the receiving program is not ready when the SEND statement is executed (an *exception condition*.) If this clause is not used and the receiving program is not ready, the sending program is suspended until the receiving program is ready.

NOT ON EXCEPTION imperative-statement

This clause provides a statement to be performed after the data transfer has successfully occurred.

Details

This format of the SEND statement uses the CRCR (core-to-core) capabilities of the MCP. CRCR is a synchronous communication method that enables a program to send data to or receive data from another program that is present in the same multiprogramming mix.

When a program issues a SEND statement, the receiving program must issue a RECEIVE statement before the data transfer can occur. If the program designated to receive the data does not execute a RECEIVE statement, the sending program is suspended until the RECEIVE statement is executed.

To prevent the sending program from being suspended if the receiving program is not ready, you can specify an alternate course of action by including the ON EXCEPTION clause. Note, however, that you must not use this clause if the RECEIVE statement in the partner program has specified an ON EXCEPTION clause.

For more information on the CRCR functionality, refer to the *Task Management Programming Reference Manual*.

Format 2: Send Data Asynchronously (STOQ)

```
SEND TO { TOP    } identifier-1
        { BOTTOM }
[ ON EXCEPTION imperative-statement]
[ NOT ON EXCEPTION imperative-statement]
[ END-SEND ].
```

Explanation

TOP
BOTTOM

This determines whether the data is to be placed at the beginning of the queue or at the end of the queue.

identifier-1

This identifier must refer to a 01-level data description entry that describes a STOQ parameter block.

ON EXCEPTION imperative-statement

This clause provides an alternate statement to be performed if the specified queue is full (an *exception condition*.) If this clause is not used and an exception condition exists, the sending program is suspended until space becomes available in the queue.

NOT ON EXCEPTION imperative-statement

This clause provides a statement to be performed after the data has been successfully added to the queue.

Details

The STOQ capability of the MCP enables programs to communicate asynchronously by means of an external memory buffer called a storage queue. The MCP maintains a predefined number of queues in main memory. Programs add data to the storage queue by using the SEND statement. Any program can retrieve the data in the storage queue by using the RECEIVE statement. Data remains in a storage queue after the sending program terminates.

For more information about the STOQ functionality, refer to the *Task Management Programming Reference Manual*.

STOQ Parameter Block

To use storage queues, a program must contain a STOQ parameter block, which is a 01-level data description entry that identifies and describes

- The storage queue used for transferring data
- The data that is to be transferred to or from the storage queue

The data description entry for a STOQ parameter block must have the following format:

```
01 Identifier-1.
        02  Queue-name              PIC X(6).
        02  Entry-name-length       PIC 9(2) COMP.
        02  Entry-name              PIC X(nn).
        02  Entry-data-length       PIC 9(4) COMP.
        02  Entry-data              PIC X(nnnn).
```

The elements of a STOQ parameter block are described as follows:

queue-name

This is the programmatically assigned symbolic name of the queue to which the request pertains.

entry-name-length

This specifies the size of the optional entry-name field. A length of 0 (zero) indicates that no subqueue name exists.

entry-name

This is the name associated with the individual queue entry (optional). This name can be used to provide a substructure to a queue. This name also provides the means to access data elements that are at locations other than the top or bottom of the queue.

More than one item in the queue can have the same name; the entry-name need not be unique. Also, the name given to an item when it is stored by the SEND verb can be longer than the name specified in entry-name for a RECEIVE request.

entry-data-length

This indicates the size of the entry data area that contains the transaction to be accessed for a storage request. The size of the entry-data field can be from 0 to 9999 bytes, inclusive. For a SEND statement, the value must be filled in by the application before each SEND statement.

This field serves as the response area for a queue inquiry request made with the ACCEPT MESSAGE COUNT statement (see Format 3 of the ACCEPT statement).

entry-data

This data area contains the data to be added to the queue in a SEND operation or the data retrieved from the queue in a RECEIVE operation. The data can include any EBCDIC character, including embedded blanks and nonprintable values. This field is not applicable to a queue inquiry request issued by the ACCEPT MESSAGE COUNT statement.

Action of the SEND Statement

The SEND statement Format 2 causes data to be sent from the entry-data-field in the STOQ parameter block specified by identifier-1 to the queue named in that parameter block. If an entry-name is also specified in the parameter block, the data is identified by that entry-name within the queue.

- If SEND TO TOP is specified, the item in the entry-data field is stored at the beginning of the queue named in the queue-name field of the STOQ parameter block.
- If SEND TO BOTTOM is specified, the item is stored at the end of the queue named in the queue-name field of the STOQ parameter block.

When the request is complete, execution resumes at the next statement.

If insufficient space exists in the queue for the storage request, the ON EXCEPTION condition exists. In that case, the following rules apply:

- If you specified the ON EXCEPTION clause, the imperative-statement is executed.
- If you did not specify the ON EXCEPTION clause, the program is suspended until the specified item is placed into the queue.

If a queue with the name given in the queue-name field does not already exist when the SEND statement is executed, the queue is created.

Determining the Number of Messages in a Storage Queue

You can determine the number of entries in a storage queue by using the ACCEPT MESSAGE COUNT statement. This statement returns a count of the number of entries in a queue as an unsigned integer in the entry-data-length field of the STOQ parameter block. You can optionally specify an entry-name to determine the number of entries for that name or name group. For more information, refer to Format 3 of the ACCEPT statement.

SET Statement

The SET statement can be used to establish reference points for table-handling operations, change the status of external switches, change the value of conditional variables, or modify a file attribute.

Format	Use
Formats 1, 2	These formats establish reference points for table-handling operations.
Format 3	This format changes the status of external switches.
Format 4	This format changes the value of conditional variables.
Format 5	This format sets or modifies a file attribute.

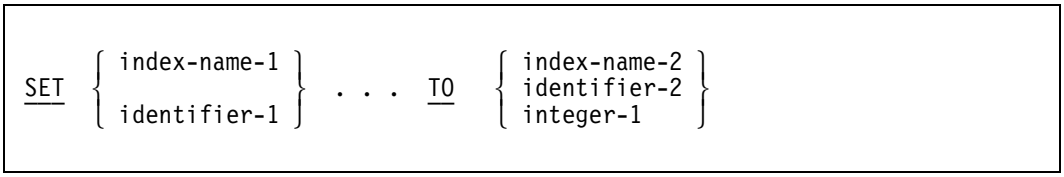
Rules for Formats 1 and 2

All references to index-name-1, identifier-1, and index-name-3 apply to all index-names and identifiers that precede the ellipsis marks.

Index-names are connected with a given table by being specified in the INDEXED BY phrase of the OCCURS clause for that table. See "OCCURS Clause" in Section 4 for details.

Formats 1 and 2 establish reference points for table-handling operations by setting indexes that are associated with table elements.

Format 1: SET . . . TO



Explanation

index-name-1
identifier-1

These elements identify the index-name or data item referred to by identifier-1 that you want to set.

TO
index-name-2
identifier-2
integer-1

These elements represent the value you want to assign to index-name-1 or the data item referred to by identifier-1.

Details

Index-names are user-defined words that name indexes associated with a specific table.

Identifiers must refer to an index data item or an elementary item that is described as an integer.

Integer-1 can be signed; however, it must be positive.

If you specify index-name-1, the value of the index after the execution of the SET statement must correspond to an occurrence number of an element in a table associated with index-name-1. The index value that is associated with an index-name can be set to an occurrence number that lies outside the range of its associated table, after the execution of a PERFORM or a SEARCH statement. The PERFORM and SEARCH statements are described earlier in this section.

If you specify index-name-2, the value of the index before the execution of the SET statement must correspond to an occurrence number of a table element that is associated with index-name-1.

Action of Set Statement, Format 1

In Format 1 operations, the following actions occur:

- Index-name-1 is set to a value that refers it to the table element that corresponds, in occurrence number, to the table element referred to by index-name-2, identifier-2, or integer-1. If identifier-2 refers to an index data item, or if index-name-2 is related to the same table as index-name-1, conversion does not occur.
- If identifier-1 refers to an index data item, it can be set equal to either the content of index-name-2, or identifier-2, where identifier-2 also refers to an index data item. In either case, conversion does not occur.
- If identifier-1 does not refer to an index data item, it can be set only to an occurrence number that corresponds to the value of index-name-2. In this case, neither identifier-2 nor integer-1 can be used.
- The value-setting process is repeated for each recurrence of index-name-1 or identifier-1, if specified. For each repetition, the value of index-name-2 or the data item referred to by identifier-2 is used as it was at the beginning of the execution of the statement. Any subscripting that is associated with identifier-1 is evaluated immediately before the value of the respective data item is changed. Refer to ''Table Handling'' in Section 5 for a description of subscripting.

Table 8–1 shows the valid operand combinations in Format 1 of the SET statement.

Table 8–1. Valid Operand Combinations for the SET . . . TO Statement

Sending Item	Receiving Item		
	Integer Data Item	**Index-Name**	**Index Data Item**
Integer Literal	No	Yes	No
Integer Data Item	No	Yes	No
Index-Name	Yes	Yes	Yes
Index Data Item	No	Yes	Yes

Examples

```
02 II USAGE IS INDEX.
02 IDM PIC 999.
02 TBL PIC 99 OCCURS 10 TIMES
      INDEXED BY J.
SET J TO II.
```

In this example, the index (J) is set to the value of the index data item referred to by II.

```
SET J TO IDM.
```

In this example, the index (J) is set to the value of the identifier referred to by IDM.

```
SET J TO 3.
```

In this example, the index (J) is set to the value of an integer.

```
SET IDM TO J.
```

In this example, the identifier referred to by IDM is set to the value of the index (J).

Format 2: SET . . . UP BY (DOWN BY)

```
SET  { index-name-3 } . . .  { UP BY   }  { identifier-3 }
                             { DOWN BY }  { integer-2    }
```

Explanation

index-name-3

This user-defined word names an index associated with a specific table.

UP BY
DOWN BY

These elements indicate an increase (UP BY) or decrease (DOWN BY) in a value.

identifier-3

This element must refer to an elementary numeric integer.

integer-2

This element can be signed.

Details

In Format 2, the value of the index both before and after the execution of the SET statement must correspond to an occurrence number of an element in the table that is associated with index-name-3.

The content of index-name-3 is increased (UP BY) or decreased (DOWN BY) by the value of integer-2, or the value of the data item referenced by identifier-3. This process is repeated for each recurrence of index-name-3. For each repetition, the value of the data item referenced by identifier-3 is used as it was at the beginning of the execution of the statement.

Examples

```
SET M UP BY C.
```

In this example, the data item referred to by identifier-3 (C) has a value of 4 and M points to the second element of the table. The content of index-name-3 (M) is increased by 4 when the SET statement is executed. After the execution of the SET statement, M points to the sixth element of the table.

```
SET M DOWN BY 1.
```

After the execution of the SET statement, M points to the fifth element of the table.

Format 3: SET an External Switch

```
SET  { { mnemonic-name-1 } . . .  TO  { ON  } } . . .
     {                                { OFF }
```

Explanation

mnemonic-name-1

This user-defined word must be associated with an external switch-name, the status of which can be changed. The external switches that can be referred to by the SET statement are described in Section 3 under "SWITCH-NAME Clause."

ON
OFF

These words represent the status of an external switch.

Details

The status of each external switch that is associated with the specified mnemonic-name-1 is modified so that an evaluation of an associated condition-name results in one of the following conditions:

- An on status, if the ON phrase is specified
- An off status, if the OFF phrase is specified

Section 3 contains a description of the switch-name clause.

Example

```
SET SW2 TO ON.
```

In this example, the status of the external switch (SW2) that is associated with mnemonic-name-1 is set to ON.

Format 4: SET a Condition TO TRUE

```
SET  { condition-name-1 } . . .  TO TRUE
```

Explanation

condition-name-1

This element is a user-defined word. Within a complete set of values, it is a name you assign to a specific value, or range of values, that a data item can assume. The data item itself is called a conditional variable.

In the SET statement, a condition-name indicates that the associated value will be moved to the conditional variable. Therefore, condition-name-1 must be associated with a conditional variable.

TRUE

This word indicates that a value is moved into the conditional variable.

Details

In the VALUE clause, the literal that is associated with condition-name-1 is inserted in the conditional variable, according to the rules for the VALUE clause. If more than one literal is specified in the VALUE clause, the conditional variable is set to the value of the first literal that appears in the clause. If you specify multiple condition-names, the results are the same as if a separate SET statement had been written for each condition-name-1. SET statement operations are executed in the order in which they are specified in the SET statement.

Example

In this example, the value assigned to FEB (02) is moved to MONTH.

```
DATA DIVISION.
  03 MONTH PIC 99.
     88 JAN VALUE 01.
     88 FEB VALUE 02.
        .
        .
        .
PROCEDURE DIVISION.
        .
        .
        .
  SET FEB TO TRUE.
```

Format 5: SET or Modify a File Attribute

```
SET file-name ( [ subscript-2 , ] file-attribute-name )

          { identifier-6                                  }
          { literal-2                                     }
          { arithmetic-expression                         }
          {                                               }
          { [ { VALUE } ]                                 }
          { [ {       }   mnemonic-attribute-value        }
          { [ { VA    } ]                                 }
          {                                               }
```

Explanation

file-name

This name identifies the file whose attribute is to be set or modified.

subscript-2

This name identifies the subfile of the file and is valid only for port files. The subscript can be an arithmetic-expression with the value of the expression identifying the subport.

file-attribute-name

This identifies the file attribute to be set or modified.

identifier-6
literal-2
arithmetic-expression
VALUE or VA

This portion of the SET statement determines the value of the file attribute after the file attribute is set or modified. For more information about file attributes in COBOL85, refer to Section 10. For information about specific file attributes and their values, refer to the *File Attribute Programming Reference Manual*.

Details

Mnemonic-attribute names can be used as data-names or procedure names provided they are not reserved words in COBOL85.

If a data-name has the same name as a mnemonic-attribute name, the value assigned to the attribute is determined by the use of the optional word VALUE. If the word VALUE is present, the attribute is set to the value of the mnemonic. if the word VALUE is omitted, the attribute is set to the current value of the data-name. Refer to "VALUE Clause" in Section 4 for more information.

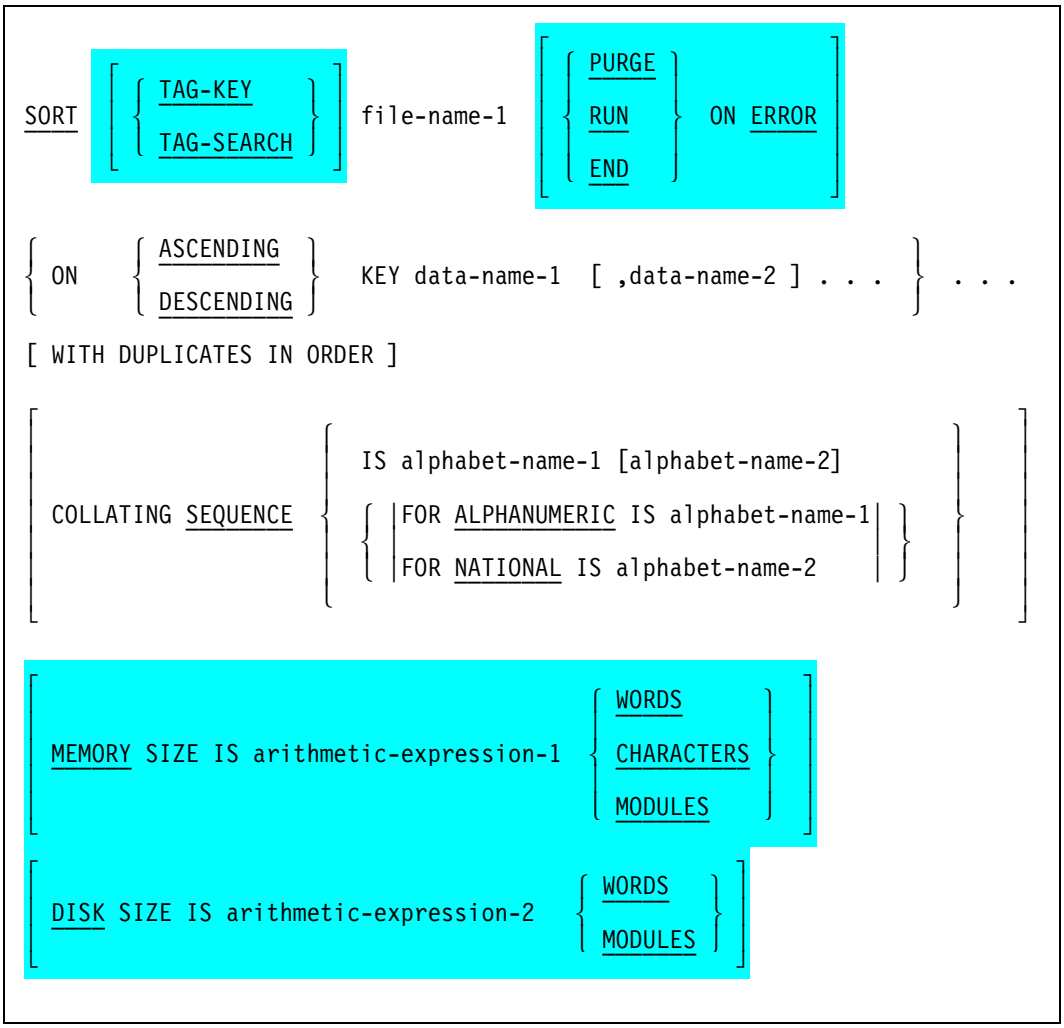
Format 5 of the SET statement is an obsolete element of COBOL85. The CHANGE statement is the preferred syntax.

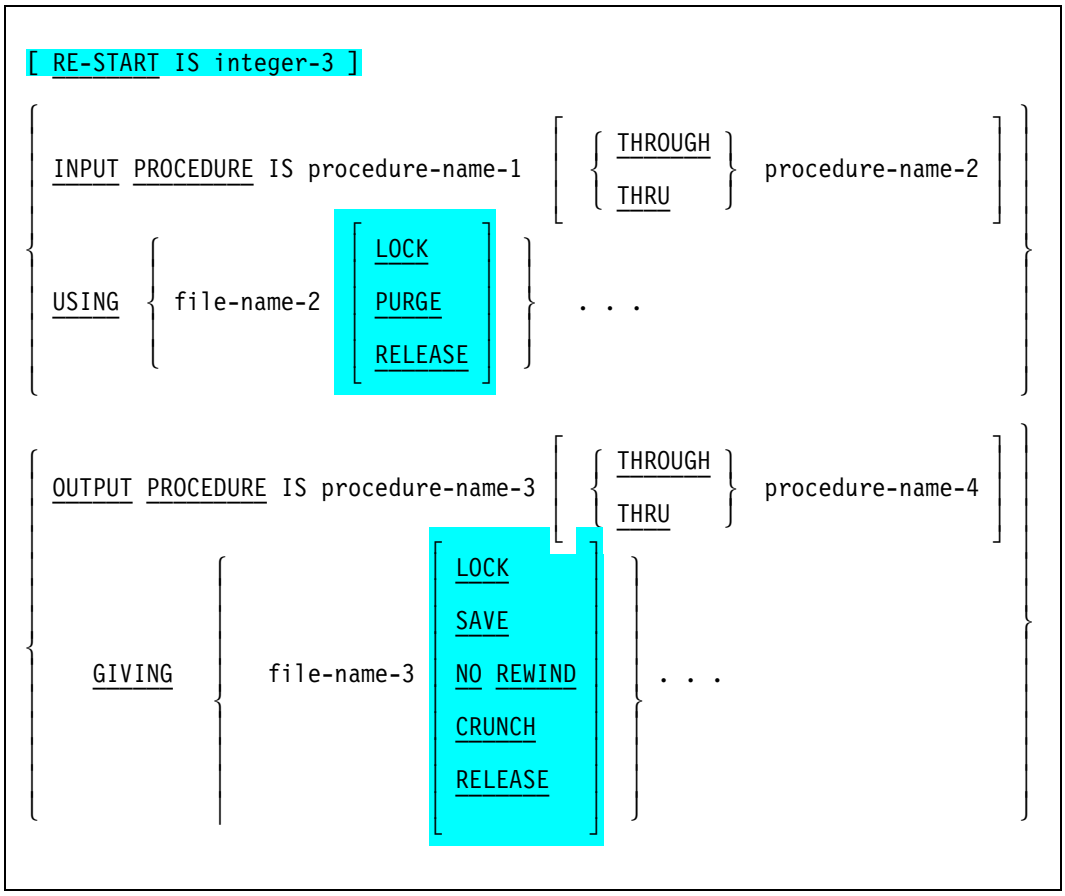
SORT Statement

The SORT statement

- Creates a sort file by executing an input procedure or by transferring records from another file.
- Sorts records in the sort file on a set of specified keys.
- Makes available each record from the sort file, in sorted order, to an output procedure or to an output file.

The SORT statement can appear anywhere in the Procedure Division except in the declarative portion. The syntax for the SORT statement is displayed on the following two pages.




[RE-START IS integer-3]
INPUT PROCEDURE IS procedure-name-1
THROUGH
THRU
procedure-name-2
USING
file-name-2
LOCK
PURGE
RELEASE
. . .
OUTPUT PROCEDURE IS procedure-name-3
THROUGH
THRU
procedure-name-4
GIVING
file-name-3
LOCK
SAVE
NO REWIND
CRUNCH
RELEASE
. . .

Explanation

TAG-KEY

This optional phrase specifies that sorting is performed on keys rather than on the entire record. The record numbers are placed in the sorted order in the GIVING file. The GIVING file is restricted to a record of eight DISPLAY digits.

The TAG-KEY option prohibits the use of the INPUT PROCEDURE and OUTPUT PROCEDURE clauses.

TAG-SEARCH

This optional phrase specifies that sorting is performed on keys rather than on the entire record. The records are placed in the GIVING file according to the sorted order of the record numbers.

The TAG-SEARCH option prohibits the use of the INPUT PROCEDURE and OUTPUT PROCEDURE clauses.

The TAG-SEARCH option is not supported for tape input files or for multiple-file input.

file-name-1

This refers to the sort file, which is an internal file.

This element is a user-defined word. It must be described in a sort-merge file description entry in the Data Division.

ON ERROR

The ON ERROR options enable you to have control over irrecoverable parity errors when input/output procedures are not present in a program.

PURGE causes all records in a block that contains an irrecoverable parity error to be dropped; processing is continued after a message is displayed on the ODT, giving the relative position in the file of the bad block.

RUN causes the bad block to be used by the program and provides the same message as defined for PURGE.

END causes a program termination; this is the default.

ASCENDING

When you specify the ASCENDING phrase, records contained in file-name-1 are sorted from the lowest value of data items identified by the key data-names to the highest value, according to the rules for comparison of operands in a relation condition. Refer to "Relation Conditions" in Section 5 for more information.

DESCENDING

When you specify the DESCENDING phrase, records contained in file-name-1 are sorted from the highest value of data items identified by the key data-names to the lowest value, according to the rules for comparison of operands in a relation condition.

KEY

Keys enable you to specify the order in which you want to sort a set of records.

The data-names following the word KEY are listed from left to right in order of decreasing significance without regard to how they are divided into KEY phrases. The leftmost data-name is the major key, the next data-name is the next most significant key, and so forth.

data-name-1
data-name-2 . . .

These user-defined words are key data-names. A key data-name is the name of a data item that is used as a sort key. Refer to "Rules for Key Data-Names" under the "Details" portion of this statement.

WITH DUPLICATES IN ORDER

This phrase determines the order in which duplicate records are returned after a SORT statement has been executed.

COLLATING SEQUENCE IS

This phrase enables you to specify alternate collating sequences. Alphabet-name-1 and alphabet-name-2 are user-defined words in the SPECIAL-NAMES paragraph of the Environment Division that assign a name to a specific character set and collating sequence.

MEMORY SIZE IS arithmetic-expression-1

This phrase is a guideline for allocating SORT memory area and overrides the same clause in the OBJECT-COMPUTER paragraph. You can allocate MEMORY SIZE as WORDS, CHARACTERS, or MODULES. If you do not specify MEMORY SIZE in the SORT statement or in the OBJECT-COMPUTER paragraph, the compiler assumes a default value of 12,000 words.

If the number of records to be sorted varies from run to run, you can allocate MEMORY SIZE by specifying arithmetic-expression-1. An arithmetic expression contains combinations of identifiers and literals, which are separated by arithmetic operators and parentheses. For details about arithmetic expressions, refer to Section 5.

DISK SIZE

This phrase is a guideline for allocating SORT disk area, and overrides the same clause in the OBJECT-COMPUTER paragraph. You can allocate DISK SIZE as WORDS or MODULES. If you do not specify DISK SIZE in the SORT statement or OBJECT-COMPUTER paragraph, the compiler assumes a default value of 900,000 words. One module of disk is the same as 1.8 million words of disk.

If the number of records to be sorted varies from run to run, you can allocate DISK SIZE by specifying arithmetic-expression-2. An arithmetic expression contains combinations of identifiers and literals, which are separated by arithmetic operators and parentheses. For details about arithmetic expressions refer to Section 5.

RE-START IS integer-3

The RE-START specification enables the sort intrinsic to resume processing at the most recent checkpoint after discontinuation of a program during the merge. The program restores and maintains variables, files, and everything that is necessary for the program to continue from the point of interruption.

The restart capability is implemented only for disk merges and sorts.

Select the type of RE-START action to be performed by choosing one of the following values for integer-3:

0	No restart capability.
1	Restart previous sort. The prior uncompleted sort must have been capable of a restart.
2	Allow restartable sort.
4 or 6	Allow a restartable sort, and enable extensive error recovery from I/O errors.
9	Restart previous sort if all input has been received. The prior uncompleted sort must have been capable of a restart.
10	Allow restartable sort after all input is received.
12 or 14	Options 4 and 10.

Refer to the MERGE section in the *System Software Utilities Operations Reference Manual* for more details on the RE-START capability of MERGE.

INPUT PROCEDURE IS

If you use a RELEASE statement to make records available to the file referred to by file-name-1, you can use an input procedure to select, modify, or copy those records.

procedure-name-1
procedure-name-2

These elements represent the beginning and ending of the range of an input procedure.

USING file-name-2

This phrase enables you to direct the SORT statement to open the file referred to by file-name-2 and to act upon it in the same manner as an input procedure. File-name-2 refers to a file that contains the records to be sorted.

You can specify up to eight file-names in the USING phrase.

If you specify an input procedure, do not specify the USING phrase.

OUTPUT PROCEDURE IS

If you use a RETURN statement to make sorted records available to the file referred to by file-name-1, you can use an output procedure to select, modify, or copy those records.

procedure-name-3
procedure-name-4

These elements represent the beginning and ending of the range of an output procedure.

GIVING file-name-3

File-name-3 refers to the output file. The GIVING phrase enables you to direct the SORT statement to open the file referred to by file-name-3 and to act upon it in the same manner as an output procedure.

You can specify up to eight file-names in the GIVING phrase.

If you specify an output procedure, do not specify the GIVING phrase.

THROUGH
THRU

These words are interchangeable. They connect two procedures that represent the range of an input or an output procedure.

LOCK
PURGE
RELEASE
SAVE
NO REWIND
CRUNCH

These options enable you to specify the type of close procedure to use on a file.

You can specify the LOCK, PURGE, and RELEASE options for file-name-2 (the USING phrase).

You can specify SAVE, LOCK, NO REWIND, CRUNCH, and RELEASE options for file-name-3 (the GIVING phrase).

For a description of these options, refer to "CLOSE Statement" in Section 6.

Details

A pair of file-names in the same SORT statement cannot be specified in the same SAME SORT AREA clause or the same SAME SORT-MERGE AREA clause. File-names associated with the GIVING phrase cannot be specified in the same SAME clause. Refer to "SAME Clause" under "I-O CONTROL Paragraph" in Section 3 and "Sort and Merge Constructs" in Section 5 for more information.

If you specify the DUPLICATES phrase and all key data items associated with one data record equal the corresponding key data items associated with one or more other data records, then the order of return of these records is one of the following:

- If there is no input procedure, records are returned in the order of the associated input files, as specified in the SORT statement. Within a given input file, records are returned in the order they are accessed.
- If there is an input procedure, records are returned in the order in which these records are released by the input procedure.

If you do not specify the DUPLICATES phrase and all key data items associated with one data record equal the corresponding key data items associated with one or more other data records, then the order of return of these records is undefined.

Action of the SORT Statement

The execution of the SORT statement consists of the following three phases:

1. Records are made available to the file referred to by file-name-1. The records are available because of the execution of RELEASE statements in the input procedure, or the implicit execution of READ statements for file-name-2. When this phase begins, the file referred to by file-name-2 must not be in the open mode. When this phase ends, the file referred to by file-name-2 is not in the open mode.
2. Records in the file referred to by file-name-1 are put in the order specified by the ASCENDING or DESCENDING options. The files referred to by file-name-2 and file-name-3 are not processed during this phase.
3. Records in the file referred to by file-name-1 are made available in sorted order. The sorted records are written to the file referred to by file-name-3, or are made available for processing by an output procedure with the execution of a RETURN statement. When this phase begins, the file referred to by file-name-3 must not be in the open mode. When this phase ends, the file referred to by file-name-3 is not in the open mode.

Rules for Key Data-Names

Key data-names can be qualified. In addition, the data items referenced by key data-names

- Must be described in a record description entry associated with file-name-1.
- Must be described in only one record description, if file-name-1 has multiple record descriptions. The same character positions referred to as a key data-name in one record description entry are taken as the key for all records in that file.
- Cannot be long numeric data items.
- Cannot be group items that contain variable-occurrence data items.
- Cannot be described by an entry that either contains an OCCURS clause or is subordinate to an entry that contains an OCCURS clause.

Collating Sequence

Alphabet-name-1 references an alphabet that defines an alphanumeric collating sequence.

Alphabet-name-2 references an alphabet that defines a national collating sequence.

The alphanumeric collating sequence that applies to the comparison of key data items for class alphabetic and class alphanumeric, and the national collating sequence that applies to the comparison of key data items of class national, are each determined separately at the beginning of the execution of the SORT statement in the following order or precedence:

1. Collating sequence established by the COLLATING SEQUENCE phrase, if specified, in this SORT statement

   The collating sequence associated with alphabet-name-1 applies to key data items of class alphabetic and alphanumeric; the collating sequence associated with alphabet-name-2 applies to key data items of class national.

2. Collating sequences established as the alphanumeric and national collating sequences

Input Procedure

The input procedure can consist of any procedure needed to select, modify, or copy the records that are made available (one at a time) by the RELEASE statement to the file referred to by file-name-1. The range of statements in the input procedure includes all statements that are executed because of a transfer of control by CALL, EXIT, GO TO, and PERFORM statements. The range also includes all statements in declarative procedures that are executed as a result of the execution of statements in the range of the input procedure. However, the range of the input procedure must not cause the execution of a MERGE, RETURN, or SORT statement.

If you specify an input procedure, control passes to the input procedure that precedes the file referred to by file-name-1, which puts the records in order (ascending or descending). The compiler inserts a return mechanism at the end of the last statement in the input procedure. When control passes to that last statement, the records that have been released to the file referred to by file-name-1 are sorted.

USING Phrase

If you specify the USING phrase, the size of the records contained in the file referred to by file-name-2 must not be larger than the largest record described for the file referred to by file-name-1.

File-name-2 and file-name-3 must be described in a file description entry (not in a sort-merge file description entry) in the Data Division. The files referred to by file-name-2 and file-name-3 can reside on the same multiple-file reel.

If you specify the USING phrase, all records in the file(s) referred to by file-name-2 are transferred to the file referred to by file-name-1. For each of the files referred to by file-name-2, the following actions occur when the SORT statement is executed:

1. File processing is initiated. The initiation is performed as if an OPEN statement with the INPUT phrase had been executed.
2. The logical records are obtained and released to the sort operation. Each record is obtained as if a READ statement with the NEXT and the AT END phrases had been executed.

   For a relative file, the content of the relative key data item is undefined after the SORT statement is executed if file-name-2 is not referred to by the GIVING phrase.
3. File processing is terminated. The termination is performed as if a CLOSE statement without optional phrases had been executed. The termination occurs before the file referred to by file-name-1 is put in either ascending or descending order.

These implicit functions are performed in a way that executes any associated USE AFTER EXCEPTION/ERROR procedures. However, the execution of such procedures must not cause the execution of any statement that manipulates the file referred to by, or that accesses the record area associated with, file-name-2.

Output Procedure

The output procedure can consist of any procedure needed to select, modify, or copy the records that are made available one at a time by the RETURN statement in sorted order from the file referred to by file-name-1. The range of statements in the output procedure includes all statements that are executed because of a transfer of control by CALL, EXIT, GO TO, and PERFORM statements. The range also includes all statements in declarative procedures that are executed as a result of the execution of statements in the range of the output procedure. The range of the output procedure must not cause the execution of a MERGE, RELEASE, or SORT statement.

If you specify an output procedure, control passes to the output procedure after the file referred to by file-name-1 puts the records in either ascending or descending order. The compiler inserts a return mechanism at the end of the last statement in the output procedure. When control passes to that last statement, the return mechanism terminates the sort operation and passes control to the next executable statement. Before entering the output procedure, the sort procedure reaches a point at which it can select the next record, in sorted order, when requested. The RETURN statements in the output procedure are the requests for the next record.

GIVING Phrase

If you specify the GIVING phrase, the size of the records contained in the file referred to by file-name-1 must not be larger than the largest record described for the file referred to by file-name-3.

If file-name-3 refers to an indexed file, the first specification of data-name-1 must be associated with an ASCENDING phrase. Also, the data item referred to by that data-name-1 must occupy the same character positions in its record as the data item that is associated with the prime record key for that file.

If you specify the GIVING phrase, all the sorted records are written in the file referred to by file-name-3 as the implied output procedure for the SORT statement. For each of the files referred to by file-name-3, the execution of the SORT statement causes the following actions:

1. File processing is initiated. The initiation is performed as if an OPEN statement with the OUTPUT phrase had been executed. Initiation occurs after the execution of any input procedure.
2. The sorted logical records are returned and written to the file. Records are written as if a WRITE statement without optional phrases had been executed.

   For a relative file, the relative key data item for the first record returned contains the value one (1); for the second record returned, the value two (2); and so forth. After the SORT statement is executed, the content of the relative key data item indicates the last record returned to the file.
3. File processing is terminated. The termination is performed as if a CLOSE statement without optional phrases had been executed.

These implicit functions are performed in a way that executes any associated USE AFTER EXCEPTION/ERROR procedures. However, the execution of such a USE procedure must not cause the execution of any statement that manipulates the file referred to by, or that accesses the record area associated with, file-name-3. On the first attempt to write beyond the externally defined boundaries of the file, any USE AFTER STANDARD EXCEPTION/ERROR procedure specified for the file is executed. If control is returned from that USE procedure or if such a USE procedure is not specified, the processing of the file is terminated as stated in step 3.

Space Fill of Records

Any record in the file referred to by file-name-2 that contains fewer character positions than the record length of the file referred to by file-name-1 is space filled from the right. Space fill begins with the first character position after the last character in the record, when that record is released to the file referred to by file-name-1.

Any record in the file referred to by file-name-3 that contains fewer character positions than the record length of the file referred to by file-name-1 is space filled from the right. Space fill begins with the first character position after the last character in the record, when that record is returned to the file referred to by file-name-3.

Examples

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MANUAL-COBOL85-SEC08-SORT.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT FILE-1      ASSIGN TO DISK.
  SELECT FILE-2      ASSIGN TO DISK.
  SELECT SRT-FIL     ASSIGN TO SORT.
DATA DIVISION.
FILE SECTION.
FD FILE-1.
01 F1REC.
   03 FILLER      PIC X(180).
FD FILE-2.
01 F2REC.
   03 FILLER      PIC X(180).
SD SRT-FIL.
01 SREC.
   03 FILLER      PIC X(10).
   03 ACC-NO      PIC 9(6).
   03 FILLER      PIC X(10).
   03 BAL-DUE     PIC 9(20).
PROCEDURE DIVISION.
BEGIN.
  SORT SRT-FIL
  ON ASCENDING KEY ACC-NO
  INPUT PROCEDURE IS PROC-1 THRU END-1
  OUTPUT PROCEDURE IS PROC-2 THRU END-2.
PROC-1.
  OPEN INPUT FILE-1.
PROC -1A.
  READ FILE-1 AT END GO TO END-1.
  RELEASE SREC FROM F1REC.
  GO TO PROC-1A.
END-1.
  CLOSE FILE-1.
PROC-2.
  OPEN OUTPUT FILE-2.
PROC-2A.
  RETURN SRT-FIL INTO F2REC AT END GO TO END-2.
  WRITE F2REC.
  GO TO PROC-2A.
END-2.
  CLOSE FILE-2.
STOP RUN.
```

In the example above, the input procedure occurs first. This procedure opens and reads FILE-1. Then, the procedure releases the record from F1REC to SORT. After the file is sorted, the output procedure begins. The output procedure opens FILE-2. Then, the sort record is returned to F2REC, and the record is written to the disk file.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MANUAL-COBOL85-SEC08-SORT.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE CONTROL.
  SELECT FILE-1      ASSIGN TO DISK.
  SELECT FILE-2      ASSIGN TO DISK.
  SELECT SRT-FIL     ASSIGN TO SORT.
DATA DIVISION.
FILE SECTION.
FD FILE-1.
01 F1REC.
  03 FILLER      PIC X(180).
FD FILE-2.
01 F2REC.
  03 FILLER      PIC X(180).
SD SRT-FIL.
01 SREC.
  03 FILLER      PIC X(10).
  03 ACC-NO      PIC 9(6).
  03 FILLER      PIC X(10).
  03 BAL-DUE     PIC 9(20).
PROCEDURE DIVISION
BEGIN.
  SORT SRT-FIL
      ON ASCENDING KEY BAL-DUE
      WITH DUPLICATES IN ORDER
      USING FILE-1
      GIVING FILE-2.
```

In this second example, the USING phrase and the GIVING phrase direct the SORT statement to open, read, and close the files in the same way that the input procedure and the output procedure opened, read, and closed the files in the previous example.

Because the DUPLICATES phrase is specified, duplicate records are returned in the order of the associated input files, as specified in the SORT statement. Within a given input file, the records are arranged in the order in which they are accessed from that file.

START Statement

The START statement provides a basis for logical positioning in relative or indexed files for the subsequent sequential retrieval of records.

This statement is partially supported in the TADS environment. Applicable exclusions are noted in this section.

```
                      ┌       ┌ IS EQUAL TO                   ┐               ┐
                      │       │ IS =                          │               │
                      │       │ IS GREATER THAN               │               │
                      │       │ IS >                          │               │
START file-name-1     │ KEY  <  IS NOT LESS THAN              >  data-name-1  │
                      │       │ IS NOT <                      │               │
                      │       │ IS GREATER THAN OR EQUAL TO   │               │
                      │       │ IS > =                        │               │
                      └       └                               ┘               ┘

[ INVALID KEY imperative-statement-1 ]

[ NOT INVALID KEY imperative-statement-2 ]

[ END-START ]
```

This format is supported in the TADS environment.

Explanation

file-name-1

This user-defined word represents the name of a file for which there is sequential or dynamic access. The file referenced by file-name-1 must be open in the input or I-O mode when the START statement is executed.

KEY

The KEY phrase enables you to specify where you want to retrieve records. If you do not specify the KEY phrase, the relational operator IS EQUAL TO is implied.IS EQUAL TO
IS =
IS GREATER THAN
IS >
IS NOT LESS THAN
IS NOT <
IS GREATER THAN OR EQUAL TO
IS > =
These relational operators are used for comparisons in the KEY phrase. Notice that IS EQUAL TO is synonymous with IS =; IS GREATER THAN is synonymous with IS >; and so forth.

data-name-1

This user-defined word can be qualified. Data-name-1 must point to the RELATIVE KEY clause, the ALTERNATE KEY clause, or the RECORD KEY clause.

INVALID KEY imperative-statement-1

The INVALID KEY phrase is required if you do not specify an applicable USE AFTER STANDARD EXCEPTION procedure for file-name-1.

The INVALID KEY phrase enables you to include an imperative statement that specifies an action to be taken when the key is invalid. A key is invalid if there is not a matching record in the file. For example, if you specify SMITH as the key for an indexed file and there is not a record called SMITH in the file, the key is invalid.

NOT INVALID KEY imperative-statement-2

The NOT INVALID KEY phrase enables you to include an imperative statement that specifies an action to be taken when the key is valid.

END-START

This phrase delimits the scope of the START statement.

Details

The execution of the START statement updates the value of the I-O status associated with file-name-1.

The execution of the START statement does not alter either of the following:

- The content of the record area
- The content of the data item referred to by the data-name specified in the DEPENDING ON phrase of the RECORD clause associated with file-name-1

TADS

Any USE procedure is not executed when a START statement that is compiled and executed in a TADS session fails.

Invalid Key Condition

When the START statement is executed and the file position indicator indicates that an optional input file is not present, the invalid key condition occurs and the START statement execution is unsuccessful.

Transfer of control after the successful or unsuccessful execution of the START operation depends on the presence or absence of the INVALID KEY and NOT INVALID KEY phrases in the START statement.

After the unsuccessful execution of a START statement, the file position indicator is set to indicate that a valid next record has not been established. Also, for indexed files, the key of reference is undefined.

Rules for Relative Files

If you specify data-name-1, it must be the data item specified in the RELATIVE KEY phrase in the ACCESS MODE clause of the associated file control entry. The comparison specified by the relational operator in the KEY phrase occurs between a key associated with a record in the file referred to by file-name-1 and the data item referred to by the RELATIVE KEY phrase of the ACCESS MODE clause associated with file-name-1. Numeric comparison rules apply. Refer to "Relation Conditions" in Section 5 for information about comparisons of numeric operands.

In addition, the following rules apply to comparisons:

- The file position indicator is set to the relative record number of the first logical record in the file whose key satisfies the comparison.
- If the comparison is not satisfied by any record in the file, the invalid key condition occurs and the execution of the START statement is unsuccessful.

Rules for Indexed Files

If you specify the KEY phrase, data-name-1 must refer to one of the following:

- A data item that is specified as a record key or alternate key associated with file-name-1
- Any alphanumeric or national data item whose leftmost character position within a record in a file corresponds to the leftmost character position of a record key associated with file-name-1, and whose length is not greater than the length of that record key

The comparison specified by the relational operator in the KEY phrase occurs between a key associated with a record in the file referred to by file-name-1 and a data item specified as follows:

- If you specify the KEY phrase, the comparison uses the data item referred to by data-name-1.
- If you do not specify the KEY phrase, the comparison uses the data item referred to by the RECORD KEY clause that is associated with file-name-1.

The comparison is made on the ascending key of reference, according to the collating sequence of the file. If the operands are of unequal size, the comparison proceeds as if the longer operand were truncated on the right, so that it equals the length of the shorter operand. All other numeric comparison rules apply.

In addition, the following rules apply to comparisons:

- The file position indicator is set to the value of the key of reference in the first logical record whose key satisfies the comparison.
- If the comparison is not satisfied by any record in the file, the invalid key condition exists and the execution of the START statement is unsuccessful.

A key of reference is established as follows:

- If you do not specify the KEY phrase, the prime record key specified for file-name-1 becomes the key of reference.
- If you do specify the KEY phrase, and you specify data-name-1 as a record key for file-name-1, that record key becomes the key of reference.
- If you specify the KEY phrase, and you specify a name other than the record key for file-name-1, the record key whose leftmost character position corresponds to the leftmost character position of the data item specified by data-name-1 becomes the key of reference.

The key of reference establishes the sequential order of records for the START statement. If the START statement executes successfully, the key of reference is also used for subsequent sequential READ statements. The READ statement is discussed earlier in this section.

Examples

```
FILE-CONTROL.
   SELECT EMP-FILE ASSIGN TO DISK
   ORGANIZATION IS RELATIVE
   ACCESS MODE IS DYNAMIC
   RELATIVE KEY IS KEY-1.
DATA DIVISION.
FILE SECTION.
FD EMP-FILE.
01 EMP-NUMBER.
   03  NAME         PIC X(10).
   03  ACC-NO       PIC X(6).
   03  BALANCE      PIC 9(6).
WORKING-STORAGE SECTION.
01 KEY-1            PIC 999.
PROCEDURE DIVISION.
BEGIN.
   OPEN I-O EMP-FILE.
   MOVE 10 TO KEY-1.
   START EMP-FILE KEY IS NOT LESS THAN KEY-1
         INVALID KEY PERFORM EDIT-KEY-TROUBLE.
   READ EMP-FILE NEXT AT END PERFORM ERR-PARA.
```

In this first example, the relative file EMP-FILE is positioned for sequential access. Before the START statement is executed, a number must be moved into the relative key. The file position indicator is set to the relative record number of the first logical record whose key is not less than KEY-1. If the comparison is not satisfied by any record in the file, the invalid key condition occurs, and EDIT-KEY-TROUBLE is performed.

```
FILE-CONTROL.
   SELECT MASTERFILE ASSIGN TO DISK
   ORGANIZATION IS INDEXED
   ACCESS MODE IS DYNAMIC
   RECORD KEY IS ACC-NO
   ALTERNATE KEY IS NAME1
   ALTERNATE KEY IS DEPT-NO WITH DUPLICATES.

DATA DIVISION.
FILE SECTION.
FD MASTERFILE.
01 REC.
   03  ACC-NO         PIC X(6).
   03  NAME1          PIC X(10).
   03  DEPT-NO        PIC X(3).
   03  BALANCE        PIC 9(6)V99.

PROCEDURE DIVISION.
BEGIN.
   OPEN INPUT MASTERFILE.
   MOVE "HARRIS" TO NAME1.
```

```
START MASTERFILE KEY IS EQUAL TO NAME1 INVALID KEY
        GO TO EDIT-KEY-TROUBLE.
READ MASTERFILE NEXT AT END PERFORM CLOSINGS.
        .
        .
        .
```

In this second example, the indexed file MASTERFILE is positioned for sequential retrieval. Before the START statement is executed, a value must be moved into the record key or an alternate key. The file position indicator is set to the value of the key of reference in the first logical record whose key satisfies the comparison. If the comparison is not satisfied by any record in the file, the invalid key condition occurs, and control is transferred to EDIT-KEY-TROUBLE.

STOP Statement

The STOP statement permanently or temporarily suspends the execution of the run unit. The STOP literal-1 construct of the STOP statement is an obsolete element and will be deleted from the next revision of standard COBOL.

Explanation

STOP RUN

The STOP RUN statement stops the execution of the run unit, and control passes to the operating system. If a STOP RUN statement appears in a consecutive sequence of imperative statements within a sentence, it must appear as the last statement in that sequence.

STOP literal-1

The STOP literal-1 statement suspends the execution of the run unit and displays literal-1 on the operator display terminal (ODT). Literal-1 must not be a figurative constant that begins with the word ALL. Section 1 contains a description of figurative constants.

If literal-1 is numeric, then it must be an unsigned integer.

The STOP statement can display more than one literal value.

Details

During the execution of a STOP RUN statement, an implicit CLOSE statement (without optional phrases) is executed for each file in the run unit that is in the open mode. USE statements that are associated with these files are not executed. CLOSE and USE statements are discussed in this section.

If the run unit has been accessing messages, the STOP RUN statement causes the message control system (MCS) to eliminate from the queue any message partially received by that run unit.

After the suspension of the run unit by STOP literal-1, you must reinitiate the run unit by typing *?OK* on your terminal. Then, execution of the run unit continues with the next executable statement.

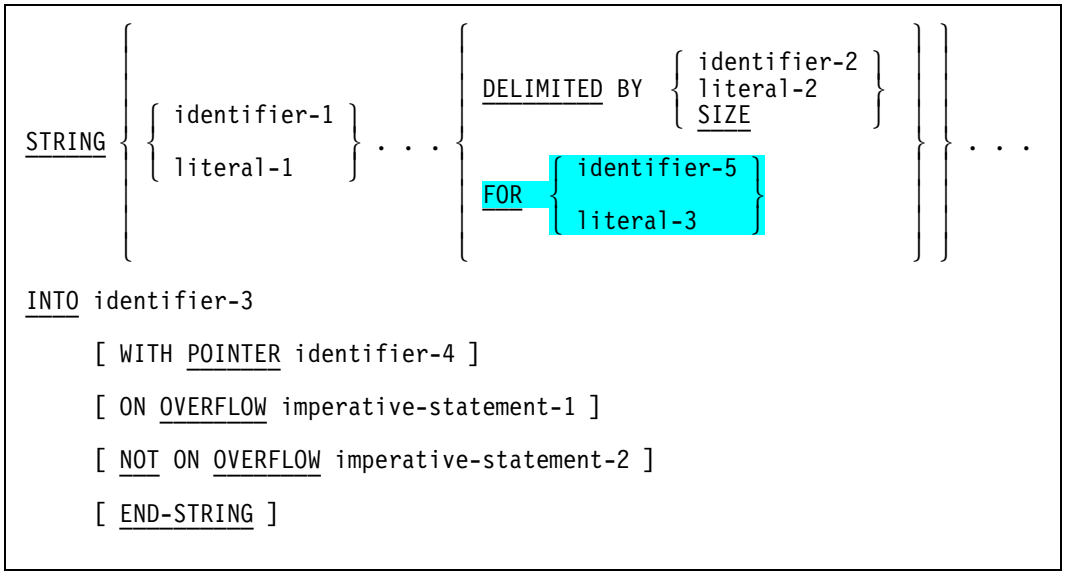
Examples

```
PROCEDURE DIVISION.
PARA-1.
    DISPLAY  "HELLO".
    STOP RUN.


PROCEDURE DIVISION.
P-1.
        STOP "STOP ""THIS ""FROM ""HAPPENING".
        DISPLAY "CONTINUED"
        STOP RUN.
```

In the first example, the STOP RUN statement terminates a program that displays HELLO on the ODT. In the second example, the STOP RUN statement terminates a program that displays STOP THIS FROM HAPPENING CONTINUED on the ODT.

STRING Statement

The STRING statement puts the partial or complete contents of one or more data items into a single data item.



Explanation

identifier-1
literal-1

These elements represent the sending field that contains the data you want to move to the data item referred to by identifier-3 (the receiving field).

Identifier-1 represents a data item you want to move. Identifier-1 cannot reference a long numeric data item. If it is an elementary numeric data item, it must be described as an integer without the symbol *P* in its PICTURE character-string.

Literal-1 represents the actual characters to be moved. Literal-1 cannot be a long numeric literal. If literal-1 is a figurative constant, it refers to an implicit one-character data item whose usage is DISPLAY or NATIONAL. Literal-1 cannot be a figurative constant that begins with the word ALL. For a description of figurative constants, see Section 1.

DELIMITED BY identifier-2
literal-2
SIZE

The DELIMITED BY phrase identifies the end of the data that will be moved.

Literal-2 or the content of the data item referred to by identifier-2 indicates the characters that delimit the move.

When literal-2 is a figurative constant it refers to an implicit one-character, data item whose usage is DISPLAY or NATIONAL. Literal-2 cannot be a figurative constant that begins with the word ALL. For a description of figurative constants, Section 1.

Identifier-2 is an elementary numeric data item, it must be described as an integer without the symbol *P* in its PICTURE character-string.

If you specify the SIZE phrase, the complete content of the data item referred to by identifier-1 or literal-1 is moved.

FOR identifier-5
FOR literal-3

This phrase specifies the number of characters to transfer.

INTO identifier-3

This phrase identifies the receiving data item. Identifier-3 must not represent an edited data item. In addition, it must not be reference-modified or include a JUSTIFIED clause in its description.

WITH POINTER identifier-4

This phrase indicates the value of the data item referred to by identifier-4. Identifier-4 must be described as an elementary numeric integer data item of sufficient size to contain a value equal to the size of the area referred to by identifier-3 plus 1. The symbol *P* cannot be used in the PICTURE character-string of identifier-4.

ON OVERFLOW imperative-statement-1

This phrase enables you to include an imperative statement that specifies an action to be taken when an overflow condition occurs.

NOT ON OVERFLOW imperative-statement-2

This phrase enables you to include an imperative statement that specifies an action to be taken when an overflow condition does not occur.

END-STRING

This phrase delimits the scope of the STRING statement.

All literals must be described as nonnumeric or national. All identifiers, except identifier-4 and identifier-5, must be described implicitly or explicitly with a usage of DISPLAY or NATIONAL. The category of all literals and identifiers, except identifier-4, identifier-5, and literal-3, must be the same.

Details

When the STRING statement is executed, characters from literal-1, or from the data item referred to by identifier-1, are transferred to the data item referred to by identifier-3. This transfer occurs according to the rules for alphanumeric-to-alphanumeric or national-to-national moves. However, space filling does not occur. MOVE operations are discussed in this section.

Effect of DELIMTED and FOR Phrases on Data Transfer

Each STRING statement must specify at least one DELIMITED phrase or FOR phrase. All data transfers occur in the sequence specified in the statement. Data transfers are repeated until all occurrences of literal-1, or data items referred to by identifier-1, are processed.

- If you specify the DELIMITED phrase without the SIZE phrase, the data item referred to by identifier-1, or the value of literal-1, is transferred to the receiving data item. This transfer begins with the leftmost character and continues from left to right until one of the following occurs:
  - The end of the sending data item is reached.
  - The end of the receiving data item is reached.
  - The characters specified by literal-2, or by the data item referred to by identifier-2, are encountered. The characters specified by literal-2 or by the data item referred to by identifier-2 are not transferred.
- If you specify the DELIMITED phrase with the SIZE phrase, the entire content of literal-1, or the content of the data item referred to by identifier-1, is transferred to the data item referred to by identifier-3. This data transfer continues until all data are transferred, or until the end of the data item referred to by identifier-3 is reached.
- If you specify the FOR phrase, the contents of identifier-1 are transferred to identifier-3 beginning with the leftmost character and continuing until one of the following occurs:
  - The end of identifier-3 is reached.
  - The number of characters specified by literal-3 or by the contents of identifier-5 have been transferred.

Effect of POINTER Phrase on Data Transfer

If you specify the POINTER phrase, the data item referred to by identifier-4 must have an initial value that is greater than zero before STRING is executed.

If you do not specify the POINTER phrase, the rules in the following paragraphs apply as if you had specified a data item with the initial value of 1 (one), which is referred to by identifier-4. When characters are transferred to the data item referred to by identifier-3, the moves occur in the following ways:

- As if each character is moved one at a time from the source into the character positions of the data item referred to by identifier-3.
- The value of the data item is determined by the value of the data item referred to by identifier-4 (if the value of the data item referred to by identifier-4 does not exceed the length of the data item referred to by identifier-3).
- As if the data item referred to by identifier-4 is increased by 1 (one) before the move of the next character or before the end of the STRING statement.

During execution of the STRING statement, the value of the data item referred to by identifier-4 is changed only as outlined here.

After the STRING statement is executed, only the portion of the data item referred to by identifier-3 that was referred to during the execution of the STRING statement is changed. All other portions of the data item referred to by identifier-3 will contain data that was present before the STRING statement was executed.

Overflow Condition

Before each move of a character to the data item referred to by identifier-3, if the value associated with the data item referred to by identifier-4 is either less than 1 (one) or exceeds the number of character positions in the data item referred to by identifier-3, the following actions occur:

- Data is not transferred to the data item referred to by identifier-3.
- The NOT ON OVERFLOW phrase, if specified, is ignored.
- Control is transferred to the end of the STRING statement, or to imperative-statement-1, if the ON OVERFLOW phrase is specified.
  - If control is transferred to imperative-statement-1, execution continues according to the rules for each statement specified in imperative-statement-1.
  - If a procedure-branching statement or a conditional statement that causes explicit transfer of control is executed, control is transferred according to the rules for that statement.
  - After imperative-statement-1 is executed, control is transferred to the end of the STRING statement.

If a STRING statement is executed with the NOT ON OVERFLOW phrase and the conditions in the previous paragraphs are not encountered, the following actions occur:

- Data is transferred according to the rules in the previous paragraphs.
- The ON OVERFLOW phrase, if specified, is ignored.
- Control is transferred to the end of the STRING statement or to imperative-statement-2, if the NOT ON OVERFLOW phrase is specified.
  - If control is transferred to imperative-statement-2, execution continues according to the rules for each statement specified in that imperative statement.
  - If a procedure-branching statement or a conditional statement that causes explicit transfer of control is executed, control is transferred according to the rules for that statement.
  - After imperative-statement-2 is executed, control is transferred to the end of the STRING statement.

If identifier-1 or identifier-2 occupies the same storage area as identifier-3 or identifier-4, or if identifier-3 and identifier-4 occupy the same storage area, the result of the execution of the STRING statement is undefined, even if these identifiers are defined by the same data description entry.

Related Information

The following table provides references for additional information related to this statement:

For information about . . .	Refer to . . .
Declaring the data items to be used as identifiers and literals for this statement	"USAGE Clause," "JUSTIFIED Clause," and "PICTURE Clause" in Section 4

Example

```
IDENTIFICATION DIVISION.
PROGRAM-ID. MANUAL-COBOL85-SEC08-STRING.
ENVIRONMENT DIVISION.

DATA DIVISION.
WORKING-STORAGE SECTION.
01 DEST PIC X(26).
01 REC-WS.
  05 A PIC X(3)   VALUE IS "DOG".
  05 B PIC X(10)  VALUE IS "NAME".
  05 C PIC X(4)   VALUE IS "XXXX".
77 PTR PIC 99 VALUE IS 1.

PROCEDURE DIVISION.
P1.
  STRING A, B, C DELIMITED BY SPACE INTO DEST
      WITH POINTER PTR
      ON OVERFLOW PERFORM NEW-FLD
  END-STRING.
  GO TO P1.
NEW-FLD.
  DISPLAY DEST.
  MOVE 1 TO PTR.
  STOP RUN.
```

In this example, let

A = DOG
B = NAME (blanks)
C = XXXX

Before the first execution of the STRING statement, PTR points to 1 (one). When the STRING statement is executed, DEST contains "DOGNAMEXXXX" and PTR points to 12. This execution of the STRING statement does not produce an overflow condition, so control passes to the imperative statement READ REC-1 INTO REC-WS, the END-STRING phrase, and the GO TO statement.

Because DEST contains only 26 characters, the third execution of a STRING statement (with the values shown here) produces an overflow condition. In this case, control passes to the imperative statement PERFORM NEW-FLD. The NEW-FLD procedure writes DEST, resets the pointer to 1, handles any excess data, and prepares a new field for more data.

SUBTRACT Statement

The SUBTRACT statement subtracts one, or the sum of two or more, numeric data items from one or more items. Then, it sets the values of one or more items equal to the result of the operation.

This statement is partially supported in the TADS environment. Supported syntax is noted in this section.

Format	Use
Format 1	The SUBTRACT . . . FROM format subtracts elementary numeric items.
Format 2	The SUBTRACT . . . FROM . . . GIVING format subtracts elementary numeric or numeric-edited items.
Format 3	The SUBTRACT CORRESPONDING format subtracts corresponding items.

The composite of operands must not contain more than 23 decimal digits.

- Format 1 determines the composite of operands by using all operands in a given statement.
- Format 2 determines the composite of operands by using all operands in a given statement except the data items that follow the word GIVING.
- Format 3 determines the composite of operands separately for each pair of corresponding data items.

The compiler ensures that enough places are carried so that significant digits are not lost during SUBTRACT operations.

Additional rules and explanations relating to the SUBTRACT statement appear in Section 5 under the following headings:

- "Arithmetic Expressions"
- "General Rules for Arithmetic Statements"
- "Multiple Results in Arithmetic Statements"
- "Statement Scope Terminators"

Format 1: SUBTRACT . . . FROM

```
SUBTRACT { identifier-1 } . . . FROM { identifier-2 [ ROUNDED ] } . . .
         { literal-1    }

   [ ON SIZE ERROR imperative-statement-1 ]

   [ NOT ON SIZE ERROR imperative-statement-2 ]

   [ END-SUBTRACT ]
```

TADS Syntax

```
SUBTRACT { identifier-1 } . . . FROM { identifier-2 [ ROUNDED ] } . . .
         { literal-1    }

   [ END-SUBTRACT ]
```

Explanation

identifier-1
literal-1

Each identifier must refer to a numeric elementary item. Each literal must be numeric. These elements represent the data item or literal you are subtracting from the value of identifier-2.

FROM

When you use Format 1, the values of the operands that precede the word FROM are added together, and the sum is stored in a temporary data item. The value in this temporary data item is subtracted from the value of the data item referred to by identifier-2. The result is stored in the data item referred to by identifier-2. This process is repeated for each occurrence of identifier-2 in the left-to-right order in which identifier-2 is specified.

identifier-2
ROUNDED

Identifier-2 refers to the data item from which you are subtracting identifier-1 or literal-1. Each identifier must refer to a numeric elementary item. The ROUNDED phrase enables you to round the result. Refer to "ROUNDED Phrase" in Section 5 for more information.

ON SIZE ERROR imperative-statement-1
NOT ON SIZE ERROR imperative-statement-2

The options ON SIZE ERROR and NOT ON SIZE ERROR enable you to include an imperative statement that specifies an action that will be taken if an error in the size of the result is or is not encountered. Refer to the "SIZE ERROR Phrase" in Section 5 for more information.

END-SUBTRACT

This phrase delimits the scope of the SUBTRACT statement.

Example

```
SUBTRACT A, B FROM C ROUNDED.
```

In this example, A and B are added together, and the sum is stored in a temporary data item. The value of this temporary data item is subtracted from C. Then, the result is rounded and stored in C.

Format 2: SUBTRACT . . . FROM . . . GIVING

```
          { identifier-1 }              { identifier-2 }
SUBTRACT  {              } . . . FROM   {              }
          { literal-1    }              { literal-2    }

 GIVING { identifier-3  [ ROUNDED ] } . . .

   [ ON SIZE ERROR imperative-statement-1 ]

   [ NOT ON SIZE ERROR imperative-statement-2 ]

   [ END-SUBTRACT ]
```

TADS Syntax

```
          { identifier-1 }              { identifier-2 }
SUBTRACT  {              } . . . FROM   {              }
          { literal-1    }              { literal-2    }

 GIVING { identifier-3  [ ROUNDED ] } . . .

   [ END-SUBTRACT ]
```

Explanation

Refer to Format 1 for descriptions of the syntax elements identifier-1, literal-1, the ON SIZE ERROR and NOT ON SIZE ERROR options, and the END-SUBTRACT phrase.

identifier-2
literal-2
ROUNDED

Each identifier must refer to a numeric elementary item. Each literal must be numeric. These elements represent the literal, or the value of the data item, from which you are subtracting identifier-1 or literal-1. The ROUNDED phrase enables you to round the result.

GIVING identifier-3

Each identifier following the word GIVING must refer to either an elementary numeric item or an elementary numeric-edited item. When you specify the GIVING phrase, the result of the subtraction is stored in each data item referred to by identifier-3.

Details

In Format 2, all literals and the values of the data items referred to by identifiers that precede the word FROM are added together. The sum is subtracted from literal-2, or the value of the data item referred to by identifier-2. The result of the subtraction is stored as the new content of each data item referred to by identifier-3.

Example

```
SUBTRACT 456 FROM 1000
    GIVING X
    ON SIZE ERROR PERFORM ERROR-PARA.
```

In this example, the literal 456 is subtracted from the literal 1000. The result (544) is stored in X. If the result contains more characters than X, an ON SIZE ERROR occurs and the ERROR-PARA is performed.

When you use the GIVING phrase, the data item referred to by identifier-3 (X in this example) can be a numeric-edited item.

Format 3: SUBTRACT CORRESPONDING

```
SUBTRACT { CORRESPONDING } identifier-1 FROM identifier-2  [ ROUNDED ]
         { CORR          }

   [ ON SIZE ERROR imperative-statement-1 ]

   [ NOT ON SIZE ERROR imperative-statement-2 ]

   [ END-SUBTRACT ]
```

TADS Syntax

```
SUBTRACT { CORRESPONDING } identifier-1 FROM identifier-2  [ ROUNDED ]
         { CORR          }

   [ ON SIZE ERROR imperative-statement-1 ]

   [ NOT ON SIZE ERROR imperative-statement-2 ]

   [ END-SUBTRACT ]
```

Explanation

Refer to Format 1 for descriptions of identifier-1 and literal-1 and the phrases ON SIZE ERROR, NOT ON SIZE ERROR, and END-SUBTRACT.

CORRESPONDING
CORR

The CORRESPONDING (or CORR) option enables you to subtract numeric data items in one group item from data items of the same name in another group item. Only elementary numeric data items can be subtracted with this phrase. Refer to the discussion of the CORRESPONDING phrase under "MOVE Statement" for rules that also apply to the SUBTRACT CORRESPONDING phrase.

CORR is an abbreviation for CORRESPONDING.

ROUNDED

Each identifier must refer to a group item. The ROUNDED phrase enables you to round the results.

Details

If you use Format 3, data items referred to by identifier-1 are subtracted from and stored in corresponding data items in identifier-2.

Example

```
DATA DIVISION.
01  group-1.
    05 A PIC 99.
    05 B PIC X(4).
    05 C PIC 9(8).
01  group-2.
    05 A PIC 99.
    05 D PIC 99.
    05 B PIC X(4).
    05 E PIC 9(4).
    05 C PIC 9(8).
    05 F PIC 9(8).
     .
     .
     .
    SUBTRACT CORR group-1 FROM group-2 ROUNDED END-SUBTRACT
```

In this example, the data items belonging to the group item group-1 are subtracted from the corresponding data items (A, B, and C) that belong to the group item group-2. The results are rounded. For details about rounding, refer to "ROUNDED Phrase" in Section 5.

UNLOCK Statement

The UNLOCK statement frees a common storage area that was previously restricted by a LOCK statement.

Explanation

event-identifier
lock-identifier

These identifiers indicate the data-name of the storage area that was previously restricted by a LOCK statement.

The event-identifier can be one or more of the following:

- The name of a data-item declared with the USAGE IS EVENT phrase. The data-name must be properly qualified and properly subscripted. See the USAGE clause in Section 4 for more information.
- A task attribute of type EVENT. The two event task attributes are ACCEPTEVENT and EXCEPTIONEVENT. For details about these task attributes, refer to the *Task Attributes Programming Reference Manual*.
- A file attribute of type EVENT. The three event file attributes are CHANGEEVENT, INPUTEVENT, and OUTPUTEVENT. For details about these files attributes, refer to the *File Attributes Programming Reference Manual*.

The lock-identifier is a data item declared with the USAGE IS LOCK clause. See the USAGE clause in Section 4 for more information.

Example

```
UNLOCK WS-01-EVENT.
```

UNLOCKRECORD Statement

The UNLOCKRECORD statement frees a record in a file that was previously restricted by a LOCKRECORD statement.

```
UNLOCKRECORD file-name
    [ON EXCEPTION imperative-statement-1]
    [NOT ON EXCEPTION imperative-statement-2]
    [END-LOCKRECORD]
```

Explanation

file-name

This user-defined word is the name of the file that contains the record to be unlocked. This file name must have been used in the previously executed LOCK statement.

ON EXCEPTION imperative-statement-1

This clause specifies an alternate statement to be performed if the UNLOCKRECORD statement is not successful.

NOT ON EXCEPTION

This clause specifies a statement to be performed after the record is successfully unlocked.

Details

The successful execution of the UNLOCKRECORD statement unlocks the record specified by the value contained in the data item referenced by the ACTUAL KEY clause in the File Control Entry of the Environment Division. No other locked records in the file are affected.

The UNLOCKRECORD statement can fail for any of the following reasons:

- The specified file
  - Does not exist.
  - Does not support locking (see the requirements for the file described with the explanation of the file-name syntax for the LOCK statement).
  - Is not open.
- The specified record key has an invalid or inconsistent value.
- The record to be unlocked is not locked.

Related Information

The following table provides references to information related to this topic.

For information about . . .	Refer to . . .
Locking a file	The LOCKRECORD statement.
I-O status codes resulting from error with locking and unlocking files	Table 3–6.

UNSTRING Statement

The UNSTRING statement separates contiguous data in a sending field and places the data into multiple receiving fields. Format 2 is a modified version of Format 1.

Format 1: UNSTRING . . . INTO

```
UNSTRING identifier-1

[ DELIMITED BY [ ALL ] { identifier-2 } [ OR [ ALL ] { identifier-3 } ] . . . ]
                       { literal-1   }               { literal-2   }
INTO { identifier-4 [ DELIMITER IN identifier-5 ]
  [ COUNT IN identifier-6 ] }. . .
     [ WITH POINTER identifier-7 ]
     [ TALLYING IN identifier-8 ]
     [ ON OVERFLOW imperative-statement-1 ]
     [ NOT ON OVERFLOW imperative-statement-2 ]
     [ END-UNSTRING ]
```

Explanation

identifier-1

The data item referenced by this identifier must be described, implicitly or explicitly, as alphanumeric or national.

The data item referenced by identifier-1 represents the sending area. Identifier-1 cannot be reference-modified.

DELIMITED BY

Each literal in this phrase must be a nonnumeric or a national literal.

Neither literal-1 nor literal-2 can be a figurative constant that begins with the word ALL.

Literal-1 or the data item referenced by identifier-2 can contain any character in the computer's character set.

The data items referenced by identifier-2 and identifier-3 must be described, implicitly or explicitly, as alphanumeric or national.

Each literal-1 or data item referenced by identifier-2 represents one delimiter. When a delimiter contains two or more characters, all of the characters must be present (in the order given in contiguous positions of the sending item) to be recognized as a delimiter. When you use a figurative constant as a delimiter, it represents a single-character nonnumeric or national literal.

When you specify two or more delimiters in the DELIMITED BY phrase, an OR condition must exist between them.

Each delimiter is compared to the sending field. If a match occurs, the character or characters in the sending field are considered to be a single delimiter. A character in the sending field cannot be considered a part of more than one delimiter.

Each delimiter is applied to the sending field in the sequence specified.

You cannot specify the DELIMITER IN phrase or the COUNT IN phrase unless you also specify the DELIMITED BY phrase.

ALL

If you specify the ALL phrase, one occurrence—or two or more contiguous occurrences—of literal-1 or the content of the data item referenced by identifier-2 is treated as only one occurrence, and this occurrence is moved to the receiving data item according to rule 4 described in "Rules for Data Transfer" in this section. This rule applies whether or not literal-1 is a figurative constant.

Without the ALL phrase, when any examination encounters two contiguous delimiters, the current receiving area is either space- or zero-filled, according to the description of the receiving area.

INTO identifier-4

The data item referenced by identifier-4 represents the receiving area. Identifier-4 can be described as alphabetic, alphanumeric, national, or numeric (except that the symbol *P* cannot be used in the PICTURE character-string). Identifier-4 must be described, implicitly or explicitly, as USAGE IS DISPLAY or USAGE IS NATIONAL.

DELIMITER IN identifier-5

The data item referenced by identifier-5 must be described, implicitly or explicitly, as alphanumeric or national.

Identifier-5 represents the receiving area for delimiters.

When you use a figurative constant as a delimiter, the delimiter must be a single-character, nonnumeric or national literal.

If two contiguous delimiters are encountered, the current receiving area is as follows:

- Space-filled if the area is described as alphabetic, alphanumeric, or national
- Zero-filled if the area is described as numeric

COUNT IN identifier-6

The data item referenced by identifier-6 must be described as an integer data item (except that the symbol *P* cannot be used in the PICTURE character-string).

The data item referenced by identifier-6 represents the number of characters in the sending item that have been isolated by the delimiters for the move to the receiving item. This value does not include a count of the delimiter character or characters.

WITH POINTER identifier-7

The data item referenced by identifier-7 must be described as an elementary numeric integer data item of sufficient size to contain a value equal to 1 plus the size of the data item referenced by identifier-1. The symbol *P* cannot be used in the PICTURE character-string of identifier-7.

The data item referenced by identifier-7 contains a value that indicates a relative character position within the area referenced by identifier-1.

The content of the data item referenced by identifier-7 is incremented by one for each character examined in the data item referenced by identifier-1. When the execution of an UNSTRING statement with a POINTER phrase is completed, the content of the data item referenced by identifier-7 contains a value equal to the initial value plus the number of characters examined in the data item referenced by identifier-1.

The program must initialize the contents of the data item used in the POINTER phrase (identifier-7).

TALLYING IN identifier-8

The data item referenced by identifier-8 must be described as an integer data item (except that the symbol *P* cannot be used in the PICTURE character-string).

The data item referenced by identifier-8 is a counter that is incremented by 1 for each occurrence of the data item referenced by identifier-4 that is accessed during the UNSTRING operation.

The program must initialize the contents of the data items in the TALLYING phrase (identifier-8).

ON OVERFLOW imperative-statement-1
NOT ON OVERFLOW imperative-statement-2

When an overflow condition exists, the UNSTRING operation is terminated. If you specify an ON OVERFLOW phrase, the imperative statement included in the ON OVERFLOW phrase is executed. If you do not specify an ON OVERFLOW phrase, control passes to the next executable statement.

When you specify a NOT ON OVERFLOW phrase and an overflow condition does not exist, control passes to imperative-statement-2.

Either of the following situations causes an overflow condition:

- An UNSTRING statement is initiated, and the value in the data item referenced by identifier-7 is less than 1 or greater than the size of the data item referenced by identifier-1.
- During execution of an UNSTRING statement, all data receiving areas have been acted upon, and the data item referenced by identifier-1 contains characters that have not been examined.

END-UNSTRING

This phrase delimits the scope of the UNSTRING statement.

Overlapping Operands

The result of the execution of the UNSTRING statement is undefined, even if the overlapping data items are defined by the same data description, if any of the following conditions exist:

- A data item referenced by identifier-1, identifier-2, or identifier-3 occupies the same storage area as a data item referenced by identifier-4, identifier-5, identifier-6, identifier-7, or identifier-8.
- A data item referenced by identifier-4, identifier-5, or identifier-6 occupies the same storage area as a data item referenced by identifier-7 or identifier-8.
- A data item referenced by identifier-7 occupies the same storage area as a data item referenced by identifier-8.

Rules for Data Transfer

When the UNSTRING statement is initiated, the current receiving area is the data item referenced by identifier-4. Data is transferred from the data item referenced by identifier-1 to the data item referenced by identifier-4 according to the following rules:

- If you specify the POINTER phrase, the string of characters referenced by identifier-1 is examined beginning with the relative character position indicated by the contents of the data item referenced by identifier-7.
- If you do not specify the POINTER phrase, the string of characters is examined beginning with the leftmost character position.
- If you specify the DELIMITED BY phrase, the examination proceeds left to right until either a delimiter specified by the value of literal-1 or the value of the data item referenced by identifier-2 is encountered.
- If you do not specify the DELIMITED BY phrase, the number of characters examined equals the size of the current receiving area. However, if the sign of the receiving item is defined as occupying a separate character position, the number of characters examined is one less than the size of the current receiving area.

- If the end of the data item referenced by identifier-1 is encountered before the delimiting condition is met, the examination terminates with the last character examined.
- The characters thus examined (excluding any delimiting character or characters) are treated as elementary alphanumeric or national data items. They are moved into the current receiving field according to the rules of the MOVE statement. See "MOVE Statement" in this section for details.
- Note that if you specify delimiters and identifier-1 begins with the specified delimiter or delimiters, the first receiving field—the data item referenced by identifier-4—is either zero- or space-filled, according to the description of identifier-4. If this is not desired, do the following:

  1. INSPECT identifier-1 TALLYING the LEADING delimiter or delimiters.
  2. UNSTRING identifier-1 using the POINTER phrase, setting identifier-7 to 1 more than the count tallied by the INSPECT statement.

- If you specify the DELIMITER IN phrase, the delimiting character or characters are treated as elementary alphanumeric or national data items and are moved into the data item referenced by identifier-5 according to the rules of the MOVE statement. See "MOVE Statement" in this section for details. If the delimiting condition is the end of the data item referenced by identifier-1, the data item referenced by identifier-5 is space-filled.
- If you specify the COUNT IN phrase, a value equal to the number of characters thus examined (excluding any delimiter character or characters) is moved into the area referenced by identifier-6 according to the rules for an elementary move (refer to "MOVE Statement" in this section).
- If you specify the DELIMITED BY phrase, the string of characters is further examined beginning with the first character to the right of the delimiter.
- If you do not specify the DELIMITED BY phrase, the string of characters is further examined beginning with the character to the right of the last character transferred.
- After data is transferred to the data item referenced by identifier-4, the current receiving area is the data item referenced by identifier-7. The actions described in paragraphs 2 through 6 are repeated until all the characters are exhausted in the data item referenced by identifier-1, or until there are no more receiving areas.

Example

In the following example, DUMMY-ITEM is a data item that contains the string:

```
+,=,?"9,abc
```

The following procedure unstrings DUMMY-ITEM:

```
UNSTRING DUMMY-ITEM
DELIMITED BY ","
OR QUOTES
INTO
PLUS-SIGN, DELIMITER IN COMMA-MARK,
EQUAL-SIGN,
QUESTION-MARK, DELIMITER IN QUOTE-MARK,
NINE, COUNT IN COUNTER-ITEM,
PLUS-WORD,
TIMES-WORD,
TALLYING IN START-END-POSITIONS,
ON OVERFLOW
DISPLAY  "WE FOUND MORE THAN"
START-END-POSITIONS  "ITEMS TO UNSTRING".
```

The following data items result from this procedure:

- PLUS-SIGN contains +
- EQUAL-SIGN contains =
- QUESTION-MARK contains ?
- NINE contains 9
- PLUS-WORD contains abc
- TIMES-WORD is empty
- COMMA-MARK contains the comma delimiter (,)
- QUOTE-MARK contains the quote delimiter (")
- COUNTER-ITEM contains 1
- START-END-POSITIONS contains 5

Format 2: UNSTRING . . . INTO . . . FOR

```
UNSTRING identifier-1

INTO { identifier-2 FOR { identifier-3 } } . . .
     {                  { literal-1    }

   [ WITH POINTER identifier-4 ]

   [ TALLYING IN identifier-5 ]

   [ ON OVERFLOW imperative-statement-1 ]

   [ NOT ON OVERFLOW imperative-statement-2 ]

   [ END-UNSTRING ]
```

Explanation

Refer to the description of Format 1 for an explanation of the INTO, WITH POINTER, TALLYING IN, and END-UNSTRING phrases, and the syntax element identifier-1.

identifier-2

The same rules apply to this data item as to identifier-4 of Format 1.

FOR
identifier-3
literal-1

This phrase specifies the number of characters to transfer. Identifier-3 must be described as an elementary numeric integer data item (except that the symbol *P* cannot be used in the PICTURE character-string).

ON OVERFLOW imperative-statement-1
NOT ON OVERFLOW imperative-statement-2

Either of the following situations causes an overflow condition:

- An UNSTRING statement is initiated, and the value in the data item referenced by identifier-3 is less than 1 or greater than the size of the data item referenced by identifier-1.
- During execution of an UNSTRING statement, all data receiving areas have been acted upon, and the number of characters acted upon is less than the value of the data item referenced by identifier-3 or the value of literal-1.

When an overflow condition exists, the UNSTRING operation is terminated. If you specify an ON OVERFLOW phrase, the imperative statement included in the ON OVERFLOW phrase is executed. If you do not specify an ON OVERFLOW phrase, control passes to the next executable statement.

When you specify a NOT ON OVERFLOW phrase and an overflow condition is not encountered, control is transferred to the statement specified in imperative-statement-2.

Details

Literal-1 or the data item referenced by identifier-3 specifies the number of characters in identifier-1 that are moved to identifier-2. If the number of characters remaining in the data item referenced by identifier-1 is less than the number of characters specified by literal-1 or the data item referenced by identifier-3, the short field is transferred according to rule 3 described in "Rules for Data Transfer" in Format 1 of this section.

Example

In the following example, the data item DUMMY-ITEM contains the string:

```
CALIFMINALNEBCONNIOWAOHTX
```

Before execution of the following procedure, the pointer P-WORD contains a value of 1:

```
UNSTRING DUMMY-ITEM INTO FIRST-ITEM FOR 5,
                         SECOND-ITEM FOR 3,
                         THIRD-ITEM FOR 2,
                         FOURTH-ITEM FOR 3,
                         FIFTH-ITEM FOR 4,
                         SIXTH-ITEM FOR 4,
WITH POINTER P-WORD,
TALLYING IN START-END-POSITIONS,
ON OVERFLOW
   DISPLAY "ONLY" P-WORD "POSITIONS WERE EXAMINED".
```

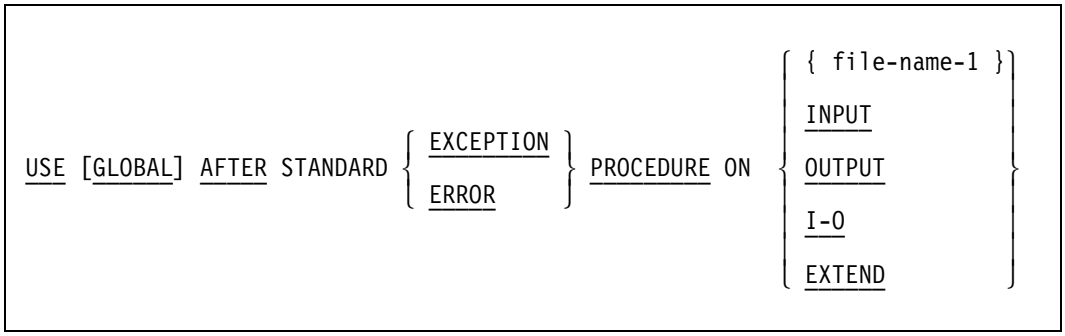
The following data items result from this procedure:

- FIRST-ITEM contains CALIF
- SECOND-ITEM contains MIN
- THIRD-ITEM contains AL
- FOURTH-ITEM contains NEB
- FIFTH-ITEM contains CONN
- SIXTH-ITEM contains IOWA
- START-END-POSITIONS contains 6
- P-WORD contains 22

USE Statement

Format	Use
Format 1	The USE AFTER format defines the conditions for the execution of USE procedures by the I/O control system for I/O error handling.
Format 2	The USE PROCEDURE format enables untyped procedures or subroutines to be declared COMMON or EXTERNAL.
Format 3	The USE AS INTERRUPT PROCEDURE format specifies a declarative statement as an interrupt procedure.
Format 4	The USE AS EPILOG PROCEDURE format specifies a declarative statement as an epilog procedure.

Format 1: USE AFTER



Explanation

USE AFTER

The USE AFTER statement is never executed itself; it merely defines the conditions calling for the execution of the USE procedures.

A USE AFTER statement must immediately follow a section header in the declaratives portion of the Procedure Division and must appear in a sentence by itself. The remainder of the section must consist of any number of procedural paragraphs that define the procedures to be used.

GLOBAL

The GLOBAL option enables any programs nested within the program that contains the GLOBAL option to use the USE procedures, if applicable.

ERROR
EXCEPTION

The words ERROR and EXCEPTION are synonymous and can be used interchangeably.

file-name-1

The files implicitly or explicitly referenced in a USE AFTER statement need not all have the same organization or access.

The appearance of file-name-1 in a USE AFTER statement must not cause the simultaneous request for execution of more than one USE AFTER procedure. That is, when file-name-1 is specified explicitly, no other USE statement can apply to file-name-1.

INPUT
OUTPUT
I-O
EXTEND

The INPUT, OUTPUT, I-O, and EXTEND phrases can each be specified only once in the declaratives portion of a given Procedure Division.

Details

Declarative procedures can be included in any COBOL source program whether or not the program contains, or is contained in, another program. Refer to Section 5 for information about declarative procedures and compiler-directing statements.

A declarative is invoked when any of the conditions described in the USE AFTER statement that prefaces the declarative occur while the program is being executed. Only a declarative in the separately compiled program, which contains the statement that caused the qualifying condition, is invoked when any of the conditions described in the USE statement, which prefaces the declarative, occurs while that separately compiled program is being executed. If a qualifying declarative does not exist in the separately compiled program, the declarative is not executed.

A declarative procedure cannot reference nondeclarative procedures when the program employs any of the following: Report Writer, a USE statement with the GLOBAL option, the USE AS INTERRUPT statement, or the USE AS EPILOG statement. Procedure-names associated with a USE AFTER statement can be referenced in a different declarative section, or in a nondeclarative procedure only with a PERFORM statement.

The procedures associated with the USE AFTER statement are executed by the input-output control system after completing the standard error retry routine if the execution of the input-output routine was unsuccessful. However, an AT END phrase can take precedence.

Rules

The following rules concern the execution of the procedures associated with the USE AFTER statement:

- If you specify file-name-1, the associated procedure is executed when the condition described in the USE AFTER statement occurs to the file.
- If you specify INPUT, the associated procedure is executed when the condition described in the USE AFTER statement occurs for any file that is open in the input mode, or that is in the process of being opened in the input mode. Those files referenced by file-name-1 in another USE AFTER statement that specify the same condition are not executed.
- If you specify OUTPUT, the associated procedure is executed when the condition described in the USE AFTER statement occurs for any file that is open in the output mode, or that is in the process of being opened in the output mode. Those files referenced by file-name-1 in another USE AFTER statement that specify the same condition are not executed.
- If you specify I-O, the associated procedure is executed when the condition described in the USE AFTER statement occurs for any file that is open in the I-O mode, or that is in the process of being opened in the I-O mode. Those files referenced by file-name-1 in another USE AFTER statement that specify the same condition are not executed.
- If you specify EXTEND, the associated procedure is executed when the condition described in the USE AFTER statement occurs for any sequential file that is open in the EXTEND mode, or that is in the process of being opened in the EXTEND mode. Those sequential files referenced by file-name-1 in another USE AFTER statement that specify the same condition are not executed.

After execution of a USE procedure, control passes to the invoking routine in the input-output control system. If the I-O status value does not indicate a critical input-output error, the input-output control system returns control to the next executable statement that follows the input-output statement whose execution caused the exception. Refer to the discussion of the STATUS IS clause in Section 3 for information on I-O status values.

In a USE procedure, a statement cannot be executed if it would cause the execution of a USE procedure that had previously been invoked and had not yet returned the control to the invoking routine.

Precedence Rules for Nested Programs

Special precedence rules are followed when programs are nested. In applying these rules, only the first qualifying declarative is selected for execution. The declarative selected for execution must satisfy the rules for execution of that declarative. The order of precedence is as follows:

1. The declarative within the program that contains the statement that caused the qualifying condition
2. The declarative in which the GLOBAL phrase is specified and that is within the program directly containing the program which was last examined for a qualifying declarative
3. Any declarative selected by applying rule 1 to each more inclusive containing program until rule 2 is applied to the outermost program. If a qualifying declarative is not found, none is executed.

The following scenario illustrates this order of precedence:

Program A contains program B, which contains program C. Each program contains USE statements.

If an I/O error occurs while program C is executing, the system first looks for the USE statement in program C. If the statement is not applicable, as in the case where an error occurred on INPUT and the USE statement specified on OUTPUT, the system then looks at the USE statement in program B. The USE statement in program B is executed if the statement contains the GLOBAL option and is applicable. Otherwise, the system looks at the USE statement in program A. The USE statement in program A is executed if it contains the GLOBAL option and is applicable.

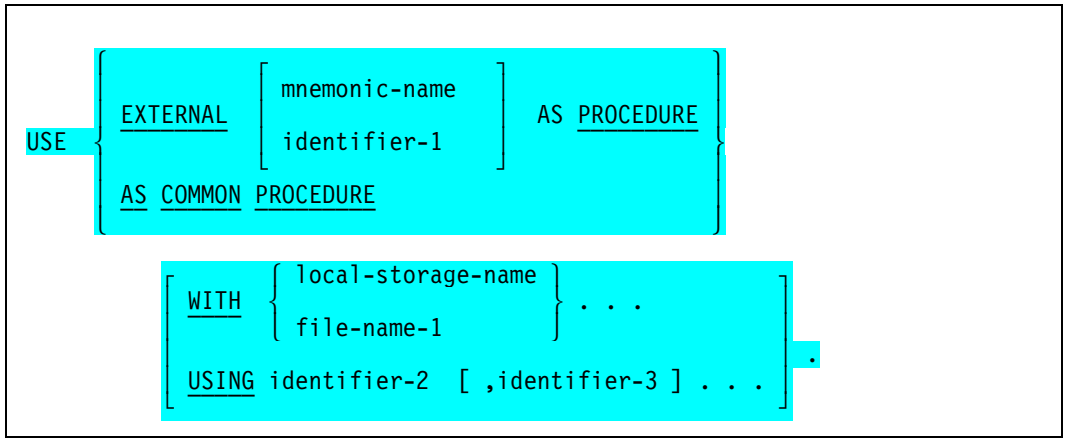
If none of the USE statements are applicable, none are executed.

Example

```
PROCEDURE DIVISION
DECLARATIVES.
PARITY-ERROR SECTION.
    USE AFTER ERROR PROCEDURE ON TAPEIN.
ERROR-ROUTINE.
     .
     .
     .
```

If an input-output error occurs for the file TAPEIN, the standard input-output error-handling procedures are followed as well as the procedures specified in ERROR-ROUTINE.

Format 2: USE PROCEDURE



Explanation

The USE EXTERNAL phrase identifies procedures to be bound from another program or the separately compiled program that is to be used as the task when this section is referenced.

The USE AS COMMON PROCEDURE phrase identifies a procedure that exists in the host program and is to be called in this bound procedure.

mnemonic-name

Mnemonic-name specifies either the program that contains the procedure to be bound or the program that is to be executed as a task. The mnemonic-name must be defined in the Special-Names paragraph of the Environment Division. In binding, the mnemonic-name can be overridden by the explicit program-name specified in the BIND statement.

local-storage-name

All local-storage-names must be defined in the Local-Storage Section. You must include a local-storage-name if the USING phrase is present in the CALL, PROCESS, or RUN statement.

identifier-1

Identifier-1 specifies the program that is to be executed as a task. Identifier-1 must be defined in the Working-Storage Section of the Data Division.

file-name-1

File-name-1 must be uniquely defined as a local file in the File Section.

identifier-2
identifier-3

Identifier-2, identifier-3 and so forth must be uniquely defined as 01-level or 77-level data items of the local-storage-name specified in the WITH phrase, or they must be defined as files in the File Section.

Format 3: USE AS INTERRUPT PROCEDURE

```
USE AS INTERRUPT PROCEDURE.
```

Explanation

The USE AS INTERRUPT PROCEDURE statement specifies a declarative as an interrupt procedure. By declaring an interrupt procedure and then attaching an event to the interrupt procedure (with the ATTACH statement), you can programmatically interrupt a process when the event attached to that procedure occurs. You must include additional statements after the USE statement to be executed when the event occurs and the interrupt procedure is allowed (ALLOW statement).

When an interrupt procedure is being executed, all other interrupts to the process being interrupted are disallowed. Thus, an interrupt procedure itself cannot be interrupted.

Related Information

The following table provides additional references for information related to this topic.

For more information about . . .	Refer to . . .
The mechanisms for handling interrupt procedures	The ALLOW, ATTACH, CAUSE, DETACH, DISALLOW, and RESET statements.
Specifying a data item as an event	The “USAGE Clause” in Section 4.

Format 4: USE AS EPILOG PROCEDURE

```
USE AS EPILOG PROCEDURE.
```

Explanation

The USE AS EPILOG PROCEDURE statement specifies a declarative as an epilog procedure. An epilog procedure enables you to designate a procedure that must be executed before exiting the program. The epilog procedure executes each time the program exits, whether the exit is normal or abnormal. This enables the user to perform necessary clean-up or to free locked resources before terminating.

To see if a program terminated normally, include a test in the epilog procedure as follows:

```
IF ATTRIBUTE HISTORYCAUSE OF MYSELF = 0 THEN
   <program terminated normally>
ELSE
   <program terminated abnormally>.
```

Restrictions on Epilog Procedures

The following restrictions apply to the epilog procedure:

- A GO TO statement cannot be used to exit from an epilog procedure.
- A program cannot have more than one EPILOG PROCEDURE declaration.
- A program with an EPILOG PROCEDURE declaration cannot be used as the host code file when running BINDER.
- If a program that contains an EPILOG PROCEDURE declaration fails because of a fatal stack overflow, the epilog procedure is not executed.
- If a program contains an EPILOG PROCEDURE declaration and the statistics option is TRUE, the epilog procedure is executed before the statistics wrap-up code.
- If certain Data Management System (DMS) statements such as OPEN or CLOSE are executed, it might not be possible to return to the epilog procedure if the executing task is discontinued.
- During a TADS session, all breakpoints are ignored when the epilog procedure is being executed after normal execution is completed.
- An ALTER statement might not reference a paragraph within the epilog procedure.

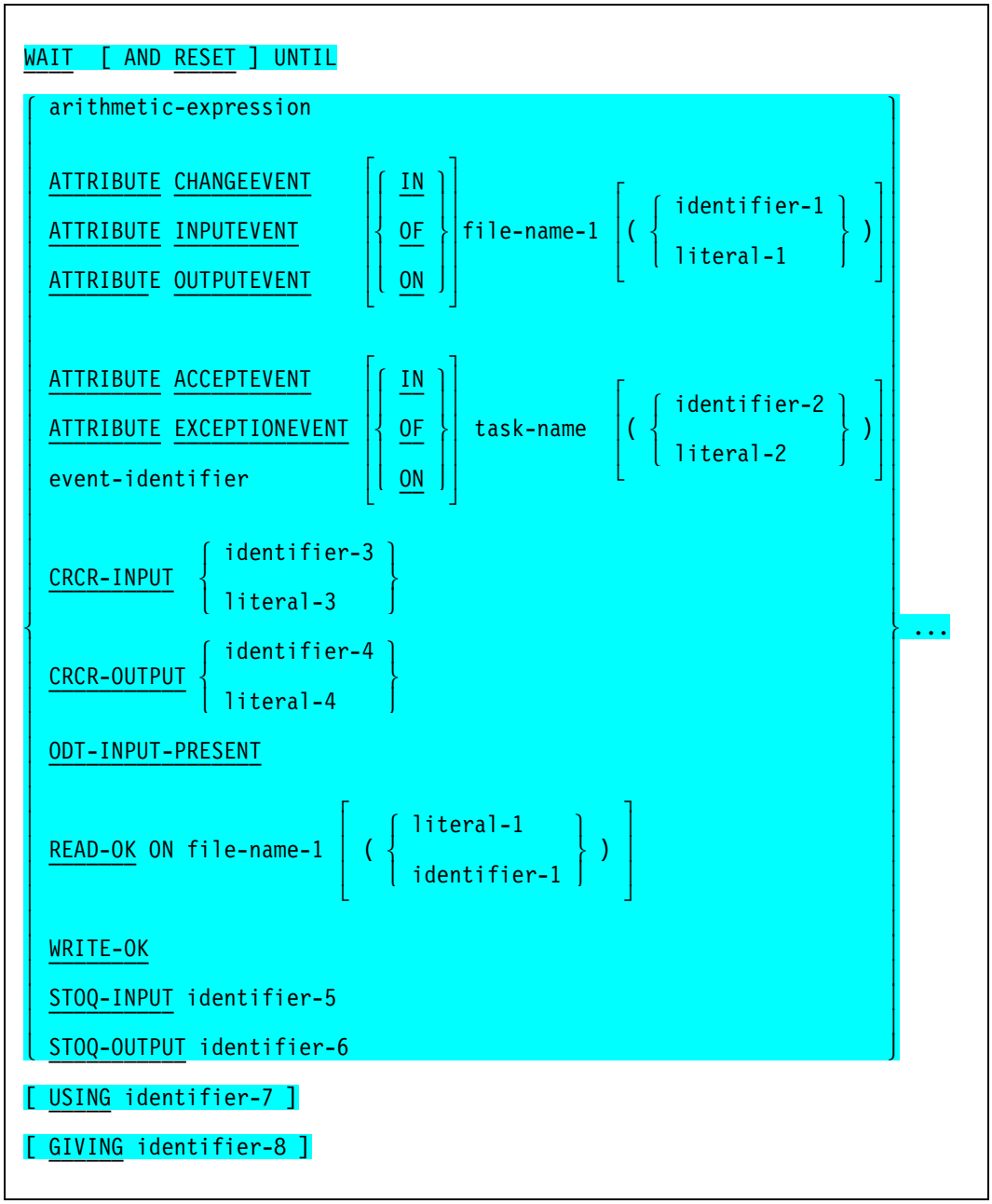
WAIT Statement

The WAIT statement suspends execution of the program for a specified length of time or until one or more conditions is true.

Format	Use
Format 1	This format suspends execution of the object program for a specified length of time or until one or more conditions is true.
Format 2	This format suspends execution of the process until one of its interrupt procedures is executed.

Format 1: Wait for Time or Condition

Following is the syntax diagram.



Explanation

AND RESET

The AND RESET phrase causes the condition or event that terminated the wait to be reset. The condition or event is specified in the UNTIL phrase of this format.

arithmetic-expression

This specifies the number of seconds the program is suspended. The maximum wait time is 164925 seconds (approximately 45.8 hours). If you specify a wait time that exceeds this maximum, the task waits only 164925 seconds.

Only one arithmetic-expression can be specified in a WAIT statement. If multiple conditions are specified, the arithmetic-expression must be the first condition in the list. If the specified number of seconds elapses before an event occurs, the AND RESET phrase has no effect.

ATTRIBUTE INPUTEVENT

This is a synonym for the READ-OK option. For details about this file attribute, refer to the *File Attributes Programming Reference Manual*.

ATTRIBUTE OUTPUTEVENT

This is a synonym for WRITE-OK. For details about this file attribute, refer to the *File Attributes Programming Reference Manual*.

ATTRIBUTE CHANGEEVENT

This condition suspends the program until the value of the FILESTATE attribute has changed. For details about this file attribute, refer to the *File Attributes Programming Reference Manual*.

READ-OK

This condition suspends the program until at least one record is available from file-name-1 (that is, until the CENSUS attribute of the file has a value greater than 1.)

For files that are not open, this condition is always FALSE.

READ-OK is synonymous with ATTRIBUTE INPUTEVENT.

WRITE-OK

This condition suspends the program until enough space exists in the file for at least one more record to be written (that is, until the CENSUS attribute of the file has a value less than the MAXCENSUS attribute).

For files that are not open, this condition is always FALSE.

file-name-1
identifier-1
literal-1

File-name-1 must name a port file. You can specify a subport by including literal-1 or identifier-1 in parentheses following the file name. If an identifier is specified, it must describe an elementary numeric data item that does not contain the symbol *P* in its PICTURE clause.

ATTRIBUTE ACCEPTEVENT
ATTRIBUTE EXCEPTIONEVENT
event-identifier

This is either a task attribute of type EVENT (either ACCEPTEVENT or EXCEPTIONEVENT) or a data item declared with the USAGE IS EVENT clause. When used, this format specifies that the program is to suspend execution until the event has been activated by the CAUSE statement. ATTRIBUTE ACCEPTEVENT performs the same function as the ODT-INPUT-PRESENT option.

task-name

Task-name must name a task variable. You can specify an entry in a task array by including literal-2 or identifier-2 in parentheses following the task-name. If an identifier is specified, it must describe an elementary numeric data item that does not contain the symbol *P* in its PICTURE clause.

CRCR-INPUT

This condition suspends the program until the sending program is ready to send the data by using the MCP core-to-core mechanism (refer to Format 1 of the SEND statement for details). Identifier-3 or literal-3 must specify the name of the program that will send the data.

CRCR-OUTPUT

This condition suspends the program until the receiving program is ready to receive the data by using the MCP core-to-core mechanism (refer to Format 1 of the RECEIVE statement for details). Identifier-4 or literal-4 must specify the name of the program that will receive the data.

ODT-INPUT-PRESENT

This condition occurs whenever input is sent to the process through the AX command. Execution of an ACCEPT statement that specifies the ODT as input causes this condition to be reset. The AND RESET phrase also resets this condition. Only one ODT-INPUT-PRESENT clause can be specified in a WAIT statement. ODT-INPUT-PRESENT is synonymous with ATTRIBUTE ACCEPTEVENT.

STOQ-INPUT

This condition suspends the program until a STOQ entry is available to be received into the program (refer to Format 2 of the RECEIVE statement for details). Identifier-5 must be a 01 level data item that defines a STOQ parameter block.

STOQ-OUTPUT

This condition suspends the program until space is available in a storage queue for data to be sent from the program (refer to Format 2 of the SEND statement for details). Identifier-6 must be a 01 level data item that defines a STOQ parameter block.

USING identifier-7

This phrase specifies a value in identifier-7 that is used to determine which event should be tested first for a true condition. When the USING phrase is not specified, the first item in the list is tested first.

GIVING identifier-8

This phrase identifies the condition that caused the wait to terminate. When this phrase is used, identifier-8 is set to the value of the position in the list of the event or condition that terminated the wait. For example, if the second event in the list terminated the wait, the data item referenced by identifier-8 is set to the value 2.

Details

If any condition specified in the WAIT statement is true, the wait terminates and control passes to the next executable statement in the program.

If none of the conditions specified in the WAIT statement are true, program execution is suspended until one of the conditions becomes true. When one of the conditions becomes true, the wait terminates and control passes to the next executable statement in the program.

Example

```
WAIT UNTIL (WAIT-RETRY-TIME + (LOAD-FACTOR * NUMBER-USERS)).
```

This suspends program execution for the number of seconds specified by the arithmetic expression WAIT-RETRY-TIME + (LOAD-FACTOR * NUMBER-USERS).

```
WAIT AND RESET WAIT-RETRY-TIME
               ODT-INPUT-PRESENT
     GIVING WAIT-ENDER.
```

This suspends program execution for the number of seconds specified by WAIT-RETRY-TIME, or until the condition ODT-INPUT-PRESENT becomes true, whichever occurs first. The condition that terminated the wait is identified by the value of the data item WAIT-ENDER. If the elapsed time exceeds the value of WAIT-RETRY-TIME, then WAIT-ENDER contains the value 1. If the condition ODT-INPUT-PRESENT becomes true, then WAIT-ENDER contains the value 2.

Format 2: Wait Until Interrupt

```
WAIT UNTIL INTERRUPT
```

Details

This statement suspends the execution of a program until one of its interrupt procedures is executed. After the interrupt procedure executes, the suspended program resumes execution. The program runs indefinitely unless one of its interrupt procedures contains a STOP RUN statement.

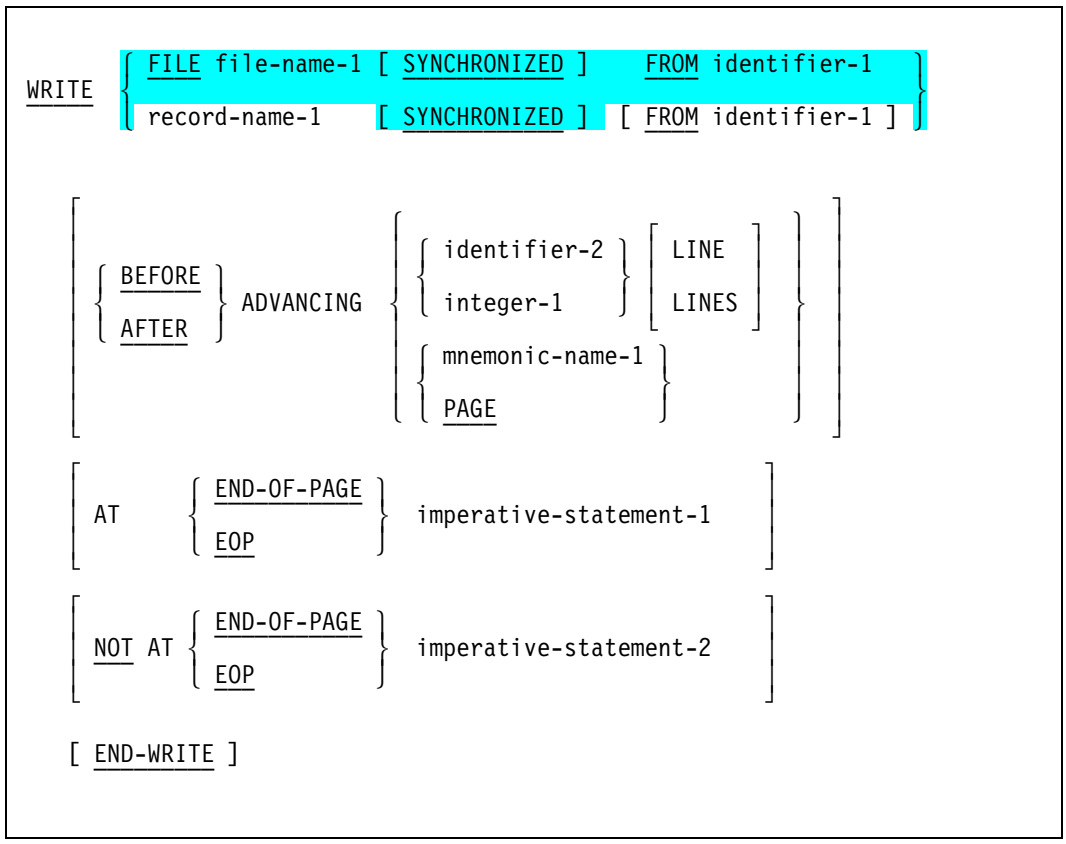
WRITE Statement

The WRITE statement releases a logical record for a file. It can also be used for vertical positioning of lines within a logical page.

This statement is partially supported in a TADS environment. Applicable exclusions are noted in this section.

Format	Use
Format 1	This format is for use with sequential files.
Format 2	This format is for use with relative and indexed files.

Format 1: WRITE (Files in Sequential Access Mode)



This format is supported in the TADS environment.

Explanation

FILE file-name-1

File-name-1 is the name of a file in the File Section of the Data Division. To use the FILE phrase, you must specify the EXTERNAL-FORMAT clause in the file description entry for file-name-1.

File-name-1 cannot reference a sort-merge description entry or a report file.

When the FILE phrase is specified, the contents of the logical record area of the file are not affected by identifier-1 specified in the accompanying FROM phrase. The result of the execution of the WRITE FILE statement with the FROM phrase is the same as the execution of the following statements:

```
MOVE identifier-1 TO implicit record.
WRITE implicit-record TO file-name.
```

The implicit-record refers to the record description that is the same as the data description entry for identifier-1.

record-name-1

Record-name-1 is the name of a logical record in the File Section of the Data Division. This name can be qualified.

Record-name-1 and identifier-1 cannot refer to the same storage area.

The file referenced by the file-name associated with record-name-1 must be in the output, I-O, or extend mode at the time the WRITE statement is executed. (You must reset the compiler option ANSICLASS (FS48) to enable the program to write to a file that is open in the I-O mode.)

The execution of the WRITE statement releases a logical record to the operating system. The execution of a WRITE statement does not affect the contents or accessibility of the record area.

If the associated file is named in the SAME RECORD AREA clause, the logical record is also available to the program as a record of other files referenced in that SAME RECORD AREA clause as the associated output file, as well as to the file associated with record-name-1. As a result, records in a sequential file opened in I-O mode cannot normally be replaced by a WRITE statement during an update operation. A READ-MODIFY-WRITE sequence accesses logical record n, modifies it, and writes it into logical position n+1 in the file. The next READ accesses logical record n+2, and so on. In order to modify a record n in place in a sequential file, the sequence READ-MODIFY-REWRITE must be used.

SYNCHRONIZED

This option enables you to override the synchronization specified by the file attribute for a specific output record.

Synchronization means that output must be written to the physical file before the program initiating the output can resume execution, thereby ensuring synchronization between logical and physical files. Synchronization of all output records can be designated with the SYNCHRONIZE file attribute. Synchronization is available for use by tape files and disk files with sequential organization only, and is not available for use by port files.

A periodic synchronous WRITE statement that follows one or more asynchronous WRITE statements can be used as a checkpoint to ensure that all outstanding records are written to the file before the program continues execution.

FROM identifier-1

If identifier-1 is a function-identifier, it must reference an alphanumeric function. If identifier-1 is not a function-identifier, it cannot reference the same storage area as record-name-1. Identifier-1 can reference a long numeric data item.

The result of the execution of the WRITE statement with the FROM phrase is the same as the execution of the following statements:

```
MOVE identifier-1 TO record-name.
WRITE record-name.
```

The contents of the record area before the execution of the implicit MOVE statement do not affect the execution of this WRITE statement. Refer to "MOVE Statement" in this section for information on MOVE rules.

BEFORE ADVANCING
AFTER ADVANCING

Identifier-2 must refer to an integer data item. The value of identifier-2 can be zero.

Integer-1 can be positive or zero, but cannot be negative.

You cannot specify ADVANCING mnemonic-name when you write a record to a file that is associated with a file description entry containing a LINAGE clause. The mnemonic-name is defined in the SPECIAL-NAMES paragraph of the Environment Division.

The mnemonic-name must be associated with a CHANNEL number.

The phrases ADVANCING PAGE and END-OF-PAGE cannot both be specified in the same WRITE statement.

Both the ADVANCING phrase and the END-OF-PAGE phrase allow control of the vertical positioning of each line on a representation of a printed page.

If you do not use the ADVANCING phrase, automatic advancing occurs as if you had specified AFTER ADVANCING 1 LINE. A WRITE BEFORE ADVANCING statement is more efficient than a WRITE AFTER ADVANCING statement. Therefore, programs that write printer files should specify a WRITE BEFORE ADVANCING statement rather than a simple WRITE statement.

If you specify the ADVANCING phrase, advancing occurs as follows:

1. If you specify integer-1 or the value of identifier-2 is positive, the representation of the printed page is advanced by that number of lines.
2. If you specify integer-1 or the value of identifier-2 is zero, repositioning of the representation of the printed page does not occur.
3. If the value of identifier-2 is negative, the results are undefined.
4. If you specify mnemonic-name-1, the representation of the printed page is advanced to the line number corresponding to the CHANNEL number.
5. If you specify the BEFORE phrase, the line is presented before the representation of the printed page is advanced according to rules 1, 2, 3, and 4.
6. If you specify the AFTER phrase, the line is presented after the representation of the printed page is advanced according to rules 1, 2, 3, and 4.
7. If you specify PAGE and the LINAGE clause is specified in the associated file description entry, the record is presented on the logical page before or after (depending on the phrase used) the device is repositioned to the next logical page. The repositioning is to the first line that can be written on the next logical page, as specified in the LINAGE clause.
8. If you specify PAGE and the LINAGE clause is not specified in the associated file description entry, the record is presented on the logical page before or after (depending on the phrase used) the device is repositioned to the next physical page. The repositioning is to CHANNEL 1 or line 1 of the next logical page, when appropriate for the hardware device.

If PAGE does not have meaning for a specific device, advancing occurs as if you had specified BEFORE or AFTER ADVANCING 1 LINE.

END-OF-PAGE
EOP

The keywords END-OF-PAGE and EOP are synonymous and interchangeable.

Both the ADVANCING phrase and the END-OF-PAGE phrase allow control of the vertical positioning of each line on a representation of a printed page.

The phrases END-OF-PAGE and ADVANCING PAGE cannot both be specified in the same WRITE statement.

If you specify the END-OF-PAGE phrase, the LINAGE clause must be specified in the file description entry for the associated file.

If the logical end of the representation of the printed page is reached during the execution of a WRITE statement with the END-OF-PAGE phrase, imperative-statement-1 specified in the END-OF-PAGE phrase is executed. The logical end is specified in the LINAGE clause associated with record-name-1.

An END-OF-PAGE condition occurs when the execution of a given WRITE statement with the END-OF-PAGE phrase causes printing or spacing in the footing area of a page body. This occurs when the execution of such a WRITE statement causes the LINAGE-COUNTER to equal or exceed the value, if specified, in the FOOTING clause (that is, the value specified by integer-2 or data-name-2 of the LINAGE clause). In this case, the WRITE statement is executed, and the imperative statement in the END-OF-PAGE phrase is then executed.

An automatic page overflow condition occurs when the execution of a given WRITE statement (with or without an END-OF-PAGE phrase) cannot be fully accommodated within the current page body. This occurs when a WRITE statement, if executed, would cause the LINAGE-COUNTER to exceed the total number of lines specified for a page (that is, the value specified by integer-1 or the data item referenced by data-name-1 of the LINAGE clause). In this case, the record is presented on the logical page before or after (depending on the phrase used) the device is repositioned to the first line that can be written on the next logical page as specified in the LINAGE clause. The imperative-statement in the END-OF-PAGE clause, if specified, is executed after the record is written and the device has been repositioned. If a FOOTING phrase is not specified in the LINAGE clause, an END-OF-PAGE condition distinct from the page overflow condition is not detected. In this case, the end-of-page condition and the page overflow condition occur simultaneously.

If a FOOTING phrase is specified in the LINAGE clause but the execution of a given WRITE statement would cause the LINAGE-COUNTER simultaneously to exceed the total number of lines allowed on a page and in the footing area, the operation proceeds as if you had not specified a footing area.

END-WRITE

This phrase delimits the scope of the WRITE statement.

Details

After the execution of the WRITE statement, the information in the area referenced by identifier-1 is available, even though the information in the area referenced by record-name-1 is not available (except as specified by the SAME RECORD AREA clause).

The file position indicator is not affected by the execution of a WRITE statement.

The execution of the WRITE statement updates the value of the I-O status of the file-name associated with record-name-1.

The maximum record size for a file is established at the time the file is created and must not be changed later.

The number of character positions on a mass-storage device required to store a logical record in a file may or may not be equal to the number of character positions defined by the logical description of that record in the program.

The number of character positions in the record referenced by record-name-1 cannot be larger than the largest or smaller than the smallest number of character positions allowed by the RECORD IS VARYING clause associated with the file-name.

If the number of character positions exceeds these bounds, the following results occur:

- The WRITE operation does not take place.
- The content of the record area is unaffected.
- The I-O status of the file associated with record-name-1 is set to a value that indicates the cause of the condition.

The current record pointer is unaffected by the execution of a WRITE statement.

The successor relationship of a sequential file is established by the order of execution of WRITE statements when the file is created. The relationship does not change except when records are added to the end of the file.

When a sequential file is open in the extend mode, the execution of the WRITE statement adds records to the end of the file as though the file were open in the output mode. If there are records in the file, the first record written after the execution of the OPEN statement with the EXTEND phrase is the successor of the last record in the file.

When you attempt to write beyond the externally defined boundaries of a sequential file, an exception condition exists and the content of the record area is unaffected. The following actions take place:

- The value of the I-O status of the file-name associated with record-name-1 is set to a value indicating a boundary violation.
- If a USE AFTER STANDARD EXCEPTION declarative is explicitly or implicitly specified for the file-name associated with record-name-1, that declarative procedure is then executed.
- If a USE AFTER STANDARD EXCEPTION declarative is not explicitly or implicitly specified for the file-name associated with record-name-1, the result is undefined.

TADS

Any USE procedure is not executed when a WRITE statement that is compiled and executed in a TADS session fails. If the end of the reel or unit is recognized and the externally defined boundaries of the file have not been exceeded, the following operations are executed:

- The standard reel or unit label procedure
- A reel or unit swap (the current volume pointer is updated to point to the next reel or unit existing for the file)
- The standard beginning reel or unit label procedure

The INVALID KEY condition exists when a maximum logical size has been specified for the file and no more logical records can be written.

If the EXTERNAL-FORMAT FOR NATIONAL clause is specified in the file description entry, the contents of elementary data items of class national are converted from internal format to external format after the record leaves the record area and before the record is written to the external medium. In determining the relevant data items, all REDEFINES and RENAMES entries in the selected record description are ignored.

The record description that defines the national data items is selected as follows:

- For a WRITE statement without the FILE phrase, the record description associated with record-name-1 in the WRITE statement is used.
- For a WRITE statement with the FILE phrase, the record description associated with identifier-1 in the FROM phrase of the WRITE statement is used.

The size of the record area is adjusted to include the size of the storage area referenced by identifier-1 in the FROM phrase plus the size of the control characters necessary to convert from internal to external format.

Requirements for Shared Files

If the file to which you are attempting to write is a shared file, which means that it has the phrase "VALUE OF BUFFERSHARING IS SHARED" in its File Description Entry (FD entry), you must precede the WRITE statement with a LOCKRECORD statement. You must follow the WRITE statement with an UNLOCKRECORD statement. If you do not lock the record before attempting the WRITE operation,

- The WRITE statement is not executed.
- The I-O status code value 9E is returned.
- The INVALID KEY action is executed.

If the value of BUFFERSHARING is altered in a CHANGE ATTRIBUTE statement or through WFL, the WRITE code is not updated.

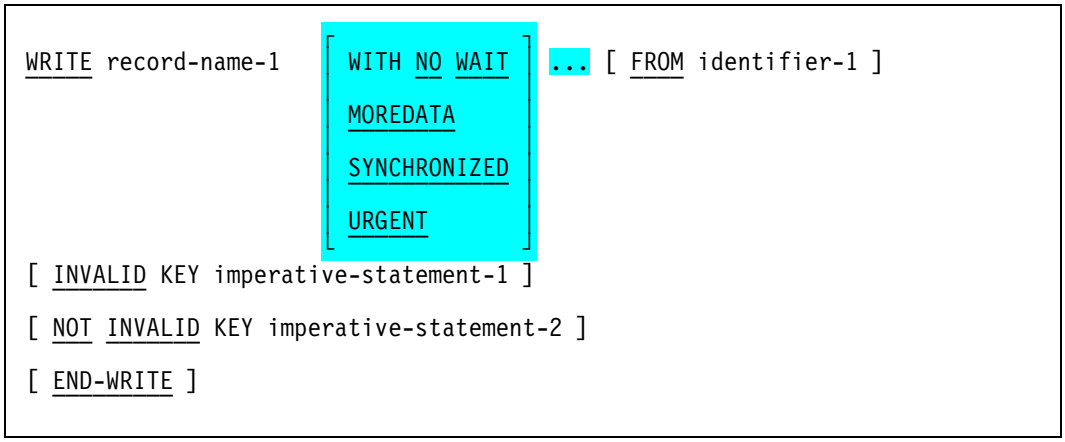
Example

```
WRITE Record-1 FROM Temp AFTER ADVANCING 2 END-WRITE
```

In this example, the record Record-1 is written to the file after advancing the representation of the printed page two lines. The record Record-1 is contained in the storage area called Temp. The END-WRITE phrase terminates the WRITE statement.

Format 2: WRITE (Files in Random Access Mode)

The format for the WRITE statement for relative and indexed files is as follows:



This format is supported in the TADS environment.

Explanation

record-name-1

Record-name-1 is the name of a logical record in the File Section of the Data Division. This name can be qualified.

Record-name-1 and identifier-1 cannot refer to the same storage area.

FROM identifier-1

If identifier-1 is a function-identifier, it must reference an alphanumeric function. If identifier-1 is not a function-identifier, it cannot reference the same storage area as record-name-1. Identifier-1 can reference a long numeric data item.

The result of the execution of the WRITE statement with the FROM phrase is the same as the execution of the following statements:

```
MOVE identifier TO record-name.
WRITE record-name.
```

The contents of the record area before the execution of the implicit MOVE statement have no effect on the execution of this WRITE statement. Refer to "MOVE Statement" for information about the MOVE rules.

WITH NO WAIT

The WITH NO WAIT phrase can be specified only for port files. The WITH NO WAIT phrase can be included only once.

A WRITE statement causes the program to wait until a buffer is available to store the record. The possibility of this suspension is prevented for a port file by using the WITH NO WAIT phrase. A status key value of 95 indicates that no buffer was available for the logical record.

MOREDATA

This option enables an OSI port file that uses the segmented I/O capability to pass a message segment with the indication that more data for the same message is forthcoming.

SYNCHRONIZED

This option enables you to override the synchronization specified by the file attribute for a specific output record.

Synchronization means that output must be written to the physical file before the program starting the output can resume execution, thereby ensuring synchronization between logical and physical files. Synchronization of all output records can be indicated with the SYNCHRONIZE file attribute. Synchronization is available for use by tape and disk files with sequential organization only, and is not available for use by port files.

A periodic synchronous WRITE statement that follows one or more asynchronous WRITE statements can act as a checkpoint to ensure that all outstanding records are written to the file before the program continues execution.

URGENT

The URGENT phrase is meaningful only when the Transmission Control Protocol/Internet Protocol (TCP/IP) is being used. This phrase sets the urgent indicator associated with the data. For information on TCP/IP, refer to the *Distributed Systems Services (DSS) Operations Guide*.

INVALID KEY imperative-statement-1

The INVALID KEY phrase must be specified if an applicable USE AFTER STANDARD EXCEPTION procedure for the file-name is not associated with the record-name.

For a sequential file, the INVALID KEY condition exists when the content of the ACTUAL KEY data item is less than 1 or is greater than the ordinal number of the last record written to the file. (The ACTUAL KEY data item is declared in the File Control Entry in the Input-Output Section.)

For a relative file, the INVALID KEY condition exists under the following circumstances:

- The access mode is random or dynamic, and the RELATIVE KEY data item specifies a record that already exists in the file. (You declare the RELATIVE KEY data item with the RELATIVE KEY clause in the File Control Entry of the Input-Output Section.) You cannot specify a long numeric data item as the RELATIVE KEY.
- An attempt is made to write beyond the externally defined boundaries of the file.
- The number of significant digits in the relative record number is larger than the size of the relative key data item described for the file.

For an indexed file, the INVALID KEY condition exists when

- The file is opened in the sequential access mode, and the file is also opened in the output or extend mode, and the value of the prime record key is not greater than the value of the prime record key of the previous record.
- The file is opened in the output or I-O mode, and the value of the prime record key is equal to the value of the prime record key of a record already existing in the file.
- The file is opened in the output, extend, or I-O mode, and the value of an alternate record key (for which duplicates are not allowed) equals the corresponding data item of a record already existing in the file.
- An attempt is made to write beyond the externally defined boundaries of the file.

For both relative and indexed files, when the INVALID KEY condition exists:

- The execution of the WRITE statement is unsuccessful.
- The contents of the record area are unaffected.
- The I-O STATUS of the file associated with record-name-1 is set to a value that indicates an INVALID KEY condition.

Refer to Section 10 for details on file attributes, file organization, and access modes.

NOT INVALID KEY imperative-statement-2

Specify the INVALID KEY phrase if an applicable USE AFTER STANDARD EXCEPTION PROCEDURE statement for the file-name is not associated with record-name-1.

If, during the execution of a WRITE statement with the NOT INVALID KEY phrase, the invalid key condition does not occur, control passes to imperative-statement-2 as follows:

- If the execution of the WRITE statement is successful, control passes after the record is written and after the I-O status of the file-name associated with record-name-1 is updated.
- If the execution of the WRITE statement is unsuccessful for a reason other than an invalid key condition, control passes after updating the I-O status of the file-name associated with record-name-1, and after executing the procedure, if any, specified by a USE AFTER STANDARD EXCEPTION PROCEDURE statement applicable to the file-name associated with record-name-1.

END-WRITE

This phrase delimits the scope of the WRITE statement.

Details

The file referenced by the file-name associated with record-name-1 must be in the output or I-O mode at the time of execution of the WRITE statement. Refer to "OPEN Statement" in this section for information on the OUTPUT, EXTEND, and I-O modes.

If the associated file is named in the SAME RECORD AREA clause, the logical record is also available to the program as a record of other files referenced in that SAME RECORD AREA clause as the associated output file, as well as to the file associated with record-name-1. Therefore, records in a sequential file opened in I-O mode cannot normally be replaced by a WRITE statement during an update operation. For information on the SAME RECORD AREA clause and the RECORD IS VARYING clause, refer to Section 4.

A READ-MODIFY-WRITE sequence accesses logical record n, modifies it, and writes it into logical position n+1 in the file. The next READ accesses the logical record n+2, and so on. In order to modify a record n in place in a sequential file, the sequence READ-MODIFY-REWRITE must be used.

After the execution of the WRITE statement, the information in the area referenced by identifier-1 is available, even though the information in the area referenced by record-name-1 is not available (except as specified by the SAME RECORD AREA clause).

The file position indicator is not affected by the execution of a WRITE statement.

The execution of the WRITE statement updates the value of the I-O status of the file-name associated with record-name-1.

The maximum record size for a file is established at the time the file is created and must not be changed later.

The number of character positions on a mass-storage device required to store a logical record in a file may or may not be equal to the number of character positions defined by the logical description of that record in the program.

The number of character positions in the record referenced by the record-name cannot be larger than the largest or smaller than the smallest number of character positions allowed by the RECORD IS VARYING clause associated with the file-name. If the number of character positions exceeds these bounds:

- The WRITE operation does not take place.
- The content of the record area is unaffected.
- The I-O status of the file associated with the record-name is set to a value that indicates the cause of the condition.

The current record pointer is unaffected by the execution of a WRITE statement.

TADS

Any USE procedure is not executed when a WRITE statement that is compiled and executed in a TADS session fails.

Port Files

Format 2 must be used for port files.

If an ACTUAL KEY is declared for a port file, your program is responsible for updating the ACTUAL KEY with an appropriate value. A WRITE statement causes the ACTUAL KEY to be passed to the I/O system to indicate the desired subfile destination. If the ACTUAL KEY is 0, a broadcast write is performed, for which the data is sent to all opened subfiles of the port file.

If no ACTUAL KEY is declared for the file, it must contain a single subfile that is written.

Relative Files

The following rules apply specifically to relative files:

- The RELATIVE KEY phrase cannot reference a long numeric data item.
- When a file is opened in the output mode, records can be placed into the file by one of the following methods:
  - If the access mode is sequential, the WRITE statement releases a record to the mass-storage control system. The first record has a relative record number of 1, and subsequent records released have relative record numbers of 2, 3, 4, and so forth. If the RELATIVE KEY data item has been specified in the file control entry for the associated file, the relative record number of the record just released is placed into the RELATIVE KEY data item during the execution of the WRITE statement.
  - If the access mode is random or dynamic, before the execution of the WRITE statement, the value of the RELATIVE KEY data item must be initialized in the program with the relative record number to be associated with the record in the record area. That record is then released to the mass-storage system.
- When a file is opened in the I-O mode and the access mode is random or dynamic, records are inserted into the associated file. The value of the RELATIVE KEY data item must be initialized by the program with the relative record number associated with the record in the record area. Execution of a WRITE statement then releases the contents of the record area to the mass-storage system.
- When a file is opened in the extend mode, records are inserted into the file. The first record released to the mass-storage control system has a relative record number of 1 greater than the highest relative record number existing in the file. Subsequent records released to the mass-storage control system have consecutively higher relative record numbers. If the RELATIVE KEY phrase is specified for the file-name associated with record-name-1, the relative record number of the record being released is moved into the RELATIVE KEY data item by the mass-storage control system during execution of the WRITE statement according to the rules for the MOVE statement.

Indexed Files

The following rules apply specifically to indexed files:

- Execution of the WRITE statement releases the contents of the record area. The mass-storage control system utilizes the contents of the record keys in such a way that subsequent access of the record may be made based upon any of those specified record keys.
- The value of the prime record key must be unique in the records in the file.
- The data item specified as the prime record key must be set by the program to the desired value before execution of the WRITE statement.
- If you specify the sequential access mode for the file, records must be released to the mass-storage control system in ascending order of prime record key values according to the collating sequence of the file.
- If you specify the extend mode for the file, the first record released to the mass-storage control system must have a prime record key whose value is greater than the highest prime record key value existing in the file.
- If you specify the random or dynamic access mode, records can be written in any program-specified order.
- When the ALTERNATE RECORD KEY clause is specified in the SELECT clause of the FILE-CONTROL paragraph for an indexed file, the value of the alternate record key can be nonunique only if the DUPLICATES phrase is specified for that data item. In this case, records are stored so that when records are accessed sequentially, the order of retrieval of those records is the order in which they are released to the mass-storage control system.
- The ALTERNATE RECORD KEY clause cannot reference a long numeric data item.

Requirements for Shared Files

If the file to which you are attempting to write is a shared file, which means that it has the phrase "VALUE OF BUFFERSHARING IS SHARED" in its File Description Entry (FD entry), you must precede the WRITE statement with a LOCKRECORD statement. You must follow the WRITE statement with an UNLOCKRECORD statement. If you do not lock the record before attempting the WRITE operation,

- The WRITE statement is not executed.
- The I-O status code value 9E is returned.
- The INVALID KEY action is executed.

If the value of BUFFERSHARING is altered in a CHANGE ATTRIBUTE statement or through WFL, the WRITE code is not updated.

Example

```
WRITE Record-1 FROM Temp INVALID KEY Imp-state END-WRITE
```

In this example, Record-1 is a relative or indexed file contained in the storage area Temp. The imperative statement Imp-state is invoked if the INVALID KEY condition exists, which is dependent on the file organization and the file access mode. The END-WRITE phrase terminates the WRITE statement.

Section 9
Intrinsic Functions

A function represents a temporary data item whose value is derived automatically when an object program makes a reference to it. This section describes the functions defined by COBOL85 that you can use throughout the Procedure Division of a COBOL program.

Summary of Functions

The COBOL85 intrinsic functions are summarized in Table 9-1. Note that the Arguments column indicates the type of argument used with a function and the number of arguments available for that function. The types of arguments are abbreviated as follows:

Abbreviation	Argument Type
A	Alphabetic
I	Integer
N	Numeric
G	National
X	Alphanumeric

Table 9–1. Intrinsic Functions

Function Name	Function Type	Arguments (Type and Number)	Value Returned
ABS	Numeric	N1	Absolute value of N1
ACOS	Numeric	N1	Arccosine of N1
ANNUITY	Numeric	N1, I2	Ratio of annuity paid for 12 periods at interest of N1 to initial investment of 1
ASIN	Numeric	N1	Arcsine of N1
ATAN	Numeric	N1	Arctangent of N1
CHAR	Alphanumeric	I1	Character in position I1 of program collating sequence
CHAR-NATIONAL	National	I1	Character in position I1 of the national character collating sequence
CONVERT-TO-DISPLAY	Alphanumeric	G1, A2 or X2	Argument converted to DISPLAY usage
CONVERT-TO-NATIONAL	National	A1 or X1, G2	Argument converted to NATIONAL usage
COS	Numeric	N1	Cosine of N1
CURRENT-DATE	Alphanumeric	None	Current date and time and difference from Greenwich Mean Time
DATE-OF-INTEGER	Integer	I1	Standard date equivalent (YYYYMMDD) of integer date
DAY-OF-INTEGER	Integer	I1	Julian date equivalent (YYYYDDD) of integer date
DIV	Integer	N1,N2	Integer part of quotient of (N1/N2)
EXP	Numeric	N1	Exponential function of N1
FACTORIAL	Numeric	I1	Factorial of I1
FIRSTONE	Integer	N1	Bit number, plus 1, of the leftmost nonzero bits in N1.
FORMATTED-SIZE	Integer	G1 or X1	Formatted size of argument
FUNCTION LINENUMBER	Integer	None	Line number of the source record
INTEGER	Integer	N1	The greatest integer not greater than N1

Table 9–1. Intrinsic Functions

Function Name	Function Type	Arguments (Type and Number)	Value Returned
INTEGER-OF-DATE	Integer	I1	Integer date equivalent of standard date (YYYYMMDD)
INTEGER-OF-DAY	Integer	I1	Integer date equivalent of Julian date (YYYYDDD)
INTEGER-PART	Integer	N1	Integer part of N1
LENGTH	Integer	A1, N1, X1, or G1	Integer length of argument
LENGTH-AN	Integer	A1, N1, X1, or G1	Length of argument in bytes
LOG	Numeric	N1	Natural logarithm of N1
LOG10	Numeric	N1	Logarithm to base 10 of N1
LOWER-CASE	Alphanumeric or national	A1, X1, or G1	All letters in the argument are set to lowercase
MAX	(Depends upon arguments)	A1..., I1..., N1..., X1..., or G1...	Value of maximum argument
MEAN	Numeric	N1...	Arithmetic mean of arguments
MEDIAN	Numeric	N1...	Median of arguments
MIDRANGE	Numeric	N1...	Mean of minimum and maximum arguments
MIN	(Depends upon arguments)	A1..., I1..., N1..., X1..., G1...	Value of minimum argument
MOD	Numeric	I1, I2	I1 modulo I2
NUMVAL	Numeric	X1	Numeric value of simple numeric string
NUMVAL-C	Numeric	X1, X2	Numeric value of simple numeric string
ONES	Integer	N1	Number of nonzero bits in N1
ORD	Integer	A1, X1, G1	Ordinal position of the argument in collating sequence
ORD-MAX	Integer	A1..., N1..., X1..., or G1...	Ordinal position of maximum argument
ORD-MIN	Integer	A1..., N1..., X1..., G1...	Ordinal position of minimum argument

Table 9–1. Intrinsic Functions

Function Name	Function Type	Arguments (Type and Number)	Value Returned
PRESENT-VALUE	Numeric	N1, N2...	Present value of a series of future period-end amounts, N2, at a discount rate of N1
RANDOM	Numeric	I1	Random number
RANGE	Numeric	I1... or N1...	Value of maximum argument minus value of minimum argument
REM	Numeric	N1, N2	Remainder of (N1/N2)
REVERSE	Alphanumeric or national	A1, X1, G1	Reverse order of the characters of the argument
SIGN	Integer	N1	Either +1, -1, or 0, depending on whether N1 is greater than, less than, or equal to zero, respectively
SIN	Numeric	N1	Sine of N1
SQRT	Numeric	N1	Square root of N1
STANDARD-DEVIATION	Numeric	N1...	Standard deviation of arguments
SUM	Numeric	I1... or N1...	Sum or arguments
TAN	Numeric	N1	Tangent or N1
UPPER-CASE	Alphanumeric or national	A1, X1, G1	All letters in the argument are set to uppercase
VARIANCE	Numeric	N1...	Variance or argument
WHEN-COMPILED	Alphanumeric	None	Date and time program was compiled

Types of Functions

Functions can be classified according to the types of values they return, as described in Table 9-2.

Table 9–2. Types of Functions

Function Type	Class	Category	Comments
Alphanumeric	Alphanumeric	Alphanumeric	Alphanumeric functions have an implicit usage of DISPLAY.
Numeric	Numeric	Numeric	Numeric functions are always assumed to have an operational sign, and yield double-precision results.
Integer	Numeric	Numeric	Integer functions are always assumed to have an operational sign.
National	National	National	National functions have an implicit usage of NATIONAL.

Rules for Using Functions

You can use functions throughout the Procedure Division as appropriate replacements for identifiers, arithmetic expressions, and integer operands.

For . . .	Use . . .
Identifiers	Alphanumeric functions
Arithmetic expressions	Numeric or integer functions
Integer operands (signed)	Integer functions

Observe the following restrictions when you use functions:

- Functions cannot be receiving operands of any statement.
- A numeric function cannot be referenced where an integer operand is required, even if the function yields an integer value.
- Numeric and integer functions can be used only in arithmetic expressions.
- Numeric functions yield double-precision results. If a result represents a value that the machine must approximate, and it is assigned to a DISPLAY, COMP, or BINARY data item, then precision can be lost. For more information, refer to USAGE IS DOUBLE of Data Description Entry Format 1 in Section 4.
- Alphanumeric functions cannot be used as identifiers in general formats where the characteristics resulting from the evaluation of the function would not meet the format's requirements for the characteristics of data items (such as class and category, size, usage, and permissible values).
- Functions used in class conditions must be alphanumeric.
- Functions used in the FROM phrase of a RELEASE, REWRITE, or WRITE statement must be alphanumeric.
- Functions cannot appear as parameters in a CALL statement.

Syntax for a Function

The syntax for a function consists of an identifier that includes the word FUNCTION, the name of a specific predefined function, and one or more arguments:

```
FUNCTION function-name-1 [ ( {argument-1} . . .  ) ] [reference-modifier]
```

Explanation

FUNCTION

This COBOL reserved word identifies the syntax as a function.

function-name-1

This indicates the name of the function. Although function names are defined by the system, function names are not reserved words. Thus, a function name can also appear in a program as a user-defined word or a system-name. The functions are listed in Table 9-1.

argument-1

This is a value to be used by the function. It can be either an identifier, a literal, or an arithmetic expression, depending on the type of function. You must separate multiple arguments with a comma (,) or a space. For more on arguments, see "Arguments" later in this section.

reference-modifier

You can use a reference-modifier only with an alphanumeric function. The format for the reference modifier is

```
    (leftmost-character-position : [length] )
```

For details about reference modifiers, refer to Section 4.

Arguments

Arguments specify values used in the evaluation of a function. Arguments can be identifiers, literals, or arithmetic expressions. Some functions do not have arguments, while other functions have one or more. Specific rules governing the number of arguments that can be used with a function, as well as the class and category of the arguments, are provided with the description of each function later in this section. Note that you must separate multiple arguments with a comma (,) or a space.

A function might restrict the values that can be used for its arguments in order to permit meaningful determination of the value of the function. If a value specified for a particular argument is outside of the permissible range as defined by the function, the returned value for the function is undefined.

Types of Arguments

The types of arguments are described in Table 9-3.

Table 9–3. Types of Arguments for Functions

Argument Type	Description
Numeric	You must specify arithmetic expressions for numeric arguments. The value of the arithmetic expression, including the operational sign, is used to determine the value of the function.
Alphabetic	You must specify an identifier that represents an elementary data item of the class alphabetic or a nonnumeric literal that contains only alphabetic characters for an alphabetic argument. The size of the argument determines the size of the result. For example, if a three character argument is used, as in FUNCTION REVERSE ("ABC"), a three-character result is produced, in this case ("CBA").
Alphanumeric	You must specify an identifier that represents an elementary data item of the class alphabetic or alphanumeric, or a nonnumeric literal. The size of the argument determines the size of the result. For example, if a three character argument is used, as in FUNCTION REVERSE ("ABC"), a three-character result is produced, in this case ("CBA").
Integer	You must specify an arithmetic expression that will always result in an integer value for an integer argument. The value of the arithmetic expression, including the operational sign, is used to determine the value of the function.
National	You must specify an identifier that represents a data item of the class national or a national-character literal for this type of argument. The size of the argument determines the size of the result. For example, if a three character argument is used, as in FUNCTION REVERSE (N"ABC"), a three-character result is produced, in this case (N"CBA").

Evaluation of Arguments

When a function has multiple arguments, the arguments are evaluated individually from left to right in the order specified. An argument can be a function-identifier itself or an expression containing function-identifiers. The function-identifier for which the argument is specified can also be referenced in the argument, as shown in the following example:

```
FUNCTION REVERSE (FUNCTION REVERSE ("ABC"))
```

The result from this syntax is "ABC."

Subscripting an Argument

Certain functions enable you to repeat an argument numerous times. Instead of repeating the argument, you can reference a table by following the argument with the subscript ALL. When the ALL subscript is specified, the effect is as if each table element associated with that subscript position were specified.

For example, consider the following three-dimensional table.

```
01 Table-1.
    02 Office OCCURS 2 TIMES.
       03 Department OCCURS 05 TIMES.
        04 Employee  OCCURS 15 TIMES.
          05 Name . . .
          05 Address . . .
```

The first dimension of this table defines 2 offices.

The second dimension defines 5 departments for each office.

The third dimension defines 15 employees for each department.

The reference Employee(1,1,ALL) specifies all the employees in the first office and the first department. The ALL subscript is incremented by 1 until the total number of employees specified in the OCCURS clause is reached:

```
Employee(1,1,1)
Employee(1,1,2)
Employee(1,1,3)
.
.
.
Employee(1,1,15)
```

The reference Employee(ALL,5,ALL) specifies all offices, department 5 of each office, and all employees in department 5 of each office. The first ALL subscript begins with 1 and remains at 1 until the second ALL subscript has been incremented by 1 through its range of values as specified in the OCCURS clause, which is 15 in the following example:

```
Employee(1,5,1)
Employee(1,5,2)
Employee(1,5,3)
.
.
.
Employee(1,5,15)
Employee(2,5,1)
Employee(2,5,2)
Employee(2,5,3)
.
.
.
Employee(2,5,15)
```

The reference Employee(ALL,ALL,ALL) specifies all offices, all departments in each office, and all employees in every department of each office. The first and second ALL subscripts begin with 1 and remain at 1 until the rightmost ALL subscript is incremented by 1 through its range of values:

```
Employee(1,1,1)
Employee(1,1,2)
Employee(1,1,3)
.
.
.
Employee(1,1,15)
```

Then the rightmost ALL subscript is reset to 1, and the ALL subscript to the left of the rightmost ALL subscript is incremented by 1 through its range of values. For each increment of 1, the subscripts to the right increment through their range of values:

```
Employee(1,2,1)
Employee(1,2,2)
Employee(1,2,3)
.
.
.
Employee(1,2,15)
Employee(1,3,1)
Employee(1,3,2)
.
.
.
```

The preceding process continues until all the subscripts are incremented through their range of values. Then the first ALL subscript is incremented to 2 and the process begins again. The process continues until the first ALL subscript has been repeated the number of times specified by the OCCURS clause, which is 2 in this example.

ABS Function

The ABS function returns a value that is the absolute value of argument-1. The type of this function is numeric.

Syntax

```
FUNCTION ABS (argument-1)
```

Explanation

Argument-1 must be of the numeric class. The returned value is the absolute value of argument-1.

Example

Function with Argument	Result
FUNCTION ABS (-6)	6

ACOS Function

The ACOS function returns a numeric value in radians that approximates the arccosine of argument-1. The type of this function is numeric. This function is fully supported in the COBOL85 TADS environment.

Syntax

```
FUNCTION ACOS (argument-1)
```

Explanation

Argument-1 must be of the numeric class. The value of argument-1 must be greater than or equal to -1 and less than or equal to +1. The returned value is the approximation of the arccosine of argument-1 and is greater than or equal to zero and less than or equal to pi.

Example

Function with Argument	Result
FUNCTION ACOS (.5)	1.047

ANNUITY Function

The ANNUITY function (annuity immediate) returns a numeric value that approximates the ratio of an annuity paid at the end of each period for the number of periods specified by argument-2 to an initial investment of one. Interest is earned at the rate specified by argument-1 and is applied at the end of the period, before the payment. The type of this function is numeric.

Syntax

```
FUNCTION ANNUITY (argument-1, argument-2)
```

Explanation

Argument-1 must be of the numeric class and must have a value that is greater than or equal to zero.

- When the value of argument-1 is zero, the value of the function is the approximation of 1 divided by argument-2.
- When the value of argument-1 is not zero, the value of the function is the approximation of the following computation:

```
argument-1 / (1 - (1 + argument-1) ** (-argument-2))
```

Argument-2 must be a positive integer.

You must separate argument-1 and argument-2 with a comma (,) or a space.

Note that an invalid argument results in a run-time error.

Example

Function with Arguments	Result
FUNCTION ANNUITY (.1,12)	.1467

ASIN Function

The ASIN function returns a numeric value in radians that approximates the arcsine of argument-1. The type of this function is numeric. This function is fully supported in the COBOL85 TADS environment.

Syntax

```
FUNCTION ASIN (argument-1)
```

Explanation

Argument-1 must be of the numeric class. The value of argument-1 must be greater than or equal to -1 and less than or equal to +1.

An invalid argument results in a run-time error.

The returned value is the approximation of the arcsine of argument-1 and is greater than or equal to -pi/2 and less than or equal to +pi/2.

Example

Function with Argument	Result
FUNCTION ASIN (.5)	.524

ATAN Function

The ATAN function returns a numeric value in radians that approximates the arctangent of argument-1. The type of this function is numeric. This function is fully supported in the COBOL85 TADS environment.

Syntax

```
FUNCTION ATAN (argument-1)
```

Explanation

Argument-1 must be of the numeric class. The returned value is the approximation of the arctangent of argument-1 and is greater than -pi/2 and less than +pi/2.

Example

Function with Argument	Result
FUNCTION ATAN (1.732)	1.047

CHAR Function

The CHAR function returns a one-character alphanumeric value that is the character in the program collating sequence that has the ordinal position equal to the value of argument-1. The type of this function is alphanumeric.

Syntax

```
FUNCTION CHAR (argument-1)
```

Explanation

Argument-1 must be an integer. The value of argument-1 must be greater than zero and less than or equal to the number of positions in the collating sequence. If more than one character has the same position in the program collating sequence, the character returned as the function value is that of the first literal specified for that character position in the ALPHABET clause.

If the current program collating sequence was not specified by an ALPHABET clause, then the default collating sequence is used.

An invalid argument results in a run-time error.

Example

Function with Argument	Result
FUNCTION CHAR (91)	]

CHAR-NATIONAL Function

The CHAR-NATIONAL (8-bit national character) function returns a one-character value that is a character in the national program collating sequence with the ordinal position equal to the value of argument-1. The type of this function is national.

Syntax

```
FUNCTION CHAR-NATIONAL (argument-1)
```

Explanation

Argument-1 must be an integer. The value of argument-1 must be greater than zero and less than or equal to the number of positions in the national program collating sequence. If more than one character has the same position in the national program collating sequence, the character returned as the function value is that of the first literal specified for that character position in the ALPHABET clause.

If the national program collating sequence was not specified in an ALPHABET clause, a compile-time error is issued for the use of this function.

An invalid argument results in a run-time error.

Example

Consider the following ENVIRONMENT DIVISION.

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
OBJECT-COMPUTER.
   PROGRAM COLLATING SEQUENCE FOR NATIONAL IS FOREIGN.
SPECIAL-NAMES.
   ALPHABET FOREIGN FOR NATIONAL IS CCSVERSION "FRANCE".
```

Function with Argument	Result
FUNCTION CHAR-NATIONAL (20)	N"A"

CONVERT-TO-DISPLAY Function

The CONVERT-TO-DISPLAY function returns a character string that contains the national characters of the argument converted to the corresponding alphanumeric character representation. The type of this function is alphanumeric.

Syntax

```
FUNCTION CONVERT-TO-DISPLAY ( argument-1 [,argument-2] )
```

Explanation

Argument-1 must be of the national class and must be at least one character in length.

Argument-2 must be of either the alphabetic or the alphanumeric class and must be one character in length. Argument-2 specifies a substitute alphanumeric character for use in the conversion of any national character for which no corresponding alphanumeric character exists.

You must separate argument-1 and argument-2 with a comma (,) or a space.

The returned value is a character string with each national character of argument-1 converted to the corresponding alphanumeric standard data format representation.

Example

Function with Argument	Result
FUNCTION CONVERT-TO-DISPLAY (N"ABC","?")	ABC

In this example, N"ABC" is a national-character literal.

CONVERT-TO-NATIONAL Function

The CONVERT-TO-NATIONAL function returns a national-character string that contains the characters of the argument converted to the corresponding national-character representation. The type of this function is national.

Syntax

```
FUNCTION CONVERT-TO-NATIONAL ( argument-1 [,argument-2] )
```

Explanation

Argument-1 must be of the alphabetic or alphanumeric class and must be at least one character in length.

Argument-2 must be of the national category and must be one character in length. Argument-2 specifies a substitute national character for use in the conversion of any alphanumeric character for which no corresponding national character exists.

You must separate argument-1 and argument-2 with a comma (,) or a space.

The returned value is a national-character string with each character of argument-1 converted to the corresponding national-character representation.

Example

Function with Argument	Result
FUNCTION CONVERT-TO-NATIONAL ("ABC",N"?")	ABC

In this example, N"?" is a national-character literal, and the result, ABC, is a national-character value.

COS Function

The COS function returns a numeric value that approximates the cosine of an angle or arc, expressed in radians, that is specified by argument-1. The type of this function is numeric. This function is fully supported in the COBOL85 TADS environment.

Syntax

The syntax of this function is as follows:

```
FUNCTION COS (argument-1)
```

Explanation

Argument-1 must be of the numeric class. The returned value is the approximation of the cosine of argument-1 and is greater than or equal to -1 and less than or equal to +1.

Example

Function with Argument	Result
FUNCTION COS (.524)	.866

CURRENT-DATE Function

The CURRENT-DATE function returns a 21-character alphanumeric value that represents the calendar date, time of day, and local time differential factor provided by the system on which the function is evaluated. The type of this function is alphanumeric.

Syntax

```
FUNCTION CURRENT-DATE
```

Explanation

The character positions returned, numbered from left to right, are as follows:

Table 9–4. CURRENT-DATE Function, Characters 1–21

Character Position	Contents
1-4	Four numeric digits of the year in the Gregorian calendar.
5-6	Two numeric digits of the month of the year, in the range 01 through 12.
7-8	Two numeric digits of the day of the month, in the range 01 through 31.
9-10	Two numeric digits of the hours past midnight, in the range 00 through 23.
11-12	Two numeric digits of the minutes past the hour, in the range 00 through 59.
13-14	Two numeric digits of the seconds past the minute, in the range 00 through 59.
15-16	Two numeric digits of the hundredths of a second past the second, in the range 00 through 99.
17	One of the following characters: Minus sign (-), which means the local time indicated in the previous character positions is behind Greenwich Mean Time. Plus sign (+), which means the local time indicated is the same as or ahead of the Greenwich Mean Time. Zero (0), which means the system on which this function is evaluated does not provide the differential factor. Only the values + and - are returned.
18-19	The returned value in character positions 18 and 19 depends upon the character in position 17 as shown in Table Section 9-5.
20-21	The returned value in character positions 20 and 21 depends upon the character in position 17 as shown in Table Section 9-6.

Table 9–5. CURRENT-DATE Function, Characters 18–19

If the 17th character is a . . .	Then the returned value is . . .
Minus sign (-)	Two numeric digits in the range 00 through 12 indicating the number of hours that the reported time is behind Greenwich Mean Time.
Plus sign (+)	Two numeric digits in the range 00 through 13 indicating the number of hours that the reported time is ahead of Greenwich Mean Time.
Zero (0)	00

Table 9–6. CURRENT-DATE Function, Characters 20–21

If the 17th character is a . . .	Then the returned value is . . .
Minus sign (-)	Two numeric digits in the range 00 through 59 indicating the number of minutes that the reported time is behind Greenwich Mean Time.
Plus sign (+)	Two numeric digits in the range 00 through 59 indicating the number of minutes that the reported time is ahead of Greenwich Mean Time.
Zero (0)	00

Example

Function	Result
FUNCTION CURRENT-DATE	1993062813195795-0700

DATE-OF-INTEGER Function

The DATE-OF-INTEGER function converts a date in the Gregorian calendar from integer date form to standard date form (YYYYMMDD). The type of this function is integer.

Syntax

```
FUNCTION DATE-OF-INTEGER  (argument-1)
```

Explanation

Argument-1 is a positive integer that represents a number of days succeeding December 31, 1600, on the Gregorian calendar. An invalid argument will return a value of 0 (zero).

The returned value represents the ISO standard date equivalent of the integer specified in argument-1 in the form (YYYYMMDD) where,

YYYY Represents a year in the Gregorian calendar

MM Represents the month of that year

DD Represents the day of that month

Example

Function with Argument	Result
FUNCTION DATE-OF-INTEGER (1096)	16040101

DAY-OF-INTEGER Function

The DAY-OF-INTEGER function converts a date in the Gregorian calendar from integer date form to Julian date form (YYYYDDD). The type of this function is integer.

Syntax

```
FUNCTION DAY-OF-INTEGER (argument-1)
```

Explanation

Argument-1 is a positive integer that represents a number of days succeeding December 31, 1600, in the Gregorian calendar. An invalid argument will return a value of 0 (zero).

The returned value represents the Julian equivalent of the integer specified in argument-1. The returned value is an integer of the form (YYYYDDD) where YYYY represents a year in the Gregorian calendar and DDD represents the day of that year.

Example

Function with Argument	Result
FUNCTION DAY-OF-INTEGER (1096)	1604001

DIV Function

The DIV function returns an integer equal to the integer part of the quotient after division. The type of this function is integer.

Syntax

```
FUNCTION DIV (argument-1, argument-2)
```

Explanation

Argument-1 represents the dividend, and argument-2 represents the divisor. You must separate each argument with a comma (,) or a space.

The result of argument-2 must not be zero.

Example

Function with Arguments	Result
FUNCTION DIV (10,3)	3

EXP Function

The EXP function returns a value that is the base of the natural system of logarithms raised to the power of argument-1. The type of this function is numeric. This function is fully supported in the COBOL85 TADS environment.

Syntax

```
FUNCTION EXP (argument-1)
```

Explanation

Argument-1 must be of the numeric class. The returned value is equal to the base of the natural system of logarithms, which has an approximate numerical value of 2.7183, raised to the power of argument-1.

Example

Function with Arguments	Result
FUNCTION EXP (2)	7.389056

FACTORIAL Function

The FACTORIAL function returns an integer that is the factorial of argument-1. The type of this function is numeric.

Syntax

The syntax of this function is as follows:

```
FUNCTION FACTORIAL (argument-1)
```

Explanation

Argument-1 must be an integer greater than or equal to zero.

- If argument-1 is zero, the value 1 is returned.
- If argument-1 is positive, its factorial is returned.

An invalid argument results in a run-time error.

Example

Function with Argument	Result
FUNCTION FACTORIAL (10)	3628800

FIRSTONE Function

The FIRSTONE function returns a value that is the bit number, plus 1, of the leftmost non- zero bits in argument-1. The type of this function is integer. This function is fully supported in the COBOL85 TADS environment.

Syntax

```
FUNCTION FIRSTONE (argument-1)
```

Explanation

Argument-1 must be of the numeric class. If argument-1 is a double-precision number, only the first word is evaluated by the function. If argument-1 is 0 (zero), the FIRSTONE function returns a value of 0 (zero). The returned value is the bit number, plus one, of the leftmost non-zero bit in argument-1.

Example

Function with Argument	Result
FUNCTION FIRSTONE (0)	0
FUNCTION FIRSTONE (3)	2

FORMATTED-SIZE Function

The FORMATTED-SIZE function returns as a value the formatted size of a data name. The type of this function is integer.

The returned value is equal to the result of the following calculation:

```
Length of data name in bytes + (number of National data items subordinate to
the data name * 2)
```

Syntax

```
FUNCTION FORMATTED-SIZE (argument-1)
```

Explanation

Argument-1 must be either a group item or any category of elementary item described implicitly or explicitly as USAGE IS DISPLAY or USAGE IS NATIONAL. Argument-1 cannot be a RENAMES entry. In addition, argument-1 must be qualified; it cannot be subscripted or indexed.

Example

Consider the following portion of code:

```
01  D-GROUP.
    02  X-ITEM        PIC 9(5).
    02  N-ITEM        PIC N(5).
```

The formatted size of D-GROUP would be derived as follows:

Function with Argument	Result
FUNCTION FORMATTED-SIZE (D-GROUP)	17

INTEGER Function

The INTEGER function returns the greatest integer value that is less than or equal to the argument. The type of this function is integer. This function is fully supported in the COBOL85 TADS environment.

Syntax

```
FUNCTION INTEGER (argument-1)
```

Explanation

Argument-1 must be of the numeric class. The returned value is the greatest integer less than or equal to the value of argument-1. For example, if the value of argument-1 is -1.5, then -2 is returned. If the value of argument-1 is +1.5, then +1 is returned.

Example

Function with Argument	Result
FUNCTION INTEGER (-2.1)	-3

INTEGER-OF-DATE Function

The INTEGER-OF-DATE function converts a date in the Gregorian calendar from standard date form (YYYYMMDD) to integer date form. The type of this function is integer.

Syntax

```
FUNCTION INTEGER-OF-DATE (argument-1)
```

Explanation

Argument-1 must be an integer in the form YYYYMMDD, whose value is obtained from the calculation (YYYY * 10000) + (MM * 100) + DD.

In the representation YYYYMMDD, . . .	Represents . . .
YYYY	The year in the Gregorian calendar. It must be an integer greater than 1600.
MM	A month. It must be a positive integer less than 13, provided that it is valid for the specified month and year combination.
DD	A day. It must be a positive integer less than 32, provided that it is valid for the specified month and year combination.

An invalid argument will return a value of 0 (zero).

The returned value is an integer that is the number of days the date represented by argument-1 succeeds December 31, 1600, in the Gregorian calendar.

Example

Function with Argument	Result
FUNCTION INTEGER-OF-DATE (16040101)	1096

INTEGER-OF-DAY Function

The INTEGER-OF-DAY function converts a date in the Gregorian calendar from Julian date form (YYYYDDD) to integer date form. This type of function is integer.

```
FUNCTION INTEGER-OF-DAY (argument-1)
```

Explanation

Argument-1 must be an integer in the form YYYYDDD, whose value is obtained from the calculation (YYYY * 1000) + DDD.

In the representation YYYYDDD, . . .	Represents . . .
YYYY	The year in the Gregorian calendar. It must be an integer greater than 1600.
DDD	The day of the year. It must be a positive integer less than 367, provided that it is valid for the year specified.

An invalid argument will return a value of 0 (zero).

The returned value is an integer that is the number of days the date represented by argument-1 succeeds December 31, 1600, in the Gregorian calendar.

Example

Function with Argument	Result
FUNCTION INTEGER-OF-DAY (1604001)	1096

INTEGER-PART Function

The INTEGER-PART function returns an integer that is the integer portion of argument-1. The type of this function is integer. This function is fully supported in the COBOL85 TADS environment.

Syntax

```
FUNCTION INTEGER-PART (argument-1)
```

Explanation

Argument-1 must be of the numeric class. The value of argument-1 determines the value of the result as shown in the following table.

If the value of argument-1 is . . .	Then the returned value is . . .
Zero	Zero.
Positive	The greatest integer less than or equal to the value of argument-1. For example, if the value of argument-1 is +1.5, then +1 is returned.
Negative	The least integer greater than or equal to the value of argument-1. For example, if the value of argument-1 is -1.5, then -1 is returned.

Example

Function with Argument	Result
FUNCTION INTEGER-PART (-5.9)	-5

LENGTH Function

The LENGTH function returns an integer equal to the length of the argument in character positions. The type of this function is integer.

Syntax

```
FUNCTION LENGTH (argument-1)
```

Explanation

Argument-1 can be either a nonnumeric literal or a data item of any class or category.

If argument-1 or any data item subordinate to argument-1 is described with the DEPENDING phrase of the OCCURS clause, the contents of the data item referenced by the data-name specified in the DEPENDING phrase are used at the time the LENGTH function is evaluated.

The type of data in argument-1 determines the returned value, as follows:

If argument-1 is . . .	Then the returned value is . . .
• A nonnumeric literal • An elementary data item • A group data item that omits a variable-occurrence data item	An integer equal to the length of argument-1 in character positions.
• A national literal consisting of multibyte characters	An integer equal to the length of argument-1 in national character positions (not in bytes).
A group data item that contains a variable-occurrence data item	An integer determined by evaluating the data item specified in the DEPENDING phrase of the OCCURS clause for the variable-occurrence data item. The evaluation is done according to the rules of the OCCURS clause for "sending" data items.

The returned value includes implicit FILLER characters, if any.

Examples

Function with Argument	Result
FUNCTION LENGTH ("ABC")	3
FUNCTION LENGTH (N"ABC") Note:N"ABC" is a national-character literal.	3

LENGTH-AN Function

The LENGTH-AN function returns an integer equal to the length of the argument in alphanumeric positions (bytes). The type of this function is integer.

Syntax

```
FUNCTION LENGTH-AN (argument-1)
```

Explanation

Argument-1 can be either a nonnumeric literal or a data item of any class or category.

If argument-1 or any data item subordinate to argument-1 is described with the DEPENDING phrase of the OCCURS clause, the contents of the data item referenced in the DEPENDING phrase are used when the LENGTH function is evaluated.

The type of data in argument-1 determines the returned value, as follows:

If argument-1 is . . .	**Then the returned value is . . .**
• A nonnumeric literal • An elementary data item • A group data item that does not contain a variable-occurrence data item	An integer equal to the length of argument-1 in alphanumeric positions.
• A national literal of multibyte (8-bit or 16-bit) characters	An integer equal to the length of argument-1 in bytes (not in national character positions).
• A group data item that contains a variable-occurrence data item	An integer determined by evaluating the data item specified in the DEPENDING phrase of the OCCURS clause for the variable-occurrence data item. The evaluation is performed according to the rules of the OCCURS clause for "sending" data items.

When argument-1 does not occupy an integral number of alphanumeric character positions, the returned value is rounded to the next larger integer value. The returned value includes implicit FILLER characters, if any.

Examples

Function with Argument	Result
FUNCTION LENGTH-AN ("ABC")	3
FUNCTION LENGTH-AN (N"ABC") ***Note:*** *N"ABC" is an 8-bit national character literal.*	3
FUNCTION LENGTH-AN (N"ABC") ***Note:*** *N"ABC" is a 16-bit national character literal.*	6

LINENUMBER Function

The LINENUMBER function returns an integer value for the sequence number of the source file record on which it appears.

Syntax

```
FUNCTION LINENUMBER
```

Explanation

The source line sequence number is returned as an integer. When used in an INCLUDE file or a COPY statement, the sequence number pertains to the line number in the included file.

Example

Function	Result
000100 DISPLAY FUNCTION LINENUMBER	DISPLAY:100.

LOG Function

The LOG function returns a numeric value that approximates the logarithm to the base e (natural log) of argument-1. The type of this function is numeric. This function is fully supported in the COBOL85 TADS environment.

Syntax

```
FUNCTION LOG (argument-1)
```

Explanation

Argument-1 must be of the numeric class. The value of argument-1 must be greater than zero. An invalid argument results in a run-time error.

The returned value is the approximation of the logarithm to the base e of argument-1.

Example

Function with Argument	Result
FUNCTION LOG (6.429)	1.860

LOG10 Function

The LOG10 function returns a numeric value that approximates the logarithm to the base 10 of argument-1. The type of this function is numeric. This function is fully supported in the COBOL85 TADS environment.

Syntax

```
FUNCTION LOG10 (argument-1)
```

Explanation

Argument-1 must be of the numeric class and must have a value greater than zero. An invalid argument results in a run-time error.

The returned value is the approximation of the logarithm to the base 10 of argument-1.

Example

Function with Argument	Result
FUNCTION LOG10 (6.429)	.808

LOWER-CASE Function

The LOWER-CASE function returns a character string that is the same length as argument-1 with each uppercase letter replaced by the corresponding lowercase letter. The type of this function is alphanumeric or national.

Syntax

```
FUNCTION LOWER-CASE (argument-1)
```

Explanation

Argument-1 must be of either the alphabetic, alphanumeric, or national class and must be at least one character in length.

The same character string as argument-1 is returned, except that each uppercase letter is replaced by the corresponding lowercase letter.

The character string returned has the same length as argument-1.

Example

Function with Argument	Result
FUNCTION LOWER-CASE ("ABC")	abc

MAX Function

The MAX function returns the content of argument-1 that contains the maximum value in the collating sequence for the program. If the type of argument-1 is National with 8-bit national characters, the function returns the content of argument-1 corresponding to the maximum ordinal number in the national collating sequence for the program. The type of this function depends upon the type of argument used with the function. The argument types and the corresponding function types are as follows:

Argument Type	Function Type
Alphabetic	Alphanumeric
Alphanumeric	Alphanumeric
Integer	Numeric
Numeric (some arguments might be integer)	Numeric
National	National

Syntax

```
FUNCTION MAX ({argument-1} ... )
```

Explanation

If more than one argument-1 is specified,

- All arguments must be of the same class.
- Multiple arguments must be separated by a comma (,) or a space.
- The returned value is the content of the argument that has the greatest value. The comparisons used to determine the greatest value are made according to the rules for simple conditions.

  If more than one argument has the same greatest value, the content of the argument returned is the leftmost argument having that value.

If the type of the function is alphanumeric or national, the size of the returned value is the same as the size of the selected argument-1.

Example

Function with Arguments	Result
FUNCTION MAX (17, 5, 11, 25, 52, 1, 17, 10)	52

Considerations for Use

When the MAX function itself is assigned to a DISPLAY, COMP, or BINARY data item, the function results in the exact value.

For example, if A is declared as PIC 9V99, the following statement yields the exact value of 1.80 for A.

```
COMPUTE A = FUNCTION MAX (1.0, 1.8)
```

When the MAX function is contained within an expression and assigned to a DISPLAY, COMP, or BINARY data item, the result of the function is an approximate value. The precision might be lost.

For example, if A is declared as PIC 9V99, the following statement yields the approximate value of 1.79 for FUNCTION MAX (1, 1.8), and yields 3.79 for A.

```
COMPUTE A = FUNCTION MAX (1.0, 1.8) + 2
```

To avoid losing precision, declare B as PIC 9V99, and use the following statements:

```
COMPUTE B = FUNCTION MAX (1.0, 1.8)
COMPUTE A = B + 2
```

MEAN Function

The MEAN function returns a numeric value that is the arithmetic mean (average) of its arguments. The type of this function is numeric.

Syntax

```
FUNCTION MEAN ({argument-1} ... )
```

Explanation

Arguments must be of the numeric class and must be separated by a comma (,) or a space.

The returned value is the arithmetic mean of the argument-1 series, defined as the sum of the argument-1 series divided by the number of occurrences referenced by argument-1.

Example

Function with Arguments	Result
FUNCTION MEAN (17, 5, 11, 25, 52, 1, 17, 10)	17.25

Considerations for Use

When the MEAN function itself is assigned to a DISPLAY, COMP, or BINARY data item, the function results in the exact value.

For example, if A is declared as PIC 9V99, the following statement yields the exact value of 1.80 for A.

```
COMPUTE A = FUNCTION MEAN (1.7, 1.8, 1.9)
```

When the MEAN function is contained within an expression and assigned to a DISPLAY, COMP, or BINARY data item, the result of the function is an approximate value. The precision might be lost.

For example, if A is declared as PIC 9V99, the following statement yields the approximate value of 1.79 for FUNCTION MEAN (1.7, 1.8, 1.9), and yields 3.79 for A.

```
COMPUTE A = FUNCTION MEAN (1.7, 1.8, 1.9) + 2
```

To avoid losing precision, declare B as PIC 9V99, and use the following statements:

```
COMPUTE B = FUNCTION MEAN (1.7, 1.8, 1.9)
COMPUTE A = B + 2
```

MEDIAN Function

The MEDIAN function returns the content of the argument whose value is the middle value in the list formed by arranging the arguments in sorted order. The type of this function is numeric.

Syntax

```
FUNCTION MEDIAN ({argument-1} ... )
```

Explanation

Arguments must be of the numeric class and must be separated by a comma (,) or a space.

The returned value is the content of the argument-1 having the middle value in the list formed by arranging all the argument-1 values in sorted order.

- If the number of occurrences referenced by argument-1 is odd, the returned value is such that at least half of the occurrences referenced by argument-1 are greater than or equal to the returned value and at least half are less than or equal.
- If the number of occurrences referenced by argument-1 is even, the returned value is the arithmetic mean of the values referenced by the two middle occurrences.

  The comparisons used to arrange the argument-1 values in sorted order are made according to the rules for simple conditions.

Example

Function with Arguments	Result
FUNCTION MEDIAN (17, 5, 11, 25, 52, 1, 17, 10)	14

Considerations for Use

When the MEDIAN function itself is assigned to a DISPLAY, COMP, or BINARY data item, the function results in the exact value.

For example, if A is declared as PIC 9V99, the following statement yields the exact value of 1.80 for A.

```
COMPUTE A = FUNCTION MEDIAN (1.0, 1.8, 2.0)
```

When the MEDIAN function is contained within an expression and assigned to a DISPLAY, COMP, or BINARY data item, the result of the function is an approximate value. The precision might be lost.

For example, if A is declared as PIC 9V99, the following statement yields the approximate value of 1.79 for FUNCTION MEDIAN (1, 1.8, 2.0), and yields 3.79 for A.

```
COMPUTE A = FUNCTION MEDIAN (1.0, 1.8, 2.0) + 2
```

To avoid the losing precision, declare B as PIC 9V99, and use the following statements:

```
COMPUTE B = FUNCTION MEDIAN (1.0, 1.8, 2.0)
COMPUTE A = B + 2
```

MIDRANGE Function

The MIDRANGE (middle range) function returns a numeric value that is the arithmetic mean (average) of the values of the minimum argument and the maximum argument. The type of this function is numeric.

Syntax

```
FUNCTION MIDRANGE ( {argument-1} ... )
```

Explanation

Arguments must be of the numeric class and must be separated by a comma (,) or a space.

The returned value is the arithmetic mean of the greatest argument-1 value and the least argument-1 value. The comparisons used to determine the greatest and least values are made according to the rules for simple conditions.

Example

Function with Arguments	Result
FUNCTION MIDRANGE (17, 5, 11, 25, 52, 1, 17, 10)	26.5

Considerations for Use

When the MIDRANGE function itself is assigned to a DISPLAY, COMP, or BINARY data item, the function results in the exact value.

For example, if A is declared as PIC 9V99, the following statement yields the exact value of 1.80 for A.

```
COMPUTE A = FUNCTION MIDRANGE (1.7, 1.8, 1.9)
```

When the MIDRANGE function is contained within an expression and assigned to a DISPLAY, COMP, or BINARY data item, the result of the function is an approximate value. The precision might be lost.

For example, if A is declared as PIC 9V99, the following statement yields the approximate value of 1.79 for FUNCTION MIDRANGE (1.7, 1.8, 1.9), and yields 3.79 for A.

```
COMPUTE A = FUNCTION MIDRANGE (1.7, 1.8, 1.9) + 2
```

To avoid the losing precision, declare B as PIC 9V99, and use the following statements:

```
COMPUTE B = FUNCTION MIDRANGE (1.7, 1.8, 1.9)
COMPUTE A = B + 2
```

MIN Function

The MIN function returns the content of argument-1 that contains the minimum value in the collating sequence for the program. If the type of argument-1 is National with 8-bit national characters, the function returns the content of argument-1 corresponding to the minimum ordinal number in the national collating sequence for the program. The type of this function depends on the argument types as follows:

Argument Type	Function Type
Alphabetic	Alphanumeric
Alphanumeric	Alphanumeric
Integer	Numeric
Numeric (some arguments might be integer)	Numeric
National	National

Syntax

```
FUNCTION MIN ({argument-1} ... )
```

Explanation

If multiple arguments are specified,

- All arguments must be of the same class.
- You must separate arguments with a comma (,) or a space.
- The returned value is the content of the argument that has the least value. The comparisons used to determine the least value are made according to the rules for simple conditions.

  If more than one argument has the same least value, the content of the argument returned is the leftmost argument having that value.

If the type of the function is alphanumeric or national, the size of the returned value is the same as the size of the selected argument-1.

Example

Function with Arguments	Result
FUNCTION MIN (17, 5, 11, 25, 52, 1, 17, 10)	1

Considerations for Use

When the MIN function itself is assigned to a DISPLAY, COMP, or BINARY data item, the function results in the exact value.

For example, if A is declared as PIC 9V99, the following statement yields the exact value of 1.80 for A.

```
COMPUTE A = FUNCTION MIN (1.8, 2.0)
```

When the MIN function is contained within an expression and assigned to a DISPLAY, COMP, or BINARY data item the result of the function is an approximate value. The precision might be lost.

For example, if A is declared as PIC 9V99, the following statement yields the approximate value of 1.79 for FUNCTION MIN (1.8, 2), and yields 3.79 for A.

```
COMPUTE A = FUNCTION MIN (1.8, 2.0) + 2
```

To avoid the losing precision, declare B as PIC 9V99, and use the following statements:

```
COMPUTE B = FUNCTION MIN (1.8, 2.0)
COMPUTE A = B + 2
```

MOD Function

The MOD function returns an integer value that is argument-1 modulo argument-2. The type of this function is numeric.

Syntax

```
FUNCTION MOD (argument-1, argument-2)
```

Explanation

Argument-1 and argument-2 must be integers. The value of argument-2 must not be zero. You must separate argument-1 and argument-2 with a comma (,) or a space.

The returned value is argument-1 modulo argument-2, defined as follows:

```
argument-1 - (argument-2 * FUNCTION INTEGER (argument-1 / argument-2))
```

Examples

Function with Arguments	Result
FUNCTION MOD (11,5)	1
FUNCTION MOD (11,–5)	-4
FUNCTION MOD (-11,5)	4
FUNCTION MOD (-11,-5)	-1

NUMVAL Function

The NUMVAL function returns the numeric value represented by the character string specified by argument-1. The NUMVAL function does not accept an argument that contains a currency sign or commas. (Use the NUMVAL-C function for that purpose.) Leading and trailing spaces are ignored.

The type of this function is numeric.

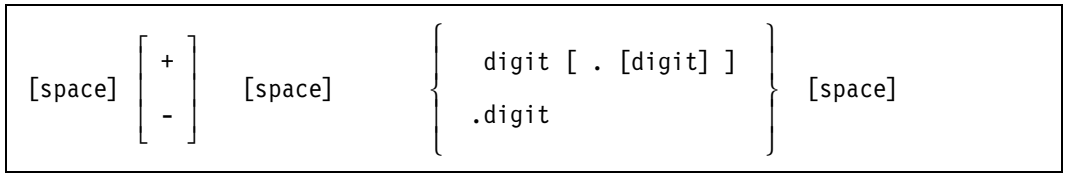
Syntax

```
FUNCTION NUMVAL (argument-1)
```

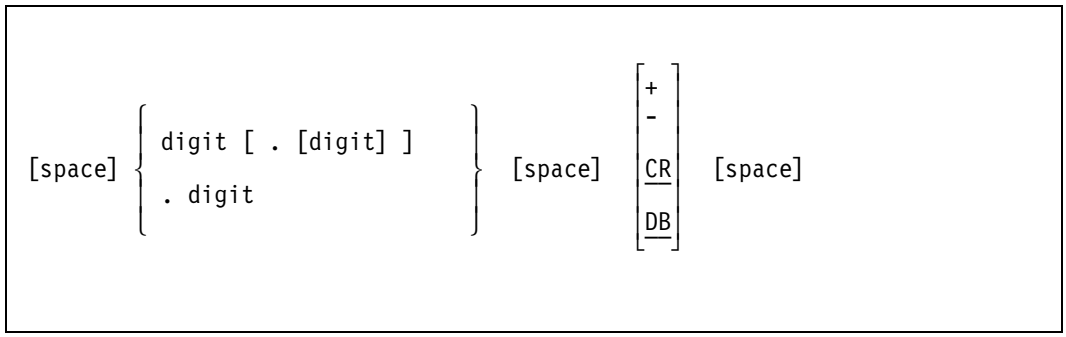
Explanation

Argument-1 must be a nonnumeric literal or alphanumeric data item whose content has one of the following formats:

Format 1:



Format 2:



In the preceding syntax . . .	Represents . . .
Space	A string of zero or more spaces.
Digit	A string of one to 18 digits. The total number of digits in argument-1 must not exceed 18.
CR	A credit.
DB	A debit.

If the DECIMAL-POINT IS COMMA clause is specified in the SPECIAL-NAMES paragraph, a comma (,) must be used in argument-1 rather than a decimal point (.).

An invalid argument results in a run-time error.

The returned value is the numeric value represented by argument-1. The number of digits returned is 18.

Example

Function with Argument	Result
FUNCTION NUMVAL ("37.125DB")	-37.125

The NUMVAL function yields a double-precision result. When the NUMVAL function is assigned to a DISPLAY, COMP, or BINARY data item, and the function is contained within an expression, then the function result is an approximate value and precision can be lost.

For example, if A is declared as PIC 9V99, the following statement yields the exact value of 1.80 for A:

```
COMPUTE A = FUNCTION NUMVAL ("1.8")
```

However, the following statement yields the approximate value of 1.79 for FUNCTION NUMVAL (“1.8”), and yield 3.79 for A:

```
COMPUTE A = FUNCTION NUMVAL ("1.8") + 2
```

The following example avoids this precision problem:

```
Declare B as PIC 9V99

COMPUTE A = FUNCTION NUMVAL ("1.8")

COMPUTE B = A + 2
```

NUMVAL-C Function

The NUMVAL-C function returns the numeric value represented by the character string specified by argument-1. Any optional currency sign specified by argument-2 and any optional commas (,) preceding the decimal point (.) are ignored. The type of this function is numeric.

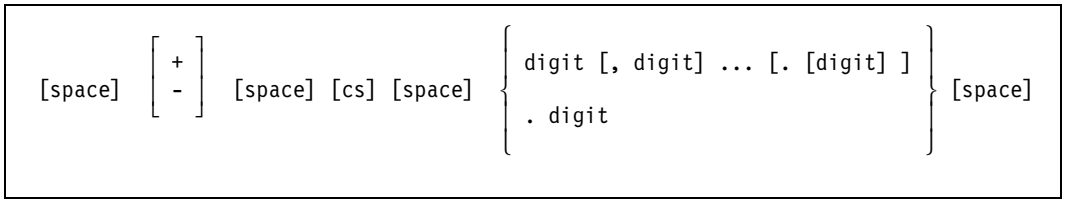
Syntax

```
FUNCTION NUMVAL-C ( argument-1 [,argument-2] )
```

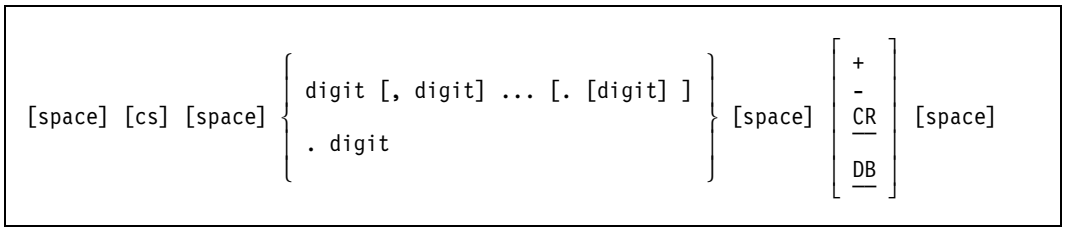
Explanation

Argument-1 must be a nonnumeric literal or an alphanumeric data item that contains a maximum of 18 digits. The content of argument-1 can have one of the following formats:

Format 1



Format 2



In the preceding syntax . . .	Represents . . .
Space	A string of zero or more spaces.
Cs	The string of one or more characters specified by argument-2.
Digit	A string of one to 18 digits. The total number of digits in argument-1 must not exceed 18.
CR	A credit.
DB	A debit.

If the DECIMAL-POINT IS COMMA clause is specified in the SPECIAL-NAMES paragraph, the functions of the comma and decimal point in argument-1 are reversed.

Argument-2, if specified, must be a nonnumeric literal or alphanumeric data item. If argument-2 is not specified, the character used for "cs" is the currency symbol specified for the program.

You must separate argument-1 and argument-2 with a comma (,) or a space.

An invalid argument results in a run-time error.

The returned value is the numeric value represented by argument-1. The number of digits returned is 18.

Example

Function with Argument	Result
FUNCTION NUMVAL-C ("$ 1,23,567.89 CR")	-1234567.89

The NUMVAL-C function yields a double-precision result. When the NUMVAL function is assigned to a DISPLAY, COMP, or BINARY data item, and the function is contained within an expression, then the function result is an approximate value and precision can be lost.

ONES Function

The ONES function returns a value that is the number of non-zero bits in argument-1. The type of this function is integer. This function is fully supported in the COBOL85 TADS environment.

Syntax

```
FUNCTION ONES (argument-1)
```

Explanation

Argument-1 must be of the numeric class. It can be either a single-precision or a double-precision expression. The returned value is the number of non-zero bits in argument-1.

Example

Function with Argument	Result
FUNCTION ONES (3)	2

ORD Function

The ORD function returns an integer value that is the ordinal position of argument-1 in the collating sequence or in the national collating sequence for the program. The lowest ordinal position is 1. The type of this function is integer.

Syntax

```
FUNCTION ORD (argument-1)
```

Explanation

Argument-1 must be one character in length and must be class alphabetic, alphanumeric, or national. An invalid argument results in a run-time error.

The returned value is the ordinal position of argument-1 in the collating sequence or in the national collating sequence for the program.

Example

Function with Argument	Result
FUNCTION ORD ("]")	91

ORD-MAX Function

The ORD-MAX function returns a value that is the ordinal number of the argument-1 that contains the maximum value in the collating sequence or in the national collating sequence for the program. The type of this function is integer.

Syntax

```
FUNCTION ORD-MAX ({argument-1} ... )
```

Explanation

If multiple arguments are specified,

- All arguments must be of the same class.
- Arguments must be separated with a comma (,) or a space.
- The returned value is the ordinal number that corresponds to the position of the argument that has the greatest value of all the arguments in the series. The comparisons used to determine the argument with the greatest value are made according to the rules for simple conditions.

  If more than one argument has the same greatest value, the number returned corresponds to the position of the leftmost argument having that value.

Example

Function with Arguments	Result
FUNCTION ORD-MAX (17, 5, 11, 25, 52, 1, 17, 10)	5

ORD-MIN Function

The ORD-MIN function returns a value that is the ordinal number of the argument-1 that contains the minimum value in the collating sequence or in the national collating sequence for the program. The type of this function is integer.

Syntax

```
FUNCTION ORD-MIN ({argument-1} ... )
```

Explanation

If multiple arguments are specified,

- All arguments must be of the same class.
- Arguments must be separated with a comma (,) or a space.
- The returned value is the ordinal number that corresponds to the position of the argument that has the least value of all the arguments in the series. The comparisons used to determine the argument with the least value are made according to the rules for simple conditions.

  If more than one argument-1 has the same least value, the number returned corresponds to the position of the leftmost argument having that value.

Example

Function with Arguments	Result
FUNCTION ORD-MIN (17, 5, 11, 25, 52, 1, 17, 10)	6

PRESENT-VALUE Function

The PRESENT-VALUE function returns a value that approximates the present value of a series of future period-end amounts specified by argument-2 at a discount rate specified by argument-1. The type of this function is numeric.

Syntax

```
FUNCTION PRESENT-VALUE (argument-1 {argument-2} ... )
```

Explanation

Argument-1 and argument-2 must be of the numeric class. The value of argument-1 must be greater than -1. Argument-1 and argument-2 must be separated with a comma (,) or a space.

An invalid argument results in a run-time error.

The returned value is an approximation of the summation of a series of calculations with each term in the following form:

```
argument-2 / (1 + argument-1) ** n
```

One term exists for each occurrence of argument-2. The exponent, n, starts at 1 and is incremented by 1 for each term in the series.

Example

Function with Arguments	Result
FUNCTION PRESENT-VALUE (.05, 5, 4, 3, 2, 1)	13.410

RANDOM Function

The RANDOM function returns a numeric value that is a pseudo-random number from a rectangular distribution. The type of this function is numeric.

Syntax

```
FUNCTION RANDOM [(argument-1)]
```

Explanation

If specified, argument-1 must be 0 (zero) or a positive integer. It is used as the seed value to generate a sequence of pseudo-random numbers. For a given seed value, the sequence of pseudo-random numbers is always the same.

- If a subsequent reference specifies argument-1, a new sequence of pseudo-random numbers is started.
- If the first reference to this function in the run unit does not specify argument-1, then some predefined seed value is used.

  In each of the preceding cases, subsequent references that do not specify argument-1 return the next number in the current sequence.

An invalid argument results in a run-time error.

The returned value is greater than or equal to 0 (zero) and less than 1.

Example

Function with Argument	Result
FUNCTION RANDOM (32)	.093

RANGE Function

The RANGE function returns a value that is equal to the value of the maximum argument minus the value of the minimum argument. The type of this function depends on the type of argument used with the function. The argument types are as follows:

Argument Type	Function Type
Integer	Numeric
Numeric (some arguments might be integer)	Numeric

Syntax

```
FUNCTION RANGE ({argument-1} ... )
```

Explanation

Arguments must be of the numeric class and must be separated by a comma (,) or a space.

The returned value is equal to the value of the maximum argument minus the value of the minimum argument. The comparisons used to determine the maximum and minimum values are made according to the rules for simple conditions.

Example

Function with Arguments	Result
FUNCTION RANGE (17, 5, 11, 25, 52, 1, 17, 10)	51

Considerations for Use

When the RANGE function itself is assigned to a DISPLAY, COMP, or BINARY data item, the function results in the exact value.

For example, if A is declared as PIC 9V99, the following statement yields the exact value of 1.80 for A.

```
COMPUTE A = FUNCTION RANGE (1.0, 2.0, 2.8)
```

When the RANGE function is contained within an expression and assigned to a DISPLAY, COMP, or BINARY data item, the result of the function is an approximate value. The precision might be lost.

For example, if A is declared as PIC 9V99, the following statement yields the approximate value of 1.79 for FUNCTION RANGE (1.0, 2.0, 2.8), and yields 3.79 for A.

```
COMPUTE A = FUNCTION RANGE (1.0, 2.0, 2.8) + 2
```

To avoid the losing precision, declare B as PIC 9V99, and use the following statements:

```
COMPUTE B = FUNCTION RANGE (1.0, 2.0, 2.8)
COMPUTE A = B + 2
```

REM Function

The REM function returns a numeric value that is the remainder of argument-1 divided by argument-2. The type of this function is numeric.

Syntax

```
FUNCTION REM (argument-1, argument-2)
```

Explanation

Argument-1 and argument-2 must be of the numeric class and must be separated by a comma (,) or a space. The value of argument-2 must not be 0 (zero).

The returned value is the remainder of (argument-1/argument-2). It is defined as the expression:

```
argument-1 - (argument-2 * FUNCTION INTEGER-PART (argument-1/argument-2))
```

Example

Function with Arguments	Result
FUNCTION REM (222.2, 70)	12.2

REVERSE Function

The REVERSE function returns a character string that contains exactly the same characters as in argument-1, except in reverse order. The type of this function is alphanumeric or national.

Syntax

```
FUNCTION REVERSE (argument-1)
```

Explanation

Argument-1 must be of the alphabetic, alphanumeric, or national class and must be at least one character in length. If argument-1 is a character string of length n, the returned value is a character string of length n such that for 1 < = j < = n, the character position j of the returned value is the character from position (n-j+1) of argument-1.

Example

Function with Argument	Result
FUNCTION REVERSE ("ABC")	CBA

SIGN Function

The SIGN function returns a value that indicates whether the contents of argument-1 is greater than, less than, or equal to 0 (zero). This function is fully supported in the COBOL85 TADS environment.

Syntax

```
FUNCTION SIGN (argument-1)
```

Explanation

Argument-1 must be of the numeric class.

If the value of argument-1 is . . .	Then the returned value is . . .
Greater than zero	+1
Less than zero	-1
Equal to zero	0

Example

Function with Argument	Result
FUNCTION SIGN (-8)	-1

SIN Function

The SIN function returns a numeric value that approximates the sine of an angle or arc, expressed in radians, that is specified by argument-1. The type of this function is numeric. This function is fully supported in the COBOL85 TADS environment.

Syntax

```
FUNCTION SIN (argument-1)
```

Explanation

Argument-1 must be of the numeric class. The returned value is the approximation of the sine of argument-1 and is greater than or equal to -1 and less than or equal to +1.

Example

Function with Argument	Result
FUNCTION SIN (.5326)	.507

SQRT Function

The SQRT function returns a numeric value that approximates the square root of argument-1. The type of this function is numeric. This function is fully supported in the COBOL85 TADS environment.

Syntax

```
FUNCTION SQRT (argument-1)
```

Explanation

Argument-1 must be class numeric. The value of argument-1 must be 0 (zero) or positive. An invalid argument results in a run-time error.

The returned value is the absolute value of the approximation of the square root of argument-1.

Example

Function with Argument	Result
FUNCTION SQRT (10)	3.162

STANDARD-DEVIATION Function

The STANDARD-DEVIATION function returns a numeric value that approximates the standard deviation of its arguments. The type of this function is numeric.

Syntax

```
FUNCTION STANDARD-DEVIATION ({argument-1} ... )
```

Explanation

Arguments must be of the numeric class and must be separated by a comma (,) or a space.

The returned value is the approximation of the standard deviation of the argument-1 series, calculated as follows:

1. The difference between each argument-1 value and the arithmetic mean of the argument-1 series is calculated and squared.
2. The resulting values are added together.
3. The sum from the preceding addition is divided by the number of values in the argument-1 series.
4. The square root of the quotient from the preceding division is calculated. The returned value is the absolute value of this square root.

If the argument-1 series consists of only one value, or if the argument-1 series consists of all variable occurrence data items and the total number of occurrences for all of them is one, the return value is 0 (zero).

Example

Function with Arguments	Result
FUNCTION STANDARD-DEVIATION (7, 22, 12, 5, 6, 7, 10, 11)	5.099

SUM Function

The SUM function returns a value that is the sum of the arguments. The type of this function depends upon the argument types as follows:

Argument Type	Function Type
Integer	Numeric
Numeric (some arguments might be integer)	Numeric

Syntax

```
FUNCTION SUM ({argument-1} ... )
```

Explanation

Arguments must be of the numeric class and must be separated by a comma (,) or a space. The returned value is the sum of the arguments.

Example

Function with Arguments	Result
FUNCTION SUM (7, 5, 11, 25, 52, 1, 17, 52)	170

Considerations for Use

When the SUM function itself is assigned to a DISPLAY, COMP, or BINARY data item, the function results in the exact value.

For example, if A is declared as PIC 9V99, the following statement yields the exact value of 1.80 for A.

```
COMPUTE A = FUNCTION SUM (0.2, 0.6, 1.0)
```

When the SUM function is contained within an expression and assigned to a DISPLAY, COMP, or BINARY data item, the result of the function is an approximate value. The precision might be lost.

For example, if A is declared as PIC 9V99, the following statement yields the approximate value of 1.79 for FUNCTION SUM (0.2, 0.6, 1.0), and yields 3.79 for A.

```
COMPUTE A = FUNCTION SUM (0.2, 0.6, 1.0) + 2
```

To avoid the losing precision, declare B as PIC 9V99, and use the following statements:

```
COMPUTE B = FUNCTION SUM (0.2, 0.6, 1.0)
COMPUTE A = B + 2
```

TAN Function

The TAN function returns a numeric value that approximates the tangent of an angle or arc, expressed in radians, that is specified by argument-1. The type of this function is numeric. This function is fully supported in the COBOL85 TADS environment.

Syntax

```
FUNCTION TAN (argument-1)
```

Explanation

Argument-1 must be of the numeric class. The returned value is the approximation of the tangent of argument-1.

Example

Function with Argument	Result
FUNCTION TAN (.7854)	1.0

UPPER-CASE Function

The UPPER-CASE function returns a character string that is the same length as argument-1 with each lowercase letter replaced by the corresponding uppercase letter. The type of this function is alphanumeric or national.

Syntax

```
FUNCTION UPPER-CASE (argument-1)
```

Explanation

Argument-1 must be at least one character in length and must be of either the alphabetic, alphanumeric, or national class.

The returned value is the same character string as argument-1, except that each lowercase letter is replaced by the corresponding uppercase letter. The returned character string has the same length as argument-1.

Example

Function with Argument	Result
FUNCTION UPPER-CASE ("abc")	ABC

VARIANCE Function

The VARIANCE function returns a numeric value that approximates the variance of its arguments. The type of this function is numeric.

Syntax

```
FUNCTION VARIANCE ({argument-1} ... )
```

Explanation

Arguments must be of the numeric class and must be separated by a comma (,) or a space.

The returned value is the approximation of the variance of the series of arguments and is defined as the square of the standard deviation of the series of arguments. For more information, refer to "STANDARD-DEVIATION Function."

The returned value is 0 (zero) in the following situations:

- If the series of arguments consists of only one value
- If the series of arguments consists only of variable-occurrence data items and the total number of occurrences for all of the data items is one

Example

Function with Arguments	Result
FUNCTION VARIANCE (7, 22, 12, 5, 6, 7, 10, 11)	26

WHEN-COMPILED Function

The WHEN-COMPILED function returns the date and time the program was compiled. The type of this function is alphanumeric.

Syntax

```
FUNCTION WHEN-COMPILED
```

Explanation

The returned value is the date and time of compilation of the program that contains the function. If the program is a contained program, the returned value is the compilation date and time associated with the separately compiled program in which it is contained.

The returned value denotes the same time as the compilation date and time if provided in the listing of the source program and in the generated object code for the source program, although their representations and precisions might differ. The characters in the returned value provide the information in the following tables.

Table 9–7. WHEN-COMPILED Function, Characters 1–21

Character Positions	Contents
1-4	Four numeric digits of the year in the Gregorian calendar
5-6	Two numeric digits of the month of the year, in the range 01 through 12
7-8	Two numeric digits of the day of the month, in the range 01 through 31
9-10	Two numeric digits of the hours past midnight, in the range 00 through 23
11-12	Two numeric digits of the minutes past the hour, in the range 00 through 59
13-14	Two numeric digits of the seconds past the minute, in the range 00 through 59
15-16	Two numeric digits of the hundredths of a second past the second, in the range 00 through 99
17	One of the following characters: Minus sign (-), which means the local time indicated in the previous character positions is behind Greenwich Mean Time. Plus sign (+), which means the local time indicated is the same as or ahead of the Greenwich Mean Time. Zero (0), which means the system on which this function is evaluated does not provide the differential factor. Only the values + and - are returned.

Table 9–7. WHEN-COMPILED Function, Characters 1–21

Character Positions	Contents
18-19	The returned value in character positions 18 and 19 depends upon the character in position 17 as shown in Table Section 9-8.
20-21	The returned value in character positions 20 and 21 depends upon the character in position 17 as shown in Table Section 9-9.

Table 9–8. WHEN-COMPILED Function, Characters 18–19

If character 17 is a . . .	Then the returned value is . . .
Minus sign (-)	Two numeric digits in the range 00 through 12 indicating the number of hours that the reported time is behind Greenwich Mean Time.
Plus sign (+)	Two numeric digits in the range 00 through 13 indicating the number of hours that the reported time is ahead of Greenwich Mean Time.
Zero (0)	00

Table 9–9. WHEN-COMPILED Function, Characters 20–21

If character 17 is a . . .	Then the returned value is . . .
Minus sign (-)	Two numeric digits in the range 00 through 59 indicating the number of minutes that the reported time is behind Greenwich Mean Time.
Plus sign (+)	Two numeric digits in the range 00 through 59 indicating the number of minutes that the reported time is ahead of Greenwich Mean Time.
Zero (0)	00

Example

Function	Result
FUNCTION WHEN-COMPILED	1993062813391247-0700

Section 10
Interprogram Communication

The ANSI COBOL85 interprogram communication (IPC) facility enables programs to communicate with each other to form a complete solution to a data processing problem.

This section explains the major concepts involved with IPC, which are as follows:

- The run unit
- Nested source programs
- How files and data are accessed
- External and internal objects
- Common and initial programs
- Naming conventions
- The four IPC forms of communication
- How ANSI IPC constructs are used

This section also includes a list of COBOL constructs necessary for IPC and some IPC coding examples.

Note that COBOL85 also provides other program communication techniques not related to IPC. These techniques and where to find information on them are listed in Table 10–1.

Table 10–1. COBOL85 Program Communication Techniques

Communication Technique	Where to Find Details
Tasking	Section 11
Storage Queue (STOQ)	Section 8, SEND Statement Format 2
Core-to-Core (CRCR)	Section 8, SEND Statement Format 1
Binding	Appendix E

Note that the COBOL IPC implementation is a subset of the capabilities available through the library facility. For more information on the library facility and COBOL85, refer to Section 9 and to "CALL Statement" in Section 6.

The Run Unit

A run unit is a complete problem solution that consists either of an object program or of several intercommunicating object programs. A run unit is an independent entity that can be executed without communicating with, or being coordinated with, any other run unit. However, a run unit can process data files and messages or set and test switches that were written or will be read by other run units.

When a program is called through a CALL statement, parameters can be passed to it by the program that calls it.

Nested Source Programs

A COBOL source program can contain, or nest, other COBOL source programs. Nested programs can include references to resources from the programs in which they are contained.

A nested program can be contained either directly or indirectly. Figure 10–1 illustrates the difference:

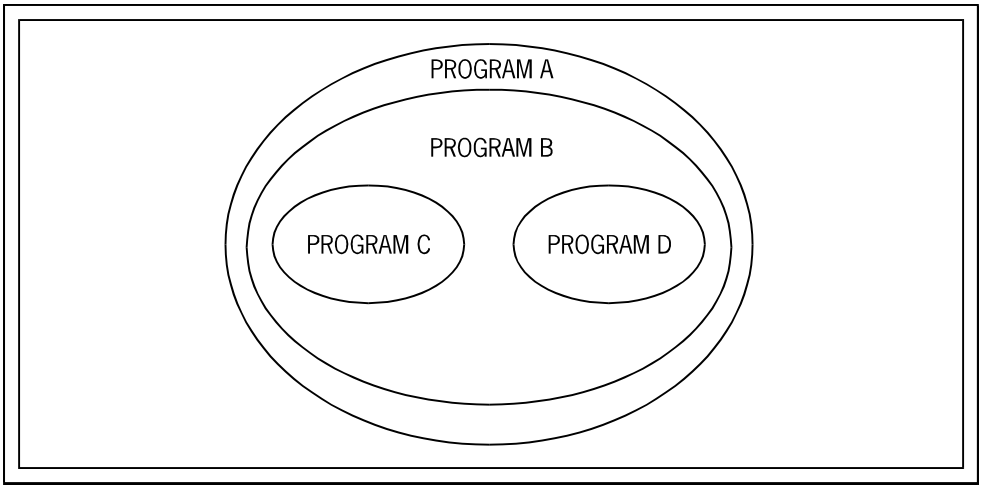


Figure 10–1. Nested Source Programs

In the preceding figure,

- Program A contains Program B
- Program B contains Program C and Program D

Program A contains Program B directly, and Program B contains Program C and Program D directly. However, Program A contains Program C and Program D indirectly because they are contained in another program (Program B), which is contained directly within Program A.

Accessing Files and Data in a Run Unit

Programs in a run unit sometimes need to access and have storage areas for

- The position and status of a file and other attributes of file processing
- Data item values and other attributes

File Connectors

A file connector is a storage area that contains information about a file. A file connector is used as the link between a file-name and a physical file, and between a file-name and its associated record area.

Global and Local Names

A data-name names a data item. A file-name names a file connector. Data-names and file-names can be either global or local.

A global name can refer to its associated object either from the program where the global name is declared or from any other program that is contained in the program that declares the global name.

Consider the following example. Program A contains Program B. Names declared as global in Program A can be accessed by Program B. However, names declared as global in Program B cannot be accessed by Program A.

A local name, however, can refer to its associated object only from the program where the local name is declared.

Some names are always global; some are always local. Other names can be either local or global depending upon specifications in the program in which the names are declared.

A file-name is global if the GLOBAL clause is specified in the file description entry for that file-name.

A record-name is global

- If the GLOBAL clause is specified in the record description entry by which the record-name is declared
- If the GLOBAL clause is specified in the file description entry for the file-name associated with the record description entry

  A data-name is global if the GLOBAL clause is specified either in the data description entry by which the data-name is declared or in another entry to which that data description entry is subordinate.

A condition-name declared in a data description entry is global if that entry is subordinate to another entry in which the GLOBAL clause is specified. However, specific rules sometimes prohibit specification of the GLOBAL clause for certain data description, file description, or record description entries. Refer to "GLOBAL Clause" in Section 4 for syntax and detailed information.

If a data-name, a file-name, or a condition-name declared in a data description entry is not global, then the name is local.

External and Internal Objects

File connectors usually require storage of certain file information. Accessible data items usually require storage of certain data representations. The storage associated with a file connector or a data item can be external or internal to the program in which the object is declared.

A file connector or data item is external if the storage associated with that object is associated with the run unit rather than with any particular program in the run unit. Any program in the run unit that describes an external object can refer to that object. Moreover, different programs that describe the same external object can refer to that object. However, there is only one representative of an external object in a run unit.

An object is internal if the storage associated with that object is associated only with the program that describes the object.

External and internal objects can have either global or local names.

A file connector receives the external attribute through the EXTERNAL clause in the associated file description entry. If the file connector does not have the external attribute, it is internal to the program in which the associated file-name is described.

A data record described in the Working-Storage Section receives the external attribute if it has the EXTERNAL clause in its data description entry. A data item is also considered external if it is described by a data description entry subordinate to an entry describing an external record. If a record or data item does not have the external attribute, it is part of the internal data of the program in which it is described.

Data records are always internal to the program that describes the file-name when they are described in one of the following ways:

- Subordinate to a file description entry that does not contain the EXTERNAL clause
- Subordinate to a sort-merge file description entry

This is also true for data items described as subordinate to the data description entries for such records. If the EXTERNAL clause is included in the file description entry, the data records and the data items receive the external attribute.

Data records, subordinate data items, and associated control information described in the Linkage Section of a program are internal to the program describing that data. Special considerations apply to data described in the Linkage Section, where an association is made between the data records described and other data items accessible to other programs. Refer to “Linkage Section” in Section 4 for more information.

Refer to “EXTERNAL Clause” in Section 4 for detailed information on this clause.

Common and Initial Programs

As an option, all programs in a run unit can have common and initial attributes.

A common program is directly contained in another program and can be called by any program contained in the program in which the common program resides.

Using the IS COMMON PROGRAM clause in a program's Identification Division enables the program to receive the common attribute. The COMMON clause makes it easier to write subprograms that will be used by all the programs contained in a program.

An initial program is one whose program state is initialized when the program is called. When an initial program is called, its program state is the same as it was when the program was first called in that run unit. The IS INITIAL PROGRAM clause in the program's Identification Division gives the program the initial attribute.

When an initial program is initialized, its internal data is also initialized. Data items whose description contains a VALUE clause will be initialized to that defined value. However, an item whose description does not contain a VALUE clause is initialized to an undefined value. When an initial program is initialized, the file connectors associated with the program are not in the open mode. Additionally, the control mechanisms for all PERFORM statements contained in the program are set to their initial states.

For the general formats of these clauses, refer to Section 2.

Scope of Names

User-defined words in a program refer only to the objects in that program. Thus, programs in a run unit, as well as nested programs and the program in which they are nested can have identical user-defined words.

You can refer to the following types of user-defined words only with statements and entries in the program in which the user-defined word is declared:

paragraph-name
section-name

However, any COBOL program can refer to the following types of user-defined words:

library-name
text-name

When the following types of names are declared in the Configuration Section of a program, you can refer to these names with statements and entries in that program. You can also refer to these names in any program that is contained in the referring program.

alphabet-name
class-name
condition-name
mnemonic-name
symbolic-character

Specific conventions for declarations and references apply to the following types of user-defined words when the conditions listed previously do not apply:

condition-name
data-name
file-name
index-name
program-name
record-name

Refer to "User-Defined Words" in Section 1 for information on the different types of user-defined words.

Conventions for Program-Names

The name of a program is declared in the PROGRAM-ID paragraph of the program's Identification Division. A program-name can be referred to only by the CALL statement, the CANCEL statement, and the end program header.

The program-names allocated to the programs of a run unit are not necessarily unique. However, when two programs in a run unit are identically named, one of them must be directly or indirectly contained in another separately compiled program that does not contain the other of those two programs.

Certain conventions apply when, in a separately compiled program, a name identical to that specified for another separately compiled program in the run unit is specified for a contained program.

Consider the situation illustrated by Figure 10–2:

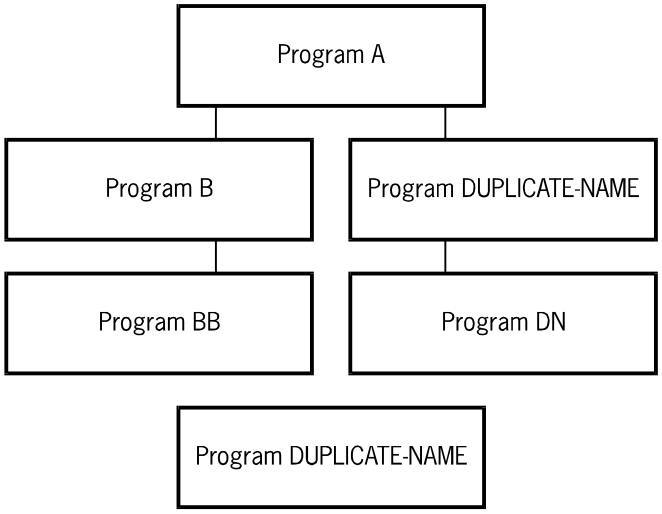


Figure 10–2. Identical Program-Names

- Program A contains program B and program DUPLICATE-NAME.
- Program B contains program BB.
- Program DUPLICATE-NAME contains program DN.
- The name DUPLICATE-NAME has also been specified for a separately compiled program.
- If program A, but not any of the programs it contains, calls program DUPLICATE-NAME, the program activated is the one contained in program A.

- If either program B or program BB calls program DUPLICATE-NAME then:
  - If the program DUPLICATE-NAME (contained in program A) possesses the common attribute, it is called.
  - If the program DUPLICATE-NAME contained in program A does not possess the common attribute, the separately compiled program is called.
- If either program DN or the program DUPLICATE-NAME contained in program A calls program DUPLICATE-NAME, the program called is the separately compiled program.
- If any other separately compiled program in the run unit or any other program contained in such a program calls the program DUPLICATE-NAME, the program called is the separately compiled program named DUPLICATE-NAME.

Conventions for Names of Data, Files, and Records

Condition-names, data-names, file-names, and record-names can be referred to by the program in which they are declared.

A program cannot refer to any condition-name, data-name, file-name, or record-name declared in any program it contains. However, a global name can be referred to in the program in which it is declared or in any programs that are directly or indirectly contained in that program.

For example, if Program B is directly contained in Program A, both programs can define a condition-name, data-name, file-name, or record-name with the same user-defined word. When such a duplicated name is referred to in Program B, the following rules determine the referenced object:

- The names you can use for the referenced object consist of all names defined in Program B and all global names defined in Program A. (This also includes global names defined any programs that directly or indirectly contain Program A.)

  The normal rules for qualification and any other rules for uniqueness of reference apply to these names until one or more objects is identified.
- If only one object is identified, it is the referenced object.
- Only one object can have a name local to Program B, even though more than one object can be identified. If none or one of the objects has a name local to Program B, the following rules apply:
  - If the name is declared in Program B, the object in Program B is the referenced object.
  - If Program A is contained in another program, the referenced object is either the object in Program A, if the name is declared in Program A or the object in the containing program, if the name is not declared in Program A and is declared in the program containing Program A. This rule is applied to other related containing programs until a single valid name has been found.

Refer to "User-Defined Words" in Section 1 for the requirements governing the uniqueness of the names allocated by a single program to be condition names, data-names, file-names, and record-names.

Conventions for Index-Names

If a data item possesses the global attribute, and it includes a table accessed by an index, that index also possesses the attribute.

Therefore, the scope of an index-name is identical to the scope of the data-name that names the table whose index is named by that index-name. The scope of name rules for data-names apply to index-names as well.

Index-names cannot be qualified.

Forms of Interprogram Communication

IPC can take four forms:

- Transfer of control
- Passing of parameters
- Reference to common data
- Reference to common files

These four forms of communication are provided when the communicating programs are separately compiled and when one of the communicating programs is contained in the other program.

Transfer of Control

The CALL statement transfers control from one program to another program in a run unit. A called program can itself contain CALL statements.

When control is transferred to a called program, execution proceeds from statement to statement from the first nondeclarative statement.

If control reaches an EXIT PROGRAM statement, this signals the logical end of the execution of the called program only. Control then reverts to the next executable statement following the CALL statement in the calling program. Thus, the EXIT PROGRAM statement terminates only the execution of the program in which it occurs, while the STOP RUN statement terminates the execution of a run unit.

The name assigned to a called program must be unique. This rule applies whether the called program is contained directly or indirectly in another program.

Scope of the CALL Statement

The following rules apply to the scope of the CALL statement:

- Any calling program can call any separately compiled program in the run unit.
- A calling program can call any program directly contained in the calling program.
- A calling program can call any program that possesses the common attribute and is directly contained in a program that itself contains (directly or indirectly) the calling program. However, this rule does not apply if the calling program is contained in the program that possesses the common attribute.
- A calling program can call a program that neither possesses the common attribute nor is separately compiled if, and only if, that program is directly contained in the calling program.

Passing Parameters to Programs

In many cases, it is necessary for the calling program to define to the called program the precise part of the problem solution to be executed. The calling program can make such data values available to the called program in one of the following ways:

- By passing the data values as parameters
- By sharing the data values as parameters
- By sharing the data values externally

The calling program can pass data values as parameters to the called program by using either the CALL...USING BY CONTENT or CALL...USING BY REFERENCE statement. For details about the BY CONTENT and BY REFERENCE phrases, refer to "CALL Statement" Format 1 in Section 6.

For a discussion of how data is shared, refer to "Sharing Data" later in this section.

Identifying Parameters

To ensure that data passed as a parameter by a calling program to another program is accessible to the calling program, the data item that will receive the data must be declared in the Data Division of the called program.

In the called program, you identify the parameters by listing references to the names assigned in that program's data description entries to the parameters in that program's Procedure Division header.

In the calling program, you identify the values of the parameters to be passed by the calling program by listing references in the CALL statement. At object time, these lists establish the correspondence between the values as they are known to each program. The correspondence is based on position. That is, the first parameter on one list corresponds to the first parameter on the other, the second to the second, and so forth.

Thus, a program can be called by another program; as shown in the following example:

```
PROGRAM-ID. EXAMPLE.
 .
 .
 .
 PROCEDURE DIVISION USING NUM, PCODE, COST.
```

The program can be called by executing:

```
CALL "EXAMPLE" USING NBR, PTYPE, PRICE.
```

This establishes the following correspondence. Only the positions of the data-names are significant, not the names themselves.

Called program (example)	Calling program
NUM	NBR
PCODE	PTYPE
COST	PRICE

Values of Parameters

The calling program controls the methods by which a called program evaluates the parameters passed to it and by which the called program returns results. Results are returned as modified parameter values.

The individual parameters referred to in the USING phrase of the CALL statement can be passed either by reference or by content.

When a parameter is passed by reference, a called program can access and modify the value of the data item referred to in the calling program's CALL statement. When a parameter is passed by content, the called program cannot modify the data item in the calling program.

The value of the parameter is evaluated when the CALL statement is executed and presented to the called program. This value can be changed by the called program during the course of its execution. Note that the value of the parameter passed by reference can be used by a called program to return to the calling program, whereas a parameter passed by content cannot be so used.

The parameters referred to in a called program's Procedure Division header must be described in the Linkage Section of that program's Data Division.

Passing Parameters Explicitly and Implicitly

Parameters can be passed either explicitly or implicitly. A parameter is passed explicitly if the parameter is specified in the USING phrase of a CALL statement. A parameter is passed implicitly in the following situations:

- The data item is subordinate to the data item specified in the USING phrase of the CALL statement, as in the following example:

```
WORKING-STORAGE SECTION.
01  Item-A.
     05 Part-1 PIC X(5).
     05 Part-2 PIC X(5).
          .
          .
          .
     CALL PROGB USING Item-A.
```

  Item-A is passed explicitly. Part-1 and Part-2 are passed implicitly.

- The data item is defined with a REDEFINES or RENAMES clause, as in the following example:

```
WORKING-STORAGE SECTION.
01  Item-A  PIC X(10).
01  Item-B REDEFINES Item-A.
     05 PIC X(5).
     05 PIC X(5).
          .
          .
          .
     CALL PROGB USING Item-A.
```

  Item-A is passed explicitly. Item-B is passed implicitly.

Sharing Data

Two programs in a run unit can refer to common data under the following circumstances:

- Any program can refer to the data content of an external data record if the referring program has described that data record.
- If a program is contained in another program, both programs can refer to data that possesses the global attribute, either in the containing program or in any program that directly or indirectly contains the containing program.
- The way a parameter value is passed by reference from a calling program to a called program establishes a common data item. The called program can refer to a data item in the calling program.

Sharing Files

Programs in a run unit can share files by referring to common file connectors.

Two programs in a run unit can refer to common file connectors under the following circumstances:

- You can refer to an external file connector from any program that describes that file connector.
- If a program contains another program, both programs can refer to a common file connector by referring to an associated global file-name, either in the containing program or in any program that directly or indirectly contains the containing program. (A global file-name is a file-name declared in only one program but which can be referred to from that program and any program contained in that program.)

Using the ANSI IPC Constructs

The following list shows the COBOL constructs to use for standard ANSI IPC operations:

IDENTIFICATION DIVISION

- The PROGRAM-ID paragraph enables you to specify the name by which a program is identified and to assign program attributes to that program.

  Do not assign the same name to a nested program as that of any other program contained in the separately compiled program that contains this program.

  You can use the COMMON clause only if the program is nested. The COMMON clause specifies that the program can be called from programs other than the one containing it.

  Use the INITIAL clause to specify that when the program is called, it and any programs it contains will be used in their initial state (that is, as they were when they first entered the run unit).

  If you are nesting programs, each program must contain an end program header. The program-name declared in the Program-ID paragraph must match the program-name in the end program header.

  Refer to "Program-ID Paragraph" in Section 2, and "End Program Header" in Section 5 for a discussion of syntax and concepts.

DATA DIVISION

- Describe data items that will be referenced by the calling program and the called program in the Linkage Section.

  Use the Linkage Section if the program will be called and the CALL statement in the calling program contains a USING phrase in its Procedure Division header.

  Data items in the Linkage Section (levels 01 or 77) can be contiguous or noncontiguous.

  The VALUE clause must not be specified in the Linkage Section except in condition-name entries (level 88).

- Use the File Description IPC Formats 1 through 3. The file description entry in the File Section determines the internal or external attributes of a file connector, the associated data records, the associated data items, and whether a file-name is local or global.

  Format 1 is the file description entry for a sequential file; Format 2 for a relative file; and Format 3 for an indexed file.

- Use the EXTERNAL clause to specify that a data item or a file connector is external. The data items and group data items of an external data record are available to every program in the run unit that describes that record. Internal is the default.
- Use the GLOBAL clause to specify that a data-name or file-name is a global name. A global name is available to every program contained in the program that declares it. Local is the default.

  Refer to "Linkage Section" in Section 4 for more information. Refer to "EXTERNAL Clause" and "GLOBAL Clause" in Section 4 for syntax information and a detailed discussion of the IPC FD formats.

  Refer also to "Procedure Division Header" in this section.

PROCEDURE DIVISION

- Use the USING clause in the Procedure Division header if the program will be called by another program. The calling program must contain a CALL statement with a USING phrase.
- The CALL statement transfers control from one object program to another in the run unit. Include the USING phrase of the CALL statement only if there is a USING clause in the Procedure Division header of the called program. The number of operands in each place USING occurs must be identical.

  The BY REFERENCE phrase specifies that the parameters will be by reference. The BY CONTENT phrase specifies that the parameters will be passed by content.
- The CANCEL statement ensures that the next time the referenced program is called, it will be in its initial state.
- The EXIT PROGRAM statement marks the logical end of a called program.
- The STOP statement with the RUN phrase stops the execution of the run unit and transfers control to the operating system.
- The USE statement specifies procedures for handling input-output errors in addition to the standard procedures provided by the input-output control system.

  Refer to "Procedure Division Header" in this section for information on the syntax for the Procedure Division header.

  Refer to "CALL Statement," "CANCEL Statement," and "EXIT Statement" in Section 6 and "STOP Statement" and "USE Statement" in Section 8 for syntax and detailed information.

IPC Examples

Example 1

```
IDENTIFICATION DIVISION.
PROGRAM-ID. P011.
PROCEDURE DIVISION.
BEGIN-P011.
    CALL   "P012".
    CALL   "P014".
    STOP RUN.
IDENTIFICATION DIVISION.
PROGRAM-ID.   P012 COMMON.
PROCEDURE DIVISION.
BEGIN-P012.
    CALL   "P013".
    EXIT PROGRAM.
IDENTIFICATION DIVISION.
PROGRAM-ID.   P013.
PROCEDURE DIVISION.
BEGIN-P013.
    EXIT PROGRAM.
END PROGRAM   P013.
END PROGRAM   P012.
IDENTIFICATION DIVISION.
PROGRAM-ID.   P014.
PROCEDURE DIVISION.
BEGIN-P014.
    CALL "P015".
    CALL "P016".
    EXIT PROGRAM.

IDENTIFICATION DIVISION.
PROGRAM-ID.  P015.
PROCEDURE DIVISION.
BEGIN-P015.
    CALL    "P012".
    EXIT PROGRAM.
END PROGRAM   P015.
IDENTIFICATION DIVISION.
PROGRAM-ID.  P016.
PROCEDURE DIVISION.
BEGIN-P016.
    EXIT PROGRAM.
END PROGRAM   P016.
END PROGRAM   P014.
END PROGRAM   P011.
```

This first example shows nested programs and the COMMON attribute. COMMON is declared in the PROGRAM-ID of P012. This enables P011 and P015 to call P012. If P012 were not declared as COMMON, P012 could be called only by P011.

Example 2

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  P001.
PROCEDURE DIVISION.
BEGIN-P001.
    PERFORM LOOP-B  10 TIMES.
    STOP RUN.
LOOP-B.
    CALL "P002".
LOOP-C.
IDENTIFICATION DIVISION.
PROGRAM-ID.  P002  INITIAL.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  AMOUNT  PIC 9(6)  VALUE 3 GLOBAL.
PROCEDURE DIVISION.
BEGIN-P002.
    ADD 7 to AMOUNT.
END-P002.
    EXIT PROGRAM.
END PROGRAM  P002.
END PROGRAM  P001.
```

This second example shows nested programs and the INITIAL attribute. Because P002 is declared as INITIAL, AMOUNT will have a value of 10 after P002 is executed (3 + 7 = 10). If INITIAL were not declared, the final value of AMOUNT would have been 73 (7 * 10 + 3 = 73). This example also shows use of the GLOBAL clause. AMOUNT is declared as global and so is available to every program contained in P001.

Example 3

```
$SET LIBRARYPROG
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  PO24.
 DATA DIVISION.
 LINKAGE SECTION.
 01  W7  PIC 9(6).
 01  W8  PIC 9(6).
 PROCEDURE DIVISION  USING W7, W8.
 BEGIN-PO24.
     ADD W8 TO W7.
     ADD 93 TO W7.
     EXIT PROGRAM.
 END PROGRAM  PO24.

$RESET LIBRARYPROG
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  PO23.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  T1  PIC 9(6)  VALUE 234.
 01  T2  PIC 9(6)  VALUE 567.
 PROCEDURE DIVISION.
 BEGIN-PO23.
     CALL "OBJECT/PO24"
     USING BY REFERENCE T1
     BY CONTENT T2.
     STOP RUN.
 END PROGRAM  PO23.


 Compile as PO23 and Run PO23.
```

This third example shows the use of the Linkage Section and the passing of parameters by content and by reference.

P023 and P024 are a pair of separately compiled programs, whose source lines are contained in one source file. Separately compiled programs generate separate, distinct object files. If PO24 had been nested within PO23, both programs would be part of the same object file.

The $SET LIBRARYPROG line forces P024 to be compiled as a library. The $RESET LIBRARYPROG line turns off the assumed library declaration for the next sequential program. These are required in order to compile P024 as a library instead of a program and then compile P023 as a program instead of a library.

Example 4

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  P021.
PROCEDURE DIVISION.
BEGIN-P021.
    CALL       "P022".
    CALL       "P022".
    CANCEL     "P022".
    CALL       "P022".
    STOP RUN.
IDENTIFICATION DIVISION.
PROGRAM-ID.  P022.
DATA DIVISION.
WORKING-STORAGE SECTION.
01  AMOUNT  PIC 9(6)   VALUE 3.
PROCEDURE DIVISION.
BEGIN-P022.
    ADD 7 to AMOUNT.
    DISPLAY AMOUNT.
END-P022.
    EXIT PROGRAM.
END PROGRAM  P022.
END PROGRAM  P021.
```

This fourth example shows the CANCEL statement. P022 is called three times by P021. The first time P022 is called, AMOUNT has a value of 3. The second time P022 is called, AMOUNT has a value of 10 (3 + 7 = 10), which is a result of the previous CALL. Before the third CALL, there is a CANCEL command that will set P022 back to its initial state. Therefore, at the third call of PO22, AMOUNT is reset to its original value of 3.

Section 11
Library Concepts

Note: *Although this section is not highlighted, all of the information is relative to Unisys extensions to COBOL ANSI-85.*

The library facility is a feature that can be used to structure processes. Unlike COBOL74, COBOL85 provides most of the library capabilities available to other extended languages such as ALGOL, Pascal, and FORTRAN.

A library program is a program that provides a procedure or a set of procedures that can be called by other programs. Each procedure made available, or "exported," by a library program is an entry point into the library program. Therefore, a library program can be thought of as a collection of procedures, each procedure accessible to other programs (including other library programs).

Programs and other library programs that use the entry points of a library program are referred to as user programs or calling programs. Throughout this section, we will refer to library programs as libraries, and procedures or entry points as nested programs.

For general information on the library facility, refer to the *System Software Utilities Operations Reference Manual*.

The library capabilities discussed in this section are considered to be extensions to ANSI COBOL85.

Do not confuse the library facility with the library program capabilities provided by COBOL74. The LIBRARYPROG compiler option provides a technique for creating COBOL74-type libraries using COBOL85. Do not use the LIBRARYPROG compiler option if a program is explicitly declared to be a COBOL85 library.

Library Programs

A library program provides a directly nested program or a set of directly nested programs that can be called by other programs. The entry points to the library provide access to nested programs within the library. These entry points are explicitly declared as exported items in the Program-Library section of the Data Division.

The PROGRAM-ID paragraph in the Identification Division must use the IS LIBRARY PROGRAM clause. A library program becomes a library when it calls the system procedure FREEZE to make the entry points available to user programs.

A program specified as a subprogram for binding purposes cannot be declared as an explicit library.

User Programs

A user program is a program that calls entry points provided by a library. The entry points are declared as imported objects in the Program-Library section of the Data Division.

A library can function as a user program and call other libraries. However, a chain of library linkages must never be circular. That is, a library cannot make a reference to itself, either directly or indirectly, through a chain of library references.

Interface between Libraries and User Programs

The MCP uses data from the compiler to match entry points declared in a calling program with entry points of a library. When the compiler creates the object code for a

- Library program, it builds a data structure called a *directory* for the library.
- User program, it builds a data structure called a *template* for each library imported by the user program.

Directory Data Structure

The description includes the name of the entry point, the type of entry point, the parameters of the entry point, and linkage information. The directory contains a description of all the entry points in the library.

NAME OF ENTRY POINT

This is the name of the exported program. The entry point name is the program-name specified by the PROGRAM-ID paragraph in the Identification Division, and exported by the ENTRY PROCEDURE clause in the Program-Library Section.

An entry point name other than the program-name can be specified by using the FOR clause of the ENTRY PROCEDURE clause in the Program-Library Section. If a name other than the program-name is assigned to an exported program, the name is referred to as the actual name of the exported program.

TYPE OF ENTRY POINT

This is the type of the program. A program can be typed or untyped depending on whether the GIVING clause is used in the Procedure Division header of the program. If the GIVING clause is used, the procedure is typed, and returns a result to the calling program. If the GIVING clause is not used, the procedure is untyped.

PARAMETERS OF ENTRY POINT

This is a description of the formal parameters expected by the exported program. Formal parameters for exported programs are declared by the USING clause of the Procedure Division header of the program. The declared formal data parameters must be described in the Linkage Section of the program, and any formal file parameters must be described in the File-Control paragraph of the program.

LINKAGE INFORMATION

This information describes the method used to link the library and the calling program (refer to "Linkage between User Programs and Libraries" in this section.

Template Data Structure

The template contains a description of all the entry points for a given library declared in the user program. The description includes the library attributes, the name of the entry point, the type of entry point, and the parameters of the entry point. The compiler creates a separate template for each library accessed by the user program.

LIBRARY ATTRIBUTES

This is a description of the library attributes declared by the user program (refer to "Library Attributes" later in this section).

NAME OF ENTRY POINT

This is the name of the imported program. The entry point name is the program-name specified by the ENTRY PROCEDURE clause of the Program-Library Section.

TYPE OF ENTRY POINT

This is the type of the imported program. A program can be typed or untyped depending on whether the GIVING clause is used in the Procedure Header of the program. If the GIVING clause is used, the procedure is typed, and returns a result to the calling program. If the GIVING clause is not used, the procedure is untyped.

PARAMETERS OF ENTRY POINT

This is a description of the formal parameters expected by the imported program. Formal parameters for imported programs are declared by the ENTRY PROCEDURE clause in the Program-Library Section of the user program, and described in the Local-Storage Section of the user program.

Library Initiation

On the first call to a library entry point, the operating system suspends execution of the user program. The description of the entry point in the template of the user program is compared to the description of the entry point with the same name in the directory associated with the referenced library. If the entry point does not exist in the library, or if the two entry point descriptions are incompatible, the operating system issues a run-time error and terminates the calling program. If the entry point exists and the two entry point descriptions are compatible, the operating system initiates the library program (if it has not already been initiated). The library program executes normally until it executes a library FREEZE. The library FREEZE makes the entry points of the library available. The operating system links to the user program all of the entry points of the library that are declared in the user program, and the user program resumes execution at the entry point of the first call.

A COBOL85 library executes a library FREEZE through a CALL statement of the form CALL SYSTEM FREEZE.

Because a library runs as a regular program until the library FREEZE request, the execution of the library FREEZE request can be conditional and can occur anywhere in the outermost program block.

If a user program causes a library program to be initiated and the library program terminates without executing a library FREEZE, the attempted linkage to the library entry points cannot be made, and the user program is terminated.

If the calling program declares an entry point that does not exist in the library, an error is not generated when the library is initiated. However, if the calling program attempts to call the nonexistent entry point, the operating system issues a "MISSING ENTRY POINT" run-time error, and terminates the user program.

Permanent and Temporary Libraries

A library can be specified as either a permanent library or a temporary library. A permanent library remains available until it is terminated by the system commands DS or THAW, or by execution of a CANCEL statement. A temporary library remains available so long as there are users of the library. A temporary library that is not in use "unfreezes" and resumes execution as a regular program with the statement that follows the library FREEZE request.

A COBOL85 library is specified as a permanent library or a temporary library through use of the PERMANENT or TEMPORARY options of the CALL SYSTEM FREEZE statement.

Linkage between User Programs and Libraries

The linkage between the user program and the library can be established directly or indirectly. The library specifies the form of linkage. Direct linkage occurs when the library program contains the procedure that is named as an exported item in the Library-Program Section of the library. Indirect linkage occurs when the library exports a procedure that is declared as an entry point of another library. When indirect linkage occurs, the operating system attempts to link the user program to this other library.

The user program can control the library to which it is linked by specifying the object code file title or the function name of the library, or by using the BYINITIATOR option. The LIBACCESS library attribute controls which of these is used. The library attribute TITLE allows you to specify the object code file title. The library attribute FUNCTIONNAME allows you to specify the System Library (SL) function name of the library. The BYINITIATOR attribute allows you to specify the library that initiated the program.

LINKLIBRARY-RESULT Identifier

During compilation, the compiler inserts a predefined identifier labeled LINKLIBRARY-RESULT. This identifier is updated to indicate whether the program is currently linked to, or is capable of being linked to, the library program when an explicit library entry procedure is called. If the user program cannot be linked to the library, the value in the LINKLIBRARY-RESULT identifier indicates the reason for the failure. The values of this identifier can be interpreted as follows.

Identifier	Description
2	Successful linkage was made to the library, but not all entry points were provided. Treated as an exception.
1	Successful linkage was made to the library and all entry points were provided.
0	The program was already linked to the library at the time of the library entry procedure call.
< 0	The program failed to link to the library. Refer to the *ALGOL Programming Reference Manual, Volume 1: Basic Implementation* for possible values and their meanings.

Creating Libraries

When a library program is written using COBOL85, certain requirements are imposed on syntactical elements in the source program. Table 11–1 summarizes these elements.

Table 11–1. Syntax Differences for COBOL85 Libraries

Program Division	Description
Identification	The IS LIBRARY PROGRAM clause identifies a COBOL85 program as a library program.
Environment	The SELECT clause of the FILE-CONTROL paragraph can be used to specify how a file is handled as a procedure parameter.
Data	The Linkage Section of each nested program (exported procedure) contains descriptions of the formal parameters of the procedure. The formal parameters are the data items declared by the USING clause of the Procedure Division header for the nested program.
Data	The Program-Library Section declares the directly nested programs to be exported. When the library freezes, the nested programs declared in the Program-Library Section are made available to user programs as entry points. Included in the LB statement is the EXPORT clause, which identifies the program as a library. The ATTRIBUTE clause permits specification of the library SHARING attribute. Library sharing is discussed later in this section. The Program-Library Section of a library might also be used to specify entry points imported by the library from other libraries. In other words, a library can call other libraries.
Procedure	The Procedure Division header of the outermost program of a library program cannot contain either the USING or the GIVING clause. The Procedure Division headers for nested programs of a library program describe the formal parameters expected by the nested program. The data items declared in the USING clause must be defined in the Linkage Section of the nested program.
Procedure	A procedural call to the library FREEZE facility explicitly freezes the library and makes the entry points of the library available to user programs. A library FREEZE can be invoked only from the outermost program of the library.

Library Sharing Specifications

The SHARING attribute controls how user programs that call the library share access to the library.

The SHARING attribute can be specified in the Program-Library Section of the library. The following table describes the available settings for this attribute:

Option Setting	Description
DONTCARE	The Master Control Program (MCP) determines the sharing.
PRIVATE	A copy of the library is invoked for each user (calling program). Any changes made to global items in the library by the actions of the user are visible only to that user of the library.
SHAREDBYRUNUNIT (default)	All invocations of the library within a run unit share the same copy of the library. The term run unit as used here refers to a program and all the libraries that are initiated either directly or indirectly by that program. Note that this definition differs slightly from the COBOL ANSI-85 definition of run unit as described in Section 8.
SHAREDBYALL	All simultaneous users share the same instance of the library.

The default value of the SHARING attribute is SHAREDBYRUNUNIT.

Making References to Libraries

When a user program that accesses libraries is written using COBOL85, certain requirements are imposed on syntactical elements in the source program. Table 11–2 summarizes these elements.

Table 11–2. Syntax Differences for COBOL85 User Programs

Program Division	Description
Data	The Local-Storage Section of a user program contains descriptions of the formal parameters expected by procedures imported from libraries. These descriptions are associated with the imported procedure through the WITH statement of an ENTRY PROCEDURE clause in the Program-Library Section of the user program.
Data	The Program-Library Section declares the procedures to be imported. The LB statement specifies the library to be imported. The ATTRIBUTE clause specifies the initial description of the imported library. The ENTRY PROCEDURE clause identifies the imported procedures. The WITH clause of the ENTRY PROCEDURE clause specifies where the description of the formal parameters of the procedure are found in the Local-Storage Section of the user program. The USING clause specifies the order of the formal parameters. The GIVING clause specifies the result returned by the entry point.
Procedure	The CALL statement transfers control from the user program to a procedure imported from a library.
Procedure	The CHANGE ATTRIBUTE statement changes a library attribute. Library attributes are discussed later in this section.

Library Attributes

The user program can assign values to certain library attributes. The library attributes available to user programs control the method used to link the library and the user program.

The user program can change library attributes dynamically. However, since the MCP ignores any changes made to library attributes of linked libraries, these changes must be made before the program is linked to the library.

When a mnemonic value is referenced in a context that is not associated with any of the library attribute mnemonic identifiers, then it is treated as a signed numeric constant.

The following paragraphs describe the library attributes available to user programs.

FUNCTIONNAME

Access	Type	Default Value
Read/Write	String (DISPLAY)	The value of INTNAME, if LIBACCESS is set to BYFUNCTION, else null string

This specifies the system function name used to find the object code file for the library. The LIBACCESS attribute controls whether the function name or the object code file title is used to find the object code file for the library.

INTERFACENAME

Access	Type	Default Value
Read/Write	String (DISPLAY)	The value of INTNAME

This identifies a particular connection library in a connection library program.

INTNAME

Access	Type	Default Value
Read/Write	String (DISPLAY)	Library-name declared in LB statement of Program-Library Section

This specifies the internal name for the library during compilation.

LIBACCESS

Access	Type	Default Value
Read/Write	Mnemonic	BYTITLE

This specifies how a library object code file is accessed when a library is called. LIBACCESS can be set to BYFUNCTION, BYTITLE or BYINITIATOR. If LIBACCESS is equal to BYTITLE, then the TITLE attribute of the library is used to find the object code file. If LIBACCESS is equal to BYFUNCTION, then the FUNCTIONNAME attribute of the library is used to access the MCP library function table, and the object code file associated with that FUNCTIONNAME is used. If LIBACCESS is equal to BYINITIATOR, then the library that initiated the program is the library that is accessed.
This specifies which information will be passed from the user program to the selection procedures of libraries that provide entry points dynamically

LIBPARAMETER

Access	Type	Default Value
Read/Write	String (DISPLAY)	(null string)

This specifies the transmission of information from the linking library to the selection procedure of the library or to the approval procedure of the connection library being linked to. The linking library can be a client library or a connection library. The primary library linked to can be a server library or a connection library. However, the selection procedure in the primary library must select a secondary library that is a server library (not a connection library). For connection libraries, the system also passes the LIBPARAMETER library or connection attribute of the requesting library to the APPROVAL procedure or procedures. The connection attribute is passed if it is set. If the connection attribute is not set, the library attribute is used. The LIBPARAMETER attribute for a single connection cannot be library-equated.

TITLE

Access	Type	Default Value
Read/Write	String (DISPLAY)	The value of INTNAME

This specifies the object code file title of the library. The LIBACCESS attribute controls whether the function name or the object code file title is used to find the object code file for the library.

Matching Formal and Actual Parameters

When a user program written in COBOL85 imports procedures from a library written in COBOL85, the data types of formal and actual parameters must be the same. For example, an imported library procedure that expects an INTEGER data item and a STRING data item must be passed an INTEGER data item and a STRING data item.

When the user program and the library are written using different programming languages, the data types of the formal and actual parameters must correspond to one another. Table 11–3 summarizes the correspondence between COBOL85 data types and data types found in the programming languages ALGOL and Pascal.

Table 11–3. Data Type Mapping between COBOL85, ALGOL, and Pascal

COBOL85 Data Type	ALGOL Data Type	Pascal Data Type
BIT, 77 SYNC RIGHT	BOOLEAN	BOOLEAN
BIT, 01 SYNC RIGHT	BOOLEAN ARRAY	BOOLEAN ARRAY
BINARY level 01	INTEGER ARRAY	ARRAY OF INTEGER
BINARY level 77, 1–11 digits	INTEGER or REAL	INTEGER
BINARY level 77, 12–23 digits	DOUBLE	DOUBLE REAL
COMPUTATIONAL and INDEX	HEX ARRAY	ARRAY OF Hexadecimal Characters
DISPLAY	EBCDIC ARRAY	ARRAY OF EBCDIC Characters
DOUBLE level 01	REAL ARRAY	ARRAY OF REAL
DOUBLE level 77	DOUBLE	DOUBLE REAL
Integer (COMPUTATIONAL) 1–11 digits	INTEGER	INTEGER
Integer (COMPUTATIONAL) 12–23 digits	DOUBLE	DOUBLE REAL
REAL level 01	REAL ARRAY	ARRAY OF REAL

Table 11–3. Data Type Mapping between COBOL85, ALGOL, and Pascal

COBOL85 Data Type	ALGOL Data Type	Pascal Data Type
REAL level 77	REAL	REAL
String (DISPLAY)	EBCDIC STRING	STRING OF EBCDIC Characters

Note: *COBOL level 01 data types are EBCDIC data by default. Therefore, if a level 01 item containing a subordinate OCCURS item of BINARY, DOUBLE, or REAL data is being passed as a parameter to a non-EBCDIC array, the COBOL 01 data item USAGE must be explicitly declared to match the data type of the receiving array.*

COBOL85 Library Example

The following COBOL85 program is a library program containing two exported library procedures (nested programs), the first named MSGLIB-PROG1 and the second named MSGLIB-PROG2. The file is named TEST/LIBRARY, and resides under the usercode COBOLUSER on the family COBOLPACK. The executable code file is named "(COBOLUSER)OBJECT/TEST/LIBRARY ON COBOLPACK" .

```
* The outer block program containing the procedures
*
*
  IDENTIFICATION DIVISION.
  PROGRAM-ID. MSGLIB IS LIBRARY PROGRAM.

 ENVIRONMENT DIVISION.

 DATA DIVISION.

 PROGRAM-LIBRARY SECTION.
  LB MSGLIB EXPORT
     ATTRIBUTE SHARING IS PRIVATE.
     ENTRY PROCEDURE MSGLIB-PROG1.
     ENTRY PROCEDURE MSGLIB-PROG2.

 PROCEDURE DIVISION.
  PARA-1.
     CALL SYSTEM FREEZE TEMPORARY.
     STOP RUN.
*
* THE NESTED PROGRAM MSGLIB-PROG1
*

 IDENTIFICATION DIVISION.
  PROGRAM-ID. MSGLIB-PROG1.

 ENVIRONMENT DIVISION.

 DATA DIVISION.

 WORKING-STORAGE SECTION.
  01 ID-STR     PIC X(80).

 LINKAGE SECTION.
  01 MSG-STR    PIC X(80).
```

```
*
*  The MSG-STR parameter is passed in by the calling program
*
  PROCEDURE DIVISION USING MSG-STR.
  PARA-1.
     MOVE "THIS IS LIBRARY PROGRAM 1" TO ID-STR.
     DISPLAY ID-STR.
     DISPLAY MSG-STR.
    EXIT PROGRAM.
 END PROGRAM   MSGLIB-PROG1.

*
* The nested program MSGLIB-PROG2
*

 IDENTIFICATION DIVISION.
  PROGRAM-ID. MSGLIB-PROG2.

 ENVIRONMENT DIVISION.

 DATA DIVISION.

 WORKING-STORAGE SECTION.
  01 ID-STR     PIC X(80).

 LINKAGE SECTION.
  01 MSG-STR    PIC X(80).
 77 MSG-NUM    PIC 9(11)  BINARY.
*
*  The MSG-STR and MSG-NUM parameters are passed in by
*  the calling program
*
  PROCEDURE DIVISION USING MSG-STR, MSG-NUM.
  PARA-1.
     MOVE "THIS IS LIBRARY PROGRAM 2" TO ID-STR.
     DISPLAY ID-STR.
     DISPLAY MSG-NUM.
     DISPLAY MSG-STR.
     EXIT PROGRAM.
  END PROGRAM   MSGLIB-PROG2.
  END PROGRAM   MSGLIB.
```

COBOL85 User Program Example

The following program is a COBOL85 program that imports the library procedures of the example COBOL85 library program provided earlier in this section:

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. USERPROGRAM.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
 01 MSG-STR    PIC X(80).
77 NUM        PIC 9(11)  BINARY.
LOCAL-STORAGE SECTION.
LD PROG1.
 01 PROG1-STR  PIC X(80).
LD PROG2.
 01 PROG2-STR  PIC X(80).
77 MSG-NUM    PIC 9(11)  BINARY.
PROGRAM-LIBRARY SECTION.
LB MSGLIB IMPORT
   ATTRIBUTE TITLE IS "(COBOLUSER)OBJECT/TEST/LIBRARY ON COBOLPACK".
   ENTRY PROCEDURE MSGLIB-PROG1 WITH PROG1
             USING PROG1-STR.
    ENTRY PROCEDURE MSGLIB-PROG2 WITH PROG2
             USING PROG2-STR, MSG-NUM.
PROCEDURE DIVISION.
 PARA-1.
*
*  Put an identifying string into the MSG-STR
*
  MOVE "USERPROGRAM SAYS HELLO TO THE TEST LIBRARY" TO MSG-STR.
*
* Call the library procedure passing the identifying string
*
  CALL MSGLIB-PROG1 USING MSG-STR.
*
*  Put an identifying string into the MSG-STR, and a value
*  into NUM
*
 MOVE "USERPROGRAM SAYS HELLO TO THE TEST LIBRARY" TO MSG-STR.
  MOVE 1000 TO NUM.
*
*  Call the library procedure passing the identifying string
*
  CALL MSGLIB-PROG2 USING MSG-STR, NUM.
 STOP RUN.
  END PROGRAM   USERPROGRAM.
```

ALGOL User Program Example

The following program is an ALGOL program that imports the library procedures of the example COBOL85 library program provided earlier in this section. When executed from a remote station, this program prompts the user for text input. The user input is passed to the COBOL85 library program named "(COBOLUSER)OBJECT/TEST/LIBRARY ON COBOLPACK.

```
BEGIN
 FILE
      REM  (KIND       = REMOTE
           ,UNITS      = CHARACTERS
           ,MYUSE      = IO
           ,BLOCKSTRUCTURE
                       = EXTERNAL
           ,MAXRECSIZE = 2040
           );

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%        SIZING DEFINES
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

DEFINE
      MAX_INPUT_LENGTHV = 2040  #
     ;

EBCDIC ARRAY
      REM_INPUT     [00:MAX_INPUT_LENGTHV]
     ,DISPLAY_ARY   [00:80]
     ;

REAL ARRAY
      MSG_ARY       [00:12]    %-- THIS IS IN WORDS
     ;

POINTER
      P_MSG_ARY
     ;

INTEGER
      INPUT_LENGTH
     ,MESSAGE_NUMBER
     ;

BOOLEAN
      DONE
     ;
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                       %%
%    THIS IS THE IMPORTED LIBRARY, A COBOL85 LIBRARY    %%
%                                                       %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
LIBRARY
      COBOL_LIBRARY
           (LIBACCESS = BYTITLE
           ,TITLE     = "(COBOLUSER)OBJECT/TEST/LIBRARY ON COBOLPACK."
           );

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                    %%
% MESSAGE_DISPLAY_1 ACCEPTS A SINGLE STRING PARAMETER AND DISPLAYS IT%%
%                                                                    %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 PROCEDURE MESSAGE_DISPLAY_1 (MSG_STRING
                             );
 EBCDIC ARRAY
      MSG_STRING   [0]
     ;
 LIBRARY
      COBOL_LIBRARY (ACTUALNAME = "MSGLIB-PROG1")
     ;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                    %%
% MESSAGE_DISPLAY_2 ACCEPTS TWO PARAMETERS, A STRING AND INTEGER     %%
%                                                                    %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
PROCEDURE MESSAGE_DISPLAY_2 (MSG_STRING
                             ,MSG_NUMBER
                             );
 VALUE
      MSG_NUMBER
     ;
 EBCDIC ARRAY
      MSG_STRING    [0]
     ;
 INTEGER
      MSG_NUMBER
     ;
 LIBRARY
      COBOL_LIBRARY (ACTUALNAME = "MSGLIB-PROG2")
     ;
 DEFINE
      TALK (MSG)      = IF MYSELF.INITIATOR NEQ 0 THEN
                            BEGIN
                            REPLACE P_MSG_ARY : POINTER (MSG_ARY [0])
                                    BY MSG;
                            WRITE (REM
                                  ,OFFSET (P_MSG_ARY)
```

```
                                  ,MSG_ARY
                                  );
                            END    #
      ;   %---- END OF DEFINES
 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%                                                                       %%
%                 O  U  T  E  R     B  L  O  C  K                       %%
%                                                                       %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
DONE := FALSE;
 MESSAGE_NUMBER := 0;
TALK ("Enter the message you want the library to display on ODT");
TALK ("Transmit a single blank space to exit");
WHILE NOT DONE DO
     BEGIN
    %
    % READ FROM THE TERMINAL
    %
    READ (REM
         ,MAX_INPUT_LENGTHV
         ,REM_INPUT
         );
    %
    % GET THE NUMBER OF CHARACTERS ENTERED, OR 80 IF GTR THAN 80
    %
    INPUT_LENGTH := MIN (REM.CURRENTRECORD
                        ,80
                       );
    %
    % IF A SINGLE BLANK SPACE WAS TRANSMITTED, EXIT
    %
    DONE := (INPUT_LENGTH = 1) AND (REM_INPUT [0] = " ");
     IF NOT DONE THEN
         BEGIN
        %
        % LOAD THE INPUT CHARS INTO THE DISPLAY_ARY
        %
        REPLACE DISPLAY_ARY [0] BY REM_INPUT [0] FOR INPUT_LENGTH;
        %
        % CALL THE COBOL LIBRARY PROCEDURE
        %
        MESSAGE_DISPLAY_1 (DISPLAY_ARY);
        %
        % CALL THE SECOND COBOL LIBRARY PROCEDURE PASSING THE NUMBER
        %
        MESSAGE_DISPLAY_2 (DISPLAY_ARY, MESSAGE_NUMBER);
         MESSAGE_NUMBER := * + 1;
         END;   % IF NOT DONE THEN
     END;   % WHILE NOT DONE DO
 END.
```

Passing a File as a Parameter

In order to pass a file as a parameter, it is necessary for the calling program to declare the file twice in the caller. The second instance of the file is the formal parameter and is used in the imported entry point declaration of the LB.

The following library program and calling program examples illustrate how this is accomplished.

Library Program Example

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. EXPRTD-ENTRY-PT IS LIBRARY PROGRAM.
 ENVIRONMENT DIVISION.
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 PROGRAM-LIBRARY SECTION.
 LB EXPRTD-ENTRY-PT EXPORT
     ATTRIBUTE SHARING IS PRIVATE.
     ENTRY PROCEDURE INNERPROG.

 PROCEDURE DIVISION.
 MAIN-PROCEDURE.
     DISPLAY "Initializing Library".
     CALL SYSTEM FREEZE TEMPORARY.
     STOP RUN.

*********************************
* INNER PROGRAM STARTS HERE      *
*********************************
 IDENTIFICATION DIVISION.
 PROGRAM-ID. INNERPROG.
 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT REFERENCE THE-FYLE ASSIGN DISK.
 DATA DIVISION.
 FILE SECTION.
 FD THE-FYLE.
 01 FYLE-REC PIC X(80).

 WORKING-STORAGE SECTION.
 LINKAGE SECTION.

 PROCEDURE DIVISION USING
     THE-FYLE.

 MAIN-PROCEDURE.
     IF ATTRIBUTE OPEN OF THE-FYLE = VALUE(TRUE)
       DISPLAY "File was already open"
     ELSE
```

```
      DISPLAY "File was closed -- opening file"
      PERFORM 200-OPEN-THE-FYLE.

 100-EXIT-PROGRAM.
     EXIT PROGRAM.

 200-OPEN-THE-FYLE.
     OPEN OUTPUT THE-FYLE.
     DISPLAY "File opened in the library".

 END PROGRAM INNERPROG.
*********************************
* END OF INNER PROGRAM           *
*********************************
 END PROGRAM EXPRTD-ENTRY-PT.
```

Calling Program Example

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT THE-FYLE ASSIGN TO DISK.
     SELECT LOCAL BY REFERENCE
  FORMAL-FYLE ASSIGN TO DISK.

 DATA DIVISION.
 FILE SECTION.
 FD THE-FYLE.
 01 FYLE-REC    PIC X(84).

 FD FORMAL-FYLE.
 01 FYLE-REC    PIC X(84).

 WORKING-STORAGE SECTION.
 77  WS-PARAM PIC S9(11) BINARY.

 LOCAL-STORAGE SECTION.
 LD  FORMAL-PARAMS.
 77  LS-PARAM PIC S9(11) BINARY.

 PROGRAM-LIBRARY SECTION.
 LB  EXPL-LIB IMPORT
     ATTRIBUTE TITLE IS "OBJECT/C85/CALL/EXLIB/FD/CALLED".
     ENTRY PROCEDURE INNERPROG
     WITH FORMAL-PARAMS
          FORMAL-FYLE
     USING LS-PARAM
           FORMAL-FYLE.
```

```
 PROCEDURE DIVISION.
 MAIN-PARAGRAPH.
    complete CALL-THE-LIBRARY.
    IF ATTRIBUTE OPEN OF THE-FYLE = VALUE(TRUE)
      DISPLAY "File was returned OPEN"
      CLOSE THE-FYLE
    ELSE
      DISPLAY "File was returned CLOSED".
    STOP RUN.

CALL-THE-LIBRARY.
    CHANGE ATTRIBUTE TITLE OF THE-FYLE TO "JUNK".
    CALL INNERPROG
      USING WS-PARAM
            THE-FYLE.
```

Section 12
File Concepts

This section discusses three important file handling concepts in COBOL: file attributes, file organization, and access mode.

File attributes enable you to define, monitor, and change file properties (attributes). File organization determines the physical arrangement of the records of a file, which includes the way records are stored on mass-storage devices. The specified organization of a file determines the access mode of that file. Access mode determines the logical method that a program uses to access the records.

This section is organized as follows:

- Overview

  A general discussion of file concepts, including file attributes, the three types of file organization (sequential, relative, and indexed), and the three types of access mode (sequential, random and dynamic).

- File Attributes

  An explanation of file attributes, port files, and subfiles.

- File Organization

  An explanation of the different ways to organize files.

- Access Mode

  An explanation of the different ways to access records in a file.

- File Organization Checklists and Examples

  A list of the COBOL elements used with each type of file organization. Each statement in the list has a reference that directs you to more detailed information on the statement. Annotated examples of programs that use each type of file organization are provided.

Overview

File information is defined by distinguishing between the physical aspects of the file and the conceptual characteristics of the data in the file.

The physical aspects of a file describe the data as it appears on the input or output medium, that is, how logical records are grouped according to the physical limits of the medium and the means by which the file can be identified.

The conceptual characteristics of a file define each logical entity in the file. In a COBOL program, the input or output statements refer to an entity called a logical record.

Physical versus Logical Records

The distinction between a physical record and a logical record is important. A COBOL logical record is a group of related information that is uniquely identifiable and treated as a unit. A physical record is a physical unit of information with a size and recording mode convenient to a particular computer for storing data on an input or output device. The size of a physical record is hardware-dependent and has no direct relationship to the size of the file contained on a device.

A single logical record can be contained in a single physical record, or several logical records can be contained in a single physical record. In a mass-storage file, however, a logical record can require more than one physical record. In this manual, references to records mean logical records unless the term "physical record" is specified.

The concept of a logical record is not restricted to file data. A logical record can apply also to the definition of working-storage. Thus, working-storage can be grouped into logical records and defined by a series of record-description entries.

Special facilities can be accessed through logical records. For example, assigning a file to REMOTE enables you to use the logical file mechanism to access a family of terminal or station devices that use traditional file-handling methods rather than the specialized data-communications-handling methods of the Communication Section.

Manipulating Files

Both the physical and the logical properties (attributes) of files can be defined, monitored, and changed using file attributes.

To gain access to a logical file, a program must declare both the organization and access mode of the file. There are three ways to organize a file and three possible ways that the system can access the file. You designate both organization and access mode in the SELECT statement of the FILE-CONTROL paragraph in the Environment Division.

The three types of file organization used in COBOL are sequential, indexed, and relative. The type of file organization determines the physical relationship between records. To choose a type of file organization, consider the way a file is used in your program and the resources of your installation.

The types of access mode for files in COBOL are sequential, random, and dynamic. Not all modes of access are available for all three different types of files (refer to Table 12–1, later in this section).

File Attributes

File attributes enable you to define, monitor, or change file properties.

File attributes provide access to functions not otherwise available in the language. Also, file attributes can be used to declare and access files. When both a file attribute and standard COBOL syntax are available to accomplish a desired function, it is always preferable to use the standard COBOL syntax, because changing the attribute can lead to unexpected results in cases when the attribute is also used or altered by the compiler.

File attributes can be initialized using the VALUE OF clause. They can be changed using the CHANGE statement, and set using the SET statement in the Procedure Division. A full explanation of each file attribute and how it can be used is available in the *File Attributes Programming Reference Manual*.

File-Attribute Identifier

File-attribute identifiers enable you to monitor, manipulate, define, or dynamically change any specific file attribute.

```
ATTRIBUTE attribute-name { OF } file-name
                         { IN }
        [ ( arithmetic-expression-1 [, arithmetic-expression-2 ] ) ]
        [ ( VALUE [(] attribute-name [)] ) ]
```

attribute-name

The attribute-name is a system-name.

arithmetic-expression

If arithmetic-expression-1 is used with a port file, the value of the expression must specify which subfile of the file is affected. A subfile index is required for accessing or changing attributes of a subfile of a port file.

If arithmetic-expression-1 is . . .	Then . . .
Not specified,	The attribute of the port is accessed.
Specified and its value is nonzero,	The value of the expression specifies a subfile index and causes the attribute of the subfile to be accessed.
Specified and its value is zero,	The attributes of all subfiles are accessed.

If an arithmetic expression is used with a disk file, the values of arithmetic-expression-1 and arithmetic-expression-2 must specify the row and copy parameters for the file.

VALUE attribute-name

The VALUE attribute-name phrase is valid only for the FILEEQUATED attribute.

Details

A file attribute belongs to one of four categories, depending on the type of attribute-name specified in the file-attribute identifier. The four file-attribute categories are described in the following paragraphs:

- Alphanumeric file-attribute identifier

  Where allowed in syntax, an alphanumeric file-attribute identifier is similar to an elementary alphanumeric DISPLAY data item that has a size equal to the maximum size allowed for the specified attribute. The contents of the alphanumeric data-identifier are left-justified with space fill. Alphanumeric file-attribute identifiers are allowed as operands in relation conditions and as sending operands in Format 1 MOVE statements.

- Numeric file-attribute identifier

  Where allowed in syntax, a numeric file-attribute identifier is similar to an elementary numeric DISPLAY data item that represents a signed integer with eight decimal digits. Numeric file-attribute identifiers are allowed as operands in arithmetic expressions and as sending operands in Format 1 MOVE statements. Some numeric file attributes represent information that accounts for the number of areas, blocks, records and so forth in the file. These attributes are "one relative" in that their value specifies the exact number of areas, blocks, records, and so forth in the file.

- Mnemonic file-attribute identifier

  Certain file attributes are associated with values best expressed as mnemonic-names, because the magnitude of the actual value is unrelated to its meaning. Mnemonic file-attribute identifiers can appear as the subject of a mnemonic-attribute relation condition using a mnemonic value associated with the specified attribute as the object. The name for the attribute value must follow the reserved word VALUE.

  Mnemonic-attribute relation conditions are allowed in any conditional expression.

```
mnemonic-attribute-identifier IS [ NOT ] { EQUAL } TO
                                         {  =  }

                { VALUE } [ ( ] mnemonic-attribute-value [ ) ]
                { VA    }
```

- File-attribute mnemonic value

  When a mnemonic value is referenced in a context that is not associated with any of the file attribute mnemonic identifiers, then it is treated as a signed numeric constant.

- Mnemonic-attribute relation conditions cannot be abbreviated. The names for the mnemonic-attribute values are system-names and are not necessarily reserved words. File attributes with a Boolean nature are considered mnemonic attributes in COBOL and are associated with the mnemonic-attribute values TRUE and FALSE.
- The parentheses surrounding the mnemonic-attribute-value are optional. If either parenthesis is used, both parentheses must be present.
- Boolean file-attribute identifier.

  These attributes are referenced in the same manner as numeric file-attribute identifiers. These attributes return the value 1 for TRUE and 0 for FALSE.

MCPRESULTVALUE Identifier

During compilation, the compiler inserts into each program a predefined identifier labeled MCPRESULTVALUE. This identifier is updated with the I/O result value returned by the MCP after the execution of an OPEN, CLOSE, READ, WRITE, REWRITE, DELETE, START, or SEEK statement. The value in the MCPRESULTVALUE identifier indicates the success or failure of the I/O statement. The MCPRESULTVALUE predefined identifier is a 48-bit word that appears as a real-valued, Working-Storage data item declared as follows:

```
77 McpResultValue            REAL.
```

For details on the values placed in MCPRESULTVALUE after an OPEN or CLOSE statement is executed, refer to the AVAILABLE attribute in the *File Attributes Programming Reference Manual.*

For details on the values placed in MCPRESULTVALUE for all other I/O operations, refer to the STATE attribute in the *File Attributes Programming Reference Manual.*

Note that an enumerated value is returned in the MCPRESULTVALUE identifier after the execution of an OPEN or CLOSE statement. The value returned for all other statements is a Boolean value in the lower part of the word with the enumerated value in the middle of the word. For convenience, you can move the enumerated value to another real or binary data item by using the following statement:

```
MOVE MCPRESULTVALUE TO identifier [26:9:10].
```

You can then query the alternate data item to determine the status value of the particular I/O request.

Examples

The following code fragments illustrate the use of the MCPRESULTVALUE identifier to aid in error recovery. In each case, it is assumed that the $FS4XCONTINUE compiler option is set (TRUE), which allows a program to continue executing when an I/O request fails as long as one of the following conditions is met:

- The FILE STATUS clause is declared (Environment Division)
- A USE routine is declared.
- An alternate statement to perform in case of an unsuccessful I/O is declared with the particular I/O statement (refer to each I/O statement for syntax)

```
WRITE F-REC.
MOVE MCPRESULTVALUE TO R [26:9:10].
IF R NOT = 0 THEN
    IF R = 91 THEN
       DISPLAY "IMPLICIT OPEN FAILED"
       OPEN OUTPUT F
       WRITE F-REC
    ELSE
       CALL SYSTEM IOTERMINATE USING F.
```

This example performs a WRITE operation, and then moves the enumerated value in the MCPRESULTVALUE identifier to the identifier, R. It then uses an IF statement to test the MCPRESULTVALUE in identifier R, and provides instructions for the failure or success of the write operation.

```
OPEN OUTPUT F.
IF MCPRESULTVALUE NOT = 1 THEN
    IF MCPRESULTVALUE = 40 THEN
       DISPLAY "FILE WAS NOT CLOSED"
    ELSE
       CALL SYSTEM IOTERMINATE USING F.
```

This example issues an OPEN statement, and then issues an IF statement to test the value in the MCPRESULTVALUE identifier.

Port Files

User processes communicate across a BNA network through the standard I/O file mechanism using a special kind of file called a port file. You can communicate with a foreign process by performing READ and WRITE operations to a port file. A port file has one or more associated subports, called subfiles, each of which can be connected to a different process. Communication between local processes can use port files without going through a BNA network.

A subfile provides a two-way, point-to-point, logical communication path between two programs. To establish this path, each program must describe the desired connection. These descriptions are declared using file attributes.

The ACTUAL KEY clause of the File-Control entry specifies the subfile index used for a port file when an I/O operation is initiated. If the ACTUAL KEY clause specifies the value zero, the OPEN statement opens all subfiles associated with the port file, the READ statement performs a nonselective read, the WRITE statement performs a broadcast write, and the CLOSE statement closes all opened subfiles associated with the port file.

If the ACTUAL KEY clause is not specified, the file must contain a single subfile. This subfile is then assumed to be the subfile associated with the I/O statements (any OPEN, READ, WRITE or CLOSE).

File Organization

File organization controls the way records of the file are related to each other. For a mass-storage file, the organization controls the way records are stored on the mass-storage device. File organization relates to attributes of the physical file.

To choose the file organization, consider the type of processing the program is to accomplish. More specifically, determine how the files in the program are used. The device associated with a file can determine the organization of the file (for example, a printer file is organized sequentially). The way that information in the file is accessed can influence your choice of organization. For example, if records must be read randomly from many different locations in the file, then the file should probably not be organized sequentially. The resources of your installation can also influence the choice of file organization (indexed files require more mass-storage space than other files).

The following paragraphs describe methods of file organization and discuss briefly how they are used.

Sequential Files

A sequential file is the simplest type of file organization. Records are organized according to the time that they are placed in the file. For example, the first record written to the file is placed at the beginning of the file; the second record is placed in the second position in the file. Frequently, files that do not reside on mass-storage devices (for example, tape or cards) are organized sequentially.

The following steps illustrate the use of a sequential file: A master file that contains records for all employees is first sorted by employee number, and then placed in a sequential file. In a sequential file, every record has a predecessor and a successor except for the first and last records. Therefore, the record for the employee number 5066 would be after the record for employee 5065 and before 5067. The records are in numerical order by employee number because the file was sorted by employee number before being placed in the sequential file. The records in a sequential file are organized by the order they are placed in the file.

Sequential files are usually used when you must process most of the records in a file. In a master payroll file, where most records are updated, a sequentially organized file is the best choice. However, if most of the records in a file will not be accessed during processing, then sequential file organization could be inefficient.

Types of sequential files

- Printer file (output)
- Tape file (input or output)
- Card file (input)
- Port file (input or output)

Important considerations for use

- All the predecessors to any record must be read before access is available.
- Empty record spaces are not allowed.
- You can only process forward in a file, from record 1 to 2 (except for tape files that allow the file to be rewound). In order to move backwards in a file, you must reset—that is, CLOSE and then OPEN—the file.
- Records cannot be deleted.

Relative Files

Relative file organization makes random processing easier. Records in a relative file are defined by the ordinal position in the file. The relative key reflects the relative position of the record in the file. The key is required for all relative files and is not part of the record itself.

To fully understand relative file organization, examine the difference between relative files and sequential files. In a sequential file, every record has a predecessor and a successor except for the first and last records. In a relative file, a record is located by the relative record number, regardless of the location of the record. Consequently, it is possible to have undefined record positions in a relative file.

Example of a relative file

An account file, where all information is accessed by account number. In such a file, the account number could be used as the relative key. If an account becomes inactive, the record is deleted, but the deleted record location remains in the file until the account number is reassigned or the file is consolidated.

Important considerations for use

- Relative files are designed to make random access easier. If a file is never accessed randomly, you should consider a different type of file organization.
- Deleted record locations are allowed in the file. The empty spaces are skipped in sequential access mode.
- Relative files must be disk files.

Indexed Files

An indexed file is a type of file organization that allows access to records according to a key field in each record. An indexed file consists of two parts: a data file containing all of the records, and an index or key file that contains record keys in sorted order. When an indexed file has multiple keys, there are multiple key files (one for each key). Each record in a key file connects a record key value with the position of the corresponding record in the data file. Therefore, when a specific record is needed for processing, the system checks the key file to determine the exact position of that record.

Index files provide a more flexible access. For example, a programmer designates that an employee number is a record key. The system then creates a key file on disk that references the positions of employee records, based on their employee numbers. To access a record during processing, the employee number is moved into the record area of the program and the system reads the indexed file to locate the desired record.

Example of an indexed file

A typical indexed file is one that must be accessed by one or more different characteristics. For example, records in an inventory file of cars for sale at a dealership could be accessed by model, year, or color.

Important considerations for use

- Flexibility: Indexed files enable you to access records sequentially by key, randomly by key, or by relative record number.
- Resources: Indexed files consume more space on disk or disk pack than other file organizations.
- Speed of processing: Indexed files generally require more I/Os and take longer to access.
- Indexed files are limited to mass-storage devices.
- Any item declared as a record key must be part of the actual data record.
- Indexed files allows record level locking.
- Records in indexed files can be deleted.

Access Mode

Access mode describes the way that records in a file are processed. There are three types of access modes in COBOL. Not all access modes can be used with all file organizations. Table 12–1 lists each type of file organization, the types of access modes that are possible with the organization, and the types of keys that are necessary. (Information on the different types of keys is included under "FILE-CONTROL Paragraph" in Section 3.) Immediately following the table are paragraphs describing each type of access mode.

Table 12–1. File Organization and Access Mode

Organization	Access Mode	Key
Sequential Files	Sequential	No key required
	Random	Actual key (numeric)
Relative Files	Sequential	Key is optional
	Random	Relative key (numeric)
	Dynamic	Relative key (numeric)
Indexed Files	Sequential	Record key (alphanumeric)
	Random	Record key (alphanumeric)
	Dynamic	Record key (alphanumeric)

Sequential Access Mode

Sequential access is valid for all three types of file organization and is the default for each type of file organization. A file declared as sequential access is processed from beginning to end, starting with the first record and finishing with the last record. A key is not required unless the file is organized as an indexed file. Use the AT END phrase of the READ statement for any in-line exception handling.

Random Access Mode

Random access is allowed for all three types of file organization. (This is an extension to COBOL ANSI-85 for sequential files.)

A file declared as random access uses a relative record number to point to a specific record in the file. The programmer must maintain the value of the key. To access a file randomly, the file must be stored on disk. For random access files, the READ statement must contain the INVALID KEY phrase for in-line exception handling.

Dynamic Access Mode

Dynamic access allows a file to be accessed in both the random and the sequential modes. The mode of a file depends on which commands are used.

Dynamic access is allowed for indexed and relative files only.

- For sequential access, the READ statement requires the AT END phrase to handle any in-line exception handling.
- For random access, the READ statement must contain the INVALID KEY phrase.

To read a file sequentially, the NEXT phrase must be used in the READ statement. The specified key is not considered. The current record parameter is changed to point to the next available record.

To read a file randomly, the value of the specified key points to the position in the file and the record in that position is delivered during the evaluation of a READ statement. The INVALID phrase determines the handling of the circumstances when the value of the key points to a nonexistent record.

File Organization Checklists

The following pages contain a list of the COBOL elements that are used to declare, initialize, and process a file. The checklist is divided into sections for each type of file organization: sequential, relative, and indexed. In each type of file organization, the elements are listed according to the division where they appear. These pages also include program examples to illustrate how these COBOL elements are used.

Each element in the list refers to more detailed information elsewhere in this manual.

Sequential File Checklists

Identification Division : None.

Environment Division

FILE-CONTROL paragraph of the Input-Output Section

- Use Format 1 of the SELECT statement for sequential files.
- Name each file used in the program in a SELECT statement and assign the file to a type of hardware using the ASSIGN TO clause.
- Let the default value of the ORGANIZATION IS clause define the organization of the file as sequential. For documentation, define the organization explicitly (ORGANIZATION IS SEQUENTIAL).
- Use the ACCESS MODE IS clause to define the access mode the program uses to access the file. The default access mode for sequential files is SEQUENTIAL.
- If the program monitors the status of the file, define a variable to receive status key values by using the FILE STATUS IS clause.

Detailed information on the FILE-CONTROL paragraph and status key values for sequential files is included in Section 3.

I-O-CONTROL paragraph of the Input-Output Section: The MULTIPLE FILE clause defines sequentially organized tape files when more than one file shares the same physical reel of tape. Detailed information on the I-O-CONTROL paragraph is included in Section 3.

Data Division

FD Entry in the File Section : A valid file description (FD) entry must be defined for all sequential files. Detailed information on the FD entry is included in Section 4.

Data-names in the WORKING-STORAGE SECTION: If the program monitors the status of a file, a data-name must be defined to receive status key values. This data-name must be the same as the data-name in the FILE STATUS IS clause of the file's SELECT entry, located in the FILE-CONTROL paragraph of the Environment Division. Details on the FILE STATUS IS clause are included under "FILE-CONTROL Paragraph" in Section 3.

Procedure Division

CLOSE statement

- Use Format 1 of the CLOSE statement.
- You can use the REEL and UNIT phrases of the CLOSE statement for sequential tape files. They are not valid for any other type of file organization.

Detailed information on the CLOSE statement is included in Section 8.

OPEN statement : You can use the REVERSED, NO REWIND, and EXTEND phrases of the OPEN statement for sequential files. These phrases are not valid for any other type of file organization. Details on the OPEN statement are included in Section 6.

READ statement: Use Format 1 of the READ statement. Detailed information on the READ statement is included in Section 6. Information relating to sequentially organized files is included in the discussion of both Format 1 and Format 2.

REWRITE statement

- The REWRITE statement is valid only for mass-storage files.
- When the access mode is sequential, a successful READ statement must be performed on the file before a REWRITE statement is performed. No other I-O statements that affect the file can be executed between the READ and the REWRITE statements.

Detailed information on the REWRITE statement is included in Section 6.

WRITE statement: Use Format 1 of the WRITE statement. Detailed information on the WRITE statement is included in Section 8.

Sequential File Program Example

This COBOL program example uses a sequential file. The program name is EXECTEST, and it creates a print file from the contents of a sequential data file named OUT-FILE.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. EXECTEST.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. A-14.
OBJECT-COMPUTER. A-14.
SPECIAL-NAMES.
    ALPHABET ASCII-SET IS ASCII.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT OPTIONAL OUT-FILE
    STATUS WA-STAT ASSIGN TO DISK.
    SELECT PRINT-FILE ASSIGN TO PRINTER.
DATA DIVISION.
FILE SECTION.
FD  OUT-FILE
    BLOCK CONTAINS 30 RECORDS
    RECORD CONTAINS 84 CHARACTERS
    VALUE OF INTNAME IS "IN"
    CODE-SET IS ASCII-SET.
01  SOURCE-IN-REC.
    05 SOURCE-IN      PIC X(72).
    05 SOURCE-SEQ     PIC 9(8)  COMP.
    05 SOURCE-JUNK    PIC X(4).
FD  PRINT-FILE.
01  PRINT-REC.
    05 PRINT-LINENUM  PIC 9(8).
    05 BLANK-SPACES   PIC X(2).
    05 PRINT-DATA     PIC X(72).
WORKING-STORAGE SECTION.
77 WA-STAT           PIC X(02).
01 LINE-NUMBER       PIC 9(8)  COMP.
PROCEDURE DIVISION.
PARA-1.
    OPEN INPUT OUT-FILE.
    OPEN OUTPUT PRINT-FILE.
    MOVE SPACES TO BLANK-SPACES.
    MOVE 1 TO LINE-NUMBER.
PARA-2.
    READ OUT-FILE AT END GO TO EOJ.
    MOVE LINE-NUMBER TO PRINT-LINENUM.
    MOVE SOURCE-IN TO PRINT-DATA.
    WRITE PRINT-REC.
    ADD 1 TO LINE-NUMBER.
    GO TO PARA-2.
EOJ.
```

Relative File Checklist

Identification Division: None.

Environment Division

FILE-CONTROL paragraph of the Input-Output Section

- Use Format 2 of the SELECT statement for relative files.
- Name each file used in the program in a SELECT statement and assign the file to a type of hardware using the ASSIGN TO clause.
- Use the ORGANIZATION IS clause to define the file organization as relative.
- Use the ACCESS MODE IS clause to define the access mode the program uses to retrieve records from the file. Allowable access modes are sequential, random or dynamic.
- If the program uses random or dynamic access mode, use the RELATIVE KEY IS clause to define the key variable.
- If the access mode is sequential, the RELATIVE KEY IS clause is optional.
- If the program monitors the status of the file, define a variable to receive status key values by using the FILE STATUS IS clause.

For details on the FILE-CONTROL paragraph and status key values, see Section 3.

Data Division

FD in the File Section: A valid file description entry (FD) must be defined for all relative files. Detailed information on the FD is included in Section 4.

Data-names in the Working-Storage Section

- Define the random access key. If the relative file is accessed randomly or dynamically, a data item must be defined to be the key used for random access. This data-name must be the same as the data-name in the RELATIVE KEY IS clause of the SELECT entry for the file.
- If the program monitors the status of a file, a data name must be defined to receive status key values. This data name must be the same as the data name in the FILE STATUS IS clause of the SELECT entry of the file.

Detailed information on the RELATIVE KEY IS clause and the FILE STATUS IS clause is included under "FILE-CONTROL Paragraph" in Section 3.

Procedure Division

CLOSE statement

- Use Format 2 of the CLOSE statement.
- The REEL and UNIT phrases are not valid for relative files.

Detailed information on the CLOSE statement is included in Section 6.

DELETE statement

- The DELETE statement is valid only for mass-storage files.
- If the file is accessed randomly or dynamically, a DELETE statement removes the record indicated by the contents of the data item specified by the RELATIVE KEY IS clause of the FILE-CONTROL paragraph.

Detailed information on the DELETE statement is included in Section 6.

OPEN statement: The REVERSED, NO REWIND and EXTEND phrases of the OPEN statement are not valid for relative files. Detailed information on the OPEN statement is included in Section 7.

READ statement: Use Format 1 of the READ statement to read the file in sequential access mode.

- If the access mode is random, use Format 2.
- For dynamic access mode, use Format 1 (with the NEXT phrase) or Format 2, depending on the needs of the program.

Details on the READ statement are included in Section 7. Information on relative files is included in the discussion of both Format 1 and Format 2.

REWRITE statement

- The REWRITE statement is valid only for mass-storage files.
- When the access mode is random or dynamic, a REWRITE statement replaces the record indicated by the data item used in the RELATIVE KEY IS clause of the FILE-CONTROL paragraph.
- When the access mode is sequential, a successful READ statement must be performed on the file before a REWRITE statement is performed. No other I-O statements that affect the current record pointer can be executed between the READ and the REWRITE statements.

Detailed information on the REWRITE statement is included in Section 7.

START statement

- The START statement is valid for sequential or dynamic access mode.
- The data item used in the START statement must match the data item used in the RELATIVE KEY IS clause of the FILE-CONTROL paragraph.

Detailed information on the START statement is included in Section 6.

WRITE statement : Use Format 2 of the WRITE statement. Details on the WRITE statement are included in Section 8. Information on relative files is included in the discussion of both Format 1 and Format 2.

Relative File Program Example

This COBOL program example uses a relative file. The name of the program is REL-EXMPL, and it prints a report of selected employee information from an employee master file using the employee number as the record key.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.             REL-EXMPL.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.        A-9.
OBJECT-COMPUTER.        A-9.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
      SELECT PERSONNEL-FILE ASSIGN TO DISK
                  ORGANIZATION IS RELATIVE
                  ACCESS MODE IS RANDOM
                  RELATIVE KEY IS KEY-FOR-RECORD.
      SELECT PRINT-FILE ASSIGN TO PRINTER.

DATA DIVISION.
FILE SECTION.
FD  PERSONNEL-FILE BLOCK CONTAINS 3 RECORDS.
01  EMPLOYEE-RECORD.
    05 EMPLOYEE-NUMBER   PIC 9999.
    05 NAME              PIC X(16).
    05 JOB-TITLE         PIC X(16).
    05 DATES             PIC X(16).
    05 MONTHLY-SALARY    PIC 9999.99.
01  DEPENDENT-RECORD.
    05 DEP-NAME          PIC X(24).
    05 DEP-SSN           PIC X(11).
    05 CHILD-FLAG        PIC X(1).
FD  PRINT-FILE.
01  PRINT-EMPLOYEE-RECORD.
    05 PRINT-EMPL        PIC X(16) OCCURS 5 TIMES INDEXED BY EP.
01  PRINT-DEPENDENT-RECORD.
    05 PRINT-DEP         PIC X(26) OCCURS 3 TIMES INDEXED BY DP.
    05 PRINT-TRAILER-DEP PIC XX.

WORKING-STORAGE SECTION.
01  KEY-FOR-RECORD               PIC 99999  VALUE IS 0.
01  RESULT-NEXT-EMPLOYEE         PIC X      VALUE IS "Y".
01  RESULT-DEPENDENT-NUMBER      PIC 9      VALUE IS 0.
01  RESULT-DESIRED-EMPLOYEE      PIC 9999   VALUE IS 0.

PROCEDURE DIVISION.
S1 SECTION.

MAIN-PROGRAM.
    OPEN OUTPUT PERSONNEL-FILE.
```

```
    OPEN OUTPUT PRINT-FILE.
    PERFORM ENTER-DATA UNTIL RESULT-NEXT-EMPLOYEE = "N".
    CLOSE PERSONNEL-FILE SAVE.
    OPEN I-O PERSONNEL-FILE.
    PERFORM GENERATE-REPORT THROUGH GENERATE-REPORT-ENDLOOP
            UNTIL RESULT-NEXT-EMPLOYEE = "N".
    CLOSE PERSONNEL-FILE.
    CLOSE PRINT-FILE.
    STOP RUN.

  ENTER-DATA.
      PERFORM GET-EMPLOYEE-DATA.
      MULTIPLY EMPLOYEE-NUMBER BY 3 GIVING KEY-FOR-RECORD.
      WRITE EMPLOYEE-RECORD; INVALID KEY DISPLAY "FILE SIZE EXCEE".
      PERFORM PROMPT-FOR-NUM-OF-DEP.
      PERFORM ENTER-DEPENDENT-DATA
              RESULT-DEPENDENT-NUMBER TIMES.
      PERFORM PROMPT-NEXT-EMPLOYEE.

    ENTER-DEPENDENT-DATA.
        PERFORM GET-DEPENDENT-DATA.
        ADD 1 TO KEY-FOR-RECORD.
        WRITE DEPENDENT-RECORD;
             INVALID KEY DISPLAY "FILE SIZE EXCEEDED".
****************************************************************

GENERATE-REPORT.
    PERFORM PROMPT-DESIRED-EMPLOYEE.
    MULTIPLY RESULT-DESIRED-EMPLOYEE BY 3 GIVING KEY-FOR-RECORD.
    READ PERSONNEL-FILE;
         INVALID KEY DISPLAY "NO EMPLOYEE FOR THAT NUMBER"
                 PERFORM INVALID-EMPLOYEE
                 GO TO GENERATE-REPORT-ENDLOOP
         NOT INVALID KEY  PERFORM WRITE-EMPL-RECORD.
    ADD 1 TO KEY-FOR-RECORD.
    READ PERSONNEL-FILE;
         INVALID KEY MOVE SPACES TO PRINT-DEPENDENT-RECORD
                 MOVE "NO DEPENDENTS" TO PRINT-DEP(1)
                 WRITE PRINT-DEPENDENT-RECORD
                 GO TO GENERATE-REPORT-ENDLOOP
         NOT INVALID KEY PERFORM WRITE-DEP-RECORD.
    ADD 1 TO KEY-FOR-RECORD.
    READ PERSONNEL-FILE;
         INVALID KEY MOVE SPACES TO PRINT-DEPENDENT-RECORD
                 MOVE "NO MORE DEPENDENTS" TO PRINT-DEP(1)
                 WRITE PRINT-DEPENDENT-RECORD
         NOT INVALID KEY PERFORM WRITE-DEP-RECORD END-WRITE.

GENERATE-REPORT-ENDLOOP.
    PERFORM PROMPT-NEXT-EMPLOYEE.

INVALID-EMPLOYEE.
```

```
    MOVE SPACES TO PRINT-EMPLOYEE-RECORD.
    MOVE "INVALID EMPL. #" TO PRINT-EMPL(1)
    MOVE RESULT-DESIRED-EMPLOYEE TO PRINT-EMPL(2)
    WRITE PRINT-EMPLOYEE-RECORD.

WRITE-EMPL-RECORD.
    MOVE SPACES TO PRINT-EMPLOYEE-RECORD.
    MOVE EMPLOYEE-NUMBER TO PRINT-EMPL(1).
    MOVE NAME TO PRINT-EMPL(2).
    MOVE JOB-TITLE TO PRINT-EMPL(3).
    MOVE DATES TO PRINT-EMPL(4).
    MOVE MONTHLY-SALARY TO PRINT-EMPL(5).
    WRITE PRINT-EMPLOYEE-RECORD.

WRITE-DEP-RECORD.
    MOVE SPACES TO PRINT-DEPENDENT-RECORD.
    MOVE DEP-NAME TO PRINT-DEP(1).
    MOVE DEP-SSN TO PRINT-DEP(2).
    MOVE CHILD-FLAG TO PRINT-DEP(3).
    WRITE PRINT-DEPENDENT-RECORD.

* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
GET-EMPLOYEE-DATA.
*                       TERMINAL COMMUNICATION PROCEDURE TO DISPLAY
*                       FORMATTED SCREEN AND RETURN ENTERED DATA ABOUT
*                       AN EMPLOYEE INTO THE FIELDS OF EMPLOYEE-RECORD.

GET-DEPENDENT-DATA.
*                       TERMINAL COMMUNICATION PROCEDURE TO DISPLAY
*                       FORMATTED SCREEN AND RETURN ENTERED DATA ABOUT
*                       A DEPENDENT INTO FIELDS OF DEPENDENT-RECORD.
PROMPT-NEXT-EMPLOYEE.
*                       TERMINAL COMMUNICATION PROCEDURE TO ASK IF
*                       DATA FOR ANOTHER EMPLOYEE MUST BE ENTERED.
*                       "Y" OR "N" IS RETURNED IN RESULT-NEXT-EMPLOYEE.
PROMPT-FOR-NUM-OF-DEP.
*                       TERMINAL COMMUNICATION PROCEDURE TO ASK HOW
*                       MANY DEPENDENTS DATA MUST BE ENTERED FOR.
*                       RETURNS A NUMBER FROM 0 THROUGH 2 IN
*                       RESULT-DEPENDENT-NUMBER.
PROMPT-DESIRED-EMPLOYEE.
*                       TERMINAL COMMUNICATION PROCEDURE TO ASK FOR
*                       THE EMPLOYEE NUMBER OF THE EMPLOYEE WHOSE
*                       RECORDS WILL BE INCLUDED IN THE PRINTED REPORT.
*                       THE EMPLOYEE'S NUMBER IS RETURNED IN
*                       RESULT-DESIRED-EMPLOYEE.
```

Indexed File Checklist

Identification Division: None.

Environment Division

- FILE-CONTROL paragraph of the Input-Output Section
- Use Format 3 of the SELECT statement for indexed files.
- Name each file used in the program in a SELECT statement. Use the ASSIGN TO clause to assign the file to a type of hardware.
- Use the ORGANIZATION IS clause to define the organization of the file as indexed. For documentation, define the organization explicitly (ORGANIZATION IS INDEXED).
- Use the ACCESS MODE IS clause to define the access mode the program uses to access the file. The access mode can be sequential, random, or dynamic.
- Required: Use the RECORD KEY IS clause to define the primary key for an indexed file.
- Optional: Use the ALTERNATE RECORD KEY IS clause to define any alternate keys for an indexed file.
- If the program monitors the status of the file, define a variable to receive status key values by using the FILE STATUS IS clause.

Detailed information on the FILE-CONTROL Paragraph is included in Section 3.

Data Division

FD in File Section

- A valid file description entry (FD) must be defined for all indexed files.
- All keys defined in the FILE-CONTROL paragraph must be fields in the record declared in the 01 Record Description.

Detailed information on the FD is included in Section 4.

Variables in the Working-Storage Section : If the program monitors the status of a file, you must define a data-name to receive status key values. This data-name must be the same as the data-name in the FILE STATUS IS clause of the SELECT entry for the file in the FILE-CONTROL paragraph of the Environment Division. Details on the FILE STATUS IS clause are included under "FILE-CONTROL Paragraph" in Section 3.

Procedure Division

CLOSE statement: Use Format 2 of the CLOSE statement. Detailed information on the CLOSE statement is included in Section 6.

OPEN statement: The REVERSED, NO REWIND and EXTEND phrases of the OPEN statement are not valid for indexed files. Details on the CLOSE statement are included in Section 7.

READ statement

- Use Format 1 of the READ statement for indexed files with sequential or dynamic access mode.
- Use Format 2 of the READ statement for indexed files with random access mode.

Detailed information on the READ statement is included in Section 7.

START statement

- Use the general format of the START statement for indexed files.
- If the KEY phrase is specified, the data-name can reference two different items. Refer to the syntax rules of the START statement for more information.

Detailed information on the START statement is included in Section 8.

REWRITE statement

- Use the general format of the REWRITE statement for indexed files.
- There are specific rules for using the REWRITE statement on indexed files. Refer to the syntax rules of the REWRITE statement for more information.

Detailed information on the REWRITE statement is included in Section 8.

WRITE statement: Use Format 2 of the WRITE statement for indexed files. Detailed information on the WRITE statement is included in Section 8.

Indexed File Program Example

This example uses an indexed file. The name of this program is IND-EXMPL, and it prints a report of the current clients of a kennel grouped by color, breed, or name.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.                     IND-EXMPL.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.                A9.
OBJECT-COMPUTER.                A9.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
     SELECT KENNEL-FILE ASSIGN TO DISK
         ORGANIZATION IS INDEXED
         ACCESS MODE IS RANDOM
         RECORD KEY IS NAME
         ALTERNATE RECORD KEY IS COLOR WITH DUPLICATES
         ALTERNATE RECORD KEY IS BREED WITH DUPLICATES.
     SELECT PRINT-FILE ASSIGN TO PRINTER.

DATA DIVISION.
FILE SECTION.
FD   KENNEL-FILE BLOCK CONTAINS 6 RECORDS
         VALUE OF TITLE IS "KENNEL/RECORDS".
01   KENNEL-RECORD.
     05 NAME             PIC X(12).
     05 COLOR            PIC X(10).
     05 BREED            PIC X(10).
     05 PRICE            PIC 9999.99.
     05 KENNEL           PIC X(10).
     05 KENNEL-NUMBER    PIC 9999.
FD   PRINT-FILE.
01   PRINT-RECORD.
     05 PRINT-ITEM       PIC X(12) OCCURS 6 TIMES INDEXED BY J.
     05 PRINT-TRAIL      PIC X(8).
WORKING-STORAGE SECTION.
01   RED                  PIC X(10)       VALUE IS "RED".
01   TAN                  PIC X(10)       VALUE IS "SPOTTED".
01   BLACK                PIC X(10)       VALUE IS "BLACK".
01   EXPENSIVE            PIC 999V99      VALUE IS 800.00.
01   CHEAP                PIC X(10)       VALUE IS "CHEAP".
01   NO-SALE              PIC X(10)        VALUE IS "NO SALE".
01   INDEX-TYPE           PIC X(10) DISPLAY.
```

```
 PROCEDURE DIVISION.
  SECTION-1 SECTION.

  PROCEDURE-1.
       OPEN OUTPUT KENNEL-FILE.
       MOVE "OTTO" TO NAME.        MOVE "GOLDEN" TO COLOR.
       MOVE EXPENSIVE TO PRICE.    MOVE BLACK TO KENNEL.
       MOVE 13 TO KENNEL-NUMBER.   MOVE "COLLIE" TO BREED.
       WRITE KENNEL-RECORD;
             INVALID KEY DISPLAY "ERROR - PRIMARY KEY NOT UNIQUE".

       MOVE "GERONIMO" TO NAME.    MOVE RED TO COLOR.
       MOVE 350.00 TO PRICE.       MOVE "ANOTHER" TO KENNEL.
       MOVE 97 TO KENNEL-NUMBER.   MOVE "RETRIEVER" TO BREED.
       WRITE KENNEL-RECORD;
             INVALID KEY DISPLAY "ERROR - PRIMARY KEY NOT UNIQUE".

       MOVE "CHARLIE" TO NAME.     MOVE "WHITE" TO COLOR.
       MOVE CHEAP TO PRICE.        MOVE "NONE" TO KENNEL.
       MOVE 01 TO KENNEL-NUMBER.   MOVE "MIXED" TO BREED.
       WRITE KENNEL-RECORD;
             INVALID KEY DISPLAY "ERROR - PRIMARY KEY NOT UNIQUE".
       CLOSE KENNEL-FILE SAVE.

 PROCEDURE-2.
  *         This procedure opens the indexed file, reads
  *         data by alternate keys, writes to a printer file.
       OPEN I-O KENNEL-FILE. OPEN OUTPUT PRINT-FILE.

       MOVE "RED" TO COLOR.
       MOVE "COLOR" TO INDEX-TYPE.
       READ KENNEL-FILE KEY IS COLOR
            INVALID KEY PERFORM INVALID-READ-MARKER
            NOT INVALID KEY PERFORM WRITE-OUT-RECORD.

       MOVE "MIXED" TO BREED.
       MOVE "BREED" TO INDEX-TYPE.
       READ KENNEL-FILE KEY IS BREED
            INVALID KEY PERFORM INVALID-READ-MARKER
            NOT INVALID KEY PERFORM WRITE-OUT-RECORD.

        MOVE "WHITE" TO COLOR.
        MOVE "COLOR" TO INDEX-TYPE.
        READ KENNEL-FILE KEY IS COLOR
             INVALID KEY PERFORM INVALID-READ-MARKER
             NOT INVALID KEY PERFORM WRITE-OUT-RECORD.
             CLOSE KENNEL-FILE. CLOSE PRINT-FILE.
        STOP RUN.

  WRITE-OUT-RECORD.
       MOVE SPACES TO PRINT-RECORD.
       MOVE NAME TO PRINT-ITEM(1).
```

```
           MOVE COLOR TO PRINT-ITEM(2).
           MOVE BREED TO PRINT-ITEM(3).
           MOVE PRICE TO PRINT-ITEM(4).
           MOVE KENNEL TO PRINT-ITEM(5).
           MOVE KENNEL-NUMBER TO PRINT-ITEM(6).
           WRITE PRINT-RECORD.

      INVALID-READ-MARKER.
           DISPLAY "ERROR - NO SUCH " INDEX-TYPE " IN FILE"
           MOVE "INVALID RECORD ACCESS" TO PRINT-RECORD.
```

Section 13
Tasking in COBOL85

Note: *Although this section is not highlighted, all of the information is relative to Unisys extensions to COBOL ANSI-85.*

COBOL85 provides a tasking capability, which enables you to write programs that initiate other programs. This section discusses the basic concepts of tasking, including:

- Terminology used to describe programs and processes
- A discussion of task attributes and task variables
- The three conditions that affect interprocess relationships
- A discussion of coroutines
- The instructions for writing a COBOL85 program that initiates other programs
- The definition of a critical block and information for preventing critical block exits

You should read Sections 1 and 2 of the *Task Management Programming Guide* before you attempt to implement tasking in a COBOL85 program.

Programs and Processes

A *program* is a sequence of statements that are stored in a source file. When the source file is compiled, an *object code file* is created. You can initiate an object code file by using a variety of commands and statements. Initiation causes the system to start performing the instructions in the object code file. At this point, the object code file is being *executed*. The system reads and performs the instructions in the object code file without altering the contents of the file.

A separate, dynamic entity called a *process* is initiated any time an object code file is initiated. The process appears in the active system mix and reflects the current state of the execution of the object code file. A process also stores information used by the object code file and keeps track of which statement is currently being executed.

Task Attributes

All processes possess specific characteristics such as a usercode, a mix number, a priority, printer defaults, and so on. These characteristics are defined by the operating system and are known as *task attributes.* Task attributes record or control many aspects of process execution, including security, processor usage, memory usage, and I/O activity.

Task attributes have a fixed meaning, but their values can vary. For example, the USERCODE task attribute always indicates the person who owns a particular process. However, one process might have a USERCODE value of JSMITH and another process might have a USERCODE value of JANEDOE.

You can display the value of any task attribute, except string-type task attributes (attributes whose values are characters strings), by using the DISPLAY statement. For string-type task attributes, you must move the attribute into a data area with the MOVE statement, and then display the value with the DISPLAY statement. Refer to the second program example in the following Examples section.

Attributes with an implicit numeric class can be used in DISPLAY statements and in place of any identifier in an arithmetic statement, except the receiving-field identifier.

You can determine the mnemonic value of a task attribute by using the task attribute in a conditional expression. For details about conditional expressions, see Section 5.

In general, the types of task attributes and the values that are valid for them are shown in the following table. For a complete list of task attributes and their possible values, refer to the *Task Attributes Programming Reference Manual.*

Attribute Type	Values Accepted and Returned
String	Alphanumeric
Boolean	Numeric (or the value associated with a mnemonic)
Integer	Numeric (or the value associated with a mnemonic)
All other attributes types	Numeric identifier, literal, arithmetic expression, or the value associated with a mnemonic

You can change the value of a task attribute by using the CHANGE statement in the Procedure Division of your COBOL85 program.

Task Variables

Because the same task attributes are common to all processes, the system must be able to determine which task attribute value belongs to which process. For example, every process has a USERCODE task attribute. When a program assigns a value to the USERCODE task attribute, the system must have some way to identify the process to which you want to apply the new USERCODE value. The system can differentiate among processes by using *task variables*.

A task variable is a name that you use to represent a particular process. The system automatically provides several predeclared task variables. Two of these variables are MYSELF and MYJOB. The MYSELF task variable refers to the process itself. The MYJOB task variable refers to the independent process in a group of related dependent processes—the process *family*. (For a discussion of familial relationships among processes, refer to the *Task Management Programming Guide.)*

In a COBOL85 program, you create a task variable by declaring a data item in the Data Division with the USAGE IS TASK clause. You can associate a task variable with a particular process by specifying the task variable in the program initiation statement (either CALL, PROCESS, or RUN) in the Procedure Division. After the program is initiated, the task variable is associated with the resulting process. For details about using task variables in program initiation statements, refer to Format 6 of the CALL statement, the PROCESS statement, or the RUN statement.

When a mnemonic value is referenced in a context that is not associated with any of the task attribute mnemonic identifiers, then it is treated as a signed numeric constant.

The following program fragment sets the BDBASE option of the OPTION task attribute. Note that the mnemonic value BDBASE is used as a destination bit location in this case. Specific options of the OPTION task attribute can be accessed by using mnemonic identifiers. The mnemonic identifiers represent specific bits in the OPTION word. One way to access these bits is to use the Format 3 MOVE statement.

```
WORKING-STORAGE SECTION.
01  OPTION-WORD  PIC 9(11)  BINARY.
01  VALUE-ONE    PIC 9(11)  BINARY  VALUE 1.
PROCEDURE DIVISION.
OPTION-TEST.
     MOVE ATTRIBUTE OPTION OF MYSELF TO OPTION-WORD.
     MOVE VALUE-ONE TO OPTION-WORD [0:VALUE BDBASE:1].
     CHANGE ATTRIBUTE OPTION OF MYSELF TO OPTION-WORD.
```

Interprocess Relationships

The type of relationship a process has with the process that initiated it depends upon whether the initiated procedure

- Exists internally or externally to the initiator
- Relies on the continued existence of its initiator
- Runs in parallel with the initiator or takes turns

The following subsections describe the way processes behave in each situation.

Internal Processes

An internal process results from the initiation of an internal procedure. A COBOL85 program cannot *initiate* an internal procedure.

External Processes

An external process results from the initiation of an external procedure. External procedures are separate programs that exist outside the main program.

External processes do not inherit task attribute values.

COBOL85 programs can initiate separate programs by using the CALL, PROCESS, and RUN statements.

Synchronous and Asynchronous Processes

Another condition that affects process relationships is the way the process shares the processor. That is, does it take turns executing with the other process (synchronous processing), or does it run in parallel with the other process (asynchronous processing). Both situations are discussed in the following paragraphs.

Synchronous Processes

A COBOL85 program can initiate a synchronous process by using the CALL statement.

When a synchronous process is initiated, the initiating process stops executing and the new process begins executing. The initiating process is still considered active during this period and its process stack still exists. When the initiated process terminates, the initiating process begins executing again, starting with the first executable statement after the process initiation statement.

The initiating program can set the attributes of a synchronous process only at initiation time and can interrogate the attributes only after the synchronous process has terminated.

Synchronous processes are sometimes referred to as coroutines, but more properly the term coroutine has a different use. For details, refer to "Coroutines" later in this section.

Asynchronous Processes

A COBOL85 program can initiate an asynchronous process by using either the PROCESS or the RUN statement.

When an asynchronous process is initiated, the new process and the initiator execute in parallel. Although they execute at the same time, they do not necessarily execute at the same speed. It is for this reason that the new process is called *asynchronous*.

The initiating process can read or assign the task attributes of an asynchronous process while the process is executing.

When you initiate an asynchronous process, you must take special measures to prevent a *critical block*exit error from occurring. For details, refer to the discussion of "Preventing Critical Block Exits" later in this section.

Note that initiating processes asynchronously can create ambiguous timing situations because it is impossible to predict exactly how long a process will take to execute. To assist you in regulating the timing of asynchronous processes, you can use events, locks, and interrupt procedures. For an overview of these mechanisms, refer to the *Task Management Programming Guide.*

For information about establishing events in COBOL85, refer to the USAGE clause in Section 4 , the ATTACH, CAUSE, and DETACH statements in Section 6, the RESET statement in Section 7, and WAIT statement in Section 8.

For information about establishing locks in COBOL85, refer to the USAGE clause in Section 4, the LOCK statement in Section 7, and the UNLOCK statement in Section 8.

For information about establishing interrupt procedures in COBOL85, refer to the ALLOW and DISALLOW statements in Section 6 and the USE and WAIT statements in Section 8.

Dependent and Independent Processes

The final condition that affects interprocess relationships is *dependency*. The concept of dependency involves two related concepts: *critical objects* and *parents*.

Critical objects are items that are declared by one process and used by another process, such as the task variable and parameters. When a process is initiated, it receives these critical objects from the initiator (also called the *parent*). Dependency is the relationship between a process and its parent process, which determines how the system stores the critical objects.

When an *independent* process is initiated, the system creates a separate copy of the critical objects for the new process to use. As a result, the independent process can continue executing if the parent process terminates.

The COBOL85 RUN statement initiates an independent process. An independent process is sometimes referred to as a *job*.

When a *dependent* process is initiated, the system creates references to the objects stored by the parent. Because of the sharing of the critical objects, a dependent process relies on the continued existence of its parent.

The COBOL85 CALL statement initiates a dependent, synchronous process, and the PROCESS statement initiates a dependent, asynchronous process. A dependent process is sometimes referred to as a *task*.

The dependency of a process remains the same throughout execution. If the process is initiated as dependent, it cannot later become independent or vice versa.

Details about Process Dependency

Observe the following details when planning the execution of dependent and independent processes. For an expanded discussion of the effects of dependency on processes, refer to the *Task Management Programming Guide*.

Independent Processes

- Only external processes that result from the initiation of separate programs can be independent.
- An independent process is always asynchronous.
- Parameters passed to an independent process can be passed only by value.
- The task variable for an independent process ceases to be associated with the parent once the independent task is initiated.

Dependent Processes

- A dependent process is asynchronous if it is initiated with the PROCESS statement, or synchronous if it is initiated with the CALL statement.
- Parameters passed to a dependent process can be passed by reference or by value.
- The task variable for a dependent process remains associated with the parent for as long as the parent exists.

Coroutines

The term *coroutines* refers to a group of processes that exist simultaneously but take turns executing, so that only one of the processes is executing at any given time. Every synchronous process is a coroutine. However, not every coroutine is a synchronous process. Unlike synchronous processes, which are terminated when exited, control can alternate between the parent process and the coroutine.

The use of coroutines offers the following benefits:

- The ability to execute a procedure repeatedly without incurring the processor time required to enter or initiate the procedure each time.
- The ability to execute a procedure repeatedly without losing the values of objects declared in the procedure between each execution.

You can implement coroutines in your COBOL85 program by using the CALL, CONTINUE, and EXIT PROGRAM statements. These statements perform the following functions:

Statement	Location	Function
CALL	Calling program	Initiates a dependent process
EXIT PROGRAM	Called program	Causes control to be returned to the parent
CONTINUE	Calling program	Returns control to the dependent process

For details about the CALL, CONTINUE, and EXIT PROGRAM statements, refer to the discussion of each statement in Section 6 of this manual. For more information about coroutines, refer to the *Task Attributes Programming Reference Manual*.

Structuring a Program to Initiate Processes

Writing a COBOL85 program that initiates a separate process requires you to programmatically perform the functions described in the following table.

Function to Perform	Division
Name the object code file to be executed as a process	Environment or Data Division
Describe any parameters to be passed between the two programs	Data Division
Declare a task variable	Data Division
Associate parameters with the called program	Procedure Division
Declare the name of the external program to be used as the procedure	Procedure Division
Change task attribute values as necessary	Procedure Division
Choose the appropriate program initiation statements	Procedure Division

Environment Division

You can specify the name of the external program to be executed in the Special-Names paragraph of the Environment Division. The format for this specification is as follows:

```
ENVIRONMENT DIVISION.
SPECIAL-NAMES.
    "OBJECT/TESTPROG" IS TESTPROG.
```

This format uses the "literal IS mnemonic-name" format. OBJECT/TESTPROG is the literal name of the object code file. TESTPROG is the mnemonic-name by which you refer to the object code file. You use the mnemonic-name that you assign in this division in the USE EXTERNAL phrase of the Declaratives Section of the Procedure Division.

Data Division

In this division you

- Name the object program to be executed (alternate method)
- Declare the task variable
- Describe parameters in the called program that are to be passed between programs
- Describe parameters in the calling program that are to be passed between programs

Naming the Program to Be Executed (Alternate Method)

You can name the external program that is to be executed by declaring the program as a data item in the Working-Storage Section of the Data Division. You can use the VALUE clause with the declaration as shown in the following example:

```
DATA DIVISION.
WORKING-STORAGE SECTION.

01 TESTPROG                    PIC X(15) VALUE IS "OBJECT/TESTPROG".
```

You use the data item you declared in this division in the USE EXTERNAL phrase in the Declaratives Section of the Procedure Division.

For details about describing data in the Data Division, refer to Section 4 of this manual.

Declaring the Task Variable

To declare a task variable, define a 77-level data item or a 01-level or subordinate data item in the Working-Storage Section of the Data Division with the USAGE IS TASK clause.

If you specify the USAGE IS TASK clause for a group item, all the elementary items in the group are task variables. The group itself is not a task variable. A group item thus defined can be used only in the USING phrase of the CALL statement Format 4 (for binding), CALL statement Format 6 (for tasking), the PROCESS statement, and the RUN statement.

An example of the syntax used for declaring a task variable is as follows:

```
DATA DIVISION.
WORKING-STORAGE SECTION.

01 TASK-VAR-1           USAGE IS TASK.
```

You use the task variable you define here with one of the program initiation statements in the Procedure Division. Doing so associates the task variable with the initiated process. You can dissociate a task variable from a process by using the DETACH statement.

For details on describing data in the Working-Storage Section, see Section 4 of this manual.

Describing the Formal Parameters in the Called Program

Parameters in the called program that are to be used as references by both programs must be described in the Linkage Section of the Data Division of the called program. The data names you describe here are used as references to formal parameters in the Procedure Division Header.

An example of this syntax is as follows:

```
DATA DIVISION.
LINKAGE SECTION.
01  RECEIVE-STRING             PIC X(6)
01  RECEIVE-NUMBER             PIC S9(22) COMP.
```

These parameters are received by the called program for use in its Procedure Division statements.

For details on describing data items in the Linkage Section, see Section 4 of this manual.

Describing the Formal Parameters in the Calling Program

Parameters in the calling program that are to be referenced by both the calling and the called programs must be described in the Local-Storage Section of the Data Division of the calling program. The data items you describe here are referenced as formal parameters in the USE EXTERNAL statement in the Declaratives Section of the Procedure Division. An example of this syntax is as follows:

```
DATA DIVISION.
LOCAL-STORAGE SECTION.
LD  PARAMS.
01  FORMAL-STRING            PIC X(6).
01  FORMAL-NUMBER            PIC S9(11) COMP.
```

These formal parameters are compared to the actual parameters specified in the process initiation statement (CALL, PROCESS, or RUN) of the calling program.

For details on describing data items in the Local-Storage Section, see Section 4 of this manual.

Describing the Actual Parameters in the Calling Program

Parameters in the calling program that are to be sent to the called program must be described in either the Working-Storage or the Linkage Section in the Data Division of the calling program. The data items you describe here are specified in the process initiation statement (CALL, PROCESS, or RUN) in the Procedure Division. These data items are also compared to the formal parameters referenced in the USE EXTERNAL statement in the Declaratives portion of the Procedure Division. An example of this syntax is as follows:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01  SEND-STRING              PIC X(6).
01  SEND-NUMBER              PIC S9(11) COMP.
```

These actual parameters are sent from the calling program to the called program.

Procedure Division

In this division, you

- Associate parameters with the called program.
- Declare the name of the external program and any parameters it references.
- Change task attribute values.
- Specify the process initiation statement.
- Specify the EXIT PROGRAM statement and the CONTINUE statement for coroutines.
- Dissociate a task variable from a process.

Procedure Division Header in the Called Program

You must begin the Procedure Division of the called program with a header that names the identifiers that the called process is to receive as parameters. The identifiers you use in this header must be defined in the Linkage Section of the Data Division. An example of this syntax is as follows:

```
PROCEDURE DIVISION USING RECEIVE-STRING, RECEIVE-NUMBER.
```

Declaratives Section

You must declare the name of the external program to be executed with a USE EXTERNAL statement. A section-name must precede the USE EXTERNAL statement. The object code file name is either the mnemonic name you defined in the Special-Names paragraph of the Environment Division or the name you declared in the Working-Storage Section of the Data Division. An example of this syntax is as follows:

```
PROCEDURE DIVISION.
DECLARATIVES SECTION.
EXTERNAL-PROG SECTION.
   USE EXTERNAL TESTPROG AS PROCEDURE WITH PARAMS
        USING FORMAL-STRING, FORMAL-NUMBER.
END DECLARATIVES.
```

The USE EXTERNAL phrase can also reference the parameters described in the Local-Storage Section of the Data Division. For details about the USE statement, refer to Section 8 of this manual.

Changing Task Attribute Values

You can include a CHANGE statement anywhere after the Declaratives Section to change the value of a task attribute. For details about the CHANGE statement, refer to Section 6 of this manual. For a description of the task attributes and their default values, refer to the *Task Attributes Programming Reference Manual*.

Initiating External Procedures

You can initiate an external procedure by using one of the statements as follows:

Use the . . .	To initiate a program as . . .
CALL statement	A synchronous, dependent process.
PROCESS statement	An asynchronous, dependent process.
RUN statement	An asynchronous, independent process.

An example of this syntax follows:

```
CALL TASK-VAR-1 WITH EXTERNAL-PROG
             USING SEND-STRING, SEND-NUMBER.
```

For details about these program-initiation statements, refer to Sections 6 through 8.

Implementing Coroutines

You can implement coroutines by using the CALL, CONTINUE, and EXIT PROGRAM statements. The CALL statement creates a synchronous task that is an active coroutine and changes the parent process into a continuable coroutine. The task can return control to its parent by executing an EXIT PROGRAM statement. The parent can return control to its task by executing a CONTINUE statement.

The EXIT PROGRAM statement, in addition to transferring control to the parent, also specifies the place where execution resumes when the parent later continues the task. The simple form EXIT PROGRAM specifies that the task resumes from the beginning. The EXIT PROGRAM RETURN HERE form specifies that the task resumes with the statement that follows the EXIT PROGRAM statement.

For details about the CALL, CONTINUE, and EXIT PROGRAM statements, refer to Section 6 of this manual.

Dissociating a Task Variable from a Process

When a task variable is used in a CALL, PROCESS, or RUN statement, the variable is associated with the process initiated by that statement. To dissociate the task variable from the process, you can use the DETACH statement. For details about using the DETACH statement, refer to Section 6 of this manual.

Examples of Declaring the Object Code File Name of the Called Program

The following example shows how you can declare the name of the object code file by changing the NAME attribute of the task variable with the CHANGE statement before using the CALL statement.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CALL-TASK-CALLER.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 DEP-TASK TASK.
PROCEDURE DIVISION.
DECLARATIVES.
INITIATE-PROCESS SECTION.
     USE EXTERNAL AS PROCEDURE.
END DECLARATIVES.
MAIN SECTION.
MAIN-PARA.
     CHANGE ATTRIBUTE NAME OF DEP-TASK TO "OBJECT/C85/CALLED".
     CALL DEP-TASK WITH INITIATE-PROCESS.
     STOP RUN.
```

The next example shows how you can declare the object code file name of the called program by declaring a mnemonic name in the Special-Names paragraph of the Environment Division and then using it in the USE EXTERNAL statement in the Declaratives. In this example, the object code file is titled OBJECT/C85/CALLED, and the mnemonic to which it is assigned is TASK-ID.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CALL-TASK-CALLER.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SPECIAL-NAMES.
     "OBJECT/C85/CALLED" IS TASK-ID.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 DEP-TASK TASK.
PROCEDURE DIVISION.
DECLARATIVES.
INITIATE-PROCESS SECTION.
     USE EXTERNAL TASK-ID AS PROCEDURE.
MAIN SECTION.
MAIN-PARA.
     CALL DEP-TASK WITH INITIATE-PROCESS.
     STOP RUN.
```

Example of Passing Control between Two Programs

The calling program initiates the program OBJECT/C85/CALLED as a separate process and contains statements to pass control between the calling and the called program.

he object code file name of the called program is declared in the calling program by

1. The definition of a data item in the Working-Storage Section
2. The inclusion of that data item in the USE EXTERNAL statement in the Declaratives of the Procedure Division
3. The assignment of an object code file title to the data item by the use of a MOVE statement in the Procedure Division

Calling Program

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CALL-TASK-CALLER.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
77 TASK-SUB                        PIC 9 VALUE 4.
01 DEP-TASK-ARRAY                  TASK.
     05 DEP-TASK                   OCCURS 5 TIMES.
77 WS-PROGID                       PIC X(40).
PROCEDURE DIVISION.
DECLARATIVES.
RUN-A-PROCESS SECTION.
     USE EXTERNAL WS-PROGID AS PROCEDURE.
END DECLARATIVES.
MAIN SECTION.
MAIN-PARA.
    MOVE "OBJECT/C85/CALLED." TO WS-PROGID.
    CALL DEP-TASK (TASK-SUB) WITH RUN-A-PROCESS.
    DISPLAY "CONTINUE " WS-PROGID.
    CONTINUE DEP-TASK (TASK-SUB).
    DISPLAY "FINAL RETURN " WS-PROGID.
    DETACH DEP-TASK (TASK-SUB).
WAIT-HERE.
    IF ATTRIBUTE STATUS OF DEP-TASK(TASK-SUB) > VALUE
         (TERMINATED) THEN
          WAIT AND RESET
          UNTIL ATTRIBUTE EXCEPTIONEVENT OF MYSELF
     GO TO WAIT-HERE.
STOP RUN.
```

In the preceding program, a group of task variables is declared in the Working-Storage Section. These task variables share the name DEP-TASK. The system distinguishes them logically by number, DEP-TASK 1, 2, 3, and so on. A task variable subscript is declared as a data item named TASK-SUB, and the value of 4 is assigned to it. A task variable is used with a subscript to indicate which specific task variable of a group is to be used, in this case DEP-TASK 4.

The called program, OBJECT/C85/CALLED, is initiated by the calling program.

Called Program

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CALL-TASK-CALLED.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
77 MY-NAME             PIC X(45).
PROCEDURE DIVISION.
MAIN SECTION.
MAIN-PARA.
     MOVE ATTRIBUTE NAME OF MYSELF TO MY-NAME.
     DISPLAY MY-NAME " WAS CALLED".
     EXIT PROGRAM RETURN HERE.
     DISPLAY MY-NAME " CALLED AGAIN".
     EXIT PROGRAM.
     STOP RUN.
```

The transfer of control between these two programs occurs in the following sequence:

1. The calling program executes the statement CALL DEP-TASK (TASK-SUB) WITH RUN-A-PROCESS.
2. The called program begins execution at the MOVE statement and continues executing until it reaches the first EXIT PROGRAM statement.
3. Upon execution of the EXIT PROGRAM statement in the called program, control is returned to the statement following the CALL statement in the calling program, which is DISPLAY “CONTINUE” WS-PROGID.
4. After executing the DISPLAY statement, the calling program executes a CONTINUE statement, and control is returned to the second DISPLAY statement in the called program: DISPLAY MY-NAME “CALLED AGAIN”.
5. After executing the DISPLAY statement, the called program executes an EXIT PROGRAM statement, and control is returned to the second DISPLAY statement in the calling program (DISPLAY “FINAL RETURN” WS-PROGID) without executing the STOP RUN in the called program.
6. After executing the DISPLAY statement, the calling program executes a DETACH statement to dissociate the called process from the task variable. The status attribute of the process attached to the task variable is updated to TERMINATED, and the process is discontinued.
7. To prevent a critical block exit, the calling program cannot terminate before the called program. So that this does not occur, the calling program executes a WAIT statement with the condition that if the status of DEP-TASK is not updated to terminated, the calling program waits until its own EXCEPTIONEVENT attribute is been caused. Then it returns to WAIT-HERE. When the attribute of DEP-TASK is equal to –1 (a status of TERMINATED), the calling program executes its STOP RUN statement and ceases to execute.

Because the called program does not execute its STOP RUN statement, the DETACH statement is used to dissociate a called process from its task variable. If the calling program contains no DETACH statement, the calling program would execute its STOP RUN statement and terminate, causing a critical block exit error.

Preventing Critical Block Exits

A critical block is a block that includes the definition of the critical objects that are to be passed from the initiating process to the initiated process. The critical objects include task variables and parameters to be passed.

The process that executes the critical block is considered to be the parent of any process it initiates. All processes initiated by the parent are considered to be *offspring* of that parent. If the parent exits the critical block while a dependent process is in use, the error message "CRITICAL BLOCK EXIT" is displayed. The parent process is terminated and all offspring are discontinued.

To prevent a critical block exit error, you can include statements such as the following to check for termination of dependent processes before terminating your program:

```
PROCWAIT SECTION.
P2.
    WAIT AND RESET UNTIL ATTRIBUTE EXCEPTIONEVENT OF MYSELF.
    IF ATTRIBUTE STATUS OF TASK-VAR-1 IS GREATER THAN
       VALUE TERMINATED THEN GO PROCWAIT.
    STOP RUN.
```

The preceding example assumes that an asynchronous offspring was initiated by using the task variable TASK-VAR-1. The COBOL85 program waits on its own EXCEPTIONEVENT task attribute, which is automatically caused whenever the offspring changes status. The program then checks the status of the offspring and returns to a waiting state if the offspring has not yet terminated.

Section 14
Report Writer

This section explains how to use Report Writer, which is a special-purpose language subset of COBOL that enables you to produce reports.

Overview

Report Writer enables you to specify the physical appearance of a report, rather than requiring specification of the detailed procedure necessary to produce that report.

A hierarchy of levels is used in defining the logical organization of a report. Each report is divided into report groups, which in turn are divided into sequences of items. This hierarchical structure enables explicit reference to a report group with implicit reference to other levels in the hierarchy. A report group contains one or more items to be presented on zero, one, or more lines.

For each report group, you must define an output file, called the *report file,* with a sequential file organization. A report file has a file description entry containing a REPORT clause. The content of a report file consists of records that are written under control of the report writer control system (RWCS). The structure of a report file is defined in the File Section of the Data Division. A report file differs from a regular sequential file in the following two ways:

- A REPORT clause is associated with a report file.
- Record description entries cannot follow the file description entry for a report file.

A report file is referred to and accessed by the following statements:

- OPEN
- GENERATE
- INITIATE
- SUPPRESS
- TERMINATE
- USE AFTER STANDARD EXCEPTION PROCEDURE
- USE BEFORE REPORTING
- CLOSE

File Section

The File Section defines the structure of data files. When you use Report Writer, the File Section defines the structure of report files. You define each report file by a file description entry containing a REPORT clause.

REPORT Clause

The REPORT clause specifies the names of reports that make up a report file.

```
{ REPORT IS  }
{            }    { report-name-1 } ...
{ REPORTS ARE}
```

REPORT IS
REPORTS ARE

These keywords indicate the number of reports in a file.

report-name-1

Each report-name specified in a REPORT clause must be the subject of a report-description entry in the Report Section. The order of appearance of the report-names is not significant. A report-name must appear in only one REPORT clause.

In the Procedure Division, you can reference the subject of a file-description entry that specifies a REPORT clause only by the USE statement, the CLOSE statement, or the OPEN statement with the OUTPUT or EXTEND phrase.

Details

The presence of more than one report-name in a REPORT clause indicates that the file contains more than one report.

After execution of an INITIATE statement and before the execution of a TERMINATE statement for the same report file, the report file is under the control of the report writer control system (RWCS). While a report file is under the control of RWCS, you cannot execute any input-output statement that references a report file.

If the associated file connector is an internal file connector, each file description entry in the run unit that is associated with that file connector must describe it as a report file.

Report Section

You must describe, in the Report Section, the format of each report named in the REPORT clause of a file description entry. The Report Section is located in the Data Division of a source program (refer to Section 4, "General Format of the DATA DIVISION," for proper placement). The Report Section consists of the following two components:

- A report description entry
- A report-group description entry

Both of these components are described in the following pages.

Report Description Entry

The report description entry contains information pertaining to the overall format and structure of a report named in the File Section. It is uniquely identified in the Report Section by the level indicator RD.

```
RD report-name-1

  [ CODE literal-1 ]
  [ { CONTROL IS  } { FINAL [ data-name-1 ] ... } ]
  [ { CONTROLS ARE} { { data-name-1 } ...       } ]

  [
  [   PAGE [ LIMIT IS  ] integer-1 [ LINE  ] ]
  [        [ LIMITS ARE]           [ LINES ] ]

      [ HEADING integer-2 ]

      [ FIRST DETAIL integer-3 ]

      [ LAST DETAIL integer-4  ]
                                 ]
      [ FOOTING integer-5 ]      |
                                 ]
```

RD

The level indicator RD (report description) identifies the beginning of a report description and must precede the report-name.

report-name-1

This is a user-defined name assigned to the report. This must appear in only one REPORT clause.

Details

The clauses that follow the report-name are optional, and the order of appearance is not significant. These clauses are the CODE clause, the CONTROL clause, and the PAGE clause. Each of these clauses is discussed in turn in the pages that follow.

Report-name is the highest permissible qualifier that you can specify for LINE-COUNTER, PAGE-COUNTER, and all data-names defined in the Report Section. Refer to "Special Counters" in this section for descriptions of LINE-COUNTER and PAGE-COUNTER.

CODE Clause

The CODE clause specifies a two-character literal that identifies each print line as belonging to a specific report.

```
CODE literal-1
```

Literal-1

Literal-1 must be a 2-character nonnumeric literal.

Details

If you specify the CODE clause for any report in a file, it must be specified for all reports in that file.

When you specify the CODE clause, literal-1 is automatically placed in the last two character positions of each generated report.

The positions occupied by literal-1 are not included in the description of the print line, but are included in the size of a logical record.

If more than one report is associated with a file and the reports are produced simultaneously, you can use the CODE clause literal to select a report to be printed individually from the WFL PB statement. The system printer-backup routine will look for printer backup files with BDREPORT as the filename prefix, and then print the report indicated by the CODE literal you specify. To take advantage of this you must set BDNAME before opening the file to write to it. Once BDNAME is set, all printer files that the program opens will use the BDREPORT prefix. If you need to open a printer file with the default BDNAME, then you must reset BDNAME to null before opening it.

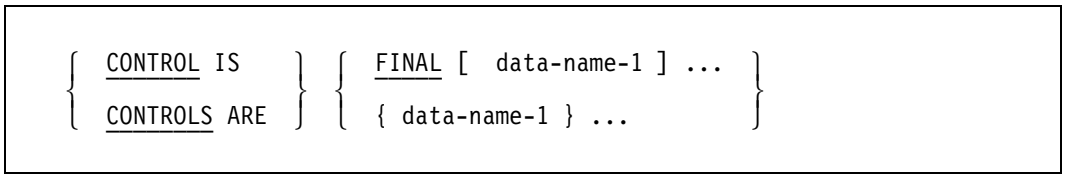
Use the CHANGE statement to set and reset BDNAME in the program. CHANGE ATTRIBUTE BDNAME OF MYSELF TO "BDREPORT" sets it. CHANGE ATTRIBUTE BDNAME OF MYSELF TO "." resets it. With BDNAME set to BDREPORT, either of the two following statements can be used to print a report, based upon its CODE literal:

```
PB D job-number KEY REPORT EQUAL literal-1.

PB D * KEY REPORT EQUAL literal-1.
```

Job-number is the mix number of the job that created the report. Literal-1 is the CODE clause literal specified in the CODE clause for the report. The asterisk (*) indicates that the job number is that of the WFL job itself. The asterisk function is useful when a PB (Printer Backup) statement is included in a WFL statement that both creates and prints the report. For further details on the use of PB, please refer to Section 3, "The SYSTEM/BACKUP Utility" of the *Printing Utilities Operations Guide* (8600 0692).

CONTROL Clause

The CONTROL clause establishes the level of the control hierarchy for the report.



data-name-1

The data-name must not be defined in the Report Section. You can qualify the data-name, however, you cannot subscript or index the data-name.

Each data-name must identify a different data item.

Data-name-1 must not have subordinate variable-occurrence data items. Control data items are subject to the same rules that apply to SORT keys.

FINAL
data-name-1

The FINAL keyword and the data-names specify the level of the control hierarchy. FINAL is the highest control. Data-name-1 is the major control. The next recurrence of data-name-1 is an intermediate control, and so on. The last recurrence of data-name-1 is the minor control.

Details

The CONTROL clause is required when you use control heading or control footing groups. The data-names specified in the CONTROL clause are the only data-names referred to by the RESET and TYPE clauses in the report-group descriptions for a report. You cannot reference a data-name, including FINAL, by more than one type-control-heading report group and one type-control-footing report group.

The execution of the first chronological GENERATE statement for a report causes the values of all control data items associated with that report to be saved. On subsequent executions of all GENERATE statements for that report, control data items are tested for a change of value. A change of value in any control data item causes a control break to occur. The control break is associated with the highest level for which a change of value is noted.

A test for a control break is made by comparing the contents of each control data item with the prior contents saved from the execution of the previous GENERATE statement for the same report.

The relation test is applied as follows:

1. If the control data item is a numeric data item, the relation test is for the comparison of two numeric operands.
2. If the control data item is an index data item, the relation test is for the comparison of two index data items.
3. If the control data item is a data item other than those described in items 1 and 2 above, the relation test is for the comparison of two nonnumeric operands.

A control break for FINAL occurs before the first detail line is printed and whenever a TERMINATE statement is executed. A control break occurring at a particular level implies a control break for each lower level in the control hierarchy. For example, if you use the CONTROL clause "CONTROLS ARE MAJ-KEY, INT-KEY, MIN-KEY", and you specify control headings and footings, they are printed in the following order on a control break on MAJ-KEY:

```
CONTROL FOOTING      (for MIN-KEY)

CONTROL FOOTING      (for INT-KEY)

CONTROL FOOTING      (for MAJ-KEY)

CONTROL HEADING      (for MAJ-KEY)

CONTROL HEADING      (for INT-KEY)

CONTROL HEADING      (for MIN-KEY)
```

PAGE Clause

The PAGE clause defines the length of a page and the vertical subdivisions within which report groups are presented.

```
PAGE [ LIMIT IS    ]   integer-1 [ LINE  ]
     [ LIMITS ARE  ]               [ LINES ]

        [ HEADING integer-2 ]

        [ FIRST DETAIL integer-3 ]

        [ LAST DETAIL integer-4 ]

        [ FOOTING integer-5 ]
```

LIMIT IS
LIMITS ARE

These keywords identify the number of lines on a page of a report.

LINE
LINES

These keywords indicate the number of lines on a page of a report.

integer-1

Integer-1 must not exceed three significant digits in length. Also, integer-1 must be greater than or equal to integer-5.

integer-2

Integer-2 must be greater than or equal to one.

integer-3

Integer-3 must be greater than or equal to integer-2.

integer-4

Integer-4 must be greater than or equal to integer-3.

integer-5

Integer-5 must be greater than or equal to integer-4.

HEADING
FIRST DETAIL
LAST DETAIL
FOOTING

These phrases can be written in any order.

Rules for Placing Report Groups

The following rules indicate the vertical subdivision of the page in which each type of report group can appear when you specify the PAGE clause:

- If you define a REPORT HEADING report group that is presented on a page by itself, then you must define the REPORT HEADING report group so that it is presented in the vertical subdivision of the page that extends from the line number specified by integer-2 to the line number specified by integer-1, inclusive.

  If you define a REPORT HEADING report group that is **not** presented on a page by itself, then you must define the REPORT HEADING report group so that it is presented in the vertical subdivision of the page that extends from the line number specified by integer-2 to the line number specified by integer-3 minus 1, inclusive.
- If you define a PAGE HEADING report group, then you must define the PAGE HEADING report group so that it is presented in the vertical subdivision of the page that extends from the line number specified by integer-2 to the line number specified by integer-3 minus 1, inclusive.
- If you define a CONTROL HEADING or DETAIL report group, then you must define the CONTROL HEADING or DETAIL report group so that they are presented in the vertical subdivision of the page that extends from the line number specified by integer-3 to the line number specified by integer-4, inclusive.
- If you define a CONTROL FOOTING report group, then you must define the CONTROL FOOTING report group so that it is presented in the vertical subdivision of the page that extends from the line number specified by integer-3 to the line number specified by integer-5, inclusive.
- If you define a PAGE FOOTING report group, then you must define the PAGE FOOTING report group so that it is presented in the vertical subdivision of the page that extends from the line number specified by integer-5 plus 1 to the line number specified by integer-1, inclusive.
- If you define a REPORT FOOTING report group on a page by itself, then you must define the REPORT FOOTING report group so that it is presented in the vertical subdivision of the page that extends from the line number specified by integer-2 to the line number specified by integer-1, inclusive.

  If you define a REPORT FOOTING report group that is **not** presented on a page by itself, then you must define the REPORT FOOTING report group so that it is presented in the vertical subdivision of the page that extends from the line number specified by integer-5 plus 1 to the line number specified by integer-1, inclusive.

You must describe all report groups so that they are presented on one page. A multiline report group is **never** split across page boundaries.

Rules for Setting the Vertical Format

You establish the vertical format of a report page using the following integer values specified in the PAGE clause:

- Integer-1 defines the size of a report page by specifying the number of lines available on each page.
- HEADING integer-2 defines the first line number on which a REPORT HEADING or PAGE HEADING report group is presented.
- FIRST DETAIL integer-3 defines the first line number on which a body group is presented. REPORT HEADING (without NEXT GROUP NEXT PAGE) and PAGE HEADING report groups cannot be presented on or beyond the line number specified by integer-3.
- LAST DETAIL integer-4 defines the last line number on which a CONTROL HEADING or DETAIL report groups is presented.
- FOOTING integer-5 defines the last line number on which a CONTROL FOOTING report group is presented. PAGE FOOTING and REPORT FOOTING report groups must follow the line number specified by integer-5.

If absolute line spacing is indicated for all report groups, you do not need to specify integer-2 through integer-5. If relative line spacing is indicated for individual detail report groups entries, you must define some or all of the limits (depending on the type of report groups within the report) for control of page formatting to be maintained.

Defaults

If you specify the PAGE clause, the following implicit values are assumed for any omitted phrases:

- If you omit the HEADING phrase, a value of one is assumed for integer-2.
- If you omit the FIRST DETAIL phrase, a value equal to integer-2 is given to integer-3.
- If you omit both the LAST DETAIL and the FOOTING phrases, the value of integer-1 is given to both integer-4 and integer-5.
- If you specify the FOOTING phrase and you omit the LAST DETAIL phrase, the value of integer-5 is given to integer-4.
- If you specify the LAST DETAIL phrase and you omit the FOOTING phrase, the value of integer-4 is given to integer-5.

If you omit the PAGE clause, the report consists of a single page of indefinite length.

Absolute line number or absolute NEXT GROUP spacing must be consistent with controls specified in the PAGE LIMIT clause.

Figure 14–1 illustrates page format control of report groups when you specify the PAGE LIMIT clause.

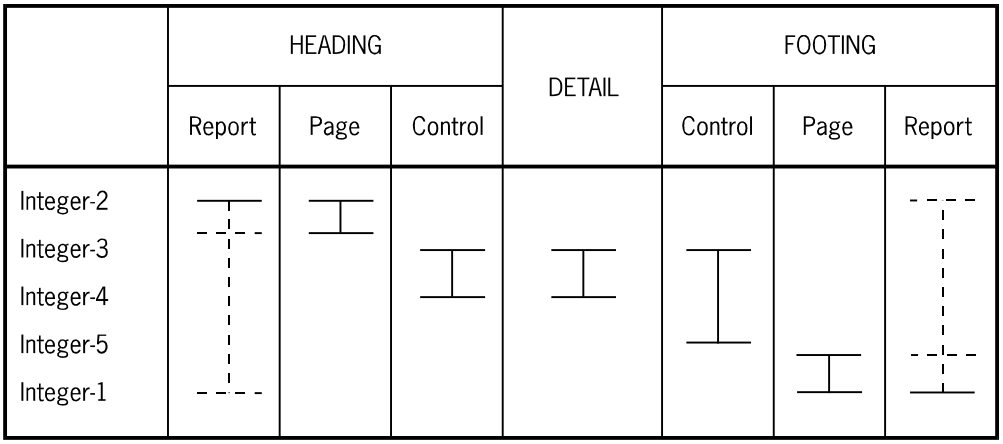


Figure 14–1. Page Format Control

Page regions established by the PAGE clause are depicted in Table 14–1.

Table 14–1. Page Regions Established by the PAGE Clause

Report Groups Presented in the Region	First Line Number of the Region	Last Line Number of the Region
REPORT HEADING described with NEXT GROUP NEXT PAGE REPORT FOOTING described with LINE integer-1 NEXT PAGE	integer-2	integer-1
REPORT HEADING not described with NEXT GROUP NEXT PAGE PAGE HEADING	integer-2	integer-3 minus 1
CONTROL HEADING DETAIL	integer-3	integer-4
CONTROL FOOTING	integer-3	integer-5
PAGE FOOTING REPORT FOOTING not described with LINE integer-1 NEXT PAGE	integer-5 plus 1	integer-1

Special Counters

The following special counters are available for each report described in the Report Section:

- LINE-COUNTER
- PAGE-COUNTER

LINE-COUNTER

LINE-COUNTER is a special register that is automatically created for each report where you specify the PAGE LIMIT clause. If more than one LINE-COUNTER register exists in a program, then you must qualify all references to LINE-COUNTER. In the Report Section, an unqualified reference to LINE-COUNTER is implicitly qualified by the name of the report in which the reference is made.

In the Report Section, a reference to LINE-COUNTER can appear only in a SOURCE clause. Outside the Report Section, LINE-COUNTER can be used in any context in which a data-name of integral value can appear. However, the content of LINE-COUNTER can be changed only by RWCS.

Execution of an INITIATE statement causes the LINE-COUNTER register for that report to reset to 0 (zero). LINE-COUNTER is also reset to 0 each time a page advance is executed for the associated report.

After a report group is printed, the LINE-COUNTER register contains the line number on which the last line of the report group was printed, unless the report group specifies the NEXT GROUP clause. In that case, LINE-COUNTER contains 0 if you specify NEXT PAGE or the line number.

For further information on line number positioning, refer to “LINE NUMBER Clause” and “NEXT GROUP Clause” in this section.

PAGE-COUNTER

PAGE-COUNTER is a special register that is automatically created for each report that you specify in the Report Section.

In the Report Section, a reference to PAGE-COUNTER can appear only in a SOURCE clause. Outside the Report section, PAGE-COUNTER can be used in any context in which a data-name of integral value can appear.

If more than one PAGE-COUNTER register exists in a program, you must qualify PAGE-COUNTER by a report-name whenever it is referenced in the Procedure Division. In the Report Section, an unqualified reference to PAGE-COUNTER is implicitly qualified by the name of the report in which the reference is made, whenever you reference the PAGE-COUNTER of a difference report. You must explicitly qualify PAGE-COUNTER by that report-name.

Execution of the INITIATE statement causes the PAGE-COUNTER of the referenced report to reset to 1.

You can alter PAGE-COUNTER by using Procedure Division statements. If you want a starting value other than 1, change the contents of PAGE-COUNTER following the INITIATE statement for that report.

Report-Group Description Entry

One or more report groups follow each report description (RD) entry. Each group describes one or more print lines related to a specific function in producing a report. A report group is described by a hierarchical data structure similar to record descriptions in the other sections of the Data Division.

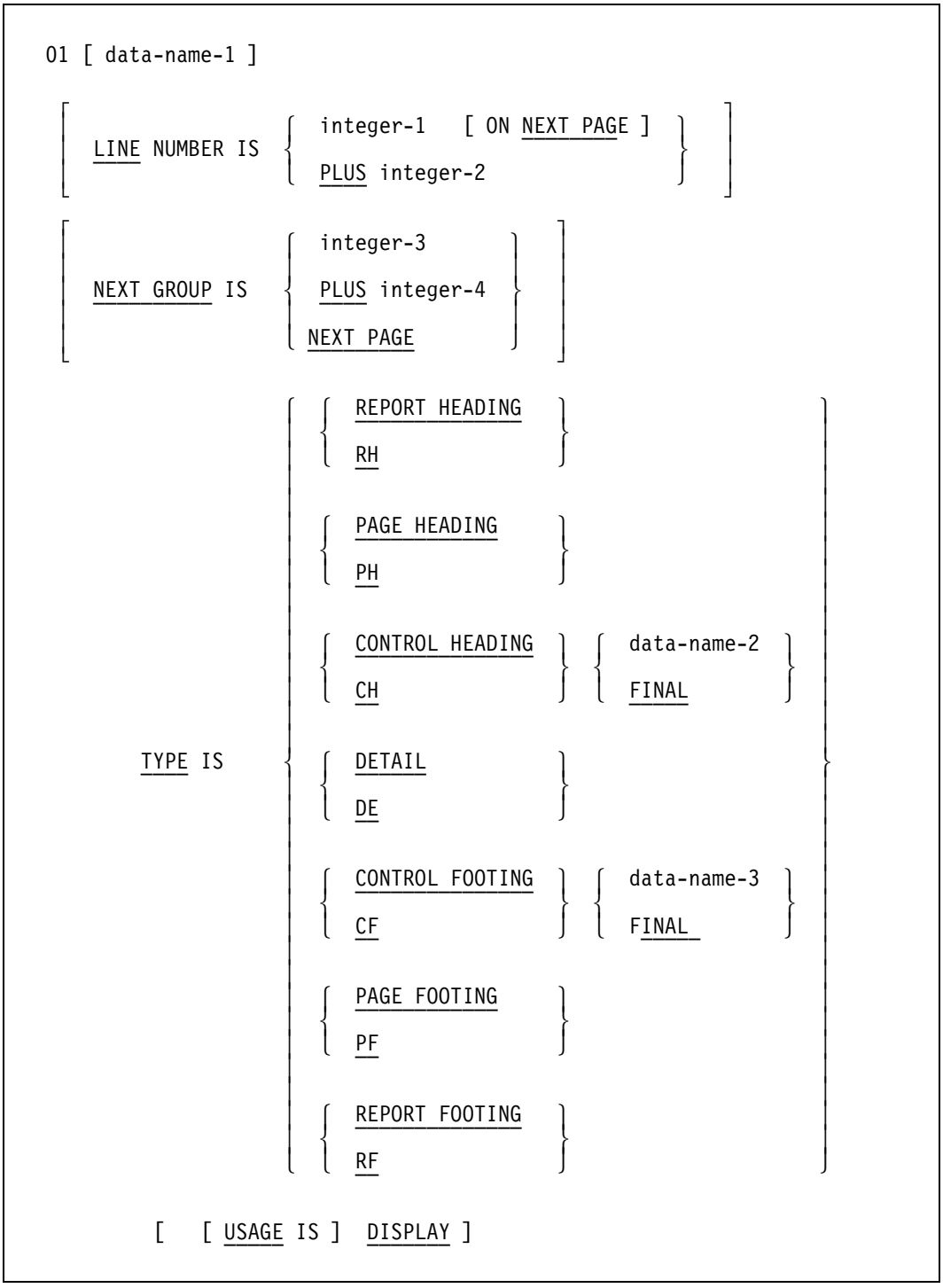
Format	Use
Format 1	This format specifies the vertical positioning and type of the report group.
Format 2	This format describes a single line of the report group.
Format 3	This format describes the single, printable items for a line, and a line that contains only one printable item.

The report-group description entry can appear only in the Report Section. Integers must be greater than 0.

The description of a report group can consist of one, two, or three hierarchical levels. The first entry of a report group must be a Format 1 entry. A Format 2 entry must be followed immediately by a Format 3 entry.

Report-Group Description Entry Format 1

This format describes the vertical positioning and type of the report group.



01

The level-number 01 identifies the first entry in a report group. A level-number is required as the first element in each data description entry. Data description entries subordinate to an RD entry must have level-numbers 01 through 49 only.

data-name-1

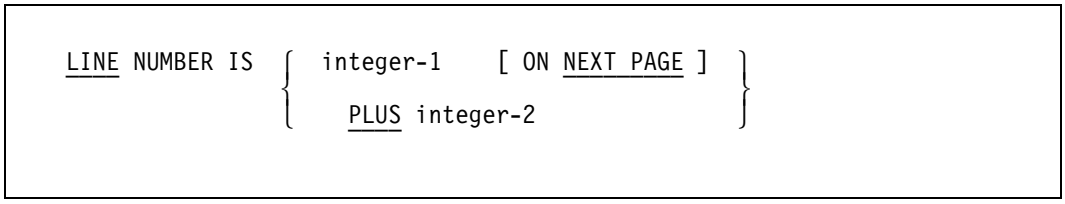
This is a user-defined name of a data file. The data-name, if used, must follow a level-number. However, you can write the clauses following a data-name in any sequence.

You can reference data-name-1 of a Format 1 entry only in the following cases:

- When a DETAIL report group is referenced by a GENERATE statement
- When a DETAIL report group is referenced by the UPON phrase of a SUM clause
- When a report group is referenced in a USE BEFORE REPORTING sentence
- When the name of a CONTROL FOOTING report group is used to qualify a reference to a sum-counter

LINE NUMBER Clause

The LINE NUMBER clause specifies vertical positioning information for its report group.



integer-1
integer-2

Integer-1 and integer-2 must not exceed three significant digits in length. Integer-2 can be zero.

You cannot specify integer-1 or integer-2 if any line of a report group is presented outside the vertical page subdivision designated for that report group type, as defined by the PAGE clause. Refer to "PAGE Clause" in this section for more information.

Integer-1 specifies an absolute line number. An absolute line number specifies the line number on which the print line is printed.

Integer-2 specifies a relative line number. If you specify a relative LINE NUMBER clause, the line number on which the print line is printed is determined by the sum of the line number on which the previous print line of the report group was printed and integer-2 of the relative LINE NUMBER clause.

ON NEXT PAGE

The ON NEXT PAGE phrase specifies that the report group is to be presented beginning on the indicated line number on a new page.

Details of LINE NUMBER Clause

You must specify the LINE NUMBER clause to establish each print line of a report group.

The vertical positioning specified by the LINE NUMBER clause occurs before the line established by that LINE NUMBER clause is printed.

In a given report-group description entry, the following rules apply:

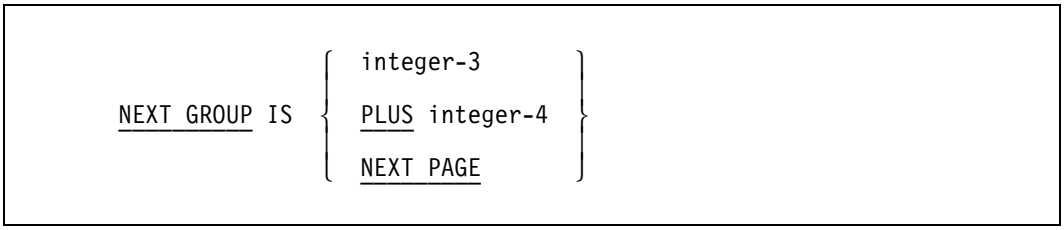
- An entry that contains a LINE NUMBER clause must not contain a subordinate entry that also contains a LINE NUMBER clause.
- All absolute LINE NUMBER clauses must precede all relative LINE NUMBER clauses.
- Successive absolute LINE NUMBER clauses must specify integers in ascending order. The integers do not need to be consecutive.
- If you omit the PAGE clause, you can specify only relative LINE NUMBER clauses in any report-group description entry in the report.
- An ON NEXT PAGE phrase can appear only once. If present, this phrase must be in the first LINE NUMBER clause. A LINE NUMBER clause with the ON NEXT PAGE phrase can appear only in the description of body groups and in a REPORT FOOTING report group.

Every entry that defines a printable item must either contain a LINE NUMBER clause or be subordinate to an entry that contains a LINE NUMBER clause.

The first LINE NUMBER clause specified within a PAGE FOOTING report group must be an absolute LINE NUMBER clause.

NEXT GROUP Clause

The NEXT GROUP clause specifies information for vertical positioning of a page following the presentation of the last line of a report group.



integer-3
integer-4

Integer-3 and integer-4 must not exceed three significant digits in length.

NEXT PAGE

You must not specify the NEXT PAGE phrase of the NEXT GROUP clause in a PAGE FOOTING report group.

Refer to the preceding discussion under "LINE NUMBER Clause" for more information on the NEXT PAGE phrase.

Details of NEXT GROUP Clause

A report-group entry must not contain a NEXT GROUP clause unless the description of the report group contains at least one LINE NUMBER clause.

If you omit the PAGE clause from the report description entry, you can specify only a relative NEXT GROUP clause in any report-group description entry in that report.

You must not specify the NEXT GROUP clause in a REPORT FOOTING report group or in a PAGE HEADING report group.

Any positioning of the page you specify using the NEXT GROUP clause takes place after the report group in which the clause appears is printed.

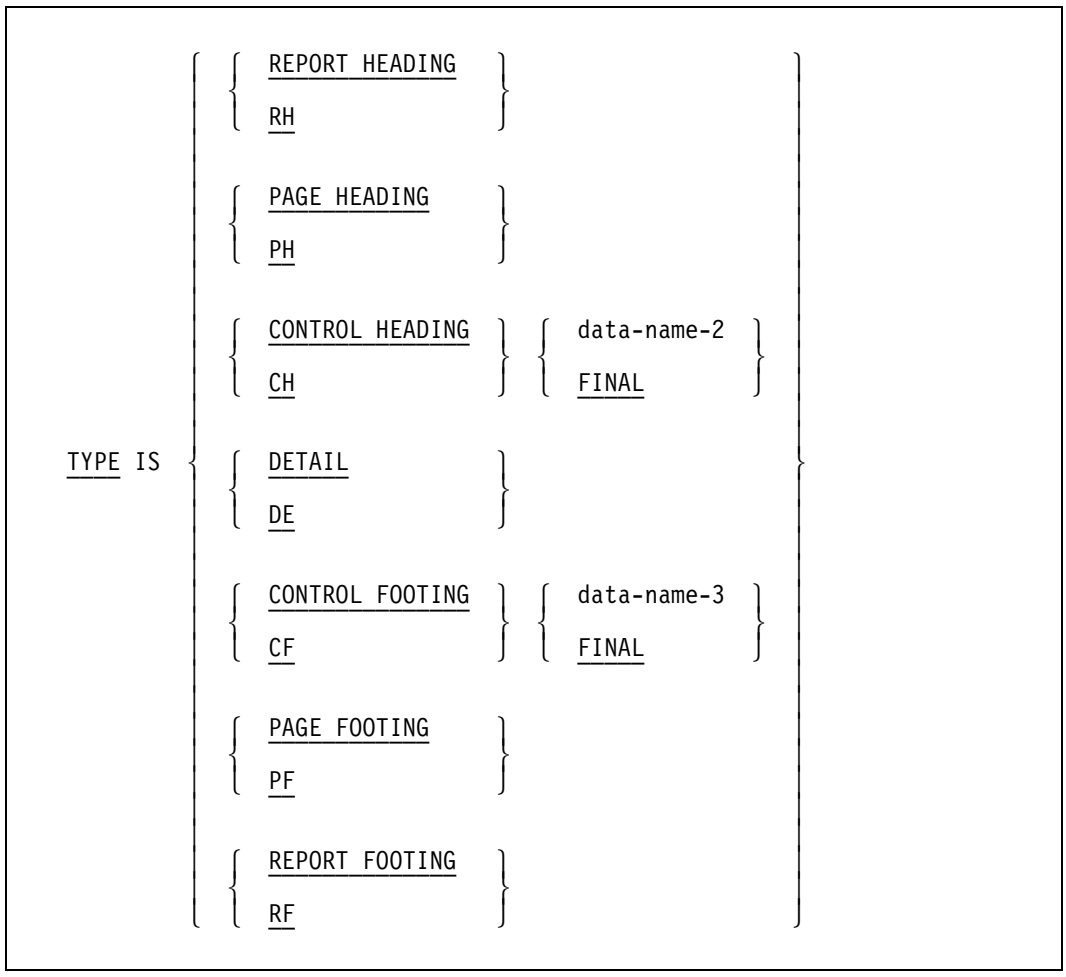
The vertical positioning information supplied by the NEXT GROUP clause is interpreted along with information from the TYPE and PAGE clauses and the value in LINE-COUNTER to determine a new value for LINE-COUNTER.

The NEXT GROUP clause is ignored when you specify it on a CONTROL FOOTING report group that is at a level other than the highest level at which a control break is detected.

The NEXT GROUP clause of a body group refers to the next body group to be printed, and therefore can affect the location at which the next body group is printed. The NEXT GROUP clause of a REPORT HEADING report group can affect the location at which the PAGE HEADING report group is printed. The NEXT GROUP clause of a PAGE FOOTING report group can affect the location at which the REPORT FOOTING report group is printed.

TYPE Clause

The TYPE clause specifies the particular type of report group described by the report description entry and indicates the time at which the report group is to be processed.



REPORT HEADING

The REPORT HEADING phrase specifies a report group that is processed only once per report as the first report group of that report. The REPORT HEADING report group is processed during execution of the first chronological GENERATE statement for that report. The REPORT HEADING phrase can appear no more than once in the description of a report.

The minimum abbreviation is RH.

PAGE HEADING

The PAGE HEADING phrase specifies a report group that is processed as the first report group on each page of that report except under the following conditions:

- A PAGE HEADING report group is not processed on a page that is to contain only a REPORT HEADING report group or only a REPORT FOOTING report group.
- A PAGE HEADING report group is processed as the second report group on a page when it is preceded by a REPORT HEADING report group that is not to be printed on a page by itself.

The PAGE HEADING report group can appear no more than once in the description of a report. Also, you can specify the PAGE HEADING only if you specify a PAGE clause in the corresponding report description entry.

The minimum abbreviation is PH.

CONTROL HEADING

The CONTROL HEADING phrase specifies a report group that is processed in one of two ways:

- At the beginning of a control group for a designated control data-name.
- During execution of the first chronological GENERATE statement for that report, in the case of FINAL.

If a control break is detected during execution of any GENERATE statement, then any CONTROL HEADING report groups associated with the highest control level of the break and lower levels are processed.

The CONTROL HEADING FINAL report group can appear no more than once in the description of a report.

The minimum abbreviation is CH.

data-name-1
data-name-2
FINAL

You must specify data-name-1, data-name-2, and FINAL, if present, in the CONTROL clause of the corresponding report-description entry. At most, you can specify one CONTROL HEADING report group and one CONTROL FOOTING report group for each data-name of FINAL in the CONTROL clause of the report-description entry. However, neither a CONTROL HEADING report group nor a CONTROL FOOTING report group is required for a data-name or FINAL specified in the CONTROL clause of the report-description entry.

DETAIL

DETAIL report groups are processed as a direct result of GENERATE statements. If a report group is other than TYPE DETAIL, processing is an automatic function.

When you specify a GENERATE report name statement in the Procedure Division, the corresponding report-description entry must include no more than one DETAIL report group. If you do not specify any GENERATE data-name statements for such a report, a DETAIL report group is not required.

The minimum abbreviation is DE.

CONTROL FOOTING

The CONTROL FOOTING phrase specifies a report group that is processed at the end of a control group for a designated control data-name. In the case of FINAL, the CONTROL FOOTING report group is processed only once for each report, as the last body group of that report. During execution of any GENERATE statement in which a control break is detected, any CONTROL FOOTING report group associated with the highest level of the control break or with minor levels is printed. All CONTROL FOOTING report groups are printed during execution of the TERMINATE statement if at least one GENERATE statement has been executed for the report.

CONTROL FOOTING FINAL can appear no more than once in the description of a report.

The minimum abbreviation is CF.

PAGE FOOTING

The PAGE FOOTING phrase specifies a report group that is processed as the last report group on each page, except under the following conditions:

- A PAGE FOOTING report group is not processed on a page that is to contain only a REPORT HEADING report group or a REPORT FOOTING report group.
- A PAGE FOOTING report group is processed as the second-to-last report group on a page when it is followed by a REPORT FOOTING report group that is not to be processed on a page by itself.

PAGE FOOTING can appear no more than once in the description of a report.

You can specify a PAGE FOOTING report group only if you specify a PAGE clause in the corresponding report-description entry.

The minimum abbreviation is PF.

REPORT FOOTING

The REPORT FOOTING phrase specifies a report group that is processed only once for each report as the last report group of that report. The REPORT FOOTING report group is processed during execution of a corresponding TERMINATE statement, if at least one GENERATE statement has been executed for the report.

REPORT FOOTING can appear no more than once in the description of a report.

The minimum abbreviation is RF.

Additional Information about the TYPE Clause

Additional information about the TYPE clause is grouped in the following topics:

- REPORT HEADING, PAGE HEADING, CONTROL HEADING, PAGE FOOTING, and REPORT FOOTING report groups processing
- CONTROL FOOTING report group processing
- DETAIL/No DETAIL report group processing
- Body groups processing
- Control break processing
- Other information

REPORT HEADING, PAGE HEADING, CONTROL HEADING, PAGE FOOTING, and REPORT FOOTING Report Group Processing

The following sequence of steps is executed when a REPORT HEADING, PAGE HEADING, CONTROL HEADING, PAGE FOOTING, or REPORT FOOTING report group is processed:

1. If a USE BEFORE REPORTING procedure is present that references the data-name of the report group, the USE procedure is executed.
2. If you have executed a SUPPRESS statement or the report group is not printable, no further processing is done for the report group.
3. If you have **not** executed a SUPPRESS statement and the report group is printable, the print lines are formatted and printed according to the rules for the given type of report group.

CONTROL FOOTING Report Group Processing

The following sequence of steps is executed when a CONTROL FOOTING report group is processed:

The GENERATE rules specify that when a control break occurs, the CONTROL FOOTING report groups, beginning at the minor level and proceeding upwards, are processed through the level at which the highest control break was detected. Even if no CONTROL FOOTING report group has been defined for a given control data-name, step 5 is executed if a RESET phrase within the report description specifies that control data-name.

1. Sum counters are *crossfooted*. That is, when the addend is a sum counter defined in the same CONTROL FOOTING report group, then the accumulation of that addend into the sum counter is termed crossfooting. Thus, all sum counters defined in this report group that are operands of SUM clauses in the same report group are added to the sum counters.
2. Sum counters are rolled forward. Thus, all sum counters defined in the report group that are operands of SUM clauses in higher-level CONTROL FOOTING report groups are added to the higher-level sum counters.
3. If a USE BEFORE REPORTING procedure references the data-name of the report group, the USE procedure is executed.
4. If you have executed a SUPPRESS statement or the report group is not printable, step 5 is executed next; otherwise, the print lines are formatted and the report group is printed according to the rules for CONTROL FOOTING report groups.
5. Sum counters that are to be reset when this level is processed in the control hierarchy are reset.

DETAIL or No DETAIL Report Group Processing

Step 1 of the following list is executed in response to a GENERATE report-name statement when the description of a report does not include DETAIL report groups. This step is performed as if the description of the report includes exactly one DETAIL report group and a GENERATE data-name statement is being executed.

Steps 1 through 5 are executed in response to a GENERATE report-name statement when the description of a report includes exactly one DETAIL report group. These steps are performed as if a GENERATE data-name statement is being executed.

1. Any subtotaling designated for the DETAIL report group is performed.
2. If a USE BEFORE REPORTING procedure refers to the data-name of the report group, the USE procedure is executed.
3. If you have executed a SUPPRESS statement or the report group is not printable, no further processing is done for the report group.
4. If the DETAIL report group is processed as a consequence of a GENERATE report-name statement, no further processing is done for the report group.

5. If a SUPPRESS statement is **not** executed, or the report group is **not** printable, or the DETAIL report group is **not** processed as a result of a GENERATE report-name statement, then no further processing of the report group occurs.

Body Group Processing

When a CONTROL HEADING, CONTROL FOOTING, or DETAIL report is processed, interruption of a previously described processing of that body group may be necessary after determining that the body group is to be printed. A page advance is executed (and PAGE FOOTING and PAGE HEADING report groups are processed) before the body group is actually printed.

Control Break Processing

During control break processing, the values of control data items used to detect a given control break are known as *prior values*. The following rules apply to prior values:

1. During control break processing of a CONTROL FOOTING report group, any references to control data items in a USE procedure or SOURCE clause associated with the CONTROL FOOTING report group are supplied with the prior values.
2. When a TERMINATE statement is executed, the prior control-data-item values are made available to SOURCE clause or USE procedure references in CONTROL FOOTING and REPORT FOOTING report groups as if a control break were detected in the highest control data-name.
3. All other data item references in report groups and USE procedures access the current values contained in the data items at the time the report group is processed.

Other Information

The description of a report must include at least one body group.

The DETAIL phrase specifies a report group that is processed when a corresponding GENERATE statement is executed.

In CONTROL FOOTING, PAGE HEADING, PAGE FOOTING, and REPORT FOOTING report groups, SOURCE clauses and USE statements must not reference any of the following: group data items containing a control data item, data items subordinate to a control data item, a redefinition or renaming of any part of a control data item.

In PAGE HEADING and PAGE FOOTING report groups, SOURCE clauses and USE statements must not reference control data-names.

Report-Group Description Entry Format 2

This format describes a single line of the report group. This entry must contain at least one of the optional clauses.

```
level-number [ data-name-1 ]

        [  LINE NUMBER IS   {  integer-1  [ ON NEXT PAGE ]  }  ]
        [                   {  PLUS integer-2               }  ]

        [  [ USAGE IS ] DISPLAY ]
```

level-number

The level-number is any integer between 02 and 48, inclusive.

data-name-1

This entry is optional. You can use this entry only to qualify a sum-counter reference, if this option is present.

A Format 2 entry must contain at least one of the optional clauses.

LINE NUMBER Clause
NEXT PAGE Clause

Refer to Report-Group Description Entry Format 1 for information on these clauses.

integer-1
integer-2

Refer to Report-Group Description Entry Format 1 for information on these integers.

USAGE Clause

A USAGE clause specifies the format of a data item in computer storage.

You can use the USAGE clause at either the elementary or 01-level. However, the USAGE of all report groups and their elementary items must be the same as the USAGE for the file on which the report is written. Refer to "USAGE Clause" in Section 4 for a more detailed description of this clause.

The USAGE IS DISPLAY clause indicates that the format of the data is a standard data format. If the USAGE is not specified for an elementary item, or for any group to which the item belongs, the usage is implicitly DISPLAY.

Report-Group Description Entry Format 3

This format describes the single, printable items for a line, and describes a line that contains only one printable item.

Format 3 entries must define elementary data items.

```
level number [ data-name-1]

    { PICTURE } IS character-string
    { PIC     }

    [  [ USAGE IS ] DISPLAY ]

    [  [ SIGN IS ] { LEADING  } SEPARATE CHARACTER  ]
                   { TRAILING }

    [  { JUSTIFIED } RIGHT  ]
       { JUST      }

    [  BLANK WHEN ZERO ]

    [  LINE NUMBER IS { integer-1 [ on NEXT PAGE ] } ]
                      { PLUS integer-2             }

    [  COLUMN NUMBER IS integer-3  ]

    {  SOURCE IS identifier-1                                  }
    {  VALUE IS literal-1                                      }
    {  { SUM  identifier-2   [ , identifier-3 ]...             }
    {      [ UPON data-name-2 [ , data-name-3 ] ... ] } ...    }
    {                                                          }
    {          [  RESET ON { data-name-4 } ]                   }
    {                      { FINAL       }                     }

    [ GROUP INDICATE ]
```

level-number

The level-number is any integer between 02 and 49, inclusive.

data-name-1

This entry is optional. If present, you can only use this entry to qualify a sum-counter reference.

PICTURE Clause

Refer to the PICTURE clause under "Data Description Entry Format 1" in Section 4 for a more detailed description of this clause.

USAGE Clause

In a Format 3 entry, the USAGE clause must define a printable item.

Refer to the USAGE clause under "Report-Group Description Entry Format 2" earlier in this section for information on this clause.

SIGN Clause

The SIGN clause specifies the position and the mode of representation of the operational sign when it is necessary to describe these properties explicitly.

Refer to the SIGN clause under "Data Description Entry Format 1" in Section 4 for detailed information.

JUSTIFIED Clause

The JUSTIFIED clause permits alternate (nonstandard) positioning of data in a receiving data item.

Refer to the JUSTIFIED clause under "Data Description Entry Format 1" in Section 4 for detailed information.

BLANK WHEN ZERO Clause

The BLANK WHEN ZERO clause fills an item with spaces when its value is zero.

Refer to the BLANK WHEN ZERO clause under "Data Description Entry Format 1" in Section 4 for detailed information.

LINE NUMBER Clause

Refer to "Report-Group Description Entry Format 1" earlier in this section for information on this clause.

COLUMN NUMBER Clause

The COLUMN NUMBER clause identifies a printable item and specifies the column-number position of the item on the print line. You can specify this clause only at the elementary level. When you use this clause, it must appear in, or be subordinate to, an entry that contains a LINE NUMBER clause.

integer-3

This must be greater than 0 (zero). Integer-3 specifies the leftmost character position of the printable item. In a given print line, you must define printable items in ascending column-number order, so that each character defined occupies a unique position.

Details of COLUMN NUMBER Clause

The COLUMN NUMBER clause indicates that the following are to be printed with the leftmost character position indicated by integer-3:

- The object of a SOURCE clause
- The object of a VALUE clause
- The sum counter defined by a SUM clause

The first or leftmost character of a print line is column number one.

The absence of a COLUMN NUMBER clause indicates that the entry is not printed.

Space characters are automatically provided for all positions of a print line that are not occupied by printable items.

SOURCE Clause

The SOURCE clause identifies the sending data item that is moved to an associated printable item defined in a report-group description entry.

identifier-1

This can be defined in any section of the Data Division. If identifier-1 is a Report Section item, it can only be PAGE-COUNTER, LINE-COUNTER, or a sum counter of the report in which the SOURCE clause appears.

Identifier-1 specifies the sending data item of the implicit MOVE statement that is executed to move identifier-1 to the printable item. You must define identifier-1 so that it conforms to the rules for sending items in the MOVE statement.

The print lines of a report group are formatted immediately before presentation of the report group. At that time, the implicit MOVE statements specified by SOURCE clauses are executed.

VALUE Clause

The VALUE clause defines the value of Report Section printable items. Refer to the VALUE clause under "Data Description Entry Format 1" in Section 4 for a more detailed discussion.

SUM Clause

The SUM clause establishes a sum counter and names the data items to be summed.

identifier-2
identifier-3

You must define identifier-2 and identifier-3 as numeric data items. When defined in the Report Section, identifier-2 and identifier-3 must be the names of sum counters.

UPON

If UPON is omitted, then you must define any identifiers in the associated SUM clause that are themselves sum counters, in the following two ways:

- In the same report group that contains this SUM clause
- In a report group at a lower level in the control hierarchy of this report

If you specify the UPON phrase, any identifiers in the associated SUM clause must not be sum counters.

data-name-2
data-name-3

These must be names of DETAIL report groups described in the same report as the CONTROL FOOTING report group in which the SUM clause appears. You can qualify data-name-2 and data-name-3 by a report name.

data-name-4

This must be one of the data-names specified in the CONTROL clause for this report. Data-name-4 must not be a lower-level control than the associated control for the report group in which the RESET phrase appears.

FINAL

If this is specified in the RESET phrase, FINAL must also appear in the CONTROL clause for this report.

Details of SUM Clause

A SUM clause can appear only in the description of a CONTROL FOOTING report group.

The highest permissible qualifier of a sum counter is the report-name.

The SUM clause establishes a sum counter. The sum counter is a numeric data item with an operational sign. At execution time, each of the values identifier-1, identifier-2, and so forth is added directly into the sum counter. This addition is performed under the rules of the ADD statement.

The size of the sum counter is equal to the number of receiving character positions specified by the PICTURE clause that accompanies the SUM clause in the description of the elementary item.

Only one sum counter exists for an elementary report entry, regardless of the number of SUM clauses you specify in the elementary report entry.

If the elementary report entry for a printable item contains a SUM clause, the sum counter serves as a source data item. The data contained in the sum counter is moved, according to the rules of the MOVE statement, to the printable item for printing.

If the data-name appears as the subject of an elementary report entry that contains a SUM clause, the data-name is the name of the sum counter, not the name of the printable item that the entry can also define.

Procedure Division statements can alter the contents of sum counters.

Addition of the identifiers into sum counters is performed during execution of GENERATE and TERMINATE statements. Each individual addend is added into the sum counter at a time that depends on the characteristics of the addend. The following three categories of sum-counter incrementing describe the characteristics of the addends and the timing of the addition process:

- Subtotaling

  When the addend is not a sum counter, the accumulation into a sum counter of such an addend is called *subtotaling*. If the SUM clause contains the UPON phrase, the addends are subtotaled when a GENERATE statement for the designated DETAIL report group is executed. If the SUM clause does not contain the UPON phrase, the addends that are not sum counters are subtotaled when any GENERATE data-name statement is executed for the report in which the SUM clause appears.

- Crossfooting

  When the addend is a sum counter defined in the same CONTROL FOOTING report group, the accumulation of that addend into the sum counter is termed *crossfooting*. Crossfooting occurs when a control break takes place and at the time the CONTROL FOOTING report group is processed. Crossfooting is performed according to the sequence in which sum counters are defined in the CONTROL FOOTING report group. Thus, all crossfooting into the first sum counter defined in the CONTROL FOOTING report group is completed first. Then, all crossfooting into the second sum counter defined in the CONTROL FOOTING report group is completed. This procedure is repeated until all crossfooting operations are completed.

- Rolling forward

  When the addend is a sum counter defined in a lower-level CONTROL FOOTING report group, the accumulation of that addend into the sum counter is termed *rolling forward*. A sum counter in a lower-level CONTROL FOOTING report group is rolled forward when a control break occurs and at the time that the lower-level CONTROL FOOTING report group is processed.

Subtotaling is accomplished only during execution of GENERATE statements after any control break is processed but before the DETAIL report group is processed. Crossfooting and rolling forward are accomplished during the processing of CONTROL FOOTING report groups.

The UPON phrase enables selective subtotaling for the DETAIL report groups named in the phrase.

If two or more identifiers specify the same addend, the addend is added into the sum counter as many times as the addend is referenced in the SUM clause. Two or more data-names can specify the same DETAIL report group. When a GENERATE data-name statement for such a DETAIL report group is given, the incrementing occurs as many times as data-names appear in the UPON phrase.

In the absence of an explicit RESET phrase, a sum counter is set to 0 at the time the CONTROL FOOTING report group with which the sum counter is defined is processed. If you specify an explicit RESET phrase, the sum counter is set to 0 at the time the designated level of the control hierarchy is processed. Sum counters are initially set to 0 during execution of the INITIATE statement for the report containing the sum counter.

GROUP INDICATE Clause

The GROUP INDICATE clause indicates that this elementary item is to be produced only on the first occurrence of the item after any control or page break.

The GROUP INDICATE clause can appear only in a DETAIL report group at the elementary-item level within an entry that defines a printable item.

If you specify a GROUP INDICATE clause, it causes the SOURCE or VALUE clauses to be ignored, and spaces to be provided, except in the following cases:

- On the first printing of the DETAIL report group in the report
- On the first printing of the DETAIL report group after a page advance
- On the first printing of the DETAIL report group after every control break

If the report-description entry specifies neither a PAGE clause nor a CONTROL clause, then a GROUP INDICATE printable item is printed the first time the DETAIL is printed after the INITIATE statement is executed. Thereafter, spaces are supplied for indicated items with SOURCE or VALUE clauses.

Table 14–2 shows the permissible clause combinations for a Format 3 entry. The table is read from left to right along the selected row.

In the following table, the *M* indicates that the clause is mandatory, the *P* indicates that the clause is permitted but not required, and a *blank* indicates the clause is **not** permitted.

Table 14–2. Permissible Clause Combinations in Format 3 Report Group Description Entries

PIC	Column	Source	Sum	Value	Just	Blank When Zero	Group Indicate	Usage	Line	Sign
M			M						P	P
M	M		M			P		P	P	P
M	P	M			P		P	P	P	P
M	P	M				P	P	P	P	P
M	M			M	P		P	P	P	P

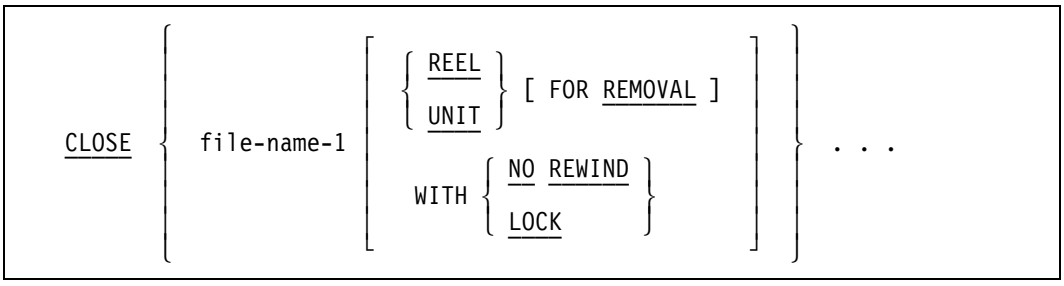
Procedure Division

You can use the statements described in the following pages with Report Writer. The statements are listed in alphabetical order.

CLOSE Statement

The CLOSE statement terminates the processing of a reel/unit of a file or a file with optional rewind and lock or removal where applicable. All reports associated with a report file that have been initiated must end with the execution of a TERMINATE statement before a CLOSE statement is executed for that report file.

The availability of the phrases within the CLOSE statement is dependent on the level of the Sequential I-O module supported by the implementation.



file-name-1

This is a user-defined word that specifies the name of the file to be closed.

The specified file must be in an open mode.

Files referenced in the CLOSE statement can have different organizations and access modes.

REEL
UNIT

These are equivalent.

The reel or unit is closed and rewound.

Treatment of sequential mass storage files is logically equivalent to the treatment of a file on tape or similar sequential medium.

Treatment of a file contained in a multiple file tape environment is logically equivalent to the treatment of a sequential single-reel/unit file, if the file is contained on one reel.

The REEL or UNIT phrase and the NO REWIND option cannot be specified together in a CLOSE statement.

FOR REMOVAL

This option is used for sequential single-reel/unit files and multi-reel/unit files. The reel/unit is closed, and the system waits for the next reel/unit.

NO REWIND

The file is closed, and the current reel/unit is left in its current position.

The NO REWIND option and the REEL or UNIT phrase cannot be specified together in a CLOSE statement.

LOCK

The logical file is marked as locked, so that it cannot be reopened during the execution of the program. If the file is a mass-storage file, it becomes a permanent file before it is made unavailable. If the file is assigned to tape, the physical unit is made not ready.

Refer to "CLOSE Statement" in Section 6 for more information.

GENERATE Statement

The GENERATE statement links the Procedure Division to the Report Writer (described in the Report Section of the Data Division) at process time.

```
GENERATE identifier
```

identifier

This represents a TYPE DETAIL report group or an RD entry.

Details

If identifier represents the name of a TYPE DETAIL report group, the GENERATE statement performs all automatic operations in the Report Writer and produces an output DETAIL report group on the output medium at process time. This type of reporting is called *detail reporting*.

If identifier represents the name of a report description (RD) entry, the GENERATE statement performs all automatic operations of the Report Writer, and updates the FOOTING report groups in a particular report description, without producing a DETAIL report group associated with the report. In this case, all sum counters associated with the report description are algebraically incremented. This type of reporting is called *summary reporting*. For summary reporting, no more than one TYPE DETAIL report group and at least one body group must be present, and the CONTROL clause must be specified for the report.

A GENERATE statement implicitly produces the following automatic operations, if defined, in both detail and summary reporting:

- Steps and tests LINE-COUNTER and/or PAGE-COUNTER to produce appropriate PAGE FOOTING and/or PAGE HEADING report groups
- Recognizes any specified control breaks to produce appropriate CONTROL FOOTING or CONTROL HEADING reporting groups
- Accumulates all specified identifiers into the sum counters, resets the sum counters on an associated control break, and performs an updating procedure between control-break levels for each set of sum counters
- Executes any specified routines defined by a USE statement before generation of the associated report groups

During execution of the first GENERATE statement, the following report groups associated with the report, if specified, are produced in the following order:

- REPORT HEADING report group
- PAGE HEADING report group
- All CONTROL HEADING report groups, in the following order:
  - Final
  - Major
  - Minor
- The DETAIL report group, if specified in the GENERATE statement

If a control break is recognized at the time of execution of a GENERATE statement (other than the first statement executed for a report), all CONTROL FOOTING report groups specified for the report are produced from the minor report group up to, and including, the report group specified for the identifier that caused the control break. Next, the CONTROL HEADING report groups specified for the report from the report group specified for the identifier that caused the control breakdown to the minor report group are produced in that order. The DETAIL report group specified in the GENERATE statement is then produced.

Data is moved to the data item in the report-group description entry of the Report Section. This data is edited under the control of the Report Writer according to the same rules for movement and editing described for the MOVE statement.

GENERATE statements for a report can be executed only after an INITIATE statement for the report has been executed and before a TERMINATE statement for the report has been executed.

INITIATE Statement

The INITIATE statement begins processing of a report.

```
INITIATE { report-name-1 } ...
```

report-name-1

You must define each report-name by a report-description entry in the Report Section of the Data Division.

Details

The INITIATE statement resets all data-name entries that contain SUM clauses associated with the report.

The PAGE-COUNTER register, if specified, is set to 1 during execution of the INITIATE statement. If a starting value other than 1 is desired for the associated PAGE-COUNTER, you can reset the counter after execution of the INITIATE statement is completed.

The LINE-COUNTER register, if specified, is set to 0 before or during execution of the INITIATE statement.

The INITIATE statement does not open the file with which the report is associated; however, the associated file must be open at the time the INITIATE statement is executed.

A second INITIATE statement for a particular report-name cannot be executed unless a TERMINATE statement has been executed for that report-name after execution of the first INITATE statement.

OPEN Statement

The OPEN statement initiates the processing of report files.

```
OPEN { OUTPUT { file-name-1 [ WITH NO REWIND ] } ... } ...
     { EXTEND { file-name-2 } ...                     }
```

OUTPUT

Upon successful execution of an OPEN statement with the OUTPUT phrase specified, a file is created. At that time, the associated file does not contain any data records.

When you open a sequential file with the OUTPUT option, all files on the associated multiple-file reel whose position numbers are less than the position number of the file being opened must already exist on the reel. In addition, there cannot be a file with a position number greater than the position number of the file being opened.

file-name-1
file-name-2

The file-name in a file description entry for a file must be equivalent to the file-name used when the file was created.

If you specify more than one file-name in an OPEN statement, the result is the same as if you had written separate OPEN statements for each file.

The minimum and maximum record sizes for a file are established at the time the file is created and cannot be subsequently changed.

EXTEND

This option enables you to write additional records to the end of a sequential file.

The EXTEND option can be used only with the following:

- Sequential single reel/unit files
- Files for which the LINAGE clause has not been specified

This option requires file-name-2 to be a previously created file (that is, already in the disk or pack directory, or on tape).

When you specify the EXTEND option, execution of the OPEN statement positions the file immediately after the last logical record for that file (that is, the last record written in the file). Subsequent WRITE statements that reference the file add records to the file as though the file had been opened with the OUTPUT phrase.

For an optional file that is unavailable, the successful execution of an OPEN statement with an EXTEND phrase creates the file as if the following statements had been executed:

```
OPEN OUTPUT file-name.
CLOSE file-name.
```

NO REWIND

This option can be used only with the following:

- Sequential single reel/unit files
- Sequential files that are wholly contained in a single reel of tape within a multiple-file tape environment

These phrases are ignored if they do not apply to the storage medium on which the file resides.

If the medium on which the file resides permits rewinding, the following rules apply:

- If you do not specify EXTEND or NO REWIND, execution of the OPEN statement causes the file to be positioned at its beginning.
- If you specify NO REWIND, execution of the OPEN statement does not cause the file to be repositioned; that is, the file must already be positioned at the beginning before the execution of the OPEN statement.

Details

Refer to "OPEN Statement" in Section 7 for additional information.

SUPPRESS Statement

The SUPPRESS statement causes the report writer control system (RCWS) to inhibit the presentation of a report group named in the USE procedure.

```
SUPPRESS PRINTING
```

Details

The SUPPRESS statement can only appear in a USE BEFORE REPORTING procedure.

The SUPPRESS statement must be executed each time the presentation of the report group is to be inhibited. When you execute the SUPPRESS statement, the RCWS is instructed to inhibit the processing of the following report group functions:

- The presentation of the print lines of the report group
- The processing of all LINE clauses in the report group
- The processing of the NEXT GROUP clause in the report group
- The adjustment of LINE-COUNTER

TERMINATE Statement

The TERMINATE statement terminates the processing of a report.

```
TERMINATE { report-name-1 } ...
```

report-name-1

You must define each report-name given in a TERMINATE statement by a report description (RD) entry in the Report Section of the Data Division.

Details

The TERMINATE statement produces all CONTROL FOOTING groups associated with this report as if a control break had just occurred at the highest level and completes the Report Writer functions for the named reports. The TERMINATE statement also produces the last PAGE FOOTING and the REPORT FOOTING report groups associated with this report.

If no GENERATE statements have been executed for a report during the interval between the execution of an INITATE statement and a TERMINATE statement for the same report, associated FOOTING groups are not produced.

Appropriate PAGE HEADING and/or PAGE FOOTING report groups are prepared in their respective order for the report description.

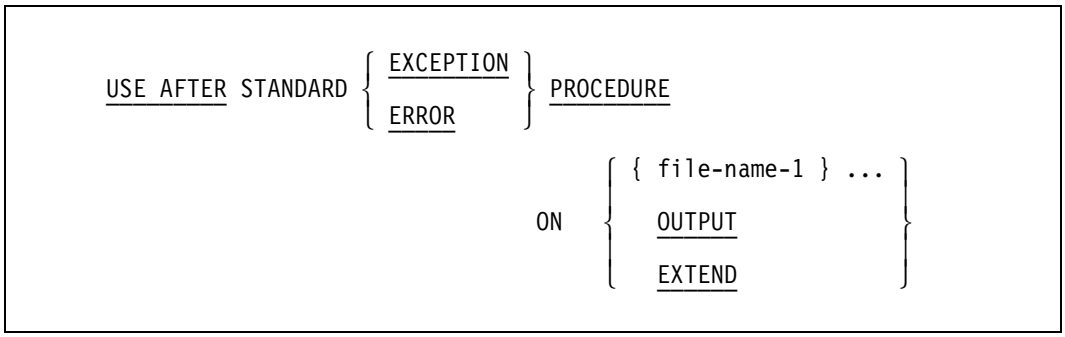
You cannot execute a second TERMINATE statement for a particular report unless a second INITIATE statement has been executed for a report-name. If a TERMINATE statement has been executed for a report, a GENERATE statement for that report must not be executed unless an intervening INITIATE statement for that report is executed.

The associated file must be open at the time the TERMINATE statement is executed. The TERMINATE statement does not close the file with which the report is associated. You must specify a CLOSE statement. The TERMINATE statement performs Report Writer functions for individually described reports analogous to the input/output functions that the CLOSE statement performs for individually described files.

SOURCE clauses used in the CONTROL FOOTING FINAL or REPORT FOOTING report groups refer to the values of the items at execution time of the TERMINATE statement.

USE AFTER STANDARD EXCEPTION PROCEDURE Statement

The USE AFTER STANDARD EXCEPTION PROCEDURE statement specifies procedures for input-output error handling that are in addition to the standard procedures provided by the input-output control system.



USE AFTER

The USE AFTER statement is never executed itself; it merely defines the conditions calling for the execution of the USE procedures.

A USE AFTER statement must immediately follow a section header in the declaratives section and must appear in a sentence by itself. The remainder of the section must consist of any number of procedural paragraphs that define the procedures to be used.

ERROR
EXCEPTION

The words ERROR and EXCEPTION are synonymous and can be used interchangeably.

file-name-1

The files implicitly or explicitly referenced in a USE AFTER statement need not all have the same organization or access.

The appearance of file-name-1 in a USE AFTER statement must not cause the simultaneous request for execution of more than one USE AFTER procedure. That is, when file-name-1 is specified explicitly, no other USE statement can apply to file-name-1.

OUTPUT
EXTEND

The OUTPUT and EXTEND phrases can each be specified only once in the declaratives portion of a given Procedure Division.

Details

Declarative procedures can be included in any COBOL source program whether or not the program contains, or is contained in, another program. Refer to Section 5 for information about declarative procedures and compiler-directing statements.

A declarative is invoked when any of the conditions described in the USE AFTER statement that prefaces the declarative occur while the program is being executed. The declarative is invoked only if it exists in the separately compiled program that contains the statement that caused the qualifying condition. If the declarative does not exist in the separately compiled program, the declarative is not executed.

A declarative procedure must not reference any nondeclarative procedures.

Procedure-names associated with a USE AFTER statement can be referenced in a different declarative section, or in a nondeclarative procedure only with a PERFORM statement.

The procedures associated with the USE AFTER statement are executed by the input-output control system after completing the standard error retry routine if the execution of the input-output routine was unsuccessful. However, an AT END phrase can take precedence.

The following rules concern the execution of the procedures associated with the USE AFTER statement:

- If you specify file-name-1, the associated procedure is executed when the condition described in the USE AFTER statement occurs to the file.
- If you specify OUTPUT, the associated procedure is executed for any file that is open in the output mode, or that is in the process of being opened in the output mode when the condition described in the USE AFTER statement occurs. Those files referenced by file-name-1 in another USE AFTER statement that specifies the same condition are not executed.
- If you specify EXTEND, the associated procedure is executed for any sequential file that is open in the EXTEND mode, or that is in the process of being opened in the EXTEND mode when the condition described in the USE AFTER statement occurs. Those sequential files referenced by file-name-1 in another USE AFTER statement that specifies the same condition are not executed.

After execution of a USE procedure, control passes to the invoking routine in the input-output control system. If the I-O status value does not indicate a critical input-output error, the input-output control system returns control to the next executable statement that follows the input-output statement whose execution caused the exception. Refer to the discussion of the STATUS IS clause in Section 3 for information on I-O status values.

In a USE procedure, a statement cannot be executed if it would cause the execution of a USE procedure that had previously been invoked and had not yet returned the control to the invoking routine.

USE BEFORE REPORTING Statement

This statement specifies Procedure Division statements that are executed just before a report group named in the Report Section of the Data Division is presented.

```
USE BEFORE REPORTING identifier-1
```

USE BEFORE REPORTING

The USE BEFORE REPORTING statement is never executed itself; it merely defines the conditions calling for the execution of the USE procedures.

This statement must immediately follow a section header in the declaratives portion of the Procedure Division and must appear in a sentence by itself. The remainder of the section must consist of any number of procedural paragraphs that define the procedures to be used.

A USE BEFORE REPORTING procedure must not alter the value of any control data item.

The GENERATE, INITIATE, or TERMINATE statements must not appear in a paragraph within a USE BEFORE REPORTING procedure. A PERFORM statement in a USE BEFORE REPORTING procedure must not have GENERATE, INITIATE, or TERMINATE statements in its range.

identifier-1

Identifier-1 must be a reference to a report group. It must not appear in more than one USE BEFORE REPORTING statement.

Details

Declarative procedures can be included in any source program whether or not the program contains, or is contained in, another program. A declarative is invoked just before the named report group is produced during the execution of the program. The report group is named by identifier-1 in the USE BEFORE REPORTING statement that prefaces the declaratives.

A declarative procedure must not make a reference to any nondeclarative procedures.

Procedure-names associated with a USE BEFORE REPORTING statement can be referenced in a different declarative section, or in a nondeclarative procedure only with a PERFORM statement.

In the USE BEFORE REPORTING statement, the designated procedures are executed by the report writer control system (RWCS) just before the named report group is produced.

In a USE procedure, a statement cannot be executed if it would cause the execution of a USE procedure that had previously been invoked and had not yet returned the control to the invoking routine.

Report Writer Examples

The data file input of the following program (Example 1) uses the Report Writer program to produce the report shown in Example 2. The output file is shown in example 3.

Example 1

```
IDENTIFICATION DIVISION.
PROGRAM-ID. FED-SCHOOL-SYSTEM.
AUTHOR. BERKOWITZ.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT PENNI ASSIGN TO SORT DISK.
    SELECT INFILE ASSIGN TO DISK.
    SELECT REPORTFILE ASSIGN TO PRINTER.
DATA DIVISION.
FILE SECTION.
FD  INFILE BLOCK CONTAINS 30 RECORDS.
01  IN-REC          PICTURE X(84).
SD  PENNI.
01  FROMM.
    02  FILLER       PICTURE XX.
    02  STUDENT.
       03  NAME-L    PICTURE X(30).
       03  NAME-F    PICTURE X(10).
    02  FILLER       PICTURE XX.
    02  GRADE        PICTURE 99.
    02  FILLER       PICTURE XX.
    02  ROOM         PICTURE 999.
    02  FILLER       PICTURE 99.
    02  MONTHH       PICTURE 99.
    02  DAYY         PICTURE 99.
    02  YR           PICTURE 99.
    02  FILLER       PICTURE X(2).
    02  TAL          PICTURE 9.
    02  FILLER       PICTURE X(22).
FD REPORTFILE    REPORT IS ABS-REPORT.
WORKING-STORAGE SECTION.
77  SAVED-MONTH     PICTURE 99 VALUE IS 0.
77  CONTINUED       PICTURE X(11) VALUE IS SPACE.
77  ABSS PIC X(8)  VALUE "ABSENCES".
77  CA PIC X(19)   VALUE "CUMULATIVE ABSENCES".
77  TAL-CTR  BINARY  PIC 9999.
77  MTHIX           PICTURE 99.
```

```
01  HEAD-1.
     02  FILLER  PIC X(22) VALUE SPACES.
     02 HEAD-LINE  PIC X(74) VALUE " MONTH        DAY
-         "GRADE        ROOM                  NAME  ".
     02  FILLER  PIC X(36) VALUE SPACES.
 01  MONTH-TABLE.
     02  MONTH-1.
         03  FILLER  PICTURE A(9) VALUE IS "JANUARY  ".
         03  FILLER  PICTURE A(9) VALUE IS "FEBRUARY ".
         03  FILLER  PICTURE A(9) VALUE IS "MARCH    ".
         03  FILLER  PICTURE A(9) VALUE IS "APRIL    ".
         03  FILLER  PICTURE A(9) VALUE IS "MAY      ".
         03  FILLER  PICTURE A(9) VALUE IS "JUNE     ".
         03  FILLER  PICTURE A(9) VALUE IS "JULY     ".
         03  FILLER  PICTURE A(9) VALUE IS "AUGUST   ".
         03  FILLER  PICTURE A(9) VALUE IS "SEPTEMBER".
         03  FILLER  PICTURE A(9) VALUE IS "OCTOBER  ".
         03  FILLER  PICTURE A(9) VALUE IS "NOVEMBER ".
         03  FILLER  PICTURE A(9) VALUE IS "DECEMBER ".
         03  FILLER  PICTURE A(9) VALUE SPACES.
     02 MONTH-2 REDEFINES MONTH-1.
         03  MONTHNAME  PICTURE A(9) OCCURS 13 TIMES.
 REPORT SECTION.
 RD  ABS-REPORT   CONTROLS ARE FINAL, MONTHH, DAYY, GRADE
     PAGE LIMIT IS 56 LINES      HEADING 2
     FIRST DETAIL 10   LAST DETAIL 45     FOOTING 55.
*
* THE FOLLOWING LINES PRODUCE THE REPORT HEADING.
* SEE <--1 IN SAMPLE REPORT WRITER REPORT.
*
 01 TYPE IS REPORT HEADING.
    02 LINE NUMBER IS 2   COLUMN 57       PIC X(17)
      VALUE "FED SCHOOL SYSTEM".
*
* THE FOLLOWING LINES PRODUCE THE PAGE HEADING.
* SEE <--2 IN SAMPLE REPORT WRITER REPORT.
*
 01  PAGE-HEAD   TYPE IS PAGE HEADING.
     02  LINE NUMBER IS 3  COLUMN 52       PIC X(26)
        VALUE "STUDENT ABSENTEEISM REPORT".
     02  LINE NUMBER IS 6.
       03  COLUMN IS 56    PIC X(9)
           SOURCE IS MONTHNAME OF MONTH-2(MONTHH).
       03  COLUMN IS 66    PIC X(8)       SOURCE IS ABSS.
       03  COLUMN IS 76    PIC X(11)      SOURCE IS CONTINUED.
     02  LINE IS 8.
       03  COLUMN IS 1     PIC X(132)     SOURCE HEAD-1.
*
```

```
* THE FOLLOWING LINES PRODUCE THE DETAIL LINES.
* SEE <--3 IN SAMPLE REPORT WRITER REPORT.
*
 01  DETAIL-LINE TYPE IS DETAIL LINE NUMBER IS PLUS 1.
     02  COLUMN IS 24 GROUP INDICATE PIC X(9)
         SOURCE IS MONTHNAME OF MONTH-2(MONTHH).
     02  COLUMN IS 41 GROUP INDICATE PICTURE IS 99
         SOURCE IS DAYY.
     02  COLUMN IS 54 GROUP INDICATE PIC  99 SOURCE IS GRADE.
     02  COLUMN IS 67 PIC 999 SOURCE IS ROOM.
     02  COLUMN IS 80 PIC X(20) SOURCE IS  NAME-L.
     02  COLUMN IS 101 PIC X(10) SOURCE IS NAME-F.
*
* THE FOLLOWING LINES PRODUCE THE CONTROL FOOTING GRADE.
* SEE <--4 IN SAMPLE REPORT WRITER REPORT.
*
 01  TYPE IS CONTROL FOOTING GRADE.
     02  LINE NUMBER IS PLUS 2.
       03  COLUMN 1 PIC X(132) VALUE SPACE.
*
* THE FOLLOWING LINES PRODUCE THE CONTROL FOOTING DAYY.
* SEE <--5 IN SAMPLE REPORT WRITER REPORT.
*
 01  TESTER   TYPE IS CONTROL FOOTING DAYY.
     02  LINE NUMBER IS PLUS 2.
       03  COLUMN 2 PIC X(12)          VALUE "ABSENCES FOR".
       03  COLUMN 24  PICTURE Z9 SOURCE SAVED-MONTH.
       03  COLUMN 26  PICTURE X VALUE "-".
       03  COLUMN 27  PICTURE 99 SOURCE DAYY.
       03  NO-ABS          COLUMN 49      PIC 999   SUM TAL.
       03  COLUMN 65 PIC X(19) SOURCE CA.
       03  COLUMN 85  PIC 999 SUM TAL RESET ON FINAL.
     02 LINE PLUS 1 COLUMN 24   PIC X(84)      VALUE ALL "*".
* THE FOLLOWING LINES PRODUCE THE CONTROL FOOTING MONTHH.
* SEE <--6 IN SAMPLE REPORT WRITER REPORT.
*
 01  TYPE CONTROL FOOTING MONTHH
     LINE PLUS 2 NEXT GROUP NEXT PAGE.
     02 COLUMN 42 PIC X(28) VALUE "TOTAL NUMBER OF ABSENCES FOR".
     02 COLUMN IS 72 PIC X(9)
        SOURCE MONTHNAME OF MONTH-2(SAVED-MONTH).
     02  COLUMN 83 PIC XXX VALUE "WAS".
     02  TOT      COLUMN 87 PIC 999        SUM NO-ABS.
*
* THE FOLLOWING LINES PRODUCE THE PAGE FOOTING.
* SEE <--7 IN SAMPLE REPORT WRITER REPORT.
*
 01  TYPE PAGE FOOTING LINE PLUS 1.
     02  COLUMN 59   PICTURE X(12) VALUE "REPORT-PAGE-".
     02  COLUMN 71   PICTURE 99 SOURCE PAGE-COUNTER.
*
```

```
* THE FOLLOWING LINES PRODUCE THE REPORT FOOTING.
* SEE <--8 IN SAMPLE REPORT WRITER REPORT.
*
 01   TYPE REPORT FOOTING.
     02  LINE PLUS 1 COLUMN 59 PICTURE A(13)
         VALUE "END OF REPORT".
 PROCEDURE DIVISION.
 DECLARATIVES.
 PAGE-HEAD-RTN SECTION.
   USE BEFORE REPORTING PAGE-HEAD.
 TEST-CONT.
     IF MONTHH = SAVED-MONTH MOVE "(CONTINUED)" TO CONTINUED
        ELSE MOVE SPACES TO CONTINUED
           MOVE MONTHH TO SAVED-MONTH.
 END DECLARATIVES.
 SORTING SECTION.
 SORTER.
     SORT PENNI
         ON ASCENDING KEY
         MONTHH, DAYY, GRADE, ROOM, STUDENT
         USING INFILE
         OUTPUT PROCEDURE IS REPORTER.
     DISPLAY MONTHH.
     MOVE MONTHH TO MTHIX.
 END-OF-THE-SORT.      STOP RUN.
 REPORTER SECTION.
 INITIATE-REPORT.
     OPEN OUTPUT REPORTFILE.
     INITIATE ABS-REPORT.
 UNWIND-THE-SORT.
     RETURN PENNI RECORD AT END
         TERMINATE ABS-REPORT STOP RUN.
     GENERATE DETAIL-LINE GO TO UNWIND-THE-SORT.
     STOP RUN.
```

Example 2: Input File

The input data file called *INFILE* results in the output shown in example 3.

```
CODDINGTON              KIMBERLY         03   125   091288   1
MILLSTEIN               SANDRA           03   121   091288   1
BURKLAND                JOSEPH           03   121   091288   1
MCCOY                   JUDY             01   142   091088   1
LUBASCH                 DANIEL           01   142   091088   1
JOFFEE                  JOHN             01   142   091088   1
EAGLE                   MIKE             05   153   090788   1
DANIELSON               FRED             05   153   090788   1
HUBERT                  THOMAS           03   115   090788   1
WONG                    SUSIE            03   111   090788   1
CODDINGTON              DARIN            02   103   090788   1
CARROLL                 JENNIFER         02   102   090788   1
HANSON                  KAREN            02   102   090788   1
AUSTIN                  EUGENE           02   101   090788   1
```

Example 3: Output File

```
                           FED SCHOOL SYSTEM                           <--1
                        STUDENT ABSENTEEISM REPORT                     <--2

                            SEPTEMBER ABSENCES                         <--2

MONTH              DAY     GRADE    ROOM           NAME                <--2

SEPTEMBER          07      02       101     AUSTIN       EUGENE        <--3
                                    102     CARROLL      JENNIFER      <--3
                                    102     HANSON       KAREN         <--3
                                    103     CODDINGTON   DARIN         <--3
                                                                       <--4
SEPTEMBER          07      03       111     WONG         SUSIE         <--3
                                    115     HUBERT       THOMAS        <--3
                                                                       <--4
SEPTEMBER          07      05       153     DANIELSON    FRED          <--3
                                    153     EAGLE        MIKE          <--3
                                                                       <--4

ABSENCES FOR          9-07      008     CUMULATIVE ABSENCES  008       <--5
*****************************************************************      <--5
SEPTEMBER          10      01       142     JOFFEE      JOHN           <--3
                                    142     LUBASCH     DANIEL         <--3
                                    142     MCCOY       JUDY           <--3
                                                                       <--4
```

```
ABSENCES FOR          9-10      003     CUMULATIVE ABSENCES  011       <--5
*************************************************************       <--5
SEPTEMBER          12      03       121     BURKLAND    JOSEPH        <--3
                                       121     MILLSTEIN   SANDRA      <--3
                                        125     CODDINGTON  KIMBERLY   <--3
                                                                       <--4

ABSENCES FOR          9-12      003     CUMULATIVE ABSENCES  014       <--5
*************************************************************       <--5

        TOTAL NUMBER OF ABSENCES FOR SEPTEMBER WAS 014                 <--6

                    REPORT-PAGE-01                                     <--7
                    END OF REPORT                                      <--8
```

Section 15
Compiler Operations

Note: *Although this section is not highlighted, all of the information is relative to Unisys extensions to COBOL ANSI-85.*

This section describes COBOL compiler operations. It contains three parts.

Input and Output Data Flow

This part of the section discusses the flow of data to and from the COBOL compiler. It also describes the input and output files of the compiler.

Compiling and Executing a COBOL Program

This part of the section summarizes how to:

- Create an object code file using the compiler
- Execute the object code file generated by the compiler
- Prevent abnormal terminations caused by stack overflows

Compiler Control Options

This part of the section describes the compiler control options that are available in COBOL85. It contains:

- General material about using any compiler option
- Reference material about each compiler option, in order of option name

Input and Output Data Flow

The source code of a program can be submitted to the COBOL compiler in the form of disk files or magnetic tape files. When more than one file is used for input, the compiler merges the files using the sequence numbers of the records in the files. The compiler also merges files according to any COPY statements included in the source code.

The primary output of the COBOL compiler is an object code file. The compiler can also generate several optional output files. The available optional files include:

- Updated symbolic file
- Error message file
- Printed listing containing the source records and compiler control records used by the compiler

Figure 15–1 shows the flow of data during a COBOL compile.

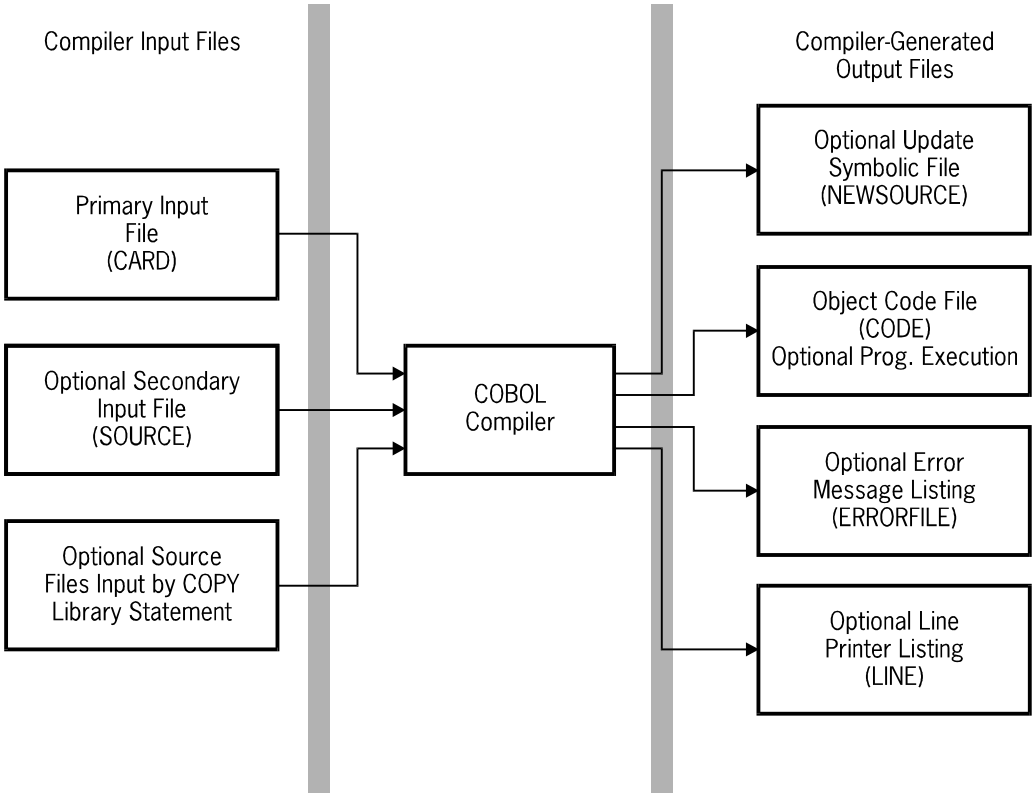


Figure 15–1. COBOL Compiler Input and Output Files

COBOL Compiler Files

The following tables list the logical input and output files used by the compiler. The values for the file attributes INTNAME, KIND, INTMODE, MAXRECSIZE, BLOCKSIZE, and FILETYPE are listed for each file. You can change the values of certain attributes by using a file equation when you initiate the compiler. For more information on file attributes, refer to the *File Attributes Programming Reference Manual*.

Each type of input and output file is discussed in the paragraphs following the tables.

Table 15–1. Compiler Input Files

INTNAME	Initiation	KIND	INTMODE	MAXRECSIZE and BLOCKSIZE	FILETYPE
CARD	WFL	READER	EBCDIC	Taken from physical file	8
	CANDE	DISK			
SOURCE	WFL and CANDE	DISK	EBCDIC	Taken from physical file	8
COPY files	WFL and CANDE	DISK	EBCDIC	Taken from physical file	8
INCLUDE files	WFL and CANDE	DISK	EBCDIC	Taken from physical file	8
INITIALCCI	WFL and CANDE	DISK	EBCDIC	Taken from physical file	8

Note: *COPY and INCLUDE are not internal file names. Rather, COPY files are files used with the COBOL COPY statement. INCLUDE files are files used with the INCLUDE compiler option.*

Table 15–2. Compiler Output Files

INTNAME	Initiation	KIND	INTMODE	MAXRECSIZE	BLOCKSIZE	FILETYPE
CODE	WFL/CANDE	DISK	HEX	30 words	270 words	—
NEWSOURCE	WFL/CANDE	DISK	EBCDIC	15 words	450 words	—
LINE	WFL /CANDE	PRINTER	EBCDIC	22 words	22 words	—
ERRORFILE	WFL	DISK	EBCDIC	12 words	12 words	—
	CANDE	REMOTE				
XREFFILE	WFL/CANDE	DISK	EBCDIC	510 words	510 words	0

Input Files

The compiler can receive input from a primary file (named CARD), a secondary file (named SOURCE), or from files in the COBOL library that are accessed through a COPY statement. Input received from more than one file is merged according to sequence numbers or according to instructions in the COPY statement.

The EXTMODE (character type) of the input files can be EBCDIC or ASCII. The MAXRECSIZE of the files must be large enough for a minimum of 72 characters. The attributes MAXRECSIZE and BLOCKSIZE need not be explicitly defined; the values for these attributes are taken from the physical file (FILETYPE=8).

CARD File

The file named CARD is the primary input file of the compiler. It must be present for each compilation. The default KIND attribute of the CARD file depends on how the compiler is initiated.

- Initiated through WFL: If no file equate statements are used, the CARD file is assumed to be a card reader file.
- Initiated through CANDE: If no file equate statements are used, the CARD file is assumed to be a disk file.

SOURCE File

The file named SOURCE is the secondary input file of the COBOL compiler. It is an optional file. If no file equate statements are used, the SOURCE file is assumed to be a disk file (regardless of how the compiler is initiated).

The SOURCE file is used only for input if the MERGE option is TRUE. When MERGE is TRUE, records from the SOURCE file are merged with those of the CARD file on the basis of sequence numbers. If a record from the CARD file and a record from the SOURCE file have the same sequence number, the CARD file record is used and the SOURCE file record is ignored. Refer to "MERGE Option" later in this section for more information.

COPY Library Files

The COBOL compiler can obtain additional source input from files in the COBOL library. The compiler incorporates records from library files in response to a COPY statement in the CARD or SOURCE file. Input from library files is added to input from the CARD and SOURCE files, as directed by the COPY statement.

For more information, refer to the "COPY Statement" in Section 6.

INCLUDE Files

The COBOL compiler can be directed to use alternate sources of source language input through the INCLUDE compiler option. When the compiler encounters an INCLUDE option, input from the file containing the INCLUDE option is suspended and all or a specified portion of the INCLUDE file is processed by the compiler. When the compiler completes processing of the INCLUDE file, input from the file containing the INCLUDE option resumes.

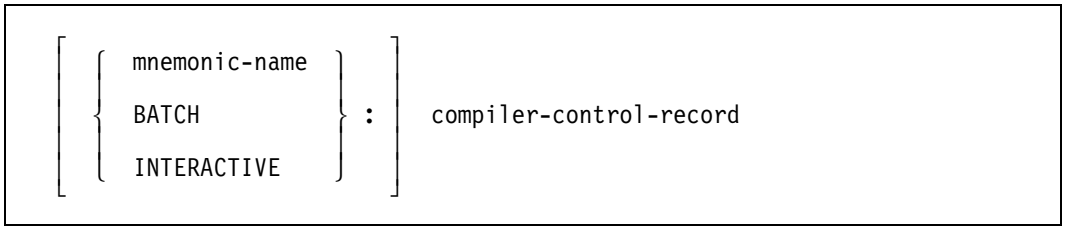
For more information, refer to "INCLUDE Option" in this section.

INITIALCCI File

The INITIALCCI file is an optional input file used to specify the initial settings of compiler options. Different initial settings can be invoked depending on whether the compilation originates from CANDE or from WFL.

Each user can use a customized INITIALCCI file, or a system-wide global INITIALCCI file can be used. The compiler searches for the INITIALCCI file using the standard usercode and family name conventions. If the file is not found, the compiler proceeds without it.

The INTNAME attribute of the INITIALCCI file is INITIALCCI. The file can be file-equated at compilation time. The FILEKIND attribute of the INITIALCCI file does not need to match the FILEKIND attribute of the program being compiled.



The following is a summary of the components of an INITIALCCI record:

mnemonic-name
BATCH
INTERACTIVE

These optional identifiers specify how the compiler-control-record is processed. If a mnemonic-name is specified, and it matches a valid file kind recognized by the system, then the compiler-control-record that follows is used only when a source file with that file kind is being compiled. If the mnemonic-name does not match a valid file kind, the entire record, including the compiler-control-record, is handled as a comment.

The keyword BATCH specifies that the compiler-control-record is used only when the compilation originates through WFL.

The keyword INTERACTIVE specifies that the compiler-control-record is used only when the compilation originates through CANDE.

compiler-control-record

This can be any valid compiler control record. If an optional identifier followed by a colon is specified (a mnemonic-name, or the keywords INTERACTIVE or BATCH), then the compiler-control-record is used only under the conditions described earlier. If an optional identifier does not precede the compiler-control-record, then the compiler-control-record is used for all compilations, regardless of the source language, or how the compilation originated. Compiler control records can begin with a currency sign ($), but it is not required. Note that the currency sign is optional only in the INITIALCCI file. For more information on compiler control records, refer to "Compiler Control Options" later in this section.

Before the INITIALCCI file is read, all compiler control options are set to their default values. The INCLUDE compiler control option is not permitted in the INITIALCCI file.

The following is an example of an INITIALCCI file:

```
COBOL85:      SET OPTIMIZE XREFFILES RESET BOUNDS
INTERACTIVE:  ERRORLIMIT = 20 OPTION (SET USEROPTION)
BATCH:        ERRORLIMIT = 50 SET ERRORLIST
              RESET LIST
              TARGET = LEVEL4
              PAGESIZE = 122      % USE LASER PRINTER
              SET NEW
```

This INITIALCCI file has the following effects:

- If the primary input source file is a COBOL85 source file, the OPTIMIZE and XREFFILES compiler options are set to TRUE, and the BOUNDS compiler option is set to FALSE.
- If the compilation originated through CANDE, the ERRORLIMIT compiler option is set to 20, and the user option USEROPTION is set to TRUE.
- If the compilation originated through WFL, the ERRORLIMIT compiler option is set to 50, and the ERRORLIST option is set to TRUE.
- For all compilations, the LIST compiler option is set to FALSE, the TARGET compiler option is set to LEVEL4, the PAGESIZE compiler option is set to 122, and the NEW compiler option is set to TRUE.

As shown by the PAGESIZE entry in the preceding example, the percent sign character can be used to delimit a comment. The compiler ignores all characters that follow a percent sign character.

Controlling Compiler Input

The following general capabilities are available for controlling the source language input processed by the compiler:

- Redirecting compiler input to an alternate source
- Ignoring defined blocks of source language input based on evaluation of a condition

The INCLUDE compiler option instructs the compiler to temporarily redirect compiler input to a specified alternate source. The compiler uses the alternate source as input until either the alternate source is exhausted or a specified range within the alternate source is exceeded. Such a range can be a single sequence number, or a sequence number range. Also, the specified range can be a string that identifies an area of the alternate source previously demarcated with the COPYBEGIN and COPYEND compiler options. For more information, refer to the INCLUDE, COPYBEGIN, COPYEND, SEARCH, and TITLE compiler options later in this section.

The IF, ELSE IF, ELSE and END compiler options are "conditional compilation phrases." These phrases define areas of the source language input that are compiled only if a given condition is true. For example, program debug code can be conditionally compiled based on the setting of a user option as shown in the following example:

```
$ OPTION (SET USERDEBUG)
           .
           .
           .
 $ IF USERDEBUG
 MOVE ERROR-CODE TO DEBUG-CODE.
     PERFORM WRITE-DEBUG-INFO.
 $ ELSE
 PERFORM NONFATAL-ERR-RECOVERY.
 $ END
```

The IF, ELSE, ELSE IF, and END compiler options are discussed in "Conditional Compilation Options" later in this section.

Output Files

The COBOL compiler creates from one to four files, depending on the options used during compilation. The four possible output files are:

- Object code file (named CODE)
- Updated symbolic file (named NEWSOURCE)
- Line printer compilation listing (named LINE)
- Error message listing (named ERRORFILE)

CODE File

The COBOL compiler generates the file named CODE, unless the compiler is directed to check for syntax only. The file CODE contains the executable object code of the program.

The status of this file after compilation depends on directions in the COMPILE statement and on the presence of syntax errors. This file can be:

- Stored permanently
- Executed and then discarded
- Discarded after compilation

Refer to the COMPILE statement in the *Work Flow Language (WFL) Programming Reference Manual* and the CANDE Operations Reference Manual.

NEWSOURCE File

The file named NEWSOURCE is produced only if the NEW option is TRUE. It is an updated source file that contains actual compiled source input from the CARD and SOURCE input files. If no file equate statements are used, the NEWSOURCE file is created on disk. For more information, refer to "NEW Option" later in this section.

LINE File

The file named LINE is always produced unless the LIST option is set to FALSE. The default value of the LIST option differs depending on how the compiler is initiated.

- Initiated through CANDE: the default value of LIST is FALSE. The LIST option must be set to TRUE before the LINE file can be produced.
- Initiated through WFL: the default value of LIST is TRUE, and the LINE file is produced.

If no file equate statements are used, the LINE file is written to a printer.

The content of the LINE file depends on the CODE option. The minimum amount of information in the LINE file is:

- Program source code used as input to the compiler
- Code segmentation information
- Error messages and error count (if syntax errors have occurred)

ERRORFILE File

The file named ERRORFILE (External name ERRORS) is produced only if the ERRORLIST option is TRUE. The default value of the ERRORLIST option and the default KIND attribute of the ERRORFILE file depend on how the compiler is initiated, as shown in the following table:

If the compiler is initiated . . .	Then the Default value of ERRORLIST is . . .	And the ERRORFILE is . . .
Through CANDE	TRUE	Produced. If a file equate statement is used, the error file is written to a printer; otherwise, the error file is written to the remote station that initiated the compiler.
Through WFL	FALSE	Not produced.

ERRORFILE has a copy of every source record that contains a syntax error, followed by all syntax errors that occurred for the record. If no syntax errors occur during compilation, ERRORFILE is not produced (regardless of the ERRORLIST option).

Using System Support Libraries

The following system support libraries are available for creating COBOL85 programs:

Library Name	Provides support for . . .
SLICESUPPORT	Intrinsics, error and warning message handling, and other miscellaneous compile and runtime support.
COBOL85SUPPORT	Certain types of Inter Program Communication (IPC) at run time.

Be sure that the release level of each support library is greater than or equal to the release level of the COBOL85 compiler used to create executing code files.

Compiling and Executing COBOL Programs

There are three ways to compile or execute a COBOL program. The method you use depends on decisions made at your installation. The three methods are:

- Through WFL
- Through CANDE
- From the ODT

The following paragraphs briefly summarize all three methods.

Compiling and Executing through WFL

Work Flow Language (WFL) is a job control language that lets you compile or execute programs. It also enables you to pass files between programs and perform other job control functions.

WFL is a powerful tool that can significantly streamline your data processing operations. Before using WFL, however, become familiar with the WFL language and how it is used at your installation. For detailed information on WFL, refer to the *Work Flow Language (WFL) Programming Reference Manual*.

WFL provides both the Compile and Run functions. Procedures for using WFL vary greatly from site to site, but the following examples provide a brief overview of how WFL functions with the COBOL85 compiler. Procedures at your installation might be significantly different.

Examples of WFL Job Files

The following are sample WFL statements that illustrate the syntax used to compile a program:

```
?BEGIN JOB EXAMPLE;
 COMPILE < program title> WITH COBOL85 LIBRARY;
 COMPILER DATA CARD
 <COBOL85 source program>
 ?END JOB;
```

WFL permits many different compilation options. For more information, refer to the *WFL Reference Manual*.

To run an existing compiled program in WFL, you must create a WFL job file. To create a WFL job file, simply create a WFL job file with a job header, followed by a RUN statement, and finally an END JOB statement.

```
?BEGIN JOB RUN/A/PROGRAM;
RUN SAMPLE/PROGRAM;
?END JOB;
```

Compiling and Executing through CANDE

CANDE provides an easy, flexible way to create, edit, and execute COBOL programs.

To create a COBOL file, enter the MAKE command (abbreviated as M) and designate the file type as COBOL. You can then enter the needed source code. To modify an existing COBOL file, use the GET command (abbreviated as G) with the correct file name to retrieve the file. You can then edit the existing code.

When you finish editing or entering code, you can compile the code with the COMPILE command (abbreviated as C). To compile on your system, enter *C*, followed by *WITH COBOL85*. To use specific compiler options, enter the complete COMPILE command. Syntax for the COMPILE command is provided in the CANDE Reference Manual.

When the code begins to compile, you receive a beginning of task (BOT) message and any syntax errors. When compilation is finished, an end of task (EOT) message is displayed.

If the compilation finishes without syntax errors, you can use the RUN command (abbreviated as R) to run the program. You can also enter *R* before compiling the program. This will compile the program and then execute it. (This option is normally used only when you are certain there are no syntax errors in the program.)

When you create a file with the file type "C85", you will use the COBOL85 compiler or default compiler. This way, you don't need to specify the compiler name along with the compile command.

To compile and run a COBOL program through CANDE, follow these steps:

Step	CANDE Command	Description
1	MAKE <file name> C85 —or— GET <file name> G	CANDE recognizes C85 as a valid file kind for the MAKE command. This is the command to retrieve an existing file.
2	COMPILE C —or— C WITH COBOL85	This compiles the current workfile. This explicitly compiles a file with the COBOL85, regardless of the file kind of the file.
3	RUN R	This executes the current workobject (object code file produced by compiling the current workfile).

Compiling and Executing from the ODT

Compiling and executing a COBOL program from the ODT is different from working in CANDE or through a user-created WFL deck. When you compile or run from the ODT, the MCP converts the commands entered through the ODT into the appropriate WFL commands using the WFL Formatter.

To compile from the ODT, identify the source file name and the compiler name in the COMPILE command, as in the following examples:

Unsecured ODT

```
COMPILE <program title> WITH COBOL85 LIBRARY;
FILE CARD (TITLE = <source file name>);
```

Secured ODT

```
USER = <user code> /<password>;
COMPILE <program title> WITH COBOL85 LIBRARY;
FILE CARD (TITLE = <source file name>, KIND = DISK);
```

In the preceding examples, the WITH COBOL85 LIBRARY option directs the system to use the COBOL85 compiler and to save the resultant code file. The FILE CARD syntax element is a file equate statement that identifies to the compiler the name of the source file and the file's location.

To execute an existing program from the ODT, enter the program name as part of the RUN command. If the program does not explicitly define its files, use a file equate statement to associate the files in the program with the correct existing files. The following are examples of the appropriate syntax to execute a program from the ODT:

Unsecured ODT

```
RUN <program title>
```

Secured ODT

```
USER = <user code> /<password>; RUN <program title>
```

Displaying the Compiling Progress

A user can view the compiling progress by entering the ?HI command for the compiler mix number at any time during compilation. The compiler responds with the same information provided by the ?CS command, augmented with the program name specified in the PROGRAM-ID paragraph. When compiling a multi-program symbol file, such as BINDSTREAM, the ?HI command is useful in determining which program is currently being compiled. Refer to "BINDSTREAM Option" later in this section.

Preventing Stack Overflows

The size of a process stack is controlled by the STACKLIMIT task attribute. When a process cannot proceed further without exceeding the limit established by this attribute, the system discontinues the process and returns the error message *STACK OVERFLOW*.

The STACKLIMIT task attribute defaults to a value of 6000 words. It is unlikely that any process stack will reach this size if it is running as intended. The *STACK OVERFLOW* error usually indicates that a process has entered into an infinite loop of procedure calls. Because each procedure call adds an activation record to the process stack, the process stack quickly exceeds the STACKLIMIT value.

If a process receives a *STACK OVERFLOW* error, and you determine that the process was running as intended, then you can remedy the problem by assigning a higher value to the STACKLIMIT task attribute before initiating the process. The highest value STACKLIMIT can be set to is approximately 64000 words.

Because a process stack is built exclusively in save memory, the save memory restrictions also effectively limit the size of the process stack. For more information, refer to the *Task Management Programming Reference Manual*.

Types of Compiler Control Options

Compiler control options are divided into the following types:

- Boolean
- Boolean Title
- Boolean Class
- Enumerated
- Immediate
- String
- User-defined
- Value

Boolean Compiler Options

A Boolean option is either enabled (set to TRUE) or disabled (set to FALSE). When enabled, the compiler applies the option to all subsequent processing until the option is disabled. The following is a list of the available Boolean options. Where a synonym exists for an option, it is shown in parentheses.

ANSI	ANSICLASS	ASCII
AUTOINSERT	BINARYCOMP	BINARYEXTENDED
BINDINFO	BINDSTREAM	BOUNDS
CODE	COMMON	CONCURRENTEXECUTION
CORRECTOK	CORRECTSUPR	DELETE(VOIDT)
FS4XCONTINUE	FREE	INCLNEW
LIBRARYLOCK	LIBRARYPROG	LINEINFO
LIST	LISTDOLLAR	LISTINCL (INCLLIST)
LISTINITIALCCI	LISTOMITTED (LISTO)	LISTP
LIST1	LOCALTEMP	MAP (STACK)
MUSTLOCK	NEWSEQERR	OMIT
OPT1	OPT2	OPT3
OPT4	OPTION	OWN
SEPARATE	SEQUENCE (SEQ)	SHOWOBSOLETE
SHOWWARN	STRICTPICTURE	SUMMARY (TIME)
TEMPORARY	UDMTRACK	VOID
WARNFATAL	WARNSUPR	XREF

Boolean Title Compiler Options

A Boolean title option sets the value of a Boolean option and optionally associates a file name with the option. The following is a list of the available Boolean title options:

ERRORLIST (ERRLIST)
LIBRARY
MERGE
NEW
XREFFILES

Boolean Class Compiler Options

A Boolean class option is a logical grouping of related Boolean options into a single Boolean option. An action performed against the class option affects all of the subordinate options, while an action performed against one of the subordinate options affects only that subordinate option and none of the others that comprise the class. For example, the OPTIMIZE compiler option is a class option. The GAMBLE, GRAPH, LEVEL, TIMING, UNRAVEL, and VECTOR_OPS subordinate options comprise the OPTIMIZER. An action can be performed against the OPTIMIZE option altering the setting of all of the subordinate options. The following shows the OPTIMIZE option being set to FALSE.

```
$ RESET OPTIMIZE
```

Also, an action can be performed against the LEVEL subordinate option (for example) without affecting the other subordinate options (the GAMBLE, GRAPH, TIMING, UNRAVEL, and VECTOR_OPS subordinate options retain their setting). The following shows the LEVEL subordinate option of the OPTIMIZE option being set to TRUE.

```
$ SET OPTIMIZE (LEVEL)
```

An action can be performed against the class option, and a different action performed against subordinate options in the same compiler control record (CCR). The following shows the OPTIMIZE option being set to FALSE, but the LEVEL subordinate option being set to TRUE.

```
$ RESET OPTIMIZE (SET LEVEL)
```

The following is a list of the available Boolean class options:

ANSICLASS	FARHEAP
BOUNDS	OPTIMIZE
COMPATIBILITY	

Enumerated Compiler Options

An enumerated option is an option whose setting is limited to a predetermined set of symbolic values. A symbolic value is a keyword that represents value meaningful to the compiler. For example, the STRINGS compiler option is an enumerated option. The set of values to which the STRINGS compiler option can be set is limited to the symbolic values EBCDIC and ASCII. The following is a list of the available enumerated options:

MEMORY_MODEL	STRINGS
SHARING	TARGET

Immediate Compiler Options

An immediate option is applied by the compiler when the option is encountered in source code. The function performed by an immediate option is independent of any subsequent processing by the compiler. Immediate options can have associated parameters. The following immediate options are available:

BINDER_MATCH	ELSE
END (END IF)	ELSE IF
IF	PAGE

String Compiler Options

A string option is an option to which a string is associated. The string can be delimited by either single or double quotation marks, so long as they are used consistently. A string delimited at the beginning by a single quotation mark must be delimited at the end by a single quotation mark. An example of a string option is the FOOTING compiler option used to specify a string to be placed at the bottom of each page of the listing. The following is a list of the available string options:

COPYBEGIN
COPYEND
FOOTING
LI_SUFFIX
MODULEFAMILY
MODULEFILE
NEWID
SEARCH
TITLE

User-Defined Compiler Options

Users can define compiler control options in addition to the standard options that are listed under "Types of Compiler Control Options." A user-defined option is a Boolean option that can be manipulated through the SET, RESET, and POP options. To create a user-defined option, the user declares the option implicitly by using it in a DOLLAR ($) statement. The first 31 characters of a user-defined option must be unique.

Value Compiler Options

A value option stores a specific value that the compiler uses when applying the option. The following is a list of the available value options:

ERRORLIMIT (LIMIT)
FEDLEVEL
LEVEL
IPCMEMORY
PAGESIZE
PAGEWIDTH
Sequence Base
Sequence Increment
VERSION

The INCLUDE option is a specially handled option, and is described later in this section.

Syntax for Compiler Control Options

There are two ways to specify compiler control options:

- On compiler control records (CCRs)
- By using commands at compile time

These two methods are discussed in the following two sections.

Compiler Control Records

Compiler control records (CCRs) are source code records with a currency sign ($) in column 7 and an optional $ in column 8. You can enable, disable, or change the value of an option by inserting a CCR into the source code of the program. A CCR contains a compiler control option and its parameters (if any).

Occasionally, the value of an option may take effect prematurely because of the compiler's look-ahead processing of source records. When this occurs, the value of an option is applied to a source record that preceded it. To prevent this premature effect, use the semicolon (;) to inhibit the compiler's look-ahead processing. If a source line is followed by a CCR, and you want to isolate the source line from the semicolon between the source line and the CCR, you can place the semicolon at the end of the source line or insert it on a line between the source line of interest and the CCR.

Generally, CCRs can be inserted at any point in the source code of a program. Some specific CCRs must be included at the beginning of the source code, before the Identification Division of the program, or before the first syntactical item in a separately compiled program. (CCRs that must be included at the beginning are noted in the option descriptions on the following pages.)

The compiler retains the setting of a compiler option for the duration of the compilation, unless the setting of the option is explicitly changed. This means that if the source language input file contains sequential programs to be compiled separately, then the compiler retains the settings of compiler options from one program to the next. To prevent this from occurring, you must explicitly change the setting of compiler options between the separate programs (that is, between the END PROGRAM statement of one program and the first syntactical element of the next program).

Frequently, a CCR contains more than one option. All options on a CCR follow the currency sign ($). At least one space must follow each option. No option can continue past column 72 of a CCR.

The general syntax used for all compiler control records (CCRs) is shown in the following syntax diagrams and discussed on subsequent pages.

```
                 [  SET   ]
$ [ $ ]          |  RESET |  [ option-name ]   [ option-parameters ]     ...
                 [  POP   ]
                    or
 $ [ $ ]   [ SET ]   { Boolean-option-name } = { Boolean-expression }...
```

$

CCRs must have a currency sign ($) in column 7 and may have an optional $ in column 8. The $ determines how the CCR is affected by the NEW compiler control option.

The NEW option directs the compiler to create a new source file (named NEWSOURCE). NEWSOURCE contains all of the source code records used during compilation.

- If a CCR has a $ (currency sign) in column 7, but not in column 8, then the CCR is not included in the new source file created by the NEW option. For example, frequently a MERGE option is not included in the newly generated source code. A CCR with only one $ is called a temporary CCR.
- If a CCR has a $ in columns 7 and 8, the CCR is included in the source code created by NEW. This is called a permanent CCR.

A CCR with a $ and no following options has no effect except in the following special cases:

- When the MERGE option is TRUE
- When a record in the primary input file (named CARD) contains a blank CCR
- When a record in the secondary input file (named SOURCE) has the same sequence number as the blank CCR in the primary input file

In any of these cases, the record in the secondary input file is ignored.

SET

SET saves the current setting of each option of the CCR and sets each option to TRUE (enabled or ON). This option has a register that stores the last 47 settings of the option. The option can also be set to the value of an optional Boolean-expression.

RESET

RESET saves the current setting of each option of the CCR and sets each option to FALSE (disabled or OFF). This option has a register that stores the last 47 settings of the option.

POP

POP discards the current setting of each option of the CCR and sets each option to its previous setting. This option has a register that stores the last 47 settings of the option. If no previous setting exists, POP sets the option to FALSE.

option-name

This element represents the name of any valid COBOL85 compiler control option. The option can be a Boolean option, a value option, or an immediate option.

Boolean-option-name

This element represents the name of a Boolean compiler control option. A Boolean option can only be set to TRUE (enabled, ON), or FALSE (disabled, OFF). A list of the Boolean options is included earlier in this section.

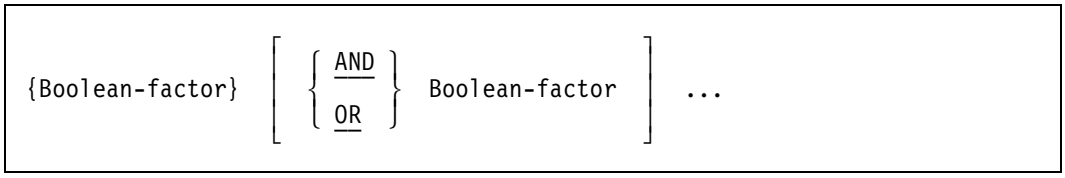
When the SET option action indicator is used, a Boolean option can be assigned the value of Boolean-expression.

Boolean-expression

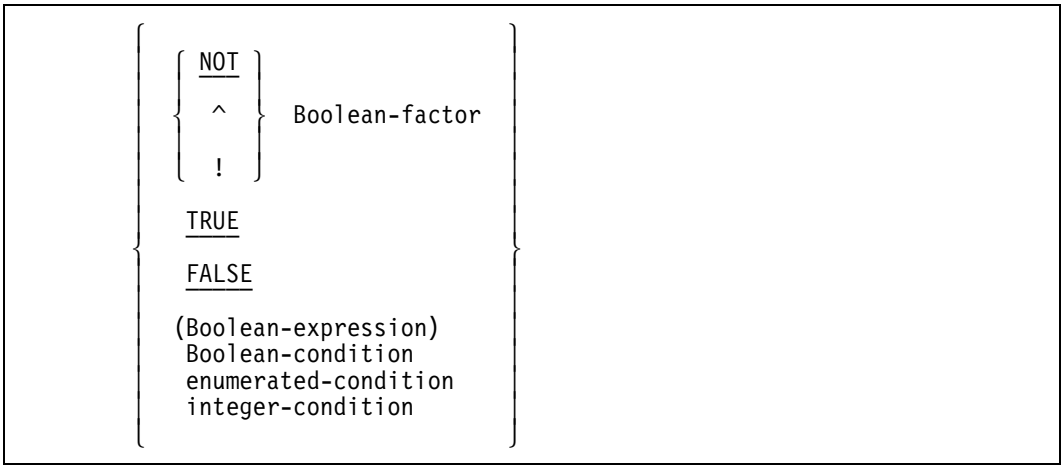
This element represents an expression that can be evaluated as a Boolean value according to the standard rules of Boolean algebra.

A Boolean-expression can be simple (for example, the value of a different Boolean option), or it can be modified by the standard Boolean operators AND, OR, or NOT. You can even nest Boolean-expressions within other Boolean-expressions by enclosing them in parentheses. Complicated Boolean-expressions should be examined to make certain that they yield the correct value in all situations.

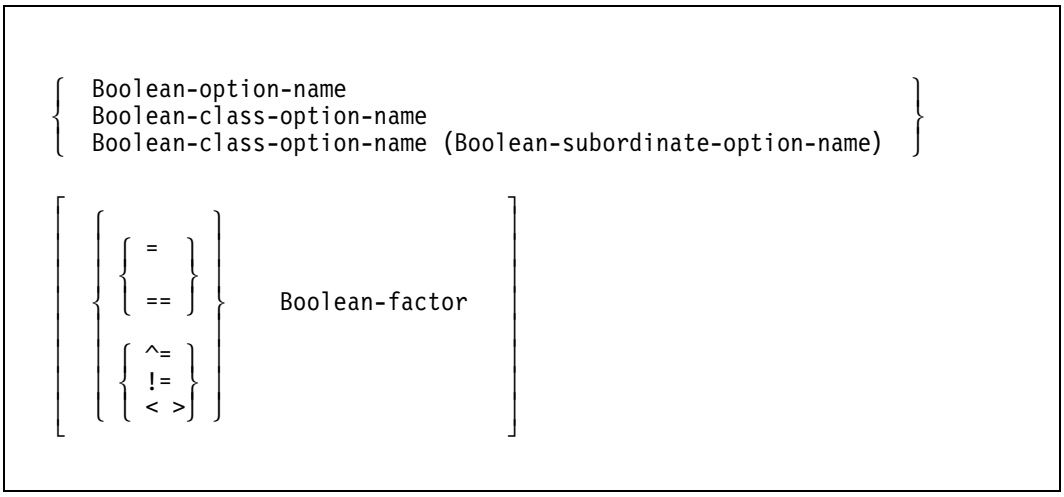
Boolean-Expression Syntax



Boolean-Factor Syntax



Boolean-Condition Syntax



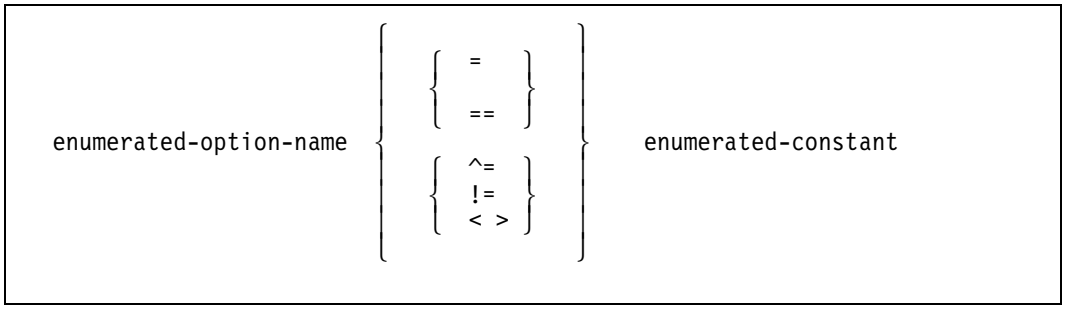
Boolean-class-option-name

This element represents the name of a Boolean class compiler control option. A Boolean class option can be set to TRUE or reset to FALSE. In addition, it possesses subordinate options which can be set or reset. To specify a subordinate option, you place the option name in parentheses following the Boolean class option, as in $SET OPTIMIZE (LEVEL).

Boolean-subordinate-option-name

This element represents the name of a Boolean subordinate option. It must be subordinate to the Boolean class option.

Enumerated-Condition Syntax



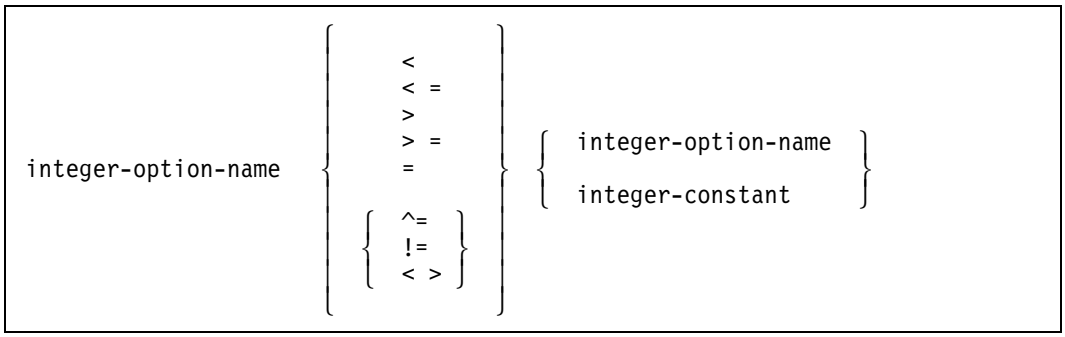
enumerated-option-name

This element represents the name of an enumerated option.

enumerated-constant

This element represents the symbolic value of the preceding enumerated option.

Integer-Condition Syntax



integer-option-name

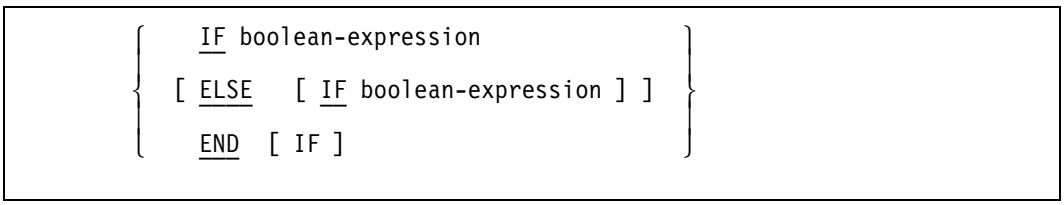
This element represents the name of a value option that takes an unsigned integer value.

integer-constant

This element represents an unsigned integer constant. affect any nonsettable options of the CCR. If an option action indicator is not specified, the SET action is assumed.

Conditional Compilations Options

Conditional compilations options are used to conditionally include or omit certain source records in the compilation. Compiler control options encountered in the source language input between any IF, ELSE IF, ELSE, or END compiler control options are always processed in the normal fashion, regardless of the value of the Boolean-expression of the IF option.



Type: Immediate

If the Boolean-expression in the IF option is TRUE, the records between the IF and a subsequent ELSE IF, ELSE, or END compiler control option are compiled, and all records between any subsequent ELSE IF or ELSE, and END compiler control options are ignored. Similarly, if the IF compiler control option is FALSE, but a subsequent ELSE IF option is TRUE, then the records between the ELSE IF and subsequent ELSE or END are compiled, with the other records ignored. If the IF and ELSE IF options resolve to FALSE, the records between the ELSE (if specified) and END options are compiled.

For the syntax of boolean-expression, refer to "Syntax for Compiler Control Options" earlier in this section. The following are examples of conditional compilations.

The IF Option

Program debug source language input can be compiled only if a user option called USERDEBUG is set to TRUE, as in the following example:

```
$ OPTION (SET USERDEBUG)
          .
          .
          .
$ IF USERDEBUG
     MOVE SPACES TO DEBUG-LOCATION.
     MOVE "GETTING NEXT ENTRY FROM LIST".
     PERFORM DEBUGGER.
$ END
     MOVE NEXT-LIST-ENTRY TO WORK-ITEM.
          .
          .
          .
```

The IF and ELSE Options

The behavior of a program can be modified through the setting of a user option, as in the following example:

```
$ OPTION (RESET OLDFORMAT)
            .
            .
            .
 DATA DIVISION.
 FILE SECTION.
 FD  ACCOUNT-FILE BLOCK CONTAINS 3 RECORDS.
*  SUPPORT BOTH THE OLD FORMAT AND THE NEW FORMAT FOR NOW
*  FIRST THE OLD FORMAT
$  IF OLDFORMAT
 01  MAJ-ACCT-INFO.
      05 ACCT-NUMBER.
          07 BRANCH           PIC 9999.
          07 DISTRICT         PIC 999.
          07 PIN              PIC 99.
      05 ASSOC-FILE-NO        PIC 999.
      05 CURRENT-STATE        PIC X(2).
      05 LAST-ACCESS-DATE     PIC X(6).
      05 EXPIRATION-DATE      PIC X(6).
*  THE NEW FORMAT CONTAINS THE FOLLOWING CHANGES
*     PIN EXPANDED FROM 2 TO 4 DIGITS
*     ASSOC-FILE-NO EXPANDED FROM 3 TO 4 DIGITS
$  ELSE
 01  MAJ-ACCT-INFO.
      05 ACCT-NUMBER.
          07 BRANCH          9999.
          07 DISTRICT        999.
          07 PIN             9999.
      05 ASSOC-FILE-NO        9999.
      05 CURRENT-STATE        PIC X(2).
      05 LAST-ACCESS-DATE     PIC X(6).
      05 EXPIRATION-DATE      PIC X(6).
$ END
           .
           .
           .
```

The IF and ELSE IF Options

A series of condition tests can be used to alter the behavior of a program, as in the following example:

```
$ OPTION (SET ALPHA RESET USER1 USER2 USER3)
          .
          .
          .
 *
 * USE USER OPTIONS TO DETERMINE THE SECURITY LEVEL OF THIS VERSION
 *
 $ IF ALPHA
      PERFORM ALPHA-INITIALIZE.
 $ ELSE IF USER1
      PERFORM STANDARD-INITIALIZE THROUGH LEVEL-ONE-USER.
 $ ELSE IF USER2
      PERFORM STANDARD-INITIALIZE THROUGH LEVEL-TWO-USER.
 $ ELSE IF USER3
      PERFORM STANDARD-INITIALIZE THROUGH LEVEL-THREE-USER.
 $ END
          .
          .
          .
```

For more information on conditional compilation, refer to "Controlling Compiler Input" earlier in this section.

Setting Compiler Options When Initiating the Compiler

You can set compiler options at compile time by including the TASKSTRING task attribute with the command you use to initiate compiling.

This method of setting options can be used instead of, or in addition to, any compiler control records (CCRs) in the INITIALCCI file and/or in the program source. The COBOL85 compiler interprets the value of the TASKSTRING task attribute as if it were a compiler control option, acting on it after the INITIALCCI file but before any CCRs in the program source.

Here is an example of setting the LIST option at compile time. This example uses CANDE, although use of this method is not limited to CANDE.

```
COMPILE; COMPILER TASKSTRING = "LIST"
```

For further discussion of the TASKSTRING task attribute, refer to the *Task Attributes Programming Reference Manual*.

Compiler Options

The following pages describe the compiler control options available in the COBOL85 compiler. Syntax diagrams are included for those options that have parameters or more than one valid name. Diagrams are not included for options where the syntax is the same as the option name.

Some options cause a file to be created or direct the compiler to use a specific file. The compiler supplies default values for the names and locations of these files. You can override the default values by using a file equate statement when the compiler is initiated. For example:

```
FILE CODE = <file-name> ON DISK
```

ANSI Option

Note: *Setting $ANSI is effectively equivalent to setting $ANSICLASS(ALL). It is recommended that you use the $ANSICLASS option instead of the $ANSI option. The $ANSICLASS option is intended to supercede the $ANSI option in a future release.*

Type: Boolean
Default: FALSE

The ANSI option affects the placement of the flagging warnings that contain nonconforming or obsolete syntax. When ANSI is set, all flagging warnings point to the specific clause, statement, or header that contains the nonconforming or obsolete syntax. When ANSI is reset, all flagging warnings point to the specific nonconforming or obsolete syntax in the source program line.

The ANSI option also affects processing of the ACCEPT statement and the COPY statement. For details, refer to "ACCEPT Statement" and "COPY Statement" in Section 6.

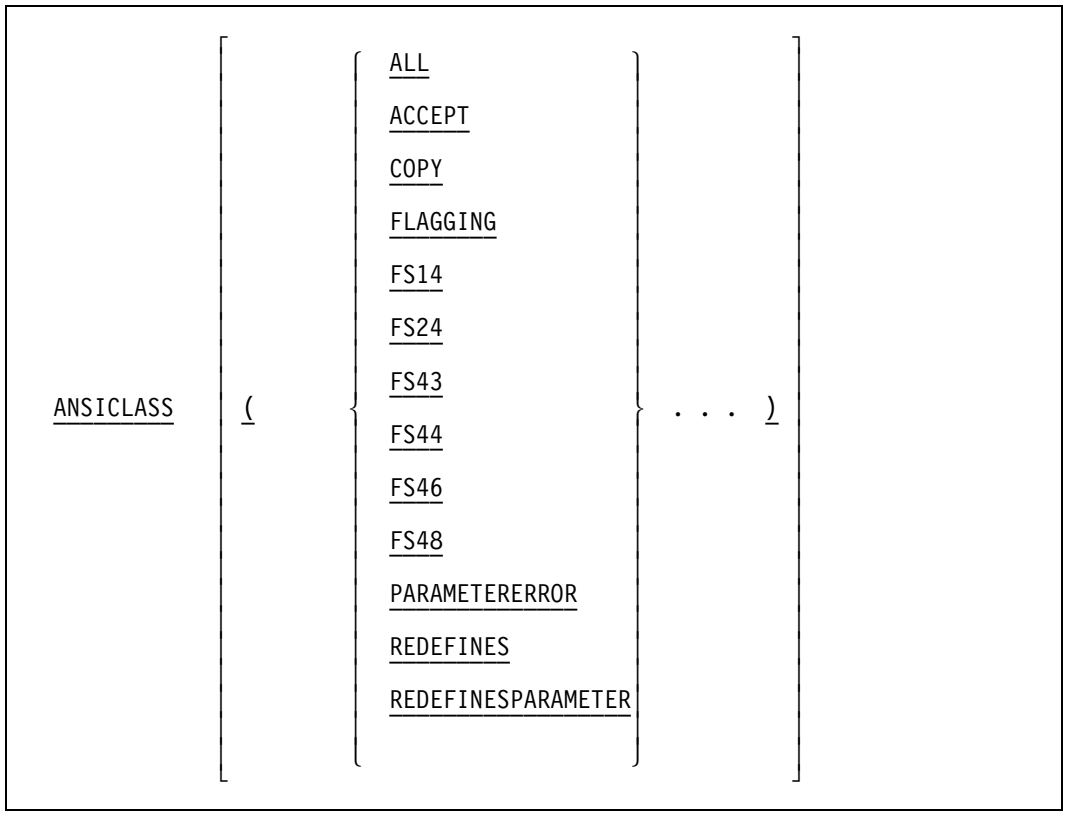
The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	The compiler uses ANSI COBOL74 rules in situations where differences exist.
SET	TRUE	The compiler uses ANSI COBOL85 rules whenever it encounters a situation where the ANSI COBOL85 rules differ from previous versions of COBOL.

ANSICLASS Option

It is recommended that you use the $ANSICLASS option instead of the $ANSI option. The $ANSICLASS option is intended to supercede the $ANSI option in a future release.

Type: Boolean Class



The ANSICLASS option provides various suboptions that you can set to

- Affect the processing of the ACCEPT and COPY statements
- Determine which I/O error conditions will be flagged with warnings
- Affect the placement of flagging warnings that contain nonconforming or obsolete syntax

  The options that begin with "FS" describe semantic errors which were not detected by the COBOL74 compiler. Because these suboptions can increase the amount of processor time required to execute certain I/O statements, users migrating from COBOL74 should determine if the benefit of detecting these errors is worth the cost of performance.

The $ANSICLASS suboptions are described in the following list. Note that all suboptions except ALL are boolean-valued.

ALL

Default Value: Not applicable

When set, the ALL suboption implies the inclusion of all of the suboptions available for the ANSICLASS option.

ACCEPT

Default Value: False (RESET)

When set, the ACCEPT suboption affects the processing of the ACCEPT statement. For details, refer to "ACCEPT Statement" in Section 6.

COPY

Default Value: False (RESET)

When set, the COPY suboption affects the processing of a COPY statement that includes the REPLACING phrase. For details, refer to the discussion of continuation lines and additional lines that accompanies the COPY statement in Section 6.

FLAGGING

Default Value: False (RESET)

When set, the FLAGGING suboption causes all flagging warnings to point to the specific clause, statement, or header that contains obsolete syntax or nonconforming syntax.

When reset, flagging warnings point only to the source program line that contains obsolete or nonconforming syntax.

FS14

Default Value: False (RESET)

The FS14 suboption must be set for the compiler to detect the I/O errors defined by file status value 14. For more information about this file status value, refer to "I-O Status Codes" in Section 3.

Users migrating from COBOL74 might want to reset this suboption, as COBOL74 did not provide this functionality.

Note that setting this suboption can increase the amount of processor time used to execute certain READ statements.

COBOL85 users can improve performance of all I/O statements acting upon either a sequential file declared with an actual key, or a relative file declared with a relative key, by declaring the appropriate key as follows:

```
77 USERSKEY       REAL.
```

FS24

Default Value: False (RESET)

The FS24 suboption must be set for the compiler to detect the I/O errors defined by file status value 24. For details on this file status value, see "I-O Status Codes" in Section 3.

Users migrating from COBOL74 might want to reset this suboption, as COBOL74 did not provide this functionality.

Note that setting this suboption can increase the amount of processor time used to execute certain WRITE statements.

COBOL85 users can improve the performance of all I/O statements acting upon either a sequential file declared with an actual key, or a relative file declared with a relative key, by declaring the appropriate key as follows:

```
77 USERSKEY       REAL.
```

FS43

Default Value: False (RESET)

The FS43 suboption must be set for the compiler to detect the I/O errors defined by file status value 43. For more information about this file status value, refer to "I-O Status Codes" in Section 3.

Note that setting this suboption can increase the amount of processor time used to execute certain DELETE and REWRITE statements.

FS44

Default Value: False (RESET)

The FS44 suboption must be set for the compiler to detect the I/O errors defined by file status value 44. For more information about this file status value, refer to "I-O Status Codes" in Section 3.

Note that setting this suboption can increase the amount of processor time used to execute certain WRITE and REWRITE statements.

FS46

Default Value: False (RESET)

The FS46 suboption must be set for the compiler to detect the I/O errors defined by file status value 46. For more information about this file status value, refer to "I-O Status Codes" in Section 3.

Note that setting this suboption can increase the amount of processor time used to execute certain READ statements.

FS48

Default Value: False (RESET)

The FS48 suboption must be set for the compiler to detect the I/O errors defined by file status value 48. For details on this file status value, see "I-O Status Codes" in Section 3.

Setting this suboption can increase the amount of processor time used to execute certain WRITE statements.

PARAMETERERROR

Default Value: False (RESET)

When set, the PARAMETERERROR suboption changes compiler message 213: FORMAL PARAMETER ERROR from a warning to an error message. This change enables stricter checking of incompatible parameter types within ANSI-85 COBOL and conforms to the COBOL85 parameter checking scheme. When reset, checking is released to conform to the COBOL74 parameter checking scheme.

The COBOL85 compiler originally supported a different parameter mapping scheme for ANSI-85 COBOL specifications. In order to support migration, the COBOL85 compiler has been enhanced to allow a parameter mapping scheme more similar to A Series COBOL74. The compile-time checking for this parameter mapping scheme, however, was not put into place as syntax error checking.

REDEFINES

Default Value: False (RESET)

When set, the REDEFINES suboption causes no warnings to be issued on REDEFINES items. When reset, a warning message is emitted for a 01 REDEFINES item whose size is different from the redefined item and for a 02 or higher REDEFINES item whose size is smaller than the redefined item.

REDEFINESPARAMETER

Default Value: False (RESET)

When set, the REDEFINESPARAMETER suboption changes the following error messages, which are too stringent within ANSI-85 COBOL, to warning messages

- 966: PARAMETER WITH REDEFINES IS ILLEGAL
- 1062: A REDEFINED PARAMETER IS ILLEGAL
- 1063: A PARAMETER FROM AN IMPLICIT REDEFINITION IS ILLEGAL

This change enables all parameters that are with or of REDEFINES to be passed. When a parameter is redefined, it is possible that the actual parameter length could be different. This difference is due to the way A Series handles some types. If the redefined item is of a different length than the REDEFINES item, the extra data becomes accessible in the callee. As the entire record is sent to the callee with no length restriction, A Series data corruption can result.

ASCII Option

Type: Boolean
Default: FALSE

This option controls the default character type used for the compilation. The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	Sets the default character type to EBCDIC.
SET	TRUE	Sets the default character type to ASCII.

The default character type is assumed for all strings when a character type has not been explicitly specified. The default character type is also used as the default value of the INTMODE file attribute.

Unless overridden by the ASCII option or the STRINGS = ASCII option, the default character type is EBCDIC.

STRINGS = ASCII is a synonym for the ASCII option.

AUTOINSERT Option

Type: Boolean
Default: TRUE

This option controls whether or not the Unisys extension described as Automatic insertion editing under the material associated with the PICTURE clause is to be enabled. Refer to "Automatic simple insertion editing" in Section 4 of this manual.

Option Setting	Value	Description
RESET	FALSE	Automatic simple insertion editing disabled
SET	TRUE	Automatic simple insertion editing enabled

Although the default setting for the AUTOINSERT option is TRUE, the option itself is scheduled for deimplementation in a future software release. Upon deimplementation, the default behavior will be as if the option were RESET.

It is recommended that this option be RESET. For those programs in which simple insertion of arbitrary symbols using the PICTURE string is needed, use Manual insertion editing and the (I) symbol as described in the material associated with the PICTURE clause.

Manual insertion editing and Automatic insertion editing cannot both occur in the same PICTURE character string. AUTOINSERT and STRICTPICTURE cannot both be set to TRUE at the same time.

BINARYCOMP Option

Type: Boolean
Default: FALSE

This option controls how the compiler handles COMPUTATIONAL data items. The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	COMPUTATIONAL data items are treated as COMPUTATIONAL data items. This inhibits the use of COMPUTATIONAL data items as actual parameters in calls to procedures written in languages other than COBOL.
SET	TRUE	COMPUTATIONAL data items are treated as if they were declared USAGE BINARY EXTENDED.

BINARYEXTENDED Option

Type: Boolean
Default: FALSE

This option controls how the compiler handles BINARY data items. The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	BINARY data items are treated as BINARY data items.
SET	TRUE	BINARY data items are treated as if they were declared as USAGE IS BINARY EXTENDED. If the TRUNCATED phrase is explicitly specified as USAGE IS BINARY TRUNCATED, that binary item is treated as BINARY.

For more information about BINARY EXTENDED data items, refer to "USAGE IS BINARY" in Section 4.

BINDER_MATCH Option

```
{ BINDER_MATCH = (string-1,string-2) . . . }
```

Type: Immediate
Default: Not applicable

This option verifies that object code files being bound with the Binder are compiled with the same set of compile-time options. Setting the BINDER_MATCH option adds string-1 and string-2 to the object code file.

String-1 can contain a maximum of 255 characters. This string must not begin with a star (*).

String-2 specifies the value of string-1.

The strings must match in casing of alphabetic characters. For example, an uppercase letter “A” does match a lowercase letter “a.”

You can specify a maximum of 200 BINDER_MATCH occurrences. A maximum of 10,000 characters is allowed for the total number of name and value strings.

When the Binder encounters a BINDER_MATCH option in one program that has an identical first string as a BINDER_MATCH option in the other program, Binder verifies that the second strings are also identical. If the second strings are different, then different values were used for the same compile-time option in the two programs. Binder prints an error message and the bind is aborted.

Note that having two BINDER_MATCH options in the same file with the same first string and different second strings causes an error at the time of compilation.

BINDINFO Option

Type: Boolean
Default: FALSE

This option controls whether information used for program binding is placed in the output object code file. The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	Binder information is not placed in the object code file.
SET	TRUE	Binder information is placed in the object code file.

The BINDINFO option does not affect programs compiled with the LEVEL options set to 3 or higher.

This option must be included in the source code before the Identification Division of the program.

If the program is a HOST program and an external procedure is to be bound in, this option must be set. COBOL74 users must set this option for HOST program files.

BINDSTREAM Option

Type: Boolean
Default: False

This option controls whether or not the symbolic file to be compiled contains a definition program and a list of multi-procedure programs separated by a LIBRARY option. The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	The symbolic file to be compiled does not contain a definition program and a list of multi-procedure programs separated by a LIBRARY option.
RESET	TRUE	The symbolic file to be compiled contains a definition program and a list of multi-procedure programs separated by a LIBRARY option.

When the BINDSTREAM option is set, a code file title must be specified for each LIBRARY option.

Example

```
$BINDSTREAM
<Definition Program>
$LIBRARY "OBJECT/1"
  <procedure 1>
          :
```

```
  <procedure n1>
$LIBRARY "OBJECT/2"
  <procedure 1>
          :
  <procedure n2>
$LIBRARY "OBJECT/N"
  <procedure 1>
            :
  <procedure nN>
```

When the preceding symbolic is compiled, multiple multi-procedure code files (OBJECT/1, OBJECT/2, ..., OBJECT/N) are created for binding purposes.

A definition program contains the declaration of global data items that can be shared by a list of multi-procedure programs.

Data items declared in the definition program with or without the COMMON clause are treated as data items declared in the WORKING-STORAGE SECTION with the COMMON clause. These data items are matched by name and type to the global directory of the host.

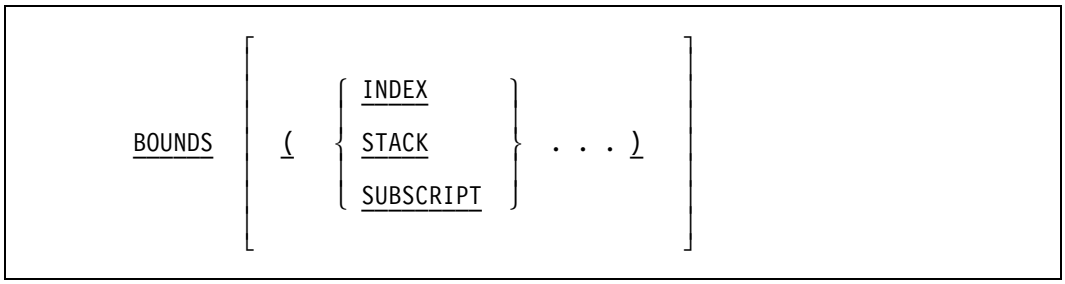
The VALUE clause is not allowed in the definition program if the associated data item is not a condition name.

A data item in the definition program that is referenced in a subprogram is declared at lex level 2. Data items declared in a subprogram is declared at the appropriate lex level as specified in the LEVEL option for the subprogram. If the LEVEL option is set to 2 in the subprogram, then the lex level is treated as 3.

If a data item of the definition program is referenced in a subprogram, the data item is included in the code file, otherwise it is ignored.

BOUNDS Option

Type: Boolean Class
Default: True (SET) for INDEX, True (SET) for STACK, True (SET) for SUBSCRIPT



The BOUNDS option provides suboptions that you can use to control the checking of the upper bounds of indexes and subscripts

Caution

Unpredictable results may be obtained if BOUNDS(INDEX) is FALSE and a subscript or index is not valid. Results may include data corruption as well as abnormal program termination.

The effect of this option depends on the following considerations:

- An index or subscript, or an expression (constant or variable) used as a subscript, is valid if its value is not less than 1 and not greater than the number of elements in the table.
- The value of the subscript or index can be known or unknown at compile time:
  - If the subscript is a literal or a constant expression, then the value of the subscript is known at compile time. Otherwise the value of the subscript or index is unknown at compile time.
  - The value of an index is always treated as unknown at compile time.
- The size of the table can be known or unknown at compile time:
  - If the OCCURS clause does not include the DEPENDING ON phrase, then the number of elements in the table is known at compile time.
  - If the OCCURS clause includes the DEPENDING ON phrase, then the number of elements in the table is unknown at compile time.

- Some bounds-checking is done regardless of the setting of the respective BOUNDS option:
  - If both the number of elements in the table and the value of the subscript are known at compile time, and the value of the subscript is greater than the number of elements in the table, then the compiler produces a syntax error message.
  - If the value of the subscript is known at compile time, and it exceeds the maximum number of elements specified in the OCCURS clause, then the compiler produces a syntax error message. This validation is done without reference to the setting of the BOUNDS option.
- Other bounds-checking is done only if the appropriate BOUNDS option is set to TRUE:
  - If the value of the subscript or index, or the number of elements in the table, or both, are unknown at compile time, and if, when these values are computed at execution time, the subscript or index is greater than the number of elements in the table, then the program terminates abnormally. This validation is done only if the BOUNDS option is set to TRUE.

When the appropriate BOUNDS option is set to TRUE for a program that has OCCURS clauses, the compiled program will have extra code to deal with subscripting and will use additional processor resources.

INDEX

This option controls whether the compiler checks for range violations when index names are used to access tables.

STACK

This option controls whether the boundaries of the software stack are checked at execution time. Refer to "MEMORY_MODEL Option" in this section for details on the software stack.

SUBSCRIPT

The option controls whether the compiler checks for range violations when subscripts are used to access tables.

Examples

The following examples illustrate the use of the BOUNDS option:

$SET BOUNDS

Range checking is active and provided for all suboptions that are not currently disabled.

$RESET BOUNDS

Range checking is inactive.

$RESET BOUNDS(INDEX)

No range checking features are active, and the INDEX option is specifically disabled. If the BOUNDS option is set later by itself, the INDEX option is still disabled.

$SET BOUNDS(INDEX)

The INDEX option is enabled, all other options are unaffected, and range checking is active for all enabled options.

$SET BOUNDS(INDEX RESET SUBSCRIPT)

The INDEX option is enabled, the SUBSCRIPT option is disabled, and range checking is provided for all enabled suboptions.

$SET BOUNDS(SET INDEX RESET SUBSCRIPT)

The INDEX option is enabled, the SUBSCRIPT option is disabled, and range checking is provided for all enabled suboptions.

$RESET BOUNDS(SET INDEX)

No range checking is active, although the INDEX option is enabled. If BOUNDS is set again later, then range checking is active and provided for indexes and any other options that are enabled.

CALL MODULE Option

Type: Boolean
Default: FALSE

This option causes the object code file produced by the compilation to contain the code necessary to enable it to be called by a CALL MODULE statement. For more information, refer to Format 7 of the CALL Statement in Section 6.

C68MOVEWARN Option

Type: Boolean
Default: FALSE

This option issues a warning message for MOVE statements in which one of the operands is a group item and the other operand is an elementary numeric item. The warning messages produced by the C68MOVEWARN option are not suppressed by the WARNSUPR option.

The C68MOVEWARN option is useful for migrating programs from COBOL(68) to COBOL85 because it identifies the MOVE statements that are expecting the results contrary to the requirements of the COBOL-1985 standard.

CALLNESTED Option

Type: Boolean
Default: FALSE

This option governs the declaration of a nested program as either an internal or external call. For a program with many nested programs, the number of cells allocated on the D2 stack may exceed the hardware limit of the D2 stack size. D2 stack cells are needed only if the called program is an external call. They are not needed if the call is to an internal nested program.

However, because the calls are usually seen before the actual declaration of the nested program, the compiler cannot distinguish a specific call to an internal or an external program. CALLNESTED is only applicable to CALL "literal". CALLNESTED can be SET and RESET throughout the program. The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	Calls of this type are assumed to be external.
SET	TRUE	Calls of this type are assumed to be internal.

CODE Option

Type: Boolean
Default: FALSE

This option controls whether the printed listing of the compiled program contains the generated object code. The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	The generated object code is not included in the printed listing of the compiled program.
SET	TRUE	The generated object code is included in the printed listing of the compiled program.

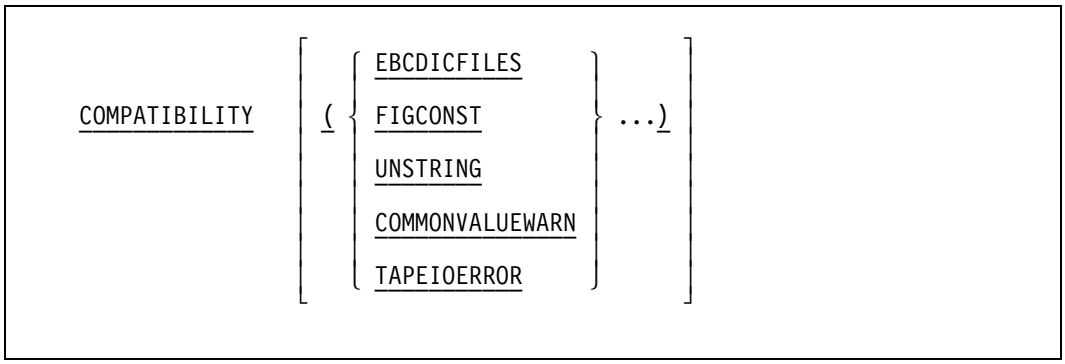
Note: *If the $OPTIMIZE option is set, the code displayed will not have any relation to the source lines because the optimizer moves code in order to achieve optimization.*

COMMON Option

This option causes all data items in Working-Storage to be COMMON except those specifically declared LOCAL or OWN. The COMMON option is ignored for host files, or if the compilation is at level 2. (COBOL74 users using the GLOBAL option must use the COMMON option for binding purposes.) This option has no effect on the Environment Division or File Section.

COMPATIBILITY Option

Type: Boolean Class
Default: True (SET) for EBCDICFILES, false (RESET) for FIGCONST, false (RESET) for UNSTRING, false (RESET) for COMMONVALUEWARN, false (RESET) for TAPEIOERROR



EBCDICFILES

This option specifies whether EBCDIC or hex files will be created.

In previous releases, the value of the INTMODE attribute of a file created by a COBOL85 program would vary depending on the USAGE clause of the first 01 level entry under the FD statement.

The COBOL85 compiler now, by default, always creates files with an INTMODE value of EBCDIC, regardless of the USAGE clause.

Resetting the suboption with

```
COMPATIBILITY (RESET EBCDICFILES)
```

will still create hex files (INTMODE=HEX) and word files (INTMODE=SINGLE) if the first 01 level entry under the FD statement is a hex item (for example, USAGE COMP) or a word item (for example, USAGE REAL).

This option is intended to facilitate migration. It is recommended that the default setting be used.

FIGCONST

When set, the FIGCONST option affects the processing of the MOVE statement for the figurative constants HIGH-VALUE, LOW-VALUE, and ALL "literal" to numeric data items of usage COMP or DISPLAY. For details, see Format 1 of the MOVE Statement in Section 7, “Compiler Operations.”

UNSTRING

When set, UNSTRING INTO <numeric_ item> ignores the implied decimal point. If you reset this option to the default setting, the UNSTRING INTO <numeric-item> obeys the normal MOVE rules. For details, see Format 1 of the MOVE Statement in Section 7, “Compiler Operations.”

This option should be used only for EVA 85 migration programs which assume the implied decimal point is ignored.

COMMONVALUEWARN

When set, the COMMONVALUEWARN option emits a warning once for a VALUE clause on a COMMON item in a bindable sub-program. When this option is false, every affected item will be flagged.

This option should be set for all EVA 85 migration programs.

TAPEIOERROR

The compiler control option COMPATIBILITY(TAPEIOERROR) permits a COBOL85 code file to perform input and output operations on tape files in a fashion similar to COBOL74. When the option is set, COBOL85 masquerades its I-O operations as though it were COBOL74. This permits I-O to continue in COBOL74 fashion after what would otherwise be a permanent and disabling error. Setting the option prevents a COBOL85 program from returning file status codes 43 through 49 for tape files.

The option can be set universally in the INITIALCCI file for your compiling environment or it can be set in individual programs. If all tape files in a given program are to be affected by the option, it can be set at the start of the program. If only selected tape files in a program are to be affected, the option must be set and reset around the SELECT clause of the file in the program ENVIRONMENT DIVISION.

The option can be manipulated as shown in the following examples:

```
$ SET COMPATIBILITY(TAPEIOERROR)
```

This option sets the IOERROR suboption, while enabling any other suboptions that were already set.

```
$ SET COMPATIBILITY(SET TAPEIOERROR)
```

This option, like the preceding option, sets the IOERROR suboption, while enabling any other suboptions that were already set.

```
$ SET COMPATIBILITY(RESET TAPEIOERROR)
```

This option resets the IOERROR suboption, while enabling any other suboptions that were already set.

```
$ RESET COMPATIBILITY(TAPEIOERROR)
```

This option resets the IOERROR suboption, while disabling all other suboptions that were already set.

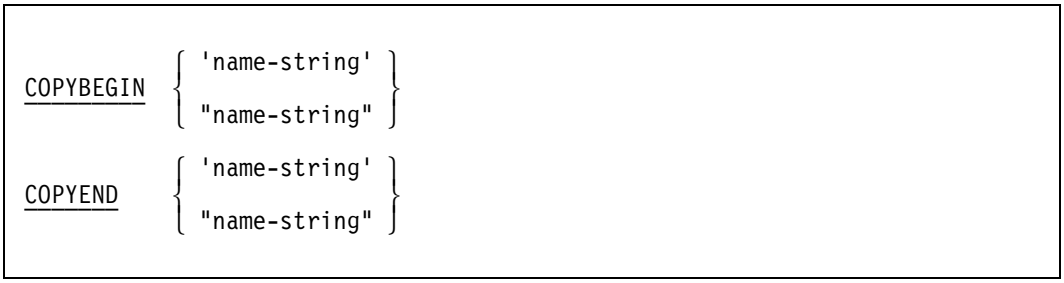
```
$ RESET COMPATIBILITY
```

This option disables any suboptions that were already set.

```
$ SET COMPATIBILITY
```

This option enables any suboptions that were already set.

Copy Boundary Options



Type: String

The COPYBEGIN and COPYEND options delimit a symbolic subfile. The name-string associates a symbolic name with that subfile. A symbolic subfile consists of all records situated between a COPYBEGIN CCR and a COPYEND CCR with matching symbolic names (name-strings). Source code file records delimited by COPYBEGIN and COPYEND CCRs can be included by other programs with the name-string range option of the INCLUDE compiler option. Symbolic subfiles can be nested within one another, and they can overlap one another. The symbolic names are matched without regard to case.

The maximum length of a name-string is 30 characters.

The following example illustrates the use of the INCLUDE compiler option in conjunction with the COPYBEGIN and COPYEND options.

A source code file contains the following INCLUDE option:

```
$ INCLUDE LOCALDEFINES = '(GAS)LOCAL/DEFINES ON RDLS' ('LOCAL_FAIL_DEFINE')
```

The file (GAS)LOCAL/DEFINES ON RDLS contains the following COPYBEGIN and COPYEND options:

```
 WORKING-STORAGE SECTION.
$ COPYBEGIN 'LOCAL_FAIL_DEFINE'
 01  LOCAL-FAIL-DEFINE.
     03  REC-TYPE.
       05  MAJOR-TYPE                          PIC 99.
       05  MINOR-TYPE                          PIC 99.
     03  CAUSE-CODE                            PIC 9(4).
     03  SYS-JULIAN-DATE.
       05  DAY                                 PIC 999.
       05  YEAR                                PIC 99.
     03  SYS-TIME                              PIC X(6).
     03  LOCAL-TEXT                            PIC X(50).
$ COPYEND 'LOCAL_FAIL_DEFINE'
```

When the INCLUDE option is encountered in the first file, all records situated between the $ COPYBEGIN 'LOCAL_FAIL_DEFINE' CCR and the $ COPYEND 'LOCAL_FAIL_DEFINE' CCR in the second file are included in the first file.

CONCURRENTEXECUTION Option

This option is for internal use in the system software only.

CORRECTOK Option

Type: Boolean
Default: FALSE

This option controls whether the compiler corrects certain minor syntax errors. The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	The compiler does not perform any automatic syntax correction.
SET	TRUE	The compiler corrects certain minor syntax errors and issues warning messages instead of error messages.

CORRECTSUPR Option

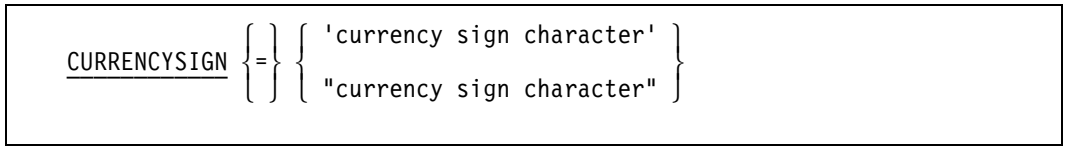
Type: Boolean
Default: FALSE

This option controls whether the compiler issues warning messages pertaining to minor syntax errors it has encountered and corrected. The compiler can be directed to correct certain minor syntax errors with the CORRECTOK option. The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	The compiler produces warning messages that pertain to automatically corrected syntax errors.
SET	TRUE	The compiler suppresses the warning messages that pertain to automatically corrected syntax errors.

CURRENCYSIGN Option

Type: String



This option alters the default currency sign in the program for floating insertion editing without requiring the explicit specification of the CURRENCY SIGN clause.

This option must be included in the source code before the Data Division of the program to be in effect.

The currency sign character must be nonnumeric and is limited to a single character. For more information, refer to the CURRENCY SIGN Clause of the SPECIAL-NAME Paragraph in Section 3.

If both the CURRENCY SIGN clause and the CURRENCYSIGN compiler control option are specified, the currency sign specified with the CURRENCY SIGN clause is used in the program.

DELETE Option

Type: Boolean
Default: FALSE

The DELETE option controls whether the compiler incorporates source language records from the secondary input file (SOURCE) into the compiled program when the MERGE option is TRUE.

If the MERGE option is FALSE, the DELETE option is ignored.

The following table describes the effects of this option:

Option	Value	Description
RESET	FALSE	Source language records from the secondary input file are incorporated into the compiled program when the MERGE option is TRUE.
SET	TRUE	Source language records from the secondary input file are discarded when the MERGE option is TRUE. The discarded source language records are not included in the output symbolic file (NEWSOURCE) if the NEW option is TRUE.

The DELETE option can appear only on a CCR in the primary source language file (CARD).

The DELETE option and the VOIDT option are synonymous.

ELSE and ELSE IF Options

Type: Immediate

The ELSE and ELSE IF options are conditional compilation options. For information on conditional compilation options, refer to "Conditional Compilation Options" and "Controlling Compiler Input" earlier in this section.

EMBEDDEDKANJI Option

Type: Boolean

Default: FALSE

This option controls the processing of an EBCDIC literal in which a double octet literal is embedded.

The EMBEDDEDKANJI option causes the compiler to ignore the quote (hex '7F') character in an EBCDIC literal and treat it as part of the double octet character code whenever it is surrounded by SDO (hex '2B') and EDO (hex '2C') delimiters.

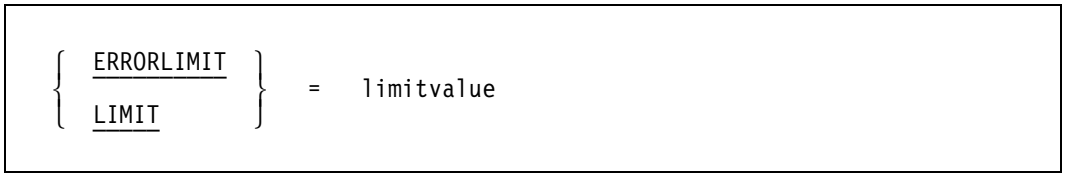
The EMBEDDEDKANJI option is useful only for the users of double octet character sets that contain hex '7F' as part of the character code. Use of this option should be limited to such users due to the impact of the option on EBCDIC literal scanning.

END Option

Type: Immediate

The END option is a conditional compilation option. For information on conditional compilation options, refer to "Conditional Compilation Options" and "Controlling Compiler Input" earlier in this section.

ERRORLIMIT Option



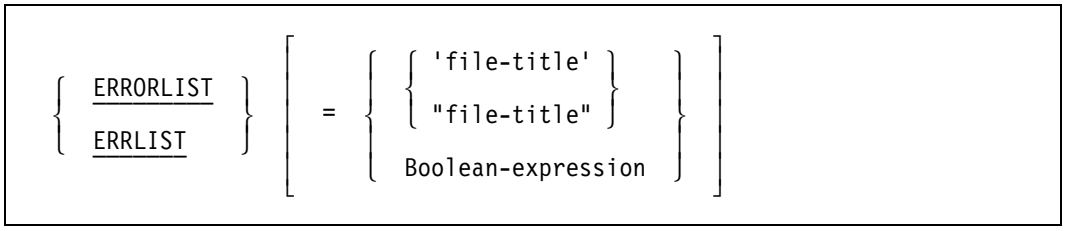
Type: Value
Default: 10 for compilations originating from CANDE; otherwise, 150

This option determines the maximum number of errors that the compiler can detect before compilation is terminated.

When the error limit is exceeded, the compiler creates a listing of the errors and informs you that compilation was terminated because of an excess number of errors.

If the NEW option is TRUE and the error limit is exceeded, the new symbolic file (NEWSOURCE) is purged.

ERRORLIST Option



Type: Boolean Title
Default: TRUE for CANDE-originated compilations; otherwise, FALSE

This option controls whether the compiler creates an error message listing file. The following table describes the effects of this option:

Setting	Value	Description
RESET	FALSE	The compiler does not create an error message listing file.
SET	TRUE	The compiler creates an error message listing file. • If no file title is specified, the error file is named ERRORFILE. • If you compile with CANDE, the error file is automatically file equated to the remote device from which the compiler was initiated.

Syntax

file title

Specifies an alternate file title for the error file. You can qualify the file-title with a usercode and/or location, so long as a foreign host is not specified.

Boolean-expression

Creates the file title according to an expression that can be evaluated as a Boolean value according to the standard rules of Boolean algebra

For details about the syntax of a Boolean-expression, refer to "Syntax for Compiler Control Options" in this section.

Error Message Listing

When a syntax error is detected in the source input, two lines of text are inserted in the ERRORFILE. These two lines show

- The source record that contained the error
- An error message
- A pointer to the item in the record where the error occurred

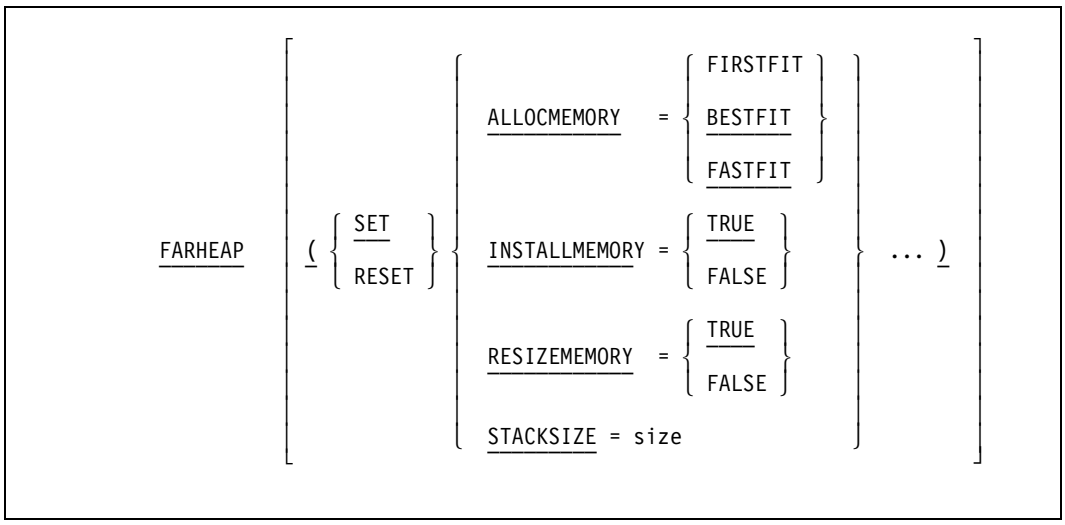
If a syntax error occurs before the ERRORLIST compiler option in the source file, the error message listing does not include information pertaining to that error.

Example

```
000100$$SET ERRLIST="(CHARLIE)ERR/BIT ON QUAL"
```

This statement creates an error file under the CHARLIE usercode if any syntax error occurred during compilation. If the location or file title is invalid, a default file title is used.

FARHEAP Option



Type: Boolean Class
Default: FALSE

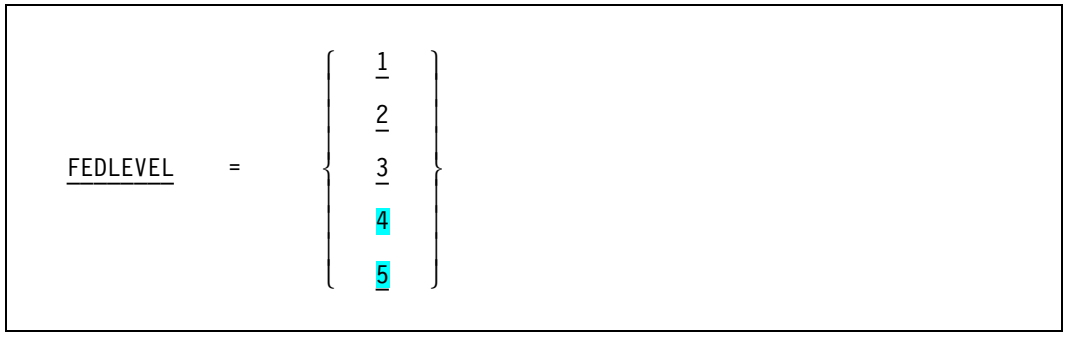
The FARHEAP option enables a program or library to select the far heap memory management mechanism instead of the default heap memory management mechanism.

The FARHEAP option must appear before any program text.

You must set the FARHEAP option when a separately compiled COBOL85 module is bound into a C language program that uses the far heap management mechanism.

The FARHEAP option is not currently used by the COBOL85 compiler. This syntax is provided for use when binding a COBOL85 program with a C language program. Refer to the *C Programming Reference Manual* for more information.

FEDLEVEL Option



Type: Value
Default: 4

The FEDLEVEL option causes the compiler to produce nonfatal warnings for constructs not available at the level at which the program was compiled. For example, if FEDLEVEL is set to 2, all constructs allowed only for level 3 and higher produce warnings.

The FEDLEVEL option measures compliance with the U.S. Government COBOL standards as specified in the *Federal Information Processing Standards (FIPS) Publication 21-2* for COBOL dated March 18, 1986.

FEDLEVEL provides the following levels of compliance:

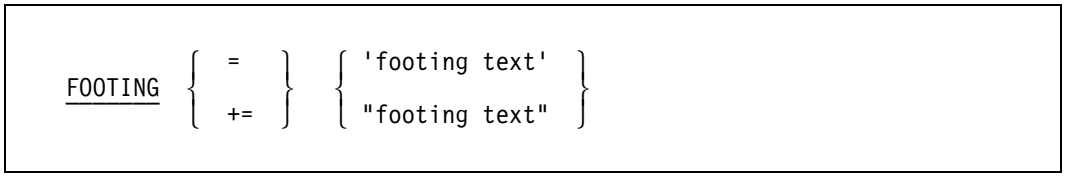
Value	Description
1	Minimum level
2	Intermediate level
3	High level
4	Extensions to ANSI standard
5	Extensions to ANSI standard

If the FEDLEVEL option is set to 5, the program-name specified in the PROGRAM-ID clause is used as the entry-point-name for the program; otherwise, the entry-point-name is PROCEDUREDIVISION. The option FEDLEVEL = 5 should not be used in programs that are explicitly declared to be libraries. This option is retained to ease the migration from ANSI COBOL74 to ANSI COBOL85, for programs using the interprogram communication (IPC) facility to call a library program.

All warnings appear under the source program line with an indication of the beginning location of the nonconforming language element. However, if the ANSI compiler control option is set, the warning points only to the start of the source line.

FEDLEVEL warning messages are not printed if the WARNSUPR option is TRUE.

FOOTING Option



Type: String

This option specifies a string of characters to be placed in the footer of each output listing page. Only the final value declared in the program as the footing appears on the output listing. If the LIST option is FALSE, the FOOTING option is ignored.

FOOTING =

Assigns a character string to the footer of each output listing page.

FOOTING +=

Appends a character string to a previously defined footing string.

'footing text'
"footing text"

Either apostrophes (' ') or quotation marks (" ") may enclose the footing text.

FREE Option

Type: Boolean
Default: TRUE for CANDE-originated compilations; otherwise, FALSE

This option controls whether COBOL margin restrictions are enforced during the compilation. The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	COBOL margin restrictions are enforced by the compiler.
SET	TRUE	Most COBOL margin restrictions are ignored by the compiler outside of the Identification Division. Violations of COBOL margin restrictions are not considered syntax errors.

Note: *When the FREE option is on, the margin restrictions in the Data Division, Environment Division, and Procedure Division are not enforced. But the Identification Division still has its margin restriction. This restriction is released as soon as one of the other three divisions are encountered.*

Any division that follows the Identification Division should still follow margin restrictions. For example:

```
$ SET FREE
 IDENTIFICATION DIVISION.
  PROGRAM-ID
       .
       .
       .
 * The following division is free from margin restrictions
 DATA DIVISION.
       .
      .
       .
 * The following division is free from margin restrictions
 PROCEDURE DIVISION.
       .
       .
       .
```

FS4XCONTINUE Option

Type: Boolean
Default: False

This option controls whether a program is terminated or is allowed to continue after a file status value of 41, 42, 43, 44, 46, 47, 48, or 49 is returned. The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	When a file status value in the 4x range is returned, the program executes any applicable use routines, and the task is terminated with an I-DS.
SET	TRUE	When a file status value in the 4x range is returned, the program executes any applicable use routines and continues executing.

Even with the FS4XCONTINUE option set, one of the following conditions must be met for a program to continue executing after a failed I/O request:

- The FILE STATUS clause is declared (Environment Division).
- A USE routine is specified.
- An alternate statement to perform in case of an unsuccessful I/O is declared in the syntax of the I/O statement (refer to each I/O statement for details).

For the meanings of the file status codes, refer to "I-O Status Codes" in Section 3.

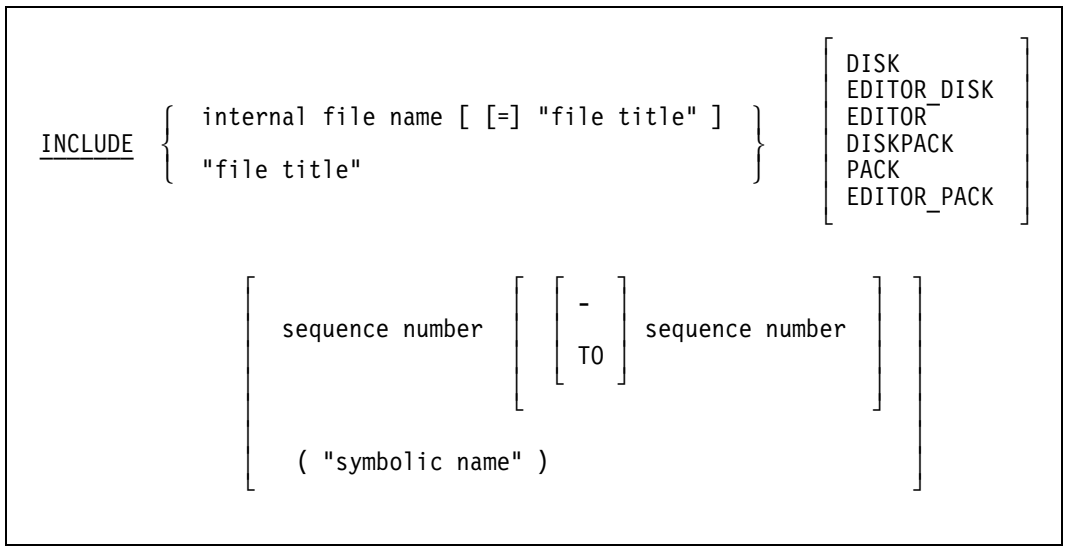
INCLNEW Option

Type: Boolean
Default: FALSE

This option controls whether source language records included during the compilation through use of the INCLUDE option are written to the updated symbolic file (NEWSOURCE). If the NEW option is FALSE, the INCLNEW option is ignored. The following table describes the effects of the INCLNEW option when the NEW option is TRUE:

Option Setting	Value	Description
RESET	FALSE	If the NEW option is TRUE, the updated symbolic file (NEWSOURCE) does not contain any source language records included through use of the INCLUDE option.
SET	TRUE	If the NEW option is TRUE, included source language records are written to the updated symbolic file (NEWSOURCE).

INCLUDE Option



Type: Special

This option instructs the compiler to temporarily accept input from a specified alternate source. The compiler uses the alternate source as input until the alternate source is exhausted or a specified range within the alternate source is exceeded.

file title

The alternate source for compiler input can be specified by a file title, an internal file-name, or a combination of an internal file-name and a file title. If the INCLUDE compiler option specifies a file title, then the compiler examines the partial file-names specified by the SEARCH compiler option (if available) to construct the file title or file titles used for the included file.

If an internal file-name is specified, the compiler assigns the internal file name to the INTNAME file attribute of the file. Use of internal file names for included files permits file equation of files at compilation time. If both an internal file name and a file title are specified, then any compilation time file equation overrides the file equation of the INCLUDE compiler option.

If only an internal file name is specified, and file equation is not used, the compiler uses specifications supplied by the SEARCH compiler option to construct the file title or file titles used for the included file.

Device name

The device names (DISK, EDITOR_DISK, EDITOR, PACK, DISKPACK, EDITOR_PACK) are included only to facilitate the migration of code from V Series to ClearPath and A Series platforms. Including these names has no effect on the compiler.

File range

The specification of a range within the included file can be a single sequence number or a sequence number range. Also, it can be a symbolic name that identifies a symbolic subfile. The symbolic name can contain a maximum of 30 characters. A symbolic subfile consists of all records situated between a COPYBEGIN compiler option and a COPYEND compiler option with matching symbolic names.

Additional details

The following special characters have meaning when they occur at the beginning of the file title specification of an INCLUDE option:

- / (slash)
- > (greater than symbol)

A slash character (/) instructs the compiler to use only the file title specification when searching for the included file. The partial file names specified in the SEARCH option (if applicable) are not applied to the file title specification. For example, if the file title was specified as `INCLUDE DEFINES ="/LOCAL/DEFINE"`, the compiler would search for the included file using the following file title:

```
(<current usercode>)LOCAL/DEFINES ON <current family>
```

A greater-than symbol (>) instructs the compiler to skip one partial file name in the partial file name list specified by the SEARCH option. More than one greater than symbol may be concatenated to skip more than one partial file title. Two greater-than symbols (>>) would cause the compiler to skip two partial file names in the partial file name list, and begin forming file title combinations with the third entry. For example, a program might contain the following CCRs:

```
$ SET SEARCH ="SYSTEM/= ON LOCALPACK; (USER)SOURCE/="
$ SET SEARCH+= ";*SYSTEM/= ON EXTERNALS"
              .
              .
              .
$ INCLUDE DEFINES = ">LOCAL/DEFINES"
```

Based on these specifications, the compiler would search for the included file using the following file title combinations.

```
(USER)SOURCE/LOCAL/DEFINES
*SYSTEM/LOCAL/DEFINES ON EXTERNALS
```

For more information about the COPYBEGIN and COPYEND compiler options, refer to "Copy Boundary Options" earlier in this section.

INLINEPERFORM Option

This option causes the next PERFORM statement and all nested PERFORM statements to be replaced, if possible, by in-line code when the OPTIMIZE option is set. A warning message is displayed if it is not possible for the PERFORM statement to be in-lined.

If you set the INLINEPERFORM option, a PERFORM statement might not be in-lined for the following reasons:

- The INLINEPERFORM option performs a PERFORM statement (nested PERFORM) that cannot be in-lined.
- The INLINEPERFORM option executes a GO TO statement that may not end up at the PERFORM statement return point (the end of the last performed paragraph).
- Another PERFORM statement overlaps with an earlier PERFORM statement but has a different PERFORM statement return point.
- The amount of code that needs to be duplicated to replace the PERFORM statement is deemed excessive (approximately 400 statements).

IPCMEMORY Option

```
$IPCMEMORY = integer-value
```

Type: Value
Default: 5000

This option controls the amount of memory you can declare to support the array used to process nested program calls at run time. Statistics printed on the output listing summary show the total memory value required for the array.

The specified size value should be reasonably close to the value of the memory required as reported in the summary statistics. If the IPCMEMORY size is not declared large enough to handle all the nested programs contained within the main program, then a compiler message is emitted. If the size value is specified to be larger than the known required array size to support 254 nested programs (74213 bytes), then the size is minimized by the compiler.

Example

The following example shows an output listing summary:

```
            $SET LIST IPCMEMORY=74214

  0001000    IDENTIFICATION DIVISION.

  000200     PROGRAM-ID. NEST-255.
  000300     ENVIRONMENT DIVISION.

  ------------------------------------------------------------------------
                                NESTED PROGRAM STATISTICS FOR     NEST-255
                                ----------------------------------------

  MEMORY ALLOCATED FOR PROGRAM  (X)              1537  99%  of     1538 BYTES
  MEMORY ALLOCATED FOR PROGRAM-IDS  (Y)          7629  99%  of     7650 BYTES
  MEMORY ALLOCATED FOR COMMON PROGRAMS  (Z)     64772  99%  of    65025 BYTES
  TOTAL MEMORY REQUIRED  (X+Y+Z)                73938  99%  of    74213 BYTES

  ACTUAL IPCMEMORY REQUESTED                    74213 BYTES

                  006100   *END PROGRAM NEST-255.
```

LEVEL Option

```
LEVEL = lexicographical-level
```

Type: Value
Default: 2

This option specifies the lexicographical level at which the program is compiled. Programs compiled at the default lexicographical level of 2 can only serve as the host file during a binding operation.

The lexicographical level must be an integer greater than 1 and less than or equal to 14. The level option must appear prior to the Identification Division.

LIBRARY Option

```
{  LIBRARY   }   [  { 'file-title'  } ]
{            }   [  {              } ]
{            }   [  { "file-title"  } ]
```

The file-title specifies an alternative file title for the code file. You can qualify the file-title with a usercode and/or location as long as a foreign host is not specified.

When the BINDSTREAM option is set, a file-title must be specified. Multiple LIBRARY options are allowed in a stacked program. Refer to "BINDSTREAM Option" for more information.

Type: Boolean Title
Default: FALSE

This option, when set, directs the compiler to generate a multi-procedure code file for binding. The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	A separate executable object code file that does not contain binding information is generated for each program unit found in the source file.
SET	TRUE	A single non-executable object code file that contains binding information for each program unit found in the source file is generated.

LIBRARY must be included in the source code before the Identification Division of the program. The resultant object code file can be bound to a host file through the use of the Binder utility.

LIBRARYLOCK Option

Type: Boolean
Default: FALSE

When TRUE, the LIBRARYLOCK option provides the locking needed to maintain private library data integrity.

Note: *This option has no effect unless used with the LIBRARYPROG compiler control option.*

For information on specifying the way a program is shared when it is called as a library, refer to the "SHARING" option.

LIBRARYPROG Option

Type: Boolean
Default: FALSE

This option is used to compile COBOL74-type libraries with COBOL85 code. The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	The resulting object code file must be executed as a stand-alone program, and cannot be executed by a library call from another program.
SET	TRUE	The resulting object code file cannot be executed as a stand-alone program, but can be executed only when called by another program.

LIBRARYPROG must be included in the source code before the Identification Division of the program. LIBRARYPROG must be set to TRUE in a program that is called by another program. However, LIBRARYPROG should not be used if that program is declared explicitly to be a library.

After it is set to TRUE, the option remains TRUE throughout compilation. If the source language input consists of separately compiled programs, be careful to ensure that this option is set appropriately for each program unit.

Note: *Using this option to compile COBOL85-type libraries results in a syntax error.*

LINEINFO Option

Type: Boolean
Default: TRUE for CANDE-originated compilations; otherwise, FALSE

This option controls whether source language sequence numbers are included in the object code file. Setting this option gives you the convenience of investigating lines of code by sequence number rather than by code address if your program terminates abnormally. Note that you must set or reset the LINEINFO option before program text.

If the LINEINFO option is set and the program terminates abnormally, the compiler displays the source language sequence number associated with the point of program termination. Sequence number 0 (zero) refers to special code generated by the compiler. The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	The compiler does not include source language sequence numbers in the object code file.
SET	TRUE	The compiler includes source language sequence numbers in the object code file.

LIST Option

```
LIST   [  =  Boolean-expression  ]
```

Type: Boolean
Default: FALSE for CANDE-originated compilations; otherwise, TRUE

This option controls whether the compiler creates a compilation listing of the program. The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	The compiler does not produce a compilation listing of the program.
SET	TRUE	The compiler produces a compilation listing of the program in a file named LINE. The minimum content of this listing includes the source language, a compilation summary, and any error messages.

Boolean-expression

Creates the file title according to an expression that can be evaluated as a Boolean value according to the standard rules of Boolean algebra. For details about the syntax of Boolean-expression, refer to "Syntax for Compiler Control Options" in this section.

LISTDOLLAR Option

```
LISTDOLLAR   [ = Boolean expression ]
```

Type: Boolean
Default: FALSE

This option controls whether temporary CCRs are included in the compilation listing of the program (the file named LINE). Temporary CCRs use a $ (currency sign) in column 7, and no $ (currency sign) in column 8. The following table shows the effects of this option:

Option Setting	Value	Description
RESET	FALSE	The compiler does not include temporary CCRs in the compilation listing of the program.
SET	TRUE	If LIST is TRUE, the compiler includes all temporary CCRs in the compilation listing of the program.

If you use the currency sign to specify this compiler option, place it in columns 9 through 72. The synonym LIST$ is no longer valid, and if specified causes erroneous results.

LISTINCL Option

```
{ LISTINCL  }
{           } [ = Boolean-expression ]
{ INCLLIST  }
```

Type: Boolean
Default: FALSE

This option controls whether the compiler places source language records included with the INCLUDE option in the compilation listing of the program (the file named LINE). The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	The included records are not contained in the compilation listing.
SET	TRUE	If the LISTINCL option is TRUE, then the compilation listing contains source language records included through use of the INCLUDE option.

If the LIST option is FALSE, then the LISTINCL option is ignored.

LISTINITIALCCI Option

Type: Boolean
Default: FALSE

This option controls whether the compilation listing includes the contents of the INITIALCCI file used for the compilation (the file named LINE). The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	The contents of the INITIALCCI file are not included in the compilation listing.
SET	TRUE	If the LIST option is TRUE, then the compilation listing includes the contents of the INITIALCCI file.

If the LIST option is FALSE, then the LISTINITIALCCI option is ignored.

LISTIPCMEMORY Option

Type: Boolean
Default: TRUE

The nested program statistics for each nested program are included in the compilation listing by default.

To suppress the nested program statistics from the list output, RESET the LISTIPCMEMORY option.

Include the LISTIPCMEMORY option in the source code only once, before the Identification Division of the first nested program. This action will SET/RESET the printing of the nested program statistics for all the nested programs in the source code.

For more information, refer to the IPCMEMORY option earlier in this section.

LISTOMITTED Option

```
{ LISTOMITTED }
{ LISTO       }
```

Type: Boolean
Default: FALSE

This option controls whether the compiler includes in the compilation listing any source language input that was omitted because of the OMIT option. More information on the OMIT option is included later in this subsection. The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	The compilation listing does not include source language records omitted through use of the OMIT option.
SET	TRUE	The compilation listing includes source language records omitted through use of the OMIT option.

LISTP Option

Type: Boolean
Default: FALSE

This option controls whether the compiler lists source language records that originate from the primary input file. The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	The compiler does not generate a listing of source language records that originated from the primary input file.
SET	TRUE	The compiler generates a listing of source language records that originated from the primary input file.

If the LIST option is TRUE, the LISTP option has no effect.

LIST1 Option

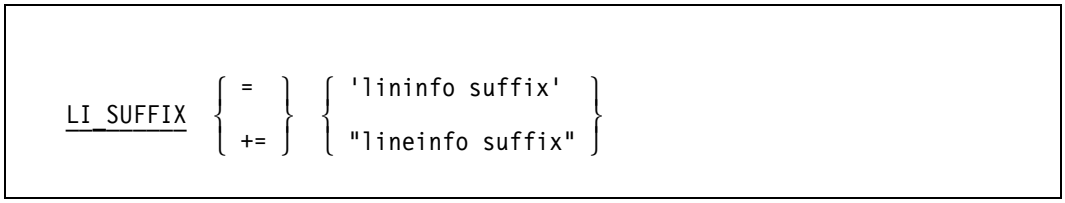
Type: Boolean
Default: FALSE

This option controls whether the compiler produces a listing during the first pass. The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	The compiler does not generate a listing during the first pass through the source input records.
SET	TRUE	The compiler generates a listing during the first pass through the source input records.

If the LIST option is TRUE, the LISTP option has no effect.

LI_SUFFIX Option



Type: String

This option specifies a string to be associated with the sequence numbers of the program. The syntax LI_SUFFIX = "lineinfo suffix" or LI_SUFFIX = 'lineinfo suffix' assigns a string of characters to the lineinfo string. The syntax LI_SUFFIX += "lineinfo suffix" or LI_SUFFIX += 'lineinfo suffix' appends a string of characters to a previously defined lineinfo string. This option is meaningful only when the LINEINFO compiler option is set to TRUE.

If a program failure occurs and the LINEINFO compiler option was set when the program was compiled, a series of sequence numbers are displayed identifying the sequence number at which the program or programs failed. If the LI_SUFFIX compiler option was used, the string associated with the program is displayed following the sequence number. This permits easy identification of the program or library associated with the displayed sequence number.

For best results, the LI_SUFFIX option should be set within a program before any nested programs are specified. For nested programs, this means after IDENTIFICATION DIVISION and before END PROGRAM.

LOCALTEMP Option

Type: Boolean
Default: True

The LOCALTEMP option specifies where temporary arrays are to be created by the compiler.

Allocating temporary arrays locally within a program is ideal for optimizing subprogram memory usage and reducing the use of lexicographic level 2 stack cells. There is, however, a performance penalty when local temporary arrays are used.

Each time a program containing a local array is entered and the array is first used, the system performs a p-bit interrupt to allocate space in memory for the local array and assign it to the program. The time required to enter and exit a program containing a local array is roughly twenty times longer than the time required to enter and exit the same program without the local array. Thus, local temporary arrays should be avoided in subprograms that are expected to be entered very frequently.

The LOCALTEMP option defaults to TRUE, causing temporary arrays to be allocated locally in the program. Resetting this option to FALSE causes the temporary arrays to be allocated globally at lexicographic level 2.

This option must be included in the source code before the Identification Division of the program.

LOCALTEMPWARN Option

Type: Boolean
Default: False

The LOCALTEMPWARN option enables the compiler to emit the following warning against a statement when the statement causes the compiler to generate a local array temporary.

```
A LOCAL ARRAY TEMPORARY HAS BEEN GENERATED FOR THIS STATEMENT, WHICH MAY
CAUSE AN INITIAL PBIT TO OCCUR. TO AVOID A PERFORMANCE PROBLEM CAUSED BY THE
PBIT, RESET THE LOCALTEMP CCI OR MODIFY THE STATEMENT.
```

LONGLIMIT Option

```
LONGLIMIT     =     limitvalue
```

Type: Value
Default: 10922 (Equivalent to 64KB)

This option specifies the maximum permissible size in words of an unpaged array.

Unpaged arrays (sometimes called LONG arrays) provide greater efficiency in accessing elements at the expense of increased actual memory usage at any given instant. Arrays that are less than or equal to the LONGLIMIT are not paged, while all other arrays are paged.

The limitvalue must be in the range from 171 words through 10922 words. In addition, an individual site may have a system-wide LONG array limit that is less than the LONGLIMIT setting. Setting the LONGLIMIT greater than the site limit causes the object program to abnormally terminate at run-time.

MAPONELINE Option

Type: Boolean
Default: FALSE

This option controls whether the MAP information should be on one line in the output listing. The MAP option must be set together with the MAPONELINE option.

If the default (FALSE) is used, the MAP option is not affected.

MAP or STACK Option

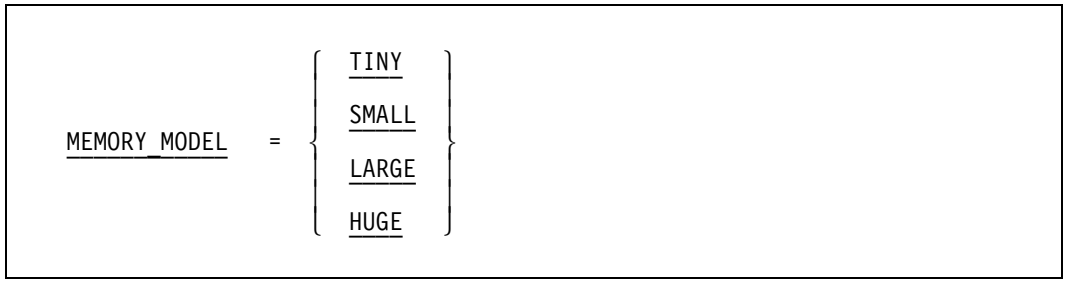
Type: Boolean
Default: FALSE

This option controls whether the compiler includes information on variable allocation in the output listing. The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	The output listing generated by the compiler does not contain any information concerning the allocation of variables.
SET	TRUE	The output listing generated by the compiler contains information concerning the allocation of variables in the object program.

MAP and STACK are synonymous.

MEMORY_MODEL Option



Type: Enumerated
Default: TINY

The MEMORY_MODEL option is not currently used by the COBOL85 compiler. This syntax is provided for use when binding a COBOL85 program with a C language program. Refer to the *C Programming Reference Manual* for more information.

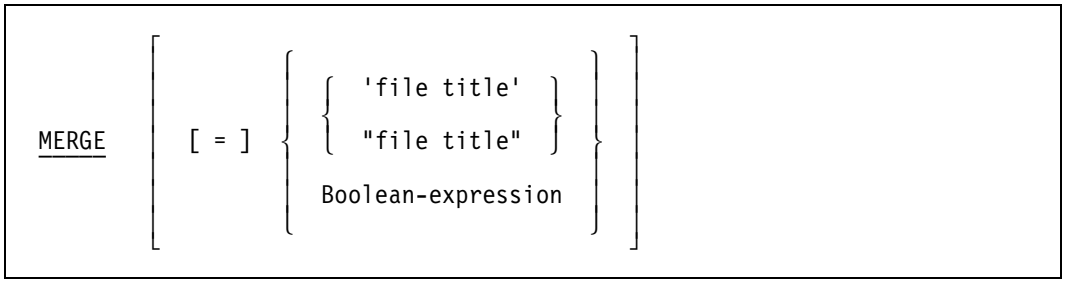
MERGE Option

Type: Boolean Title
Default: FALSE

This option controls whether the compiler merges the source language records of the primary input file (CARD) with source language records from a secondary input file (SOURCE). This option must be included in the source code before the Identification Division of the program. The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	The compiler does not merge primary source records (CARD) with secondary source records (SOURCE). The compiler ignores the secondary input file (SOURCE).
SET	TRUE	The compiler merges primary source records (CARD) with secondary source records (SOURCE). After MERGE is set to TRUE, it remains TRUE throughout compilation; any attempt to change it is treated as an error and ignored.

Syntax



file title

Specifies an alternate file title for the secondary input file, SOURCE. The file title must be a string. The device name can be DISK or TAPE.

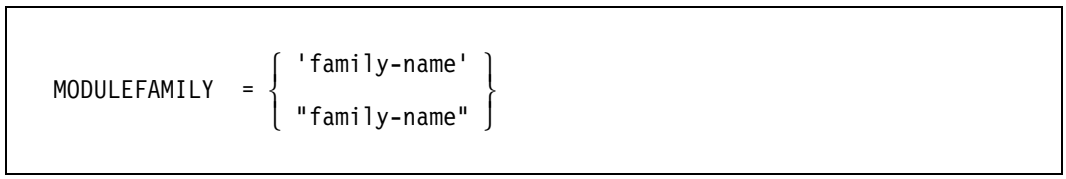
If a file title is not specified, the name "SOURCE" is assumed. File equate statements can override the default values.

Boolean-expression

Creates the file title according to an expression that can be evaluated as a Boolean value according to the standard rules of Boolean algebra.

For details about the syntax of Boolean-expression, refer to "Syntax for Compiler Control Options" in this section.

MODULEFAMILY Option



Type: String

This option specifies a default family-name to be used with the CALL MODULE statement.

MODULEFILE Option

Type: String

This option specifies the file name to be used when the MODULEFILE option is used in the CALL MODULE statement.

MUSTLOCK Option

Type: Boolean
Default: TRUE

This option enables you to specify whether the compiler is to generate code to ensure that a record has been locked by the current user before attempting to write to the record. Set this option to TRUE to enhance file integrity when multiple programs access a file concurrently.

When the MUSTLOCK option is set to TRUE, and the phrase "VALUE OF BUFFERSHARING IS SHARED" is declared in the File Description (FD) Entry for the file, code is emitted before a write operation to verify that the user has locked the record that is to be written. If the record is not locked,

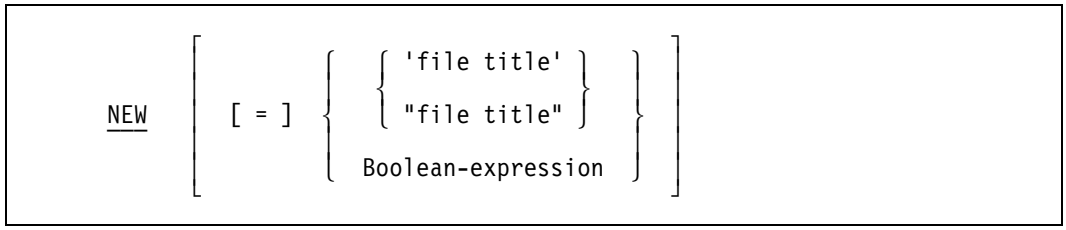
- The write operation does not occur.
- A file status value is established.
- Control passes to the statement specified in the INVALID KEY clause.

Related Information

The following table provides references for additional information related to this statement:

For Information About . . .	Refer To . . .
Locking a record	"LOCKRECORD Statement" in Section 7.
Unlocking a record	"UNLOCKRECORD Statement" in Section 8.
Performing write operations on shared files	"WRITE Statement" in Section 8.

NEW Option



Type: Boolean Title
Default: FALSE

This option controls whether the compiler creates a new source language symbolic file (NEWSOURCE). This option must be included in the source code before the Identification Division of the program. The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	The compiler does not create a new source language symbolic file.
SET	TRUE	The compiler creates a new source language symbolic file (NEWSOURCE). The new symbolic file contains all source language records used during compilation. After NEW is set to TRUE, it remains TRUE throughout compilation. Any attempt to change this option is treated as an error and ignored.

Syntax

file title

Specifies an alternate file name for the new source language symbolic file. The file title must be a string. If no file title is specified, the name NEWSOURCE is assumed. A file equate statement can override this default value.

Boolean-expression

Creates the file title according to an expression that can be evaluated as a Boolean value according to the standard rules of Boolean algebra.

For details about the syntax of Boolean-expression, refer to "Syntax for Compiler Control Options" in this section.

Source language records discarded by the DELETE or VOID options are not included in the NEWSOURCE file. Input records omitted by the OMIT option and permanent CCRs (with a $ in columns 7 and 8) **are** included.

NEWID Option

```
NEWID [ = ] new-id string
```

Type: String

This option specifies a string of characters to be placed by the compiler in the rightmost eight character positions of each source language record.

The columns 73-80 in the NEWSOURCE and the listing are replaced by this NEWID.

NEWSEQERR Option

Type: Boolean
Default: FALSE

This option controls whether sequence errors in the new source language symbolic file (NEWSOURCE) cause the compiler not to lock the NEWSOURCE file upon compilation completion. A sequence error occurs when the sequence number of a record of the NEWSOURCE file is not greater than the sequence number of the preceding record.

If the NEW option is FALSE, then the NEWSEQERR option is ignored.

The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	The compiler locks the new source language symbolic file regardless of whether sequence errors were encountered while writing the file.
SET	TRUE	The compiler does not lock the new source language symbolic file (NEWSOURCE). A message is displayed on the Operator Display Terminal (ODT) and is printed on the printer listing.

OMIT Option

Type: Boolean
Default: FALSE

This option controls whether the compiler ignores all source language records, except for other compiler control records. The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	The compiler uses all source language records in compilation.
SET	TRUE	The compiler ignores all source language records from the primary source file (CARD) and, if the MERGE option is TRUE, from the secondary source file (SOURCE). Ignored records are not used in compilation.

The OMIT option can appear on a CCR in either the primary (CARD) or the secondary(SOURCE) source language input. While the OMIT option is TRUE, CCRs encountered in the source language input are processed in the normal fashion.

If the NEW option is TRUE, the omitted records **are** carried forward to the output symbolic file (NEWSOURCE). If the LISTOMITTED option is TRUE, the records are included in the compilation listing (LINE). Otherwise, the records are not included.

OPT1 Option

Type: Boolean
Default: FALSE

This option specifies whether a numeric data item in the Working-Storage Section is treated internally as a binary item for the performance gain purpose. The following use of the OPT1 option sets the internal run-time performance tuning usage of the compiler:

```
$SET  OPT1
 01   ARITH-DATA.
      03  OP-1       PIC 9(11).
      03  OP-2       PIC 9(20).
$RESET  OPT1
```

The OPT1 option must not appear before the Identification Division in the source program. The OPT1 option affects only integer items of DISPLAY, COMPUTATIONAL, or PACKED-DECIMAL usage that are declared in the Working-Storage Section.

The total number of data items that are optimized with the OPT1 option cannot exceed 256; otherwise, a table overflow error message is displayed to the user.

The numeric data item affected by the OPT1 option is referenced in the Procedure Division only in a numeric context. Thus, the numeric data item can be the operand of an arithmetic statement or it can be used as a subscript. However, the numeric data item cannot be used as an item of a STRING statement.

An integer item with the OPT1 option set is mapped internally to a compiler- created integer stack cell, in addition to its original EBCDIC (for DISPLAY) or HEX (for COMPUTATIONAL or PACKED-DECIMAL) field. The hidden integer stack cell is in the format of a single word, if the picture size is 9(11) or less, or in the format of a double word, if the picture size is greater than 9(11).

Whenever the OPT1 integer item is used in a statement (limited to numeric context), the hidden integer stack cell is used directly without updating the original field.

If a data item that contains or overlaps an OPT1 data item is used in a statement (limited to the MOVE statement only), one of the following occurs:

- If the data item is the source, there is an implicit move from the hidden cell to the original field immediately before the move of the data item.
- If the data item is the destination, there is an implicit move from the original field to the hidden integer stack cell immediately after the move.

OPT2 Option

Type: Boolean
Default: FALSE

This option specifies that a COBOL85 source program consists entirely of uppercase characters. Programs that are in uppercase characters only will compile faster due to the overhead that is eliminated by the compiler not having to search for lowercase characters and transform the lowercase characters to uppercase.

You must place the OPT2 option anywhere in the source after the Identification Division header line. An OPT2 option placed before the Identification Division header line is ignored.

If the OPT2 option is set and lowercase letters appear in the source, the error message "Illegal Character" is issued for each lowercase letter that is found.

OPT3 Option

Type: Boolean
Default: FALSE

This option enables you to declare a constant in the Data Division. For example, the following use of the OPT3 option sets the data item CONST-1:

```
$SET  OPT3
 01   CONST-1    PIC 999 VALUE 1.
$RESET  OPT3
```

When OPT3 is set, the data item CONST-1 is treated as a constant rather than as a regular data item. Thus, CONST-1 always has the value of 1, and can be used only as a sending-only (or reference-only) operand, and **not** as a receiving operand. This will improve run-time performance.

OPT3 can be applied to any data category; however, you must ensure that the data is not used as a receiving operand.

Though the main use of the OPT3 option is for elementary data items, OPT3 can also be used for alphanumeric group data items that contain only nested group items and FILLER alphanumeric data items with VALUE clauses. This enables the INSPECT...CONVERTING statement to identify the data items as translate-table constants and to avoid generating a run-time call to build the translate table. Following is an example.

```
$ SET OPT3
 01  XLATE-IN.
     05  FILLER  PIC X(128) VALUE
         @000102030405060708090A0B0C0D0E0F101112131415161718191A1B
-        @1C1D1E1F202122232425262728292A2B2C2D2E2F3031323334353637
-        @38393A3B3C3D3E3F404142434445464748494A4B4C4D4E4F50515253
-        @5455565758595A5B5C5D5E5F606162636465666768696A6B6C6D6E6F
```

```
-        @70717273747576777879 7A7B7C7D7E7F@.
     05  FILLER  PIC X(128) VALUE
         @80818283848586878889 8A8B8C8D8E8F909192939495969798999A9B
-        @9C9D9E9FA0A1A2A3A4A5A6A7A8A9AAABACADAEAFB0B1B2B3B4B5B6B7
-        @B8B9BABBBCBDBEBFC0C1C2C3C4C5C6C7C8C9CACBCCCDCECFD0D1D2D3
-        @D4D5D6D7D8D9DADBDCDDDEDFE0E1E2E3E4E5E6E7E8E9EAEBECEDEEEF
-        @F0F1F2F3F4F5F6F7F8F9FAFBFCFDFEFF@.
 01  XLATE-OUT.
       :
$ RESET OPT3
```

OPT4 Option

Type: Boolean
Default: FALSE

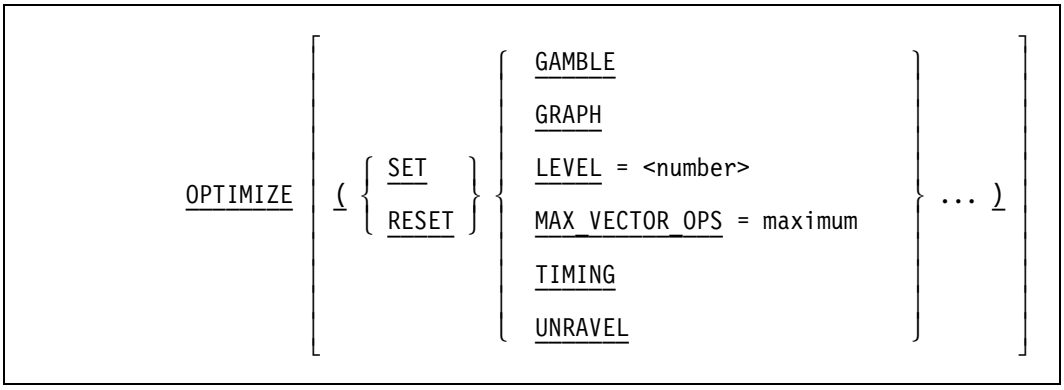
This option specifies whether or not the compiler is enabled to use a word copy descriptor optimization for short character data items. The optimization is permitted when the option has the default value of FALSE.

This optimization can provide a performance boost and is not an issue under normal conditions. However, for some COBOL subprograms generated in Enterprise Application Environment (formerly LINC), this optimization can cause problems. The OPT4 option is provided to disable the optimization in these cases.

Example

```
$SET OPT4
 01  MYDATA  COMMON.
     03  MD-ALFA1  PIC X(10).
     03  MD-ALFA2  PIC X(1).
     03  MD-ALFA2  PIC X(2).
$RESET OPT4
```

OPTIMIZE Option



Type: Boolean class
Default: FALSE

This option controls the optimizations performed before code generation. The compiler examines this option at the beginning of each procedure to determine the optimizations to be performed on that procedure. Thus, although the setting of the OPTIMIZE option can be changed at any time, only its setting at the beginning of the procedure is significant.

Because this is a Boolean class option, setting the option sets all of the suboptions, and resetting the option resets all of the suboptions. You can also set and reset each suboption individually. For details, refer to "Boolean Class Compiler Options" earlier in this section.

The six subordinate options available within the OPTIMIZE option are described as follows:

GAMBLE

This suboption allows the compiler to make certain assumptions to perform the following optimizations:

- The associative law applied to floating-point types (that is, changing a division to a multiplication by a reciprocal)
- The associative law applied to integer types (that is, changing a subtraction to an addition by an inverse)
- Conditionally executed invariant expressions moved outside of loops
- Indexes assumed to be within bounds
- Variable "strides" assumed to be positive

This option should be reset if the COBOL85 program utilizes display data items containing undigits or data with zones other than hex F.

GRAPH

This option causes a graph of the optimized procedure to be written to the output listing.

LEVEL

This option controls the amount of effort expended by the compiler in optimizing a procedure. In general, the higher the level, the greater the optimization effort. Higher levels tend to yield reduced run time at the expense of increased compilation time. The level must range from 0 to 10.

MAX_VECTOR_OPS = maximum

This option enables you to specify the maximum number of vector operators allowed per statement. The default is 3. If the target machine has vector operators and $OPTIMIZE is set, the compiler generates vector operators. A single source statement can cause multiple vector operators to be generated. Whether executing multiple vector operators is faster than the original loop depends on the number of iterations of the loop and the particular vector operators involved.

TIMING

This option causes statistics regarding the optimization phase of the compilation to be gathered and written to the output listing.

UNRAVEL

This option allows loops to be unraveled and certain functions to be generated as in-line code.

Examples

```
OPTIMIZE (SET GAMBLE UNRAVEL, RESET LEVEL)

OPTIMIZE (SET GAMBLE, RESET LEVEL, SET UNRAVEL)
```

OPTION Option

```
OPTION  { ( [option action] option name ... ) }
```

Type: Boolean
Default: FALSE

This option declares a user-defined compiler control option. A user option can be manipulated exactly like any other Boolean option using the SET, RESET and POP option actions. Also, it can be used in option expressions to assign values to standard Boolean options or to other user options.

An initial setting for the user option can be specified by using the SET, RESET, and POP option actions within the parentheses. For example, the following declares the user options SECURITY1 and SECURITY2 and sets the initial value of SECURITY1 to TRUE, and the initial value of SECURITY2 to FALSE:

```
$ SET OPTION (SET SECURITY1 RESET SECURITY2)
```

If an initial value is not specified, the initial value is supplied by the option action associated with the OPTION compiler option. For example, the preceding example could have been specified using either one of the following formats:

Example Format 1

```
$ SET OPTION (SECURITY1 RESET SECURITY2)
```

Example Format 2

```
$ RESET OPTION (SECURITY2 SET SECURITY1)
```

OWN Option

Type: Boolean
Default: FALSE

This option specifies whether data items in the Working-Storage Section are to assume the declaration OWN. The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	All data items in the Working-Storage Section except those items explicitly declared COMMON or OWN are considered to be LOCAL data items.
SET	TRUE	All data items in the Working-Storage Section except those items explicitly declared COMMON or LOCAL are considered to be OWN data items.

The OWN option and the COMMON option cannot both be TRUE.

The OWN option is ignored if the compilation is at lexicographical level 2.

PAGE Option

Type: Immediate

This option directs the compiler to begin printing on a new page in the output compilation listing (LINE).

The PAGE option is ignored if the LIST option is FALSE.

If the OMIT option is TRUE and the LISTOMITTED option is FALSE, the PAGE option is ignored.

PAGESIZE Option

```
PAGESIZE     =     pagesize value
```

Type: Value
Default: 58

This option specifies the number of lines printed on each page of the output compilation listing (LINE).

The PAGESIZE option is ignored if the LIST option is FALSE.

PAGEWIDTH Option

```
PAGEWIDTH     =     pagewidth value
```

Type: Value
Default: 132

This option specifies the number of characters printed on each line of the output compilation listing (LINE).

The PAGEWIDTH option is ignored if the LIST option is FALSE.

RPW (Report Writer) Option

Type: Boolean
Default: FALSE

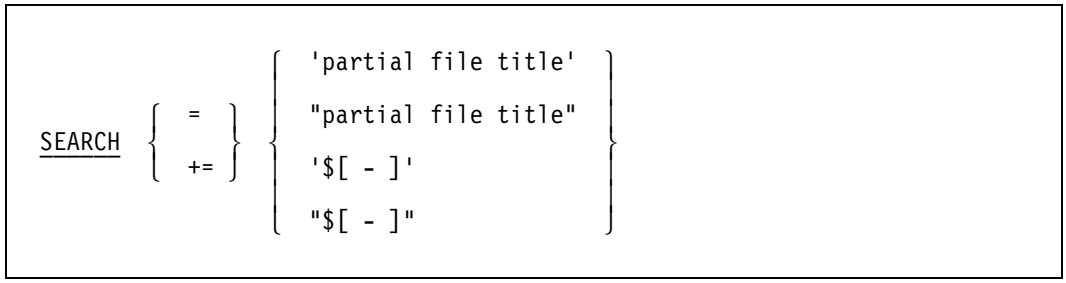
RPW controls whether the compiler identifies COBOL language constructs that are Report Writer elements as determined by the ANSI Standards Committee. The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	The compiler does not print warning messages identifying RPW elements.
SET	TRUE	The compiler prints warning messages to identify RPW elements. Warning messages are not printed if the WARNSUPR option is TRUE.

SDFPLUSPARAMETERS Option

This option is for internal use in the system software only.

SEARCH Option



Type: String

This option specifies partial file names to be used when searching for included files. The partial file names represent alternate file directories under which the included file might reside. The syntax SEARCH = "partial file title" or 'partial file title' assigns a string of characters to the list of partial file names. The syntax SEARCH += "partial file title" or SEARCH += 'partial file title' appends a string of characters to a previously defined list of partial file names.

The following table describes two special constructs for specifying partial file names in the SEARCH option:

Construct	Meaning
$	The compiler replaces this with the name of the primary source input file (CARD) followed by the characters /=. For example, if the primary source input file is named UTILITY/ACCOUNT/FIXUP, the partial file name UTILITY/ACCOUNT/FIXUP/= is added to the list of partial file names used in searching for included files.
$-	The compiler replaces this with a partial name of the primary source input file (CARD) followed by the characters /=. The partial name is formed by removing the last file name node of the primary source input file. For example, if the primary source input file is named UTILITY/ACCOUNT/FIXUP, the file name node FIXUP is removed, and the partial file name UTILITY/ACCOUNT/= is added to the list of partial file names used in searching for included files.

The partial file titles specified by the SEARCH compiler option are used to form the file titles searched for when the following condition exists:

- An INTNAME is not specified in the INCLUDE option, or if one is specified, a file equation has not been applied to the INTNAME.

When this condition exists, the compiler performs the following actions on the TITLE file attribute of the included file:

- Translates all lowercase characters to their uppercase equivalents.
- Replaces all backslashes (\) and periods (.) with slashes (/).

If partial file names have been specified with the SEARCH option, these partial file names are combined with the TITLE file attribute of the included file to form file titles. The compiler then searches for the included file using these generated file titles. For example, a program might contain the following CCRs:

```
$ SET SEARCH = "SYSTEM/= ON LOCALPACK; (USER)SOURCE/="
$ SET SEARCH+= ";*SYSTEM/= ON EXTERNALS"
              .
              .
              .
$ INCLUDE DEFINES = "LOCAL/DEFINES."
```

If a file title has not been applied to the file identified by the internal name "DEFINES" through file equation, the compiler attempts to find the file by combining the partial file names specified by the SEARCH option with the TITLE file attribute specified by the INCLUDE option. The following file titles would be searched for in the order listed. The first file title to match the file title of an existing file is used as the file title of the included file.

```
SYSTEM/LOCAL/DEFINES ON LOCALPACK
(USER)SOURCE/LOCAL/DEFINES
*SYSTEM/LOCAL/DEFINES ON EXTERNALS
```

If the file is not present under any of these names, the compiler attempts to use the TITLE file attribute itself to find the file. If this fails, the compiler attempts a file OPEN operation to change the status of the compilation to WAITING, and displays a NO FILE message. For more information, refer to "INCLUDE Option" earlier in this section.

SEPARATE Option

Type: Boolean
Default: FALSE

This option controls the resultant object code file produced by the compiler. The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	A separate executable object code file that does not contain binding information will be generated for each program unit found in the source file.
SET	TRUE	A separate non-executable object code file that contains binding information will be generated for each program unit found in the source file.

SEPARATE must be included in the source code before the Identification Division of the program. When this option is set, the resultant object code file can be bound to a host file through use of the Binder utility.

SEQUENCE or SEQ Option

Type: Boolean
Default: FALSE

This option controls whether new sequence numbers are generated for the new source language symbolic file (NEWSOURCE). New sequence numbers do not affect the compilation listing file (LINE). The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	The compiler does not generate new sequence numbers.
SET	TRUE	The compiler assigns the current sequence base to the current source language record and increments the sequence base by the sequence increment. If the resulting sequence base exceeds 999999, the compiler disables the SEQ option, and produces a sequence error.

The sequence base and sequence increment used by the compiler when assigning new sequence numbers can be specified using the Sequence Base option and Sequence Increment option described next.

Sequence Base Option

```
sequence number  [ + sequence increment ]
```

Type: Value
Default: 100

This option specifies the sequence base used by the compiler when the SEQUENCE option is TRUE. The compiler uses the specified sequence base with the sequence increment to assign new sequence numbers to source language records. The sequence increment is specified using the Sequence Increment option described next.

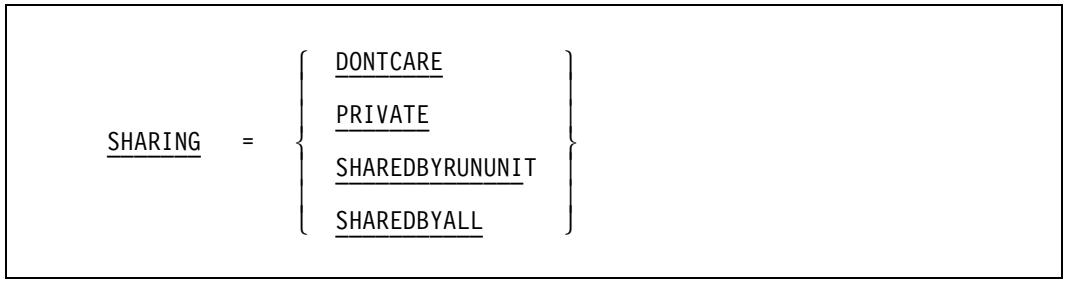
Sequence Increment Option

```
+ sequence increment
```

Type: Value
Default: 100

This option specifies the sequence increment used by the compiler when the SEQUENCE option is TRUE. The compiler uses the specified sequence increment with the sequence base to assign new sequence numbers to source language records. The sequence base is specified using the Sequence Base option described earlier in this section.

SHARING Option



Type: Enumerated
Default: SHAREDBYRUNUNIT

This option specifies how programs that call this library share access to this library. The SHARING option must be included in the source program before the Identification Division.

Note: *This option has no effect unless used with the LIBRARYPROG compiler option.*

The following table describes the available settings for the SHARING option:

Option Setting	Description
DONTCARE	The operating system determines the sharing.
PRIVATE	A copy of the library is invoked for each user (calling program). Any changes made to global items in the library by the actions of the user are visible only to that user of the library.
SHAREDBYRUNUNIT	All invocations of the library within a run unit share the same copy of the library. The term run unit as used here refers to a program and all the libraries that are initiated either directly or indirectly by that program. Note that this definition differs slightly from the COBOL ANSI-85 definition of run unit as described in Section 8.
SHAREDBYALL	All simultaneous users share the same instance of the library.

If the library is called by a COBOL74 or COBOL85 program, the library services only one user at a time, regardless of the value of the sharing option. If you have a complex environment where multiple libraries are linked together and you are using a COBOL74 type library, set the SHARING option to PRIVATE and the LIBRARYLOCK option to TRUE to ensure data integrity.

SHOWOBSOLETE Option

Type: Boolean
Default: FALSE

This option controls whether the compiler identifies COBOL language elements that are considered obsolete by the ANSI standards committee. The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	The compiler does not print warning messages identifying obsolete COBOL language elements.
SET	TRUE	The compiler prints warning messages identifying obsolete COBOL language elements. These warning messages are not printed if the WARNSUPR option is TRUE.

SHOWWARN Option

```
SHOWWARN  [ = Boolean expression]
```

Type: Boolean
Default: FALSE

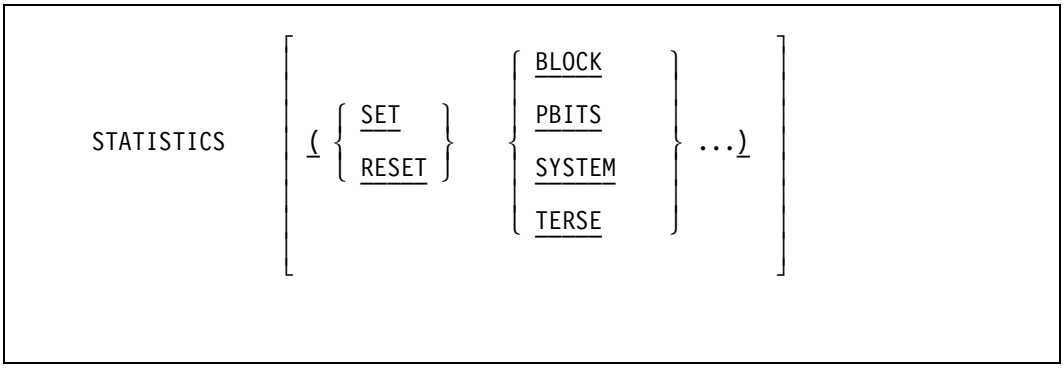
This option controls whether the compiler issues warnings and error messages to the CANDE terminal during compilation. The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	The compiler lists only error messages to the CANDE terminal.
SET	TRUE	The compiler lists warnings as well as error messages to the CANDE terminal.

STACK Option

This is a synonym for the MAP compiler option. See the "MAP or STACK Option" earlier in this section.

STATISTICS Option



Type: Boolean
Default: False (RESET) for STATISTICS option, False (RESET) for BLOCK, False (RESET) for PBITS, False (RESET) for SYSTEM, and False (RESET) for TERSE

This option causes timing statistics to be gathered for each paragraph. The option is examined at the beginning of each paragraph and, if enabled, statistics are gathered for that paragraph. Although the option setting can be changed at any time, only the setting at the beginning of a paragraph is significant. At program termination, the statistics information is printed out to the TASKFILE.

The statistics include the number of times the nested program is called and the amount of time spent in the nested program, both inclusive and exclusive of the amount of time spent in the nested programs called by that nested program.

The STATISTICS option can be used with individual subprograms of a bound program. When the bound program terminates, statistics are printed independently for each subprogram compiled with the option.

The four suboptions available within the STATISTICS option are described as follows:

BLOCK

If the BLOCK suboption is set, statistics are gathered for each execution path in the code. These statistics include the number of times each execution path is executed. The sequence number that is listed is the start of an execution path. All sequence numbers within the code that are not listed are included in the most recently listed sequence number, since these lines of code share the same execution path. Complex statements can be composed of several different execution paths, which are represented by the same sequence number being listed more than once.

PBITS

If the PBITS suboption is set, statistics are gathered about the initial pbits occurring in the program.

SYSTEM

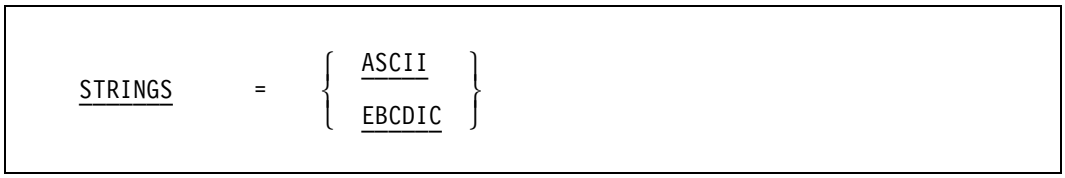
If the SYSTEM suboption is set, SLICESUPPORT functions and MCP calls are tracked separately. These items are preceded by "S:" and "M:" as follows:

- M: -- Indicates that the item is an MCP function that was added by the compiler.
- S: -- Indicates that the item is a SLICESUPPORT function that was added by the compiler.

TERSE

If the TERSE suboption is set, paragraphs that are not called are not listed in the statistics output.

STRINGS Option



Type: Enumerated
Default: EBCDIC

This option specifies the default character type used for the compilation. The following table describes the effects of this option.

Option Setting	Description
ASCII	Sets the default character type to ASCII
EBCDIC	Sets the default character type to EBCDIC

The default character type is assumed for all strings when a character type has not been explicitly specified. The default character type is also used as the default value of the INTMODE file attribute.

Unless overridden by the ASCII option or the STRINGS = ASCII option, the default character type is EBCDIC.

The ASCII option is a synonym for STRINGS = ASCII.

STRICTPICTURE Option

Type: Boolean
Default: FALSE

This option determines whether the compiler enforces a strict interpretation of the ANSI rules for PICTURE character string formation.

Option Setting	Value	Description
RESET	FALSE	Allow non-ANSI PICTURE strings
SET	TRUE	Disallow non-ANSI PICTURE strings

If this option is SET, the compiler allows only those symbols and combinations of symbols that are allowed by ANSI to comprise a PICTURE character string. The compiler issues syntax errors for those symbols and combinations of symbols that do not conform to ANSI standards. Neither of the Unisys extensions to simple insertion editing, Manual insertion editing and Automatic insertion editing, is allowed in this case.

If this option is RESET, the compiler allows non-ANSI symbols in PICTURE character strings. The choice of whether Automatic insertion editing or Manual insertion editing is allowed in this case is determined by the setting of the AUTOINSERT option.

STRICTPICTURE and AUTOINSERT cannot both be set to true at the same time.

SUMMARY Option

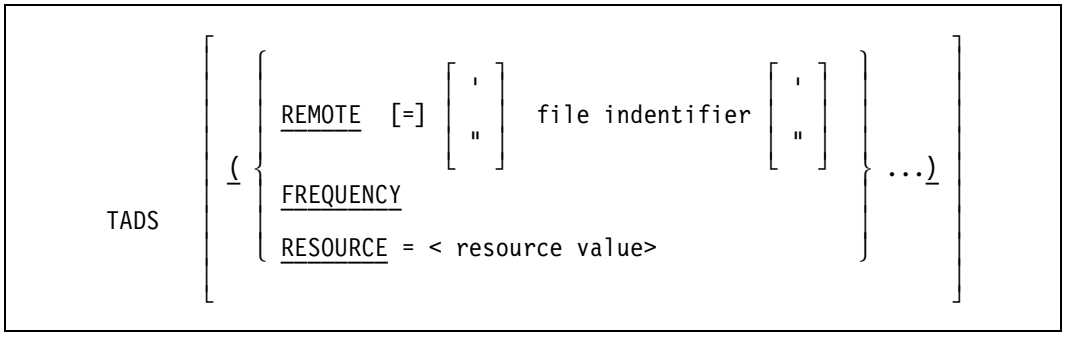
Type: Boolean
Default: FALSE

This option controls whether the compiler produces a summary listing containing information about the compilation. The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	The compiler omits the summary listing.
SET	TRUE	The compiler produces a summary listing containing the source language records used for the compilation, any error messages produced by the compilation, and a summary of the level of usage of some of the internal tables of the compiler.

TIME is a synonym for SUMMARY.

TADS Option



Type: Boolean class
Default: FALSE

When the TADS option is set to TRUE, special debugging code and tables are generated as part of the object program. The tables are generated to support the symbolic debugging environment of the COBOL85 Test and Debug System (TADS). For more information on TADS, refer to the *COBOL ANSI-85 Test and Debug System (TADS) Programming Reference Manual*.

The TADS option must be set before the first source statement or declaration of a program.

The three suboptions available within the TADS option are described as follows:

REMOTE

The REMOTE suboption enables TADS to share a REMOTE file with the program being tested. Sharing a file might be necessary because only one REMOTE input file can be open for each station. The file must have been assigned to REMOTE, and must be opened INPUT-OUTPUT. The record size must not be less than 72.

FREQUENCY

The FREQUENCY suboption enables TADS to accept the test coverage and frequency analysis commands during a test session. The test coverage and frequency analysis commands provide statistics on the execution of specified statements. The commands are: CLEAR, COVERAGE, FREQUENCY, MERGE and SAVE.

The CLEAR, COVERAGE, and FREQUENCY commands are accepted when the TADS compiler control option is specified as follows:

```
$ SET TADS (FREQUENCY)
```

Refer to the *COBOL ANSI-85 Test and Debug System (TADS) Programming Reference Manual* for detailed information on the test coverage and frequency analysis commands.

RESOURCE

The RESOURCE suboption determines the amount of resources to be used for TADS conditions. Increasing the <resource value> should improve the performance of TADS conditions, while causing more D1 stack cells and saved memory to be used.

The <resource value> can range from 20 to 2000 and has a default value of 100. The RESOURCE suboption should appear before any program text. Bound programs must have identical values for the RESOURCE suboption.

TARGET Option

```
TARGET = target-1 [ ( target-2 [ ,target-3 . . .] ) ]
```

Type: Enumerated
Default: Installation-defined

This option designates a specific computer system or group of systems as the target for which the generated object code is to be optimized. This option can be used to specify all machines on which the code file needs to run.

TARGET must appear in the source before the Identification Division of the program.

Specification of a secondary target is optional. If specified, a secondary target must be enclosed in parentheses. If more than one secondary target is specified, then the additional targets must be separated from each other by a comma and the entire list must be enclosed in parentheses.

If a secondary target is specified, the compiler does not generate any operators that are valid for the system or systems identified by the primary target but invalid for the system or systems identified by the secondary target.

See the COMPILERTARGET system command in the *System Commands Operations Reference Manual* for a complete list of the target values that are allowed.

Examples

```
TARGET=THIS
```

The compiler optimizes the object code file for the system on which it is compiled.

```
TARGET=THIS (ALL)
```

The compiler optimizes the object code file for the system on which it is compiled, but it does not generate any operators that are invalid for other machines.

TEMPORARY Option

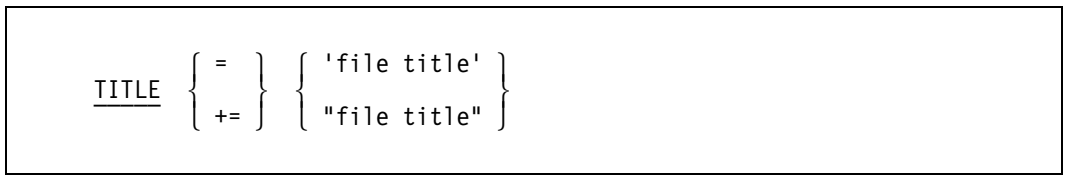
Type: Boolean
Default: FALSE

This option controls whether the object program, when called as a library, functions as a temporary or permanent library and is provided to ease migration from COBOL74 to COBOL85. The TEMPORARY compiler option should not be used if a program is declared explicitly to be a library. If a program is not declared explicitly to be a library but will be called as a library by other programs, then the LIBRARYPROG, TEMPORARY, and FEDLEVEL options should be specified.

A temporary library remains available as long as there are users of the library. A permanent library remains available until it is explicitly terminated. The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	Depending on the setting of the SHARING attribute of the library, the library can be either temporary or permanent. If SHARING is set to DONTCARE, then the object program functions as a permanent library when called as a library by another program. If SHARING is set to SHAREDBYRUNUNIT or PRIVATE, then the object program functions as a temporary library when called as a library by another program.
SET	TRUE	The object program functions as a temporary library when called as a library by another program. A temporary library remains available as long as there are users of the library.

TITLE Option



Type: String
Default: The name of the compiler

This option specifies a string of characters to be printed in the upper left corner of each output listing page. Only the final value declared in the program as the title appears on the output listing. If the LIST option is FALSE, the TITLE option is ignored.

TITLE =

Assigns a string of characters to the header of each output listing page.

TITLE +=

Appends a string of characters to a previously defined title string.

'file title'
"file title"

Either apostrophes (' ') or quotation marks (" ") may enclose the file title.

UDMTRACK Option

Type: Boolean
Default: FALSE

When set to true, the UDMTRACK option loads tracking information in the Universal Repository DMSII Model (UDM) for any of the databases loaded in the UDM. When set to false, no tracking information is loaded into the UDM. Note that the loading is done by a WFL job called DATABASE/WFL/UDM using a stream file created by COBOL.

VERSION Option

```
VERSION { version.cycle.patch     }
        { +version.+cycle.patch   }
```

Type: Value
Default: 00.000.0000

This option specifies a version number. Version numbers are used to manage software development. The VERSION option specifies an initial version number, replaces an existing version number, or updates to an existing version number.

Replacement or updating of existing version numbers occurs when all of the following conditions exist:

- The NEW option is TRUE.
- A VERSION option appears in the secondary input file (SOURCE).
- The update form of the VERSION option appears in the primary input file (CARD).
- The sequence number of the VERSION option in the primary input file (CARD) is less than or equal to the sequence number of the VERSION option in the secondary input file (SOURCE).

When all these conditions are satisfied, the VERSION option specification from the secondary input file is updated with the values specified by the VERSION option from the primary input file. The updated VERSION option record is placed in the updated symbolic file (NEWSOURCE).

VOID Option

Type: Boolean
Default: FALSE

This option controls whether the compiler ignores all source language input from both the primary input source file and the secondary input source file. Once the VOID option is SET, it can be RESET only by a CCR in the primary input file (CARD). The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	The compiler processes all source language input.
SET	TRUE	The compiler ignores all source language input from both the primary input source file and the secondary input source file. The compiler processes only CCRs. The ignored source input is neither listed in the compilation listing nor included in the updated symbolic file, regardless of the values of the LIST option and the NEW option.

WARNFATAL Option

Type: Boolean Class
Default: FALSE for the class, FALSE for MSG suboption

```
WARNFATAL [ ( MSG ( msg-number-list ) ... ) ]
```

This option controls the treatment of warning messages that might be flagged as errors. The suboption, MSG, for WARNFATAL is described in the following explanation.

MSG

The MSG suboption provides a way to designate a set of active warnings that might beflagged as errors. It is used to add to, or subtract from the active set. An MSG-number-list is a list of unsigned integers and the word ALL, separated by commas or spaces. The word ALL can be used to activate or deactivate all messages for flagging. Message numbers are activated for flagging by the SET action and deactivated by RESET.

Option Setting	Value	Description
Class SET	TRUE	Active warnings are flagged as errors.
Class RESET	FALSE	No warnings are flagged as errors.
Suboption SET	List	Adds the list of warnings to the list of active warnings.
Suboption RESET	List	Deletes the list of warnings from the list of active warnings.

Examples

The following examples illustrate the use of the WARNFATAL option.

```
$SET   WARNFATAL(MSG(ALL))
       WARNFATAL is set and all warnings will be flagged as errors.

$SET   WARNFATAL(SET MSG(ALL) RESET MSG(385))
       WARNFATAL is set and all messages are active except 385, so all but
       it will be errors, rather than warnings.

$RESET WARNFATAL(SET MSG(103,820,553))
       WARNFATAL is reset, although messages 103, 820 and 553 are armed for
       flagging once WARNFATAL is set.

$SET   WARNFATAL(RESET MSG(107))
       WARNFATAL is now set, and message 107 is marked as inactive for
       flagging.

$RESET WARNFATAL(MSG(385))
       WARNFATAL is reset and message 385 is marked as inactive for
       flagging.

$SET   WARNFATAL
       WARNFATAL is enabled and any active messages will be flagged.
```

WARNSUPR Option

Type: Boolean
Default: FALSE

The warning suppress option controls the display of warning messages. The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	The compiler prints warning messages.
SET	TRUE	The compiler suppresses printing of warning messages.

The WARNSUPR option does not affect the printing of messages related to syntax errors.

XREF Option

Type: Boolean
Default: FALSE

This option is used in conjunction with the XREFFILES option to control whether cross-reference information is collected and printed by the compiler. The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	If the XREFFILES option is TRUE, the compiler generates cross-reference information with the output stored in disk files. If the XREFFILES option is FALSE, the compiler does not generate cross-reference information.
SET	TRUE	The compiler generates cross-reference information. The information is written to the output listing. If a syntax error occurs during compilation, a cross-reference listing is not produced. If both the XREF and XREFFILES options are TRUE, then cross-reference information is both printed and stored in disk files.

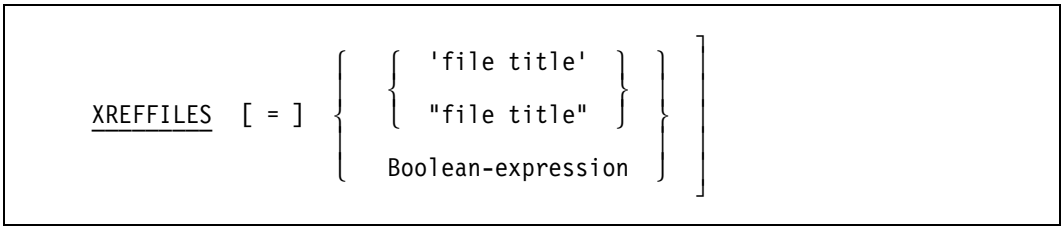
The cross-reference information consists of an alphabetized list of user-defined words that appear in the program. For each user-defined word, the compiler provides the following information:

- The type of data item named by each user-defined word
- The sequence number of the source input record on which the user-defined word is declared
- The sequence number of the source input record on which the data item is declared
- The sequence numbers of the input records on which the user-defined word is accessed

If your program contains INCLUDE or COPY files, the sequence numbers take the form fff:nnnnnn where "fff" is the number of the INCLUDE file and "nnnnnn" is the actual sequence number in the file. For example, sequence numbers that are referenced in the third INCLUDE or COPY file are listed as 3:nnnnnn. The file numbers for INCLUDE and COPY files are printed in the summary.

The XREF option should be included in the source code before the end of the Identification Division of the program.

XREFFILES Option



Type: Boolean Title
Default: FALSE

This option controls whether cross-reference information is collected and saved in disk files by the compiler. For more information on cross-reference information generated by the compiler, refer to the XREF option, earlier in this section. The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	The compiler does not generate cross-reference information.
SET	TRUE	The compiler generates cross-reference information with the output directed to disk files for use by the Editor and SYSTEM/INTERACTIVEXREF. If both the XREF option and the XREFFILES option are TRUE, then output is both printed and stored in disk files for use by the Editor and SYSTEM/INTERACTIVEXREF.

When the XREFFILES option is true, the following disk files are produced:

XREFFILES/<object code file-name>/DECS

XREFFILES/<object code file-name>/REFS

The object code file name is the name of the object code file that the compiler is generating.

The XREFFILES option should be included in the source code before the end of the Identification Division of the program.

XREFLIT Option

Type: Boolean
Default: FALSE

This option controls whether the XREF information for literals is set in the output listing. The XREFLIT option must be set together with the XREF and/or XREFFILES options.

If the default (FALSE) is used, it would have no affect on the XREF and XREFFILES options. The following table describes the effects of this option:

Option Setting	Value	Description
RESET	FALSE	The literals from the PROCEDURE DIVISION and/or literals from the WORKING-STORAGE SECTION are not included in the output listing.
SET	TRUE	The literals from the PROCEDURE DIVISION and/or literals from the WORKING-STORAGE SECTION are included in the output listing.

For example, when XREFLIT is set in the following program:

```
200          WORKING-STORAGE SECTION.

300           77  D1                 PIC X(06) VALUE "123456".

400          PROCEDURE DIVISION.

500           MOVE "ABC" TO D1.

             .
             .
             .
```

After the program is compiled, the following XREF information appears for literals in the output listing.

```
XREF FOR LITERALS:
=================

"ABC"           NONNUMERIC LITERAL          AT           500

"123456"        NONNUMERIC LITERAL          AT           300
```

Note: *The cross referencing of the literals are separate from the current XREF and are not available in the interactive XREF.*

Section 16
Internationalization

Note: *Although this section is not highlighted, all of the information is relative to Unisys extensions to COBOL ANSI-85.*

Internationalization refers to the software, firmware, and hardware features that enable you to develop and run application systems that can be customized to meet the needs of a specific language, culture, or business environment. The internationalization features provide support for various character sets, international business and cultural conventions, extensions to data communications protocols, and the ability to use multiple languages concurrently. This section describes the internationalization features you can use to customize an application for the language and conventions of a particular locality.

Localization

Using the features described in this section to write or modify an application is termed *localization*. The MultiLingual System (MLS) environment enables you to process information to localize your applications. Some of the localization methods included in the MLS environment include translating messages to another language, choosing a particular character set to be used for data processing, and defining date, time, number, and currency formats for a particular business application.

In addition to the information described in this section, refer to the *MultiLingual System (MLS) Administration, Operations, and Programming Guide* for information. The *MLS Guide* provides definitions for and detailed information about the ccsversions, character sets, languages, and conventions provided on the system. It also describes procedures for setting system values for the internationalization features.

Accessing the Internationalization Features

In order to use internationalization features, you must enable a program to access internationalization features. A program is not affected by the features described in this section unless the program specifically invokes them. Any existing programs that do not invoke internationalization features are not affected by the features.

Use the following methods either together or separately to access internationalization features:

- COBOL85 provides language syntax that enables you to localize a program. For example, if you specify a particular ccsversion in your program, the compiler uses the collating sequence associated with the ccsversion for national comparisons. The details of language syntax are described in "Summary of Language Syntax by Division," later in this section.
- The CENTRALSUPPORT system library contains procedures that enable you to localize programs. Programs can access a procedure in this library by using a CALL statement. When a call occurs, input parameters describe the type of information that is required or the action that is to be performed. Output parameters are returned with the result of the procedure call. The procedures available in the CENTRALSUPPORT library are summarized in "Summary of CENTRALSUPPORT Library Procedures," later in this section.

Using the Ccsversion, Language, and Convention Default Settings

The program can choose the specific ccsversion, language, and convention settings that it needs by setting the input parameters to a procedure. The system also has default settings for the internationalization features at other levels. The default settings can also be accessed by the program. See "Understanding the Hierarchy for Default Settings" later in this section for information on the available levels and on the features supported at each level.

One of the following two methods can be used to determine the current system default settings:

- The program calls the CENTRALSTATUS procedure in the CENTRALSUPPORT library.
- A system administrator, a privileged user, or a user who is allowed to use the system console can use Menu-Assisted Resource Control (MARC) menus and screens or the SYSTEMOPTIONS system command.

Refer to the *MultiLingual System (MLS) Administration, Operations, and Programming Guide* or the *Menu-Assisted Resource Control (MARC) Operations Guide* for the instructions about displaying the default ccsversion, language, or convention with MARC.

The system default settings are shown in Table 16–1.

Table 16–1. System Default Settings for Internationalization

System Setting	Default Value
Ccsversion	ASeriesNative
Language	English
Convention	ASeriesNative

Before you change the default settings for localization, you must consider the level at which the particular feature you want to change is defined. For example, the ccsversion can be changed only at the system operations level. You can avoid actually coding settings in a program by specifying the predefined default settings as input parameters. For example, if the system-defined ccsversion is France, the language is Francais, and the convention is FranceListing, the program can use those default settings as input parameters. See "Input Parameters" later in this section for specific information on those parameters.

In order to use the system default ccsversion, you must specify the CCSVERSION phrase in the SPECIAL-NAMES paragraph without the literal-1 option. The alphabet-name IS CCSVERSION clause identifies the collating sequence that is associated with the alphabet-name as the system default collating sequence. For more information about the CCSVERSION phrase, refer to "SPECIAL-NAMES Paragraph" in Section 3, "Environment Division," of this manual.

You can specify five different date and time formats by using the TYPE clause in the DATA DIVISION. To use the system default language and convention, do not include the USING phrase of the TYPE clause. For more information about using the TYPE clause for editing the date and time formats, refer to "TYPE Clause" in Section 4, "Data Division," of this manual.

Hierarchy for Default Settings

The default settings for the internationalization features can be established at the following levels:

Level	How Established
Task	Established at task initiation
Session	Handled by MARC or CANDE commands or by programs that support sessioning
Usercode	Established in the USERDATAFILE file
System	Established with a system or MARC command

A priority is associated with these levels. A setting at the task level overrides all other settings. A setting at the session level overrides a setting at the usercode and system levels. A setting at the usercode level overrides a setting at the system level. A language and convention can be established at any level, but the ccsversion can be established only at the system level.

Two task attributes enable you to change the language, the convention, or both. These attributes are the LANGUAGE and CONVENTION task attributes. By using these attributes, you can select a language or a convention from multiple languages and conventions when running a program. Information on the use of task attributes is provided in the *Task Attributes Reference Manual*.

The LANGUAGE task attribute establishes the language used by a program at run time.

The CONVENTION task attribute establishes the convention used by a program at run time. For example, an international bank might have a program that prints bank statements for customers in different countries. This program could have a general routine to format dates, times, currency, and numerics according to the selected conventions. To print a bank statement for a French customer, this program could set the CONVENTION task attribute to FranceBureautique and process the general routine. For a customer in Sweden, the program could set the CONVENTION task attribute to Sweden and process the general routine.

As you code your program, you can use the defaults in both the source code and the calls to the CENTRALSUPPORT library, or you can use the settings of your choice. The task level and system level are probably the most useful levels for your program. Because the language and convention features have task attributes defined, you can access or set these task attributes in your program.

Components of the MLS Environment

The following four components of the MLS environment support different languages and cultures:

- Coded character sets
- Ccsversions
- Languages
- Conventions

The following paragraphs describe the function of each of these components.

Coded Character Sets and Ccsversions

A *coded character set* is a set of rules that establishes a character set and the one-to-one relationship between the characters of the set and their code values. The same character set can exist with different encodings. For example, the LATIN1-based character set can be encoded in an International Organization for Standardization (ISO) format or an EBCDIC format. Coded character sets are defined in the *MultiLingual System (MLS) Administration, Operations, and Programming Guide*.

A coded character set name and number is given to each unique coded character set definition. This name or number can also be used to set the INTMODE or EXTMODE file attribute value for a file. For more information on these attributes, see the *File Attributes Reference Manual*.

A ccsversion is a collection of information necessary to apply a coded character set in a given country, language, or line of business. This information includes the processing requirements such as data classes, lowercase-to-uppercase mapping, ordering of characters, and escapement rules necessary for output. A ccsversion name and number is given to each unique group of information. This name and number may also be used to set the CCSVERSION file attribute for a file. For more information on these attributes, see the *File Attributes Reference Manual*.

Each system includes a data file, SYSTEM/CCSFILE, containing all coded character sets and ccsversions that are supported on the system. You cannot choose a coded character set directly, but by choosing a ccsversion, you implicitly designate the default coded character set for your system.

Data can be entered and manipulated in only one coded character set and ccsversion at a time. Although many ccsversions can be accessed, only one ccsversion is active for the entire system at one time. This ccsversion is called the system default ccsversion. All coded character set and ccsversion information can be accessed by calling CENTRALSUPPORT library procedures.

You can use any of the following ways to find out which coded character sets and ccsversions are available on the system:

- Look in the *MultiLingual System (MLS) Administration, Operations, and Programming Guide*. Your system might have a subset of the ones defined in that guide.
- Use the MARC menus and screens or the system command SYSTEMOPTIONS. Refer to the *MLS Guide* or the *System Commands Operations Reference Manual*.
- Call the CCSVSN_NAMES_NUM procedure.

You might want to refer to the *MLS Guide* for a complete understanding of ccsversions and the relationship of a coded character set and a ccsversion.

You must use language syntax to designate that a ccsversion is to be used in a program. To do this, specify a PROGRAM COLLATING SEQUENCE clause and an alphabet-name is CCSVERSION literal-1 clause in the ENVIRONMENT DIVISION.

You can enter and manipulate data in any particular coded character set and ccsversion. Though there are many ccsversions that can be accessed, only one ccsversion can be active for the entire system at one time. This ccsversion is referred to as the system default ccsversion. You can select a ccsversion other than the default ccsversion to be used during the execution of a program by including the literal-1 option in the CCSVERSION clause. If you do not use the literal-1 option, the program uses the system default ccsversion.

Many of the procedures require the specification of a coded character set or ccsversion as an input parameter. A program can choose a specific coded character set or ccsversion by using the name or number of the coded character set or the ccsversion as an input parameter when calling a procedure. A program can also use the system default setting by using predefined values as input parameters. See "Input Parameters" later in this section.

It is possible to use a different ccsversion in your program by changing the value of the literal-1 option. For example, by changing the value of literal-1, a program could process data in the AseriesNative ccsversion and then process data in the Swiss ccsversion. For more information about the CCSVERSION clause, refer to "SPECIAL-NAMES Paragraph," in Section 3, "ENVIRONMENT DIVISION," of this manual.

Mapping Tables

A mapping table is used to map one group of characters to another group of characters or another representation of the original characters. For example, a translate table can exist to translate lowercase characters to uppercase characters.

The CENTRALSUPPORT library provides three procedures that perform mapping functions: CCSTOCCS_TRANS_TEXT, CCSTOCCS_TRANS_TEXT_COMPLEX, and VSNTRANS_TEXT.

The procedure . . .	**Maps . . .**
CCSTOCCS_TRANS_TEXT	One coded character set to another coded character set (works with 8-bit character sets only).
CCSTOCCS_TRANS_TEXT_COMPLEX	One coded character set to another coded character set (works with 8-bit, 16-bit, or mixed, multibyte character sets).
VSNTRANS_TEXT	• Lowercase to uppercase characters. • Uppercase to lowercase characters. • Alternative numeric digits defined in a ccsversion to numeric digits in U.S. EBCDIC. • Numeric digits in U.S. EBCDIC to alternative numeric digits defined in a ccsversion. • Characters to their escapement directions.

Refer to the *MultiLingual System (MLS) Administration, Operations, and Programming Guide* for definitions of mapping tables for each coded character set and ccsversion.

Data Classes

A data class is a group of characters sharing common attributes such as alphabetic, upon which membership tests can be made. Some characters might not have a data class assigned to them. Many CENTRALSUPPORT library procedures store ccsversion information in ALGOL-type truthset tables as a way to define ccsversion data classes. A truthset is a method of storing the declared set of characters that defines a data class in ALGOL. It is not necessary to understand the layout of an ALGOL-type translate table because the table is usually not visible to your program. A description of translate tables is provided in the *ALGOL Reference Manual, Vol. 1: Basic Implementation.*

The internationalization features provide you with access to additional truthsets that apply to a ccsversion. These truthsets are as follows:

- Ccsversion alphabetic
- Ccsversion numeric
- Ccsversion graphics
- Ccsversion spaces
- Ccsversion lowercase
- Ccsversion uppercase

The alphabetic truthset contains those characters that are considered to be alphabetic for a specified ccsversion; the numeric truthset contains those characters that are considered to be numeric for a specified ccsversion, and so on.

The compiler automatically accesses the alphabetic truthset if you have specified a PROGRAM COLLATING SEQUENCE clause and an alphabet-name IS CCSVERSION literal-1 clause in the ENVIRONMENT DIVISION. If you then use the identifier IS ALPHABETIC clause, the compiler makes the class condition test sensitive to the ccsversion alphabetic data class.

You can use procedures from the CENTRALSUPPORT library to access these truthsets or to process data by using these truthsets. For example, if a program manipulates an employee identification number such as 555962364, it might then need to verify that the text is or is not all numeric. The program can call the VSNINSPECT_TEXT CENTRALSUPPORT library procedure to compare the text to the numeric truthset. This procedure returns the information that the text is or is not all numeric.

Refer to the *MultiLingual System (MLS) Administration, Operations, and Programming Guide* for definitions of ccsversions and data classes.

Text Comparisons

You might need to perform a text comparison to sort and merge text, to compare relationships between two pieces of text, or to index a file.

The traditional method for handling text comparisons is based on a strict binary comparison of the character values. The binary method of comparison is not meaningful when used for sorting text if the binary ordering of the coded characters does not match the ordering sequence of the alphabet. This situation is the case for most coded character sets.

Because the binary method is not sufficient for all usage requirements, the definitions of two other levels of ordering are supported.

The first level is called ORDERING. For this level, each character has an ordering sequence value (OSV). An OSV is an integer in the range 0 (zero) through 255 that is assigned to each code position in a character set. The OSV indicates a relative ordering value of a character. An OSV of 0 (zero) indicates that the character comes before a character with an OSV equal to 1. More than one character can be assigned the same OSV.

The second level is called COLLATING. For this level, each character has an OSV and a priority sequence value (PSV). A PSV is an integer in the range 1 through 15 that is assigned to each code position in a character set. A PSV indicates a relative priority value within each OSV. Each character with a unique OSV has a PSV equal to 1, but two characters with the same OSV have different PSVs to separate them.

When comparing two strings of data, the default comparison uses only the ORDERING level. This is referred to as an equivalent comparison. A comparison that uses both levels, ORDERING and COLLATING, is referred to as a logical comparison.

You can specify the three types of comparisons shown in Table 16–2 by calling procedures in the CENTRALSUPPORT library.

Table 16–2. Types of Comparisons Provided by CENTRALSUPPORT Library

The ordering type of . . .	Compares two records based on the . . .
Binary	Hexadecimal code values of the characters.
Equivalent	OSVs of the characters. This type of comparison uses the ORDERING level.
Logical	OSVs plus the PSVs of the characters. This type of comparison uses the COLLATING level.

In addition to the three types of comparisons, the two types of character substitution are also supported, as shown in Table 16–3.

Table 16–3. Valid Character Substitution Types

Substitution Type	Explanation
Many to One	A predetermined string of as many as three characters can be ordered as if it were one character, assigning it a single OSV and PSV pair. Even if a character is part of a predetermined string of characters that are ordered as a single value, the character still has an OSV and a PSV pair assigned to it to allow for cases in which the character appears in other strings or individually. For example, in Spanish, the letter pair ch is ordered as if it were a single letter, different from either c or h, and ordering between c and d.
One to Many	A single character can generate a string of two or three OSV and PSV pairs. For example, the \S (the German sharp S) character is ordered as though it were ss.

You can specify a collating sequence to be used for text comparisons. When you designate an internationalized collating sequence at the program level and you are comparing two national data items, the compiler uses the logical ordering type when generating the text comparison routines.

Historically, text was sorted by using a standard, system-provided method that is based on a strict binary comparison of the character values. Within a program, you can also specify a collating sequence to be used for text comparisons.

Your program can call the CENTRALSUPPORT library procedures listed in the category of comparing and sorting text to obtain ordering information of the ccsversion, and to sort or compare text based on this information.

Sorting and Merging

Due to the complexity of the SORT and MERGE statements, and because the compiler generates the sort compare procedure to be used in the sort and merge operations, you can specify a localized collating sequence to be used at the program level or at the SORT or MERGE statement level. To use this language syntax, specify an alphabet-name IS CCSVERSION option in the SPECIAL-NAMES paragraph. Then if you specify a localized collating sequence at the program level, the compiler generates the text comparison routines.

Supporting Natural Languages

The natural language feature enables users of your application program to communicate with the computer system in their natural language. A natural language is a human language in contrast to a computer programming language.

You must write your COBOL85 program in the subset of the standard EBCDIC character set defined by the COBOL language. Only the contents of string literals, data items with variable character data, or comments can be in a character set other than that subset.

If your program interacts with a user, has a user interface with screens or forms, displays messages or accepts user input, then those aspects of the program should be in the natural language of the user. For example, French would be the natural language of a person in France.

Refer to the *MultiLingual System (MLS) Administration, Operations, and Programming Guide* for a list of user interfaces that can be localized. The following text explains how to develop a COBOL application program that supports interaction in the natural language of the user.

Creating Messages for an Application Program

In the MLS environment, the messages handled by your application program are classified as one of the following message types:

- Output message

  An application program displays this message to the user. Some examples of output messages are error messages and prompts for input. An output message can be localized so that it can be displayed in the language of the user.

- Input message

  An interactive program receives this message either from a user or from another program in response to a prompt for input. The input message might be in a language that the program cannot recognize. In this case, the message must be translated so that it can be understood by the program.

If you develop input and output messages within an output message array, you make the localization process easier. When messages are in an output message array, the translator can use the MSGTRANS utility to localize the messages into one or more natural languages. The MSGTRANS utility finds all output message arrays in a program and presents them for translation. If messages are not in output message arrays, a translator must search the source file for each message and then translate the message.

You can create an output message array by creating an ALGOL library that contains OUTPUTMESSAGE ARRAY declarations.

An output message array contains output messages to be used by the MultiLingual System (MLS). ALGOL statements within the output message array declaration contain output messages or translate input messages. You can then call the library from your application program. The *MLS Guide* describes the procedures for creating and using output message arrays.

The program EXAMPLE/MLS/ALGOL/LIBRARY on the release media demonstrates how to create an ALGOL library containing output message arrays.

For information on how to call an ALGOL library from a COBOL85 program, refer to Section 11.

Creating Multilingual Messages for Translation

Follow these guidelines when you create messages that are to be multilingual:

- Put all output messages in output message arrays.
- Allow more space for translated messages. Because the English language is more compact than many other natural languages, a message in English generally becomes about 33 percent longer after it is translated into another language. For example, if a program can display an 80-character message, an English message should be only 60 characters long so that the translated message can expand by one-third and not exceed the maximum display size.
- Accept or display any messages through a library interface similar to that provided on the release media.
- Use complete sentences for messages because phrases are difficult to translate accurately.
- Do not use abbreviations because they are difficult to translate.

Supporting Business and Cultural Conventions

The business and cultural features enable users of an application program to display and receive data according to local conventions. A convention consists of formatting instructions for the date, the time, numbers, currency, and page size.

Convention definitions are provided for many formatting styles. For example, some of the conventions are Denmark, Italy, Turkey, and UnitedKingdom1. These convention definitions contain information to create formats for time, date, numbers, currency, and page size required by a particular locality.

Each system includes a data file named SYSTEM/CONVENTIONS that contains all the convention definitions supported on the system. Although you can access many conventions, only one convention is active at a time for the entire system. This convention is called the system default convention. You can access conventions as follows:

- Refer to the *MultiLingual System (MLS) Administration, Operations, and Programming Guide*. Your system might have a subset of the ones defined in that guide.

- Use the MARC menus and screens or the system command SYSTEMOPTIONS. Refer to the *MLS Guide* or the *System Commands Reference Manual*.
- Call the CNV_NAMES procedure to display the names of conventions available on the host computer.

If none of the existing conventions meet your needs, you can define a new convention. You must use a template to define a convention. A template is a group of predefined control characters that describe the components for the date, the time, numbers, or currency. For example, the data item 02251990 and the template !0o!/!dd!/!yyyy! produce the formatted date, 02/25/1990. To use some of the CENTRALSUPPORT library procedures, you must understand how templates are defined. The *MLS Guide* describes how to define a template.

Using the Date and Time Features

Several date and time features are provided for standard use. You can access the conventions either by using language syntax or calling a CENTRALSUPPORT library procedure.

Formatting the Date and Time with Syntax Elements

COBOL85 provides the following language syntax options to handle the formatting of date and time data items:

- You can declare a data item to have one of several date or time types in the TYPE clause of the data-description entry. You can also designate a language or convention with the TYPE clause.
- The special registers, date and time editing, and PROCEDURE DIVISION statements are provided for standard formatting of date and time. These standard features include the following.
  - The special registers TODAYS-DATE, DATE, and DAY provide the system date with MMDDYY, YYMMDD, and YYDDD formats respectively. Four digit year versions of the system date are also supported.
  - The special register TIME provides the elapsed time after midnight on a 24-hour clock, in the format HHMMSSTT, where HH equals hours, MM equals minutes, SS equals seconds, and TT equals hundredths of a second. For example, 12:01 p.m. is expressed as 12010000.
  - The special register TIMER provides the number of 2.4 microsecond intervals since midnight.
  - The special register TODAYS-NAME provides the name of the current day of the week.
  - The simple insertion editing feature of the PICTURE clause can be used to improve the legibility of date and time values in program output, as in the following example:

```
05 DATE-YYMMDD PIC 99/99/99.
05 DATE-MMDDYY PIC 99I-99I-99.
05 DATE-YYDDD PIC 99B999.
```

```
05 TIME-HHMMSS PIC 99I:99I:99.
05 AMOUNT PIC 99B999.99.
```

- Date and time editing is provided with the particular convention and language specified as a property of the receiving data item. The MOVE statement causes the editing to occur.
- The ACCEPT statement transfers the formatted system date or time into the data item specified by the identifier by using the TYPE, CONVENTION, and LANGUAGE declared for the data item.

Notes:

- *For more information about special registers, refer to "Special Registers" in Section 1, "Program Structure and Language Elements."*
- *For more information about using the "I" character in PICTURE character strings, refer to "Simple Insertion Editing," under "Editing Rules" for the PICTURE Clause in Section 4, "Data Division."*

Formatting the Date and Time with Library Calls

You can format and retrieve date and time items by calling procedures from the CENTRALSUPPORT library, as shown in Table 16–4.

Table 16–4. CENTRALSUPPORT Library Procedures for Formatting Date and Time

Procedure Type	Description
Convention	You supply the convention name and the value for the date or time. The procedure returns the date or time value in the format used by the convention. All the conventions are described in the MLS Guide.
Template	You supply the following: the format that you want for the date or time in a template parameter; the value for the date or time. You must use predefined control characters to create the template. These control characters are described in the MLS Guide.
System	The system supplies the date and time. There is a procedure that formats the system date, the system time, or both according to a convention and a procedure that formats the system date, the system time, or both according to a template that you supply.

Example

You could use the CNV_SYSTEMDATETIME_COB procedure to display the system date and time according to the language and convention you choose. If you designate the ASeriesNative convention and the ENGLISH language, the date and time are displayed as follows:

```
9:25 AM Monday, July 4, 1997
```

If you designate the FranceListing convention and the French language, the same date and time are displayed as follows:

```
9h25, lundi 4 juillet 1997
```

The "Summary of CENTRALSUPPORT Library Procedures" later in this section lists other procedures you can use to inquire about the conventions that are available on your system.

Formatting Numerics and Currencies

You can inquire about and retrieve numeric and currency symbols and format currency amounts by calling procedures in the CENTRALSUPPORT library. One of the procedures, the CNV_CURRENCYEDIT_DOUBLE_COB procedure, formats a monetary value according to the convention you choose. For example, if you designate the Greece convention, the monetary amount 12345.67 is formatted as follows:

```
DR.12 345,67
```

Formatting Page Size

Page sizes are specific to a locality. Several standard features are provided for formatting page size. You can set the number of lines per page and the number of characters per line by using language syntax or you can call the CNV_FORMSIZE procedure to obtain predetermined page size values for the convention that you specify.

Formatting Page Size with Syntax Elements

The following language syntaxes are provided to handle formatting of the page size.

- LINAGE clause of the DATA DIVISION
- Report Writer
- ADVANCING phrase of the WRITE statement

Formatting Page Size with Library Call

The CNV_FORMSIZE procedure enables you to retrieve default lines-per-page and characters-per-line values for a specified convention.

For example, the Netherlands convention definition specifies 70 lines as the default page length and 82 characters as the default page width, while the Zimbabwe convention definition specifies 66 lines as the default page length and 132 characters as the default page width.

Summary of Language Syntax by Division

The following paragraphs describe the changes you can make in the divisions of a COBOL85 program to use internationalization features. No changes are required in the Identification Division.

ENVIRONMENT DIVISION

The language syntax that you can use in the ENVIRONMENT DIVISION includes the following:

- The PROGRAM COLLATING SEQUENCE IS alphabet-name clause in the OBJECT-COMPUTER paragraph declares that the collating sequence associated with the alphabet-name is the sequence to be used in the program. The alphabet-name must be the same as the one specified in the SPECIAL-NAMES paragraph. The collating sequence is used for alphabetic comparisons in conditional statements and for sort and merge routines.
- The ALPHABET FOR NATIONAL alphabet-name IS CCSVERSION literal-1 clause in the SPECIAL-NAMES paragraph designates the system collating sequence and the ccsversion. If you do not use the literal-1 option, the system uses the system default ccsversion values for the collating sequence and the ccsversion. If you do specify literal-1, then the literal identifies the ccsversion.
- The DECIMAL-POINT IS COMMA option of the SPECIAL-NAMES paragraph causes the functions of the EBCDIC comma and decimal point to be switched.
- The CURRENCY SIGN IS literal-6 WITH PICTURE SYMBOL literal-7 clause of the SPECIAL-NAMES paragraph enables you to specify multiple characters to be used as the currency symbol in the numeric data.

DATA DIVISION

The language syntax that you can use in the DATA DIVISION includes the following:

- The TYPE option of the data-description entry for record structures allows a data item to be declared as one of the following date or time types:
  - LONG-DATE
  - SHORT-DATE
  - NUMERIC-DATE
  - LONG-TIME
  - NUMERIC-TIME
- The USING phrase of the TYPE option allows the data item to be formatted according to a designated language or convention.
- When no ALPHABET FOR NATIONAL alphabet-name IS CCSVERSION clause is specified, the NATIONAL phrase of the USAGE clause enables you to display and write messages in natural languages that require the double-octet format.
- Standard COBOL85 editing also can be used to format date and time.

PROCEDURE DIVISION

The valid language syntax for the PROCEDURE DIVISION includes the following:

- The identifier IS ALPHABETIC clause tests whether an identifier is in the alphabetic truthset. If you have specified a ccsversion in the ENVIRONMENT DIVISION, the system determines the alphabetic truthset with respect to the alphabetic data class of the ccsversion.
- The ACCEPT identifier statement transfers a formatted system date or time to the identifier. The format of the system date or time data item depends on the coding of the TYPE, LANGUAGE, and CONVENTION clauses specified for the item.
- The MOVE statement causes a receiving item with an associated TYPE clause to be formatted according to the TYPE, LANGUAGE, and CONVENTION clauses specified for the item. If no LANGUAGE and CONVENTION clauses are specified, the compiler uses the hierarchy to determine the language and convention to be used.

  If the receiving item is of the SHORT-DATE, LONG-DATE, or NUMERIC-DATE types, the sending item must be in the YYYYMMDD format, where YYYY represents the year, with a value in the range 0000 through 9999; MM represents the month, with a value in the range 01 through 12; and DD represents the day, with a value in the range 01 through 31.

  If the receiving item is of the LONG-TIME or NUMERIC-TIME types, the sending item must be in the HHMMSSPPPP format, where HH represents the hour, with a value in the range 00 through 23; MM represents the minutes, with a value in the range 00 through 59; SS represents the seconds, with a value in the range 00 through 59; and PPPP represents the partial seconds, with a value in the range 0000 through 9999.
- The CALL statement can be used to call the procedures of the CENTRALSUPPORT library.
- The SORT and MERGE statements use the collating sequence of a specified ccsversion if you establish a program collating sequence in the ENVIRONMENT DIVISION and do not override it in the COLLATING SEQUENCE IS alphabet-name option in the SORT or MERGE statement.
- The CHANGE ATTRIBUTE LIBACCESS statement can be used to change a library call by function to a library call by title or to change a library call by title to a library call by function.
- The CHANGE ATTRIBUTE FUNCTIONNAME statement enables you to specify the name of a function.

Examples

Figure 16–1 shows the five different types of date and time items and their expected values upon execution of the ACCEPT statement. The PICTURE size of the data items depends upon the definition of the type.

```
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  LONG-DATE-ITEM      PIC  X(30)      TYPE IS LONG-DATE.
 01  SHORT-DATE-ITEM     PIC  X(9)       TYPE IS SHORT-DATE.
 01  NUM-DATE-ITEM       PIC  X(8)       TYPE IS NUMERIC-DATE.
 01  LONG-TIME-ITEM      PIC  X(20)      TYPE IS LONG-TIME.
 01  NUM-TIME-ITEM       PIC  X(20)      TYPE IS NUMERIC-TIME.
*
 PROCEDURE DIVISION.
*
*Get the system date and time in the various formats defined by the
*CONVENTION and LANGUAGE task attributes for the task.   Assume a
*convention of ASERIESNATIVE and a language of ENGLISH, a current
*date of 31 August 1990, and a current time of 2:37:20 PM.
*
     ACCEPT LONG-DATE-ITEM FROM DATE.
*
*The LONG-DATE-ITEM now contains Friday, August 31, 1990.
*
     ACCEPT SHORT-DATE-ITEM FROM DATE.
*
*The SHORT-DATE-ITEM now contains Fri, Aug 31, 1990.
*
     ACCEPT NUM-DATE-ITEM FROM DATE.
*
*The NUM-DATE-ITEM now contains 08/31/90.
*
     ACCEPT LONG-TIME-ITEM FROM TIME.
*
*The LONG-TIME-ITEM now contains 14:37:20.0000.
*
     ACCEPT NUM-TIME-ITEM FROM TIME.
*
*  The NUM-TIME-ITEM now contains 14:37:20.

     STOP RUN.
```

Figure 16–1. Coding the Format 4 ACCEPT Statement

Figure 16–2 shows the five different types of date or time items and their expected values upon execution of the MOVE statement. The PICTURE size of the data-items depends on the definition of the type defined in the conventions file.

```
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  LONG-DATE-ITEM        PIC  X(30)      TYPE IS LONG-DATE.
 01  SHORT-DATE-ITEM       PIC  X(9)       TYPE IS SHORT-DATE.
 01  NUM-DATE-ITEM         PIC  X(8)       TYPE IS NUMERIC-DATE.
 01  LONG-TIME-ITEM        PIC  X(20)      TYPE IS LONG-TIME.
 01  NUM-TIME-ITEM         PIC  X(20)      TYPE IS NUMERIC-TIME.
*
 PROCEDURE DIVISION.
*Convert the literal date and time into the various formats defined
*by the CONVENTION and LANGUAGE task attributes for the task
*Assume a convention of ASERIESNATIVE and a language of ENGLISH,
*a current date of 31 August 1990, and a current time of 2:37:20 PM.

     MOVE "19900831" TO LONG-DATE-ITEM.
*
*The LONG-DATE-ITEM now contains Friday, August 31, 1990.
*
     MOVE "19900831" TO SHORT-DATE-ITEM.
*
*The SHORT-DATE-ITEM now contains Fri, Aug 31, 1990.
*
     MOVE "19900831" TO NUM-DATE-ITEM.
*
*The NUM-DATE-ITEM now contains 08/31/90.
*
     MOVE "143720000000" TO LONG-TIME-ITEM.
*
*The LONG-TIME-ITEM now contains 14:37:20.0000.
*
     MOVE "143720000000" TO  NUM-TIME-ITEM.
*
*The NUM-TIME-ITEM now contains 14:37:20.
*
     MOVE "143720000000" TO  NUM-TIME-ITEM.
*
*The NUM-TIME-ITEM now contains 14:37:20.

     STOP RUN.
```

Figure 16–2. Coding the MOVE Statement for Internationalization

For syntactical information about the statements used to localize an application, refer to the following topics in this manual: "ACCEPT Statement" and "CALL Statement" in Section 6, "Procedure Division Statements A–H;" "MERGE Statement" and "MOVE Statement" in Section 7, "Procedure Division Statements I–R;" and "SORT Statement" in Section 8, "Procedure Division Statements S–Z."

For information about the alphabetic test, refer to "Class Conditions" in Section 5, "Procedure Division Concepts."

Summary of CENTRALSUPPORT Library Procedures

The CENTRALSUPPORT library procedures are integer-valued procedures. The procedures return values in output parameters and as the procedure result.

You can check the result returned by each procedure by using standard programming practices. The result is useful in deciding if an error has occurred. The possible values for each procedure result are listed in the description of each procedure. The meanings of the result values are described at the end of this section.

The CENTRALSUPPORT library procedures are called by application programs and system software.

Following are some of the tasks your program can perform by calling CENTRALSUPPORT library procedures:

- Identify available coded characters sets and ccsversions.
- Map data from one coded character set to another.
- Process data according to ccsversion.
- Compare and sort text.
- Position characters.
- Determine available natural languages.
- Access CENTRALSUPPORT library messages.
- Identify available conventions definitions.
- Obtain convention information.
- Format dates according to convention.
- Format times according to convention.
- Format data according to monetary convention.
- Determine default page size.

The following descriptions

- Group the library procedures by the type of function they perform
- Describe the purpose of each procedure.

Identifying Available Coded Character Sets and Ccsversions

CENTRALSTATUS Procedure

Obtains the values of the default settings for internationalization features on the host computer. This procedure returns the names of the default ccsversion, language, and convention. It also returns the number of the default ccsversion.

CCSVSN_NAMES_NUMS Procedure

Returns the names and numbers of all coded character sets or all ccsversions available on the host computer. The names and numbers are listed in two arrays. These arrays are ordered so that the names in the names array correspond to the numbers in the numbers array.

VALIDATE_NAME_RETURN_NUM Procedure

Verifies that a designated coded character set or ccsversion name is valid on the host computer. If the coded character set or ccsversion is valid, the procedure returns the corresponding number.

VALIDATE_NUM_RETURN_NAME Procedure

Verifies that the designated coded character set or ccsversion number is valid on the host computer. If the coded character set or ccsversion is valid, the procedure returns the corresponding name.

Mapping Data From One Coded Character Set to Another

CCSTOCCS_TRANS_TEXT Procedure

Maps data from one 8-bit coded character set to another 8-bit coded character set by using a translate table. Characters are translated using a one-to-one mapping between the two character sets.

CCSTOCCS_TRANS_TEXT_COMPLEX Procedure

An advanced character mapping operation that supports the complex translation requirements of mixed, multibyte and 16-bit coded character sets, as well as 8-bit coded character sets. Characters may be translated using a one-to-one, one-to-many or many-to-one mapping between the two character sets.

Processing Data According to a Ccsversion

VSNINSPECT_TEXT Procedure

Compares the input text to a designated ccsversion truthset to determine whether the characters in the text are in the truthset.

VSNTRANS_TEXT Procedure

You can use this procedure to determine if the characters are in one of the following truthsets:

- Alphabetic
- Numeric
- Spaces
- Presentation
- Lowercase
- Uppercase

Translates data using a designated ccsversion. You can use this procedure to perform the following types of translations:

- Lowercase to uppercase characters
- Uppercase to lowercase characters
- The digits 0 through 9 to alternate digits
- Alternate digits to the digits 0 through 9
- Characters to their character escapement directions

Comparing and Sorting Text

VSNCOMPARE_TXT Procedure

Compares two strings using one of the following comparison methods for a designated ccsversion:

- Binary comparison, which is based on the binary values of the characters
- Equivalent comparison, which is based on the ordering sequence values of characters
- Logical comparison, which is based on the ordering sequence values and priority sequence values of characters

VSNGETORDERINGFOR_ONE_TEXT Procedure

Returns the ordering information for the input text. The ordering information determines the way that input text is collated. It includes the ordering and priority sequence values of the characters and any substitution of characters to be made when the input text is sorted. You can choose one of the following types of ordering information:

- Equivalent ordering information, which comprises only the ordering sequence values
- Logical ordering information, which comprises the ordering sequence values followed by the priority sequence values

Positioning Characters

VSNESCAPEMENT Procedure

Takes the input text and rearranges it according to the escapement rules of the ccsversion. Both the character advance direction and the character escapement direction are used. If the character advance direction is positive, then the start position of the text is the leftmost position of the starting character. If the character advance direction is negative, then the starting position for the text is the rightmost position of the last character. From that point on, the character advance direction value and the character escapement direction values, in combination, control the placement of each character in relation to the previous character.

Determining Available Natural Languages

MCPBOUND_LANGUAGES Procedure

Returns the names of the languages that are currently bound to the MCP.

Accessing CENTRALSUPPORT Library Messages

GET_CS_MSG Procedure

Returns text of the message associated with the designated CENTRALSUPPORT error number. Your program can specify the maximum message length. If the returned message is shorter, it is padded with blanks. An entire message consists of three parts:

- The header, which always takes the first 80 characters of the return message
- The general description, which takes the next 80 characters
- The specific description, which has no maximum length

Identifying Available Convention Definitions

CENTRALSTATUS Procedure

Obtains the values of the default settings for internationalization features on the host computer. This procedure returns the names of the default ccsversion, language, and convention. It also returns the number of the default ccsversion.

CNV_NAMES Procedure

Returns the names of the conventions available on the host system.

CNV_VALIDATENAME Procedure

Returns a value that indicates whether the specified convention name is currently defined on the host system.

Obtaining Convention Information

CNV_SYMBOLS Procedure

Returns the numeric and monetary symbols defined for a designated convention. The symbols in the convention are

- Numeric positive symbol
- Numeric negative symbol
- Numeric thousands separator symbol
- Numeric decimal symbol
- Numeric left enclosure symbol
- Numeric right enclosure symbol
- Numeric grouping
- Monetary positive symbol
- Monetary negative symbol
- International currency notation
- National currency symbol
- Monetary grouping
- Monetary thousands separator symbol
- Monetary left enclosure symbol
- Monetary right enclosure symbol
- Monetary decimal symbol

CNV_TEMPLATE_COB Procedure

Returns the requested template for a designated convention. You can obtain the template for the following:

- Long date format
- Short date format
- Numeric date format
- Long time format
- Numeric time format
- Monetary format
- Numeric format

Formatting Dates According to a Convention

CNV_DISPLAYMODEL_COB Procedure

Returns either the date or time display model defined for the designated convention. The components of the model are translated to the designated language.

CNV_FORMATDATE_COB Procedure

Formats a numeric date that has the form YYYYMMDD. The numeric date is passed as a parameter to the procedure according to a designated convention and language. The date can be formatted using the long, short, or numeric date format defined in the convention.

CNV_FORMATDATETIMETMP_COB Procedure

Returns the system date, the time, or both in the designated language, formatted according to a template passed to this procedure.

CNV_SYSTEMDATETIMETMP_COB Procedure

Returns the system date, the time, or both, formatted according to the designated convention template and language. You can choose from the following types of formats:

- Long date and long time
- Long date and numeric time
- Short date and long time
- Short date and numeric time
- Numeric date and long time
- Numeric date and numeric time
- Long date only
- Short date only
- Long time only
- Numeric time only

FORMATDATETMP_COB Procedure

Formats a numeric date passed as a parameter to the procedure according to a template and language passed as parameters of the procedure.

Formatting Times According to a Convention

CNV_DISPLAYMODEL_COB Procedure

Returns either the date or time display model defined for the designated convention. The components of the model are translated to the designated language.

CNV_FORMATTIME_COB Procedure

Formats a time with the form HHMMSSPPPP. The time is passed as a parameter to the procedure according to a designated convention and language. The time can be formatted using the long or numeric time format defined in the convention.

CNV_FORMATTIMETMP_COB Procedure

Formats a time passed as a parameter to the procedure according to a template and language passed as parameters of the procedure.

CNV_SYSTEMDATETIMETMP_COB Procedure

Returns the system date, the time, or both in the designated language, formatted according to a template passed to this procedure.

CNV_SYSTEMDATETIME_COB Procedure

Returns the system date, the time, or both, formatted according to the designated convention template and language. You can choose from the following types of formats:

- Long date and long time
- Long date and numeric time
- Short date and long time
- Short date and numeric time
- Numeric date and long time
- Numeric date and numeric time
- Long date only
- Short date only
- Long time only
- Numeric time only

Determining Default Page Size

CNV_FORMSIZE

Returns the default lines-per-page and characters-per-line values defined in a designated convention for formatting printer output.

Calling the CENTRALSUPPORT Library

You can access the procedures in the CENTRALSUPPORT library by using Format 5 of the CALL statement. You can structure your program and the CALL statement syntax to make either an implicit library call or an explicit call. These two types of calls are discussed in the following paragraphs.

Implicit Calls

To make an implicit library call from COBOL85, follow these steps:

1. Declare the parameters of the library procedure in the Working-Storage Section of your program.
2. Use the CHANGE statement to specify the value of the
   - LIBACCESS library attribute as BYFUNCTION
   - FUNCTIONNAME library attribute as "CENTRALSUPPORT"
3. Structure Format 5 of the CALL statement to call a library procedure by
   - Identifying the library as *procedure-name of CENTRALSUPPORT*
   - Listing the parameters in the USING clause
   - Specifying the procedure result in the GIVING clause

An example of the declarations and the syntax necessary to call the CENTRALSUPPORT library is provided in the description of each procedure later in this section.

For more information about calling library procedures, refer to the CALL statement in Section 6 of this manual.

Explicit Calls

You can also use explicit calls to access the procedures in the CENTRALSUPPORT library. To assist you in coding an explicit call, the sample file SYMBOL/INTL/COBOL85/PROPERTIES is available and is listed at the end of this section. To make an explicit library call from COBOL85, follow these steps:

1. Declare the actual parameters of the library procedure in the Working-Storage Section of your program.
2. Include the appropriate declarations for the formal parameters and library entry points using the sample file SYMBOL/INTL/COBOL85/PROPERTIES as a guide.
3. Structure Format 5 of the CALL statement to call a library procedure by
   a. Identifying the library as *procedure-name of CENTRALSUPPORT*
   b. Listing the parameters in the USING clause
   c. Specifying the procedure result in the GIVING clause

For more information about calling library procedures, refer to Format 5 of the CALL statement in Section 6 of this manual.

An example of a program that uses the properties files follows the properties file listing at the end of this section.

Parameter Categories

All integer parameters are passed by reference rather than by value. The CENTRALSUPPORT library procedures return output parameters and procedure result values. The parameter types are further described on the following pages.

Input Parameters

In many cases, one of the input parameters requires that you supply the ccsversion name or number, the language name, or the convention name. You can obtain this information in the following ways:

- If you are a system administrator, a privileged user, or are allowed to use the system console, you can use MARC menus and screens or the SYSOPS command to list the options that exist on your system. The *MLS Guide* provides the instructions needed to obtain information about ccsversion, language, or convention defaults on your system.
- You can call procedures in the CENTRALSUPPORT library that return the default ccsversion, language, and convention on the system. If you are writing an application to be used on another system, you might want to use these library procedures to verify that the ccsversion, the language, or the convention specified by the user is valid on that system.

The fields of parameters that you supply as 01-level records have fixed positions. This means that you must use blanks or zeros in any fields that you omit.

For a complete description of all the supported ccsversions, languages, and conventions, refer to the *MLS Guide*.

Input Parameters with Type Values

Many of the CENTRALSUPPORT procedures have an input parameter that indicates the type of information to be applied or returned in the procedure. The values in these parameters are referred to as type values. The values used in the convention (CNV) procedures are common across all CNV procedures. The values used in the coded character set and ccsversion (VSN) procedures are common across all CCS and VSN procedures.

For example, a parameter that specifies the formatting of the time is used in some procedures. In the examples, the parameter is named CS-NTIMEV. You must choose a type value to indicate a format. For example, a value of 3 indicates the long time format is used.

Because the parameters are passed by reference, you need to declare and assign the type values in the WORKING-STORAGE SECTION. You can then use a MOVE statement to move the value into the parameter as shown in the following example:

```
MOVE LONG-TIME-V TO CS-NTIMEV.
```

Example

Figure 16–3 shows a sample set of WORKING-STORAGE declarations for the parameters that have type values. The procedures are explained later in this section using the type value names declared in this example. Notice that numeric items must be declared PIC S9(11) USAGE BINARY. This declaration is required for parameter matching to type INTEGER.

```
77  CS-BINARYV            PIC S9(11)  USAGE BINARY  VALUE  0.
77  CS-EQUIVALENTV        PIC S9(11)  USAGE BINARY  VALUE  1.
77  CS-LOGICALV           PIC S9(11)  USAGE BINARY  VALUE  2.
77  CS-CHARACTER-SETV     PIC S9(11)  USAGE BINARY  VALUE  0.
77  CS-CCSVERSIONV        PIC S9(11)  USAGE BINARY  VALUE  1.
77  CS-NUMTOALTDIGV       PIC S9(11)  USAGE BINARY  VALUE  5.
77  CS-ALTDIGTONUMV       PIC S9(11)  USAGE BINARY  VALUE  6.
77  CS-LOWTOUPCASEV       PIC S9(11)  USAGE BINARY  VALUE  7.
77  CS-UPTOLOWCASEV       PIC S9(11)  USAGE BINARY  VALUE  8.
77  CS-ESCMENTPERCHARV    PIC S9(11)  USAGE BINARY  VALUE  9.
77  CS-ALPHAV             PIC S9(11)  USAGE BINARY  VALUE 12.
77  CS-NUMERICV           PIC S9(11)  USAGE BINARY  VALUE 13.
77  CS-PRESENTATIONV      PIC S9(11)  USAGE BINARY  VALUE 14.
77  CS-SPACESV            PIC S9(11)  USAGE BINARY  VALUE 15.
77  CS-LOWERCASEV         PIC S9(11)  USAGE BINARY  VALUE 16.
77  CS-UPPERCASEV         PIC S9(11)  USAGE BINARY  VALUE 17.
77  CS-NOTINTSETV         PIC S9(11)  USAGE BINARY  VALUE  0.
77  CS-INTSETV            PIC S9(11)  USAGE BINARY  VALUE  1.
77  CS-CMPLSSV            PIC S9(11)  USAGE BINARY  VALUE  0.
77  CS-CMPLEQV            PIC S9(11)  USAGE BINARY  VALUE  1.
77  CS-CMPEQLV            PIC S9(11)  USAGE BINARY  VALUE  2.
77  CS-CMPGTRV            PIC S9(11)  USAGE BINARY  VALUE  3.
77  CS-CMPGEQV            PIC S9(11)  USAGE BINARY  VALUE  4.
77  CS-CMPNEQV            PIC S9(11)  USAGE BINARY  VALUE  5.
77  CS-LDATE-V            PIC S9(11)  USAGE BINARY  VALUE  0.
77  CS-SDATE-V            PIC S9(11)  USAGE BINARY  VALUE  1.
77  CS-NDATEV             PIC S9(11)  USAGE BINARY  VALUE  2.
77  CS-LTIMEV             PIC S9(11)  USAGE BINARY  VALUE  3.
77  CS-NTIMEV             PIC S9(11)  USAGE BINARY  VALUE  4.
77  CS-LDATELTIMEV        PIC S9(11)  USAGE BINARY  VALUE  5.
77  CS-LDATENTIMEV        PIC S9(11)  USAGE BINARY  VALUE  6.
77  CS-SDATELTIMEV        PIC S9(11)  USAGE BINARY  VALUE  7.
77  CS-SDATENTIMEV        PIC S9(11)  USAGE BINARY  VALUE  8.
77  CS-NDATELTIMEV        PIC S9(11)  USAGE BINARY  VALUE  9.
77  CS-NDATENTIMEV        PIC S9(11)  USAGE BINARY  VALUE 10.
77  CS-LONGDATE-TEMPV     PIC S9(11)  USAGE BINARY  VALUE  0.
77  CS-SHORTDATE-TEMPV    PIC S9(11)  USAGE BINARY  VALUE  1.
77  CS-NUMDATE-TEMPV      PIC S9(11)  USAGE BINARY  VALUE  2.
77  CS-LONGTIME-TEMPV     PIC S9(11)  USAGE BINARY  VALUE  3.
77  CS-NUMTIME-TEMPV      PIC S9(11)  USAGE BINARY  VALUE  4.
77  CS-MONETARY-TEMPV     PIC S9(11)  USAGE BINARY  VALUE  5.
77  CS-NUMERIC-TEMPV      PIC S9(11)  USAGE BINARY  VALUE  6.
```

```
77  CS-DATE-DISPLAYMODEL  PIC S9(11)  USAGE BINARY  VALUE  0.
77  CS-TIME-DISPLAYMODEL  PIC S9(11)  USAGE BINARY  VALUE  1.
```

Figure 16–3. Sample Data Declarations for Type Value Data Items

Output Parameters

These parameters contain the output values produced by the procedure. For example, the translated text produced by the procedure CCSTOCCS_TRANS_TEXT is returned in an output parameter.

Result Parameter

All the library procedures return an integer value as the procedure result that indicates whether an error occurred during the execution of the procedure. In general, a returned value of 1 means that no error occurred, and any other value means that an error occurred. However, the CNV_VALIDATENAME and VSNCOMPARE_TEXT procedures are exceptions to this rule. For these procedures, the returned value can be 0 (zero), 1, or another value. A returned value of 0 (zero) means that no error occurred, and the condition is FALSE. A returned value of 1 means that no error occurred, and the condition is TRUE. Any other value means that an error occurred.

Each procedure lists the values that can be returned by that procedure. The meanings of these values are explained at the end of this section. You can use these values to call the GET_CS_MSG procedure and display the error that occurred, or you can code error routines to handle the possible errors.

Refer to the explanation of the GET_CS_MSG procedure later in this section for more information about using that procedure.

Procedure Descriptions

The following pages describe the internationalization procedures accessible from a COBOL85 program. The procedures reside in the CENTRALSUPPORT library.

Each description includes a general overview of the procedure, an example showing how to call the procedure, and an explanation of the parameters used in the example. Not all parameters used to produce the output can be displayed in the output.

You can define the name of a parameter to be whatever name you want. In the following discussions, the parameters are given names in the example, and are identified in the explanation by that name.

The following parameters are used in many of the procedures:

- CS-DATAOKV is a constant with a value of 1 that is compared with the RESULT parameter. This value indicates that the CENTRALSUPPORT library did not find errors and was able to process the information.
- CS-FALSEV is a constant with a value of 0 that is compared with the RESULT parameter. This value indicates that although the CENTRALSUPPORT library found no errors, it did not process the information.
- SUB represents a subscript.

CCSTOCCS_TRANS_TEXT

This procedure translates data from the 8-bit coded character set specified in the first parameter to the 8-bit coded character set specified in the next parameter. Characters are translated using a one-to-one mapping between two coded character sets. For example, you might want to translate data in the LATIN1EBCDIC coded character set to the LATIN1ISO coded character set.

Although there are many coded character set numbers, there is not a mapping table between every combination of coded character sets. The procedure returns an error indicating the data was not found if you pass two valid coded character set numbers for a table that does not exist.

Refer to the *MLS Guide* for a list of the coded character set to coded character set translate tables that are defined.

Example

Figure 16–4 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the CCSTOCCS_TRANS_TEXT library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example takes the input string "panuelo," which is encoded in the Latin1EBCDIC coded character set and translates it to the Latin1ISO coded character set. The string "panuelo" is represented by the following hexadecimal codes in Latin1EBCDIC: *978195A4859396*. In Latin1ISO, the hexadecimal codes are *70616E75656C6F*. You can use the *MLS Guide* to determine that the coded character set number for Latin1EBCDIC is 12 and Latin1ISO is 13. You can also retrieve these numbers by calling the procedure VALIDATE_NAME_RETURN_NUM with the coded character set names.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT OUTPUT-FILE ASSIGN TO DISK.
DATA DIVISION.
FILE SECTION.
FD  OUTPUT-FILE
    LABEL RECORD IS STANDARD
    VALUE OF TITLE IS "OUT/COBOL85/CCSTOCCSTRANSTEXT."
    PROTECTION SAVE
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS OUTPUT-RECORD.
01  OUTPUT-RECORD        PIC X(80).
WORKING-STORAGE SECTION.
01  OF-1.
    05  FILLER           PIC X(09)   VALUE "RESULT = ".
    05  OF-RESULT        PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(59)   VALUE SPACE.
01  OF-2.
    05  FILLER           PIC X(12)   VALUE "DEST-TEXT = ".
    05  OF-DEST-TEXT     PIC X(10).
    05  FILLER           PIC X(58)   VALUE SPACE.
```

```
  ****************************************************************
  ***  The following global declarations are used as parameters  ***
  ***  to the CENTRALSUPPORT procedures.                         ***
  ****************************************************************
   01  DEST-TEXT            PIC X(10).
   01  SOURCE-TEXT          PIC X(10).
   77  CCS-NUM-FROM         PIC S9(11)  USAGE BINARY.
   77  CCS-NUM-TO           PIC S9(11)  USAGE BINARY.
   77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
   77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
   77  DEST-START           PIC S9(11)  USAGE BINARY.
   77  RESULT               PIC S9(11)  USAGE BINARY.
   77  SOURCE-START         PIC S9(11)  USAGE BINARY.
   77  TRANS-LEN            PIC S9(11)  USAGE BINARY.
   PROCEDURE DIVISION.
   INTLCOBOL85.
       OPEN OUTPUT OUTPUT-FILE.
       PERFORM CCSTOCCS-TRANS-TEXT.
       CLOSE OUTPUT-FILE.
       STOP RUN.
  ***** CCSTOCSS-TRANS-TEXT ***************************************
   CCSTOCCS-TRANS-TEXT.
       CHANGE ATTRIBUTE FUNCTIONNAME OF "CENTRALSUPPORT" TO
"CENTRALSUPPORT".
       CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
       MOVE 12 TO CCS-NUM-FROM.
       MOVE 4  TO CCS-NUM-TO.
       MOVE "panuelo" TO SOURCE-TEXT.
       MOVE 10 TO TRANS-LEN.
       CALL "CCSTOCCS_TRANS_TEXT OF CENTRALSUPPORT"
          USING CCS-NUM-FROM,
                CCS-NUM-TO,
                SOURCE-TEXT,
                SOURCE-START,
                DEST-TEXT,
                DEST-START,
                TRANS-LEN,
          GIVING RESULT.
       MOVE RESULT TO OF-RESULT.
       WRITE OUTPUT-RECORD FROM OF-1.
       IF RESULT IS EQUAL TO CS-DATAOKV
         THEN MOVE DEST-TEXT TO OF-DEST-TEXT
       WRITE OUTPUT-RECORD FROM OF-2.
```

Figure 16–4. Calling the CCSTOCCS_TRANS_TEXT Procedure

Explanation

CCS-NUM-FROM

This is an integer passed to the procedure. It contains the number of the coded character set that you are formatting from.

CCS-NUM-TO

This is an integer passed to the procedure. It contains the number of the coded character set you are translating the text to. The destination text will be in this coded character set.

SOURCE-TEXT

This is passed to the procedure. It contains the text to translate. You specify the size of this record.

SOURCE-START

This is passed to the procedure. It contains the byte offset, relative to 0 (zero), in SOURCE-TEXT where the translation starts.

DEST-TEXT

This is returned by the procedure. It contains the translated text. The size of this record and the record in the SOURCE-TEXT parameter should be the same.

DEST-START

This is passed to the procedure. It contains the byte offset (0 relative) in the DEST-TEXT parameter where the translated text is to be placed.

TRANS-LEN

This is passed to the procedure. It specifies the number of characters in SOURCE-TEXT to be translated, beginning at SOURCE-START.

RESULT

This is returned as the integer value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure.

Possible values returned by CCSTOCCS_TRANS_TEXT are as follows:

1	1000	1001	1002	
3000	3001	3002	4002	4004

For more information on the error result values, see Table 16–6 later in this section.

Sample Output

The output from Figure 16–4 is as follows:

```
RESULT =            1
DEST-TEXT = panuelo
```

CCSTOCCS_TRANS_TEXT_COMPLEX

This procedure provides an advanced character mapping operation that supports the complex translation requirements of mixed, multibyte and 16-bit coded character sets. It translates data from the coded character set specified in the first parameter to the coded character set specified in the next parameter. Characters may be translated using a one-to-one, one-to-many, or many-to-one mapping between two-coded character sets. For example, you might want to translate data in the JapanEBCDICJBIS8 coded character set to the CODEPAGE932 coded character set.

Although there are many coded character set numbers, there is not a translation provided between every combination of coded character sets. The procedure returns an error indicating the data was not found if you pass two valid coded character set numbers for a translation that does not exist.

Refer to the *MLS Guide* for a list of the coded character set to coded character set translations that are defined, as well as a more detailed specification of this procedure including parameters, implementation strategies, and examples.

Example

Figure 16–5 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the CCSTOCCS_TRANS_TEXT_COMPLEX library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example takes the input string "complex," which is encoded in the JapanEBCDICJBIS coded character set and translates it to the CODEPAGE932 coded character set. The string "complex" is represented by the following hexadecimal codes in JapanEBCDICJBIS8: 839694979385A7. In CODEPAGE932, the hexadecimal codes are 636F6D706C6578. You can use the *MLS Guide* to determine that the coded character set number for JapanEBCDICJBIS8 is 100 and CODEPAGE932 is 102. You can also retrieve these numbers by calling the procedure VALIDATE_NAME_RETURN_NUM with the coded character set names.

```
 IDENTIFICATION DIVISION.
 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
   SELECT OUTPUT-FILE ASSIGN TO DISK.
 DATA DIVISION.
 FILE SECTION.
 FD OUTPUT-FILE
    LABEL RECORD IS STANDARD
    VALUE OF TITLE IS "OUT/COBOL85/CCSTOCCSTRANSTEXT/COMPLEX."
     PROTECTION SAVE
     RECORD CONTAINS 80 CHARACTERS
     DATA RECORD IS OUTPUT-RECORD.
 01 OUTPUT-RECORD PIC X(80).
 WORKING-STORAGE SECTION.
 01 OF-1.
    05 FILLER PIC X(09) VALUE "RESULT = ".
    05 OF-RESULT PIC ZZZZZZZZZZ9.
    05 FILLER PIC X(59) VALUE SPACES.
 01 OF-2.
    05 FILLER PIC X(12) VALUE "DEST-TEXT = ".
    05 OF-DEST-TEXT PIC X(20).
    05 FILLER PIC X(48) VALUE SPACES.
************************************************************
*** The following global declarations are used as parameters **
*** to the CENTRALSUPPORT procedures. ***
************************************************************
 01 SOURCE-TEXT PIC X(20).
 01 DEST-TEXT PIC X(20).
 77 CCS-NUM-FROM PIC S9(11) USAGE BINARY.
 77 CCS-NUM-TO PIC S9(11)   USAGE BINARY.
 77 CS-DATAOKV PIC S9(11)   USAGE BINARY VALUE 1.
 77 CS-FALSEV PIC S9(11)    USAGE BINARY VALUE 0.
 77 SOURCE-INX PIC S9(11)   USAGE BINARY.
 77 SOURCE-BYTES PIC S9(11) USAGE BINARY.
 77 DEST-INX PIC S9(11)     USAGE BINARY.
 77 DEST-BYTES PIC S9(11)   USAGE BINARY.
 01 STATE REAL.
    05 STATE-WRD REAL OCCURS 10 TIMES.
 77 OPTION PIC S9(11) BINARY.
 77 RESULT PIC S9(11) BINARY.
*
 LOCAL-STORAGE SECTION.
 LD LD-CCSTOCCS-TRANS-TEXT-COMPLEX.
 77  LD-COMPLEX-CCS-FROM          PIC S9(11)   BINARY CONTENT.
 77  LD-COMPLEX-CCS-TO            PIC S9(11)   BINARY CONTENT.
 01  LD-COMPLEX-SOURCE-TEXT       PIC X(100).
 77  LD-COMPLEX-SOURCE-START      PIC S9(11)   BINARY REFERENCE.
 77  LD-COMPLEX-SOURCE-BYTES      PIC S9(11)   BINARY CONTENT.
 01  LD-COMPLEX-DEST-TEXT         PIC X(100).
 77  LD-COMPLEX-DEST-START        PIC S9(11)   BINARY REFERENCE.
 77  LD-COMPLEX-DEST-BYTES        PIC S9(11)   BINARY CONTENT.
```

```
 01  LD-COMPLEX-STATE             REAL.
     05 LD-COMPLEX-STATE-WRD      REAL         OCCURS 10 TIMES.
 77  LD-COMPLEX-OPTION            PIC S9(11)   BINARY CONTENT.
 77  LD-COMPLEX-RSLT              PIC S9(11)   BINARY.
*
 PROGRAM-LIBRARY SECTION.
 LB CENTRALSUPPORT IMPORT
    ATTRIBUTE
      FUNCTIONNAME IS "CENTRALSUPPORT"
      LIBACCESS IS BYFUNCTION.

 ENTRY PROCEDURE CCSTOCCS-TRANS-TEXT-COMPLEX
    FOR "CCSTOCCS_TRANS_TEXT_COMPLEX"
    WITH LD-CCSTOCCS-TRANS-TEXT-COMPLEX
    USING
       LD-COMPLEX-CCS-FROM,
       LD-COMPLEX-CCS-TO,
       LD-COMPLEX-SOURCE-TEXT,
       LD-COMPLEX-SOURCE-START,
       LD-COMPLEX-SOURCE-BYTES,
       LD-COMPLEX-DEST-TEXT,
       LD-COMPLEX-DEST-START,
       LD-COMPLEX-DEST-BYTES,
       LD-COMPLEX-STATE,
       LD-COMPLEX-OPTION
    GIVING LD-COMPLEX-RSLT.
 PROCEDURE DIVISION.
 INTLCOBOL85.
    OPEN OUTPUT OUTPUT-FILE.
    PERFORM CCSTOCCS-T-T-COMPLEX.
    CLOSE OUTPUT-FILE.
    STOP RUN.
***** CCSTOCCS-TRANS-TEXT-COMPLEX **************************
 CCSTOCCS-T-T-COMPLEX.
    MOVE 84 TO CCS-NUM-FROM.
    MOVE 4  TO CCS-NUM-TO.
    MOVE @63006F006D0070006C00650078 00@ TO SOURCE-TEXT.
    MOVE 14 TO SOURCE-BYTES.
    MOVE 0 TO SOURCE-INX.
    MOVE 10 TO DEST-BYTES.
    MOVE 0 TO DEST-INX.
    MOVE 0 TO OPTION.
    CALL CCSTOCCS-TRANS-TEXT-COMPLEX OF CENTRALSUPPORT
      USING   CCS-NUM-FROM,
              CCS-NUM-TO,
              SOURCE-TEXT,
              SOURCE-INX,
              SOURCE-BYTES,
              DEST-TEXT,
              DEST-INX,
              DEST-BYTES,
              STATE,
```

```
          OPTION
   GIVING RESULT.
MOVE RESULT TO OF-RESULT.
WRITE OUTPUT-RECORD FROM OF-1.
IF RESULT IS EQUAL TO CS-DATAOKV
THEN MOVE DEST-TEXT TO OF-DEST-TEXT
WRITE OUTPUT-RECORD FROM OF-2.
```

Figure 16–5. Calling the CCSTOCCS_TRANS_TEXT_COMPLEX Procedure

Explanation

CCS-NUM-FROM

This is an integer passed to the procedure. It contains the number of the coded character set that you are translating from.

CCS-NUM-TO

This is an integer passed to the procedure. It contains the number of the coded character set you are translating the text to. The destination text is in this coded character set.

SOURCE-TEXT

This is passed to the procedure. It contains the text to translate.

SOURCE-INX

This is passed to the procedure. It contains the byte offset, relative to 0 (zero), in the SOURCE-TEXT where the translation starts.

SOURCE-BYTES

This is passed to the procedure. It specifies the number of characters to translate from SOURCE-TEXT (beginning at SOURCE-INX).

DEST-TEXT

This is returned by the procedure. It contains the translated text. The size of this record might need to be larger than the record in SOURCE-TEXT (twice the size of SOURCE–TEXT always works).

DEST-INX

This is passed to the procedure. It contains the byte offset, relative to 0 (zero), in DEST-TEXT where the translation starts.

DEST-BYTES

This is passed to the procedure. It specifies the maximum number of characters in DEST-TEXT (beginning at DEST–INX) that can be filled with destination characters.

STATE

This is passed to and returned by the procedure. It contains state information required by the procedure for multiple calls. It cannot be updated by the calling program.

OPTION

This is passed to the procedure. It indicates whether or not this is a continuation call to the procedure. The possible values are:

Value	Sample Data Item and Meaning
0	CS_OPT_INITIAL_COMPLETEV First call for a combination of source and destination coded character sets. The source data stream is complete and the data is mapped in one call.
1	CS_OPT_TAILV Second or higher call for a source data stream. This is the last call.
2	CS_OPT_INITIAL_HEADV First call for a combination of source and destination coded character sets. The source data is incomplete or the destination is not large enough to hold all of the data in one call.
3	CS_OPT_MIDDLEV Second or higher call for a source data stream, but not the last call.
5	CS_OPT_COMPLETEV Beginning of a new source data stream, but not the first call for this combination of source and destination coded character sets. The source data stream is complete and the data is mapped in one call.
7	CS_OPT_HEADV Beginning of a new source data stream, but not the first call for this combination of source and destination coded character sets. The source data stream is incomplete or the destination is not large enough to hold all of the data in one call.

RESULT

This is returned as the integer value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure.

Possible values returned by CCSTOCCS_TRANS_TEXT_COMPLEX are as follows:

1	1000	1001	1002	2004
2005	3000	3001	3002	4002

For more information on the error result values, see Table 16–6 later in this section.

Sample Output

The output from Figure 16–5 is as follows:

```
RESULT =            1
DEST-TEXT = complex
```

CCSVSN_NAMES_NUMS

This procedure returns a list of the coded character set names and numbers or a list of the ccsversion names and numbers that are available on your system. You specify which list you want with the first parameter to the procedure. The names and numbers are listed in two arrays. These arrays are coded so that the names in the names array correspond to the numbers in the numbers array.

You might use this procedure to create a menu that lists the ccsversions from which a user can choose. You might also use this procedure to verify that the ccsversion to be used by your program is available on the host computer.

Example

Figure 16–6 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the CCSVSN_NAMES_NUMS library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example returns a list of available ccsversion names and numbers on a system. This is an arbitrary list of ccsversions and might not be the same on every is an arbitrary list of ccsversions and might not be the same on every system.

```
IDENTIFICATION DIVISION.
   ENVIRONMENT DIVISION.
   INPUT-OUTPUT SECTION.
   FILE-CONTROL.
     SELECT OUTPUT-FILE ASSIGN TO DISK.

   DATA DIVISION.
   FILE SECTION.
   FD  OUTPUT-FILE
       LABEL RECORD IS STANDARD
       VALUE OF TITLE IS "OUT/COBOL85/CCSVSNNAMESNUMS."
       PROTECTION SAVE
       RECORD CONTAINS 80 CHARACTERS
       DATA RECORD IS OUTPUT-RECORD.

   01  OUTPUT-RECORD        PIC X(80).

 WORKING-STORAGE SECTION.

   01  OF-1.
       05  FILLER           PIC X(09)   VALUE "RESULT = ".
       05  OF-RESULT        PIC ZZZZZZZZZZZ9.
       05  FILLER           PIC X(59)   VALUE SPACE.
```

```
   01  OF-2.
       05  FILLER           PIC X(15)   VALUE "CCSversion Name".
       05  FILLER           PIC X(05)   VALUE SPACE.
       05  FILLER           PIC X(17)   VALUE "CCSversion Number".
       05  FILLER           PIC X(43)   VALUE SPACE.
   01  OF-3.
       05  FILLER           PIC X(15)   VALUE ALL "-".
       05  FILLER           PIC X(05)   VALUE SPACE.
       05  FILLER           PIC X(17)   VALUE ALL "-".
       05  FILLER           PIC X(43)   VALUE SPACE.
   01  OF-4.
       05  OF-NAMES-ELEM    PIC X(17).
       05  FILLER           PIC X(08)   VALUE SPACE.
       05  OF-NUMS-ELEM     PIC ZZZZZZZZZZ9.
       05  FILLER           PIC X(43)   VALUE SPACE.

  ****************************************************************
  ***  The following global declarations are used as parameters  ***
  ***  to the CENTRALSUPPORT procedures.                         ***
  ****************************************************************

   01  NAMES-ARY.
       05  NAMES-ELEM       PIC X(17)   OCCURS 20 TIMES.
   01  NUMS-ARY                         USAGE BINARY.
       05  NUMS-ELEM        PIC S9(11)  OCCURS 20 TIMES.
   01  SUB                  PIC 9(02).

   77  CS-CCSVERSIONV       PIC S9(11)  USAGE BINARY    VALUE 1.
   77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
   77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
   77  RESULT               PIC S9(11)  USAGE BINARY.
   77  TOTAL                PIC S9(11)  USAGE BINARY.

   PROCEDURE DIVISION.
   INTLCOBOL85.
       OPEN OUTPUT OUTPUT-FILE.
       PERFORM CCSVSNNAMESNUMS.
       CLOSE OUTPUT-FILE.
       STOP RUN.

***** CCSVSNNAMESNUMS ******************************************
   CCSVSNNAMESNUMS.
       CHANGE ATTRIBUTE FUNCTIONNAME OF "CENTRALSUPPORT" TO
"CENTRALSUPPORT".
       CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
       CALL "CCSVSN_NAMES_NUMS OF CENTRALSUPPORT"
         USING CS-CCSVERSIONV,
               TOTAL,
               NAMES-ARY,
               NUMS-ARY
         GIVING RESULT.
       MOVE RESULT TO OF-RESULT.
```

```
        WRITE OUTPUT-RECORD FROM OF-1.
        IF RESULT IS EQUAL TO CS-DATAOKV
            THEN MOVE SPACE TO OUTPUT-RECORD
                 WRITE OUTPUT-RECORD
                 WRITE OUTPUT-RECORD FROM OF-2
                 WRITE OUTPUT-RECORD FROM OF-3
                 MOVE 1 TO SUB
                 PERFORM DISPLAYARY UNTIL SUB IS GREATER THAN TOTAL.

    ***** DISPLAYARY ************************************************
     DISPLAYARY.
        MOVE NAMES-ELEM(SUB) TO OF-NAMES-ELEM.
        MOVE NUMS-ELEM(SUB) TO OF-NUMS-ELEM.
        WRITE OUTPUT-RECORD FROM OF-4.
        ADD 1 TO SUB.
```

Figure 16–6. Calling the CCSVSN_NAMES_NUMS Procedure

Explanation

CS-CCSVERSIONV

This is passed to the procedure. It enables you to specify either of the following two types of information to be returned in the output parameters:

Value	Name and Meaning
0	CS-CHARACTER-SET-V Returns the names and numbers of the coded character sets
1	CS-CCSVERSION-V Returns the names and numbers of the ccsversions

TOTAL is returned by the procedure. It contains the number of coded character set or ccsversion entries that exist.

NAMES-ARY

This is returned by the procedure. Each entry contains the name of a coded character set or ccsversion defined in the file SYSTEM/CCSFILE provided on the release media. Each name uses one element of NAMES-ARY and is 17 characters long. In this example, up to 20 names can be stored in the record. The *MLS Guide* also lists all the coded character sets and ccsversions. The recommended array size is 340 characters.

NUMS-ARY

This is returned by the procedure. NUMS-ARY contains all the coded character set or ccsversion numbers defined on the host. Each number uses one element of NUMS-ARY. Each element in NUMS-ARY corresponds to a parallel entry in NAMES-ARY and corresponds to a 17-character name. The record can hold up to 20 numbers. The *MLS Guide* also provides all the numbers for the coded character sets and ccsversions.

RESULT

This is returned as the value of the procedure. It indicates whether an error occurred in the CCSVSN_NAMES_NUMS procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure. Possible values returned by CCSVSN_NAMES_NUMS are as follows:

1	1001	1002	3000	3001	3006

For more information on the error result values, see Table 16–6 later in this section.

Sample Output

The output from Figure 16–6 is as follows:

```
RESULT =            1

Ccsversion Name     Ccsversion Number
---------------     -----------------
ASERIESNATIVE                        0
SWISS                               64
SWEDISH1                            99
SPANISH                             98
CANADAEBCDIC                        74
CANADAGP                            75
FRANCE                              35
NORWAY                              71
```

CENTRALSTATUS

This procedure returns the name and number of the system default ccsversion, the name of the system default language, and the name of the system default convention. You might use this procedure to provide a means for your application users to inquire about the default settings on the host computer.

Example

Figure 16–7 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the CENTRALSTATUS library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example returns the current values for the system default ccsversion, language, and convention. These are arbitrary system values and might not be the same on every system.

```
IDENTIFICATION DIVISION.
   ENVIRONMENT DIVISION.
   INPUT-OUTPUT SECTION.
   FILE-CONTROL.
     SELECT OUTPUT-FILE ASSIGN TO DISK.

   DATA DIVISION.
   FILE SECTION.
   FD  OUTPUT-FILE
       LABEL RECORD IS STANDARD
       VALUE OF TITLE IS "OUT/COBOL85/CENTRALSTATUS."
       PROTECTION SAVE
       RECORD CONTAINS 80 CHARACTERS
       DATA RECORD IS OUTPUT-RECORD.

   01  OUTPUT-RECORD        PIC X(80).
```

```
WORKING-STORAGE SECTION.

  01  OF-1.
      05  FILLER           PIC X(09)   VALUE "RESULT = ".
      05  OF-RESULT        PIC ZZZZZZZZZZ9.
      05  FILLER           PIC X(59)   VALUE SPACE.
  01  OF-2.
      05  FILLER           PIC X(15)   VALUE "System Defaults".
      05  FILLER           PIC X(65)   VALUE SPACE.
  01  OF-3.
      05  FILLER           PIC X(15)   VALUE ALL "-".
      05  FILLER           PIC X(65)   VALUE SPACE.
  01  OF-4.
      05  FILLER           PIC X(13)   VALUE "Field Meaning".
      05  FILLER           PIC X(29)   VALUE SPACE.
      05  FILLER           PIC X(08)   VALUE "Location".
      05  FILLER           PIC X(08)   VALUE SPACE.
      05  FILLER           PIC X(05)   VALUE "Value".
      05  FILLER           PIC X(14)   VALUE SPACE.
  01  OF-5.
      05  FILLER           PIC X(13)   VALUE ALL "-".
      05  FILLER           PIC X(29)   VALUE SPACE.
      05  FILLER           PIC X(08)   VALUE ALL "-".
      05  FILLER           PIC X(08)   VALUE SPACE.
      05  FILLER           PIC X(05)   VALUE ALL "-".
      05  FILLER           PIC X(14)   VALUE SPACE.
  01  DF-1.
      05  FILLER           PIC X(11)   VALUE "CCSVersion:".
      05  FILLER           PIC X(07)   VALUE SPACE.
      05  D1-SYS-ELEM      PIC X(17).
      05  FILLER           PIC X(45)   VALUE SPACE.
  01  DF-2.
      05  FILLER           PIC X(11)   VALUE "Language:  ".
      05  FILLER           PIC X(07)   VALUE SPACE.
      05  D2-SYS-ELEM      PIC X(17).
      05  FILLER           PIC X(45)   VALUE SPACE.
  01  DF-3.
      05  FILLER           PIC X(11)   VALUE "Convention:".
      05  FILLER           PIC X(07)   VALUE SPACE.
      05  D3-SYS-ELEM      PIC X(17).
      05  FILLER           PIC X(45)   VALUE SPACE.
  01  DF-4.
      05  FILLER           PIC X(39)
          VALUE "System Default CCSVersion Number:      ".
      05  FILLER           PIC X(06)   VALUE SPACE.
      05  FILLER           PIC X(03)   VALUE "(1)".
      05  FILLER           PIC X(03)   VALUE SPACE.
      05  D4-CONTROL-ELEM  PIC ZZZZZZZZZZ9.
      05  FILLER           PIC X(17)   VALUE SPACE.
```

```
 ****************************************************************
 ***  The following global declarations are used as parameters  ***
 ***  to the CENTRALSUPPORT procedures.                         ***
 ****************************************************************

  01  CONTROL-INFO                    USAGE BINARY.
      05  CONTROL-ELEM     PIC S9(11)  OCCURS 8 TIMES.
  01  SUB                  PIC 9(02).
  01  SYS-INFO.
      05  SYS-ELEM         PIC X(17)   OCCURS 3 TIMES.
  77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
  77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
  77  RESULT               PIC S9(11)  USAGE BINARY.

  PROCEDURE DIVISION.
  INTLCOBOL85.
      OPEN OUTPUT OUTPUT-FILE.
      PERFORM CENTRALSTATUS.
      CLOSE OUTPUT-FILE.
      STOP RUN.

 ***** CENTRALSTATUS ********************************************
  CENTRALSTATUS.
      CHANGE ATTRIBUTE FUNCTIONNAME OF "CENTRALSUPPORT" TO
"CENTRALSUPPORT".
      CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
      CALL "CENTRALSTATUS OF CENTRALSUPPORT"
        USING SYS-INFO,
              CONTROL-INFO
        GIVING RESULT.
      MOVE RESULT TO OF-RESULT.
      WRITE OUTPUT-RECORD FROM OF-1.
      IF RESULT IS EQUAL TO CS-DATAOKV
        THEN MOVE SPACE TO OUTPUT-RECORD
             WRITE OUTPUT-RECORD
             WRITE OUTPUT-RECORD FROM OF-2
             WRITE OUTPUT-RECORD FROM OF-3
             MOVE 1 TO SUB
             PERFORM DISPLAYSYSTEMINFO UNTIL SUB IS GREATER THAN 3
             MOVE SPACE TO OUTPUT-RECORD
             WRITE OUTPUT-RECORD
             WRITE OUTPUT-RECORD FROM OF-4
             WRITE OUTPUT-RECORD FROM OF-5
             MOVE 1 TO SUB
             PERFORM DISPLAYCONTROLINFO UNTIL SUB IS GREATER THAN 8.
```

```
***** DISPLAYSYSTEMINFO ****************************************
  DISPLAYSYSTEMINFO.
      IF SUB IS EQUAL TO 1
        THEN MOVE SYS-ELEM(SUB) TO D1-SYS-ELEM
             WRITE OUTPUT-RECORD FROM DF-1.
      IF SUB IS EQUAL TO 2
        THEN MOVE SYS-ELEM(SUB) TO D2-SYS-ELEM
             WRITE OUTPUT-RECORD FROM DF-2.
      IF SUB IS EQUAL TO 3
        THEN MOVE SYS-ELEM(SUB) TO D3-SYS-ELEM
             WRITE OUTPUT-RECORD FROM DF-3.
      ADD 1 TO SUB.
 ***** DISPLAYCONTROLINFO ***************************************
  DISPLAYCONTROLINFO.
      IF SUB IS EQUAL TO 1
        THEN MOVE CONTROL-ELEM(SUB) TO D4-CONTROL-ELEM
             WRITE OUTPUT-RECORD FROM DF-4.
      ADD 1 TO SUB.
```

Figure 16–7. Calling the CENTRALSTATUS Procedure

Explanation

SYS-INFO

This is returned by the procedure. It is recommended that the size of the record be 51 characters. It contains three items, each 17 characters long, in the following order:

1. System default ccsversion name
2. System default language name
3. System default convention name

Each name is 17 characters long. Names shorter than 17 characters are padded on the right with blanks.

CONTROL-INFO

This is returned by the procedure. It is eight words long and contains the following information:

Location	Information
Word [0]	System default ccsversion number
Word [1] through [7]	Reserved

RESULT

This is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure. Possible values returned by CENTRALSTATUS are as follows:

1	1001	1002	3000	3001

For more information on the error result values, see Table 16–6 later in this section.

Sample Output

The output from Figure 16–7 is as follows:

```
RESULT =            1

System Defaults
---------------
CCSVersion:       ASERIESNATIVE
Language:         ENGLISH
Convention:       ASERIESNATIVE

Field Meaning                              Location        Value
-------------                              --------        -----
System Default CCSVersion Number:             (1)              0
```

CNV_CURRENCYEDITTMP_DOUBLE_COB

This procedure receives a double-precision integer and converts it to a formatted monetary value. The procedure uses the monetary template of the convention you specify to accomplish the formatting.

The *MLS Guide* describes all the conventions and the type of currency formatting associated with each convention.

For example, you might want to print a report with the numeric and currency formats for the CostaRica conventions, such as CRC 89.99, or for the Norway conventions, such as NKR 89.99. Figure 16–8 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the CNV_CURRENCYEDITTMP_DOUBLE_COB library procedure. The declarations identify the category of data-item required for parameter matching.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example converts a double-precision integer and edits monetary symbols from the Denmark convention into an EBCDIC string.

```
 IDENTIFICATION DIVISION.
 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
   SELECT OUTPUT-FILE ASSIGN TO DISK.
 DATA DIVISION.
 FILE SECTION.
 FD OUTPUT-FILE
     LABEL RECORD IS STANDARD
     VALUE OF TITLE IS "OUT/COBOL85/CUREDITTMPDOUB."
     PROTECTION SAVE
     RECORD CONTAINS 80 CHARACTERS
     DATA RECORD IS OUTPUT-RECORD.
 01 OUTPUT-RECORD PIC X(80).
 WORKING-STORAGE SECTION.
 01 OF-1.
    05 FILLER PIC X(09) VALUE "RESULT = ".
    05 OF-RESULT PIC ZZZZZZZZZZZ9.
    05 FILLER PIC X(59) VALUE SPACE.
 01 OF-2.
    05 FILLER PIC X(09) VALUE "CE-ARY = ".
    05 OF-CE-ARY PIC X(30).
    05 FILLER PIC X(41) VALUE SPACE.
*****************************************************************
*** The following global declarations are used as parameters ***
*** to the CENTRALSUPPORT procedures. ***
*****************************************************************
 01 CE-ARY PIC X(30).
 01 CNV-NAME PIC X(17).
 01 TMP-ARY PIC X(48).
 77 AMT PIC S9(23) USAGE BINARY.
```

```
 77 PRECISION PIC S9(11) USAGE BINARY.
 77 CS-DATAOKV PIC S9(11) USAGE BINARY VALUE 1.
 77 CS-FALSEV PIC S9(11) USAGE BINARY VALUE 0.
 77 RESULT PIC S9(11) USAGE BINARY.
 PROCEDURE DIVISION.
 INTLCOBOL85.
   OPEN OUTPUT OUTPUT-FILE.
   PERFORM CNV-CURRENCYEDITTMP-DOUBLE-COB.
   CLOSE OUTPUT-FILE.
   STOP RUN.
***** CNV-CURRENCYEDITTMP-DOUBLE-COB ****************************
 CNV-CURRENCYEDITTMP-DOUBLE-COB.
   CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
   MOVE 1234567 TO AMT.
   MOVE 2 TO PRECISION.
   MOVE "ASERIESNATIVE" TO CNV-NAME.
   MOVE "!N[-]CT[,:0,3]D[.]#!" TO TMP-ARY.
   CALL "CNV_CURRENCYEDITTMP_DOUBLE_COB OF CENTRALSUPPORT"
     USING AMT,
           PRECISION,
           TMP-ARY,
           CNV-NAME,
           CE-ARY
     GIVING RESULT.
   MOVE RESULT TO OF-RESULT.
   WRITE OUTPUT-RECORD FROM OF-1.
   IF RESULT IS EQUAL TO CS-DATAOKV
     THEN MOVE CE-ARY TO OF-CE-ARY
          WRITE OUTPUT-RECORD FROM OF-2.
```

Figure 16–8. Calling the CNV_CURRENCYEDITTMP_DOUBLE_COB Procedure

Explanation

AMT

This is a double-precision integer passed to the procedure. It contains the monetary value to be formatted.

PRECISION

This is an integer passed to the procedure. It specifies the number of digits in AMT to be placed after the decimal symbol.

CNV-NAME

This is passed to the procedure. It contains the name of the convention to be used to format the monetary value. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the convention to be used. Refer to the *MLS Guide* for the list of convention names and the explanation of the hierarchy.

CE-ARY

This is returned by the procedure. It contains the formatted monetary value. The recommended size of the formatted amount is 20 characters.

RESULT

This is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. You should check the procedure result whenever you use this procedure. Possible values returned by CNV_CURRENCYEDITTMP_DOUBLE_COB are as follows:

1	1001	1002	2002	3000	3002
3009	3038				

An explanation of the error result values can be found in Table 16–6 later in this section.

Sample Output

The output from Figure 16–8 is as follows:

```
RESULT =            1
CE-ARY = kr.12.345,67
```

CNV_CURRENCYEDIT_DOUBLE_COB

This procedure receives a double-precision integer and converts it to a formatted monetary value. The procedure uses the monetary template of the convention you specify to accomplish the formatting.

The *MLS Guide* describes all of the conventions and the type of currency formatting associated with each convention.

For example, you might want to print a report with the numeric and currency formats for the CostaRica conventions, such as CRC 89.99, or for the Norway conventions, such as NKR 89.99. Figure 16–9 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the CNV_CURRENCYEDIT_DOUBLE_COB library procedure. The declarations identify the category of data-item required for parameter matching.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example converts a double-precision integer and edits monetary symbols from the Denmark convention into an EBCDIC string.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT OUTPUT-FILE ASSIGN TO DISK.
DATA DIVISION.
FILE SECTION.
FD OUTPUT-FILE
   LABEL RECORD IS STANDARD
   VALUE OF TITLE IS "OUT/COBOL85/CUREDITDOUB."
   PROTECTION SAVE
   RECORD CONTAINS 80 CHARACTERS
   DATA RECORD IS OUTPUT-RECORD.
01 OUTPUT-RECORD PIC X(80).
WORKING-STORAGE SECTION.
01 OF-1.
   05 FILLER PIC X(09) VALUE "RESULT = ".
   05 OF-RESULT PIC ZZZZZZZZZZ9.
   05 FILLER PIC X(59) VALUE SPACE.
01 OF-2.
   05 FILLER PIC X(09) VALUE "CE-ARY = ".
   05 OF-CE-ARY PIC X(30).
   05 FILLER PIC X(41) VALUE SPACE.
*****************************************************************
*** The following global declarations are used as parameters ***
*** to the CENTRALSUPPORT procedures. ***
*****************************************************************
01 CE-ARY PIC X(30).
01 CNV-NAME PIC X(17).
77 AMT PIC S9(23) USAGE BINARY.
77 PRECISION PIC S9(11) USAGE BINARY.
```

```
 77 CS-DATAOKV PIC S9(11) USAGE BINARY VALUE 1.
 77 CS-FALSEV PIC S9(11) USAGE BINARY VALUE 0.
 77 RESULT PIC S9(11) USAGE BINARY.
 PROCEDURE DIVISION.
 INTLCOBOL85.
   OPEN OUTPUT OUTPUT-FILE.
   PERFORM CNV-CURRENCYEDIT-DOUBLE-COB.
   CLOSE OUTPUT-FILE.
   STOP RUN.
***** CNV-CURRENCYEDIT-DOUBLE-COB ******************************
 CNV-CURRENCYEDIT-DOUBLE-COB.
   CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
   MOVE 1234567 TO AMT.
   MOVE 2 TO PRECISION.
   MOVE "Denmark" TO CNV-NAME.
   CALL "CNV_CURRENCYEDIT_DOUBLE_COB OF CENTRALSUPPORT"
     USING AMT,
           PRECISION,
           CNV-NAME,
           CE-ARY
     GIVING RESULT.
   MOVE RESULT TO OF-RESULT.
   WRITE OUTPUT-RECORD FROM OF-1.
   IF RESULT IS EQUAL TO CS-DATAOKV
     THEN MOVE CE-ARY TO OF-CE-ARY
          WRITE OUTPUT-RECORD FROM OF-2.
```

Figure 16–9. Calling the CNV_CURRENCYEDIT_DOUBLE_COB Procedure

Explanation

AMT

This is a double-precision integer passed to the procedure. It contains the monetary value to be formatted.

PRECISION

This is a integer passed to the procedure. It specifies the number of digits in AMT to be placed after the decimal symbol.

CNV-NAME

This is passed to the procedure. It contains the name of the convention to be used to format the monetary value. If this parameter contains all blanks or zeros, the procedure will use the hierarchy to determine the convention to be used. Refer to the *MLS Guide* for the list of convention names and the explanation of the hierarchy.

CE-ARY

This is returned by the procedure. It contains the formatted monetary value.

RESULT

This is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure. Possible values returned by CNV_CURRENCYEDIT_DOUBLE_COB are as follows:

1	1001	1002	2002	3000	3002
3009	3038				

For more information on the error result values, see Table 16–6 later in this section.

Sample Output

The output from Figure 16–9 is as follows:

```
RESULT =            1
CE-ARY = Kr.12 345,67
```

CNV_DISPLAYMODEL_COB

This procedure returns either a numeric date or numeric time display model. A display model is a format that you can display to the user to show the form of the requested input. For example, YYDDMM is a display model that shows a user that the date must be entered in that form. The procedure creates the display model according to the convention and language that you specify.

Example

Figure 16–10 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the CNV_DISPLAYMODEL_COB library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example obtains a date display model from the ASeriesNative convention. The display model is translated to English and returned in DM-ARY.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT OUTPUT-FILE ASSIGN TO DISK.

DATA DIVISION.
FILE SECTION.
FD  OUTPUT-FILE
    LABEL RECORD IS STANDARD
    VALUE OF TITLE IS "OUT/COBOL85/CNVDSPMODELCOB."
    PROTECTION SAVE
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS OUTPUT-RECORD.

01  OUTPUT-RECORD        PIC X(80).

WORKING-STORAGE SECTION.
  01  OF-1.
    05  FILLER           PIC X(09)   VALUE "RESULT = ".
    05  OF-RESULT        PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(59)   VALUE SPACE.
01  OF-2.
    05  FILLER           PIC X(09)   VALUE "DM-ARY = ".
    05  OF-DM-ARY        PIC X(10).
    05  FILLER           PIC X(61)   VALUE SPACE.
```

```
  ****************************************************************
  ***  The following global declarations are used as parameters  ***
  ***  to the CENTRALSUPPORT procedures.                         ***
  ****************************************************************

   01  CNV-NAME             PIC X(17).
   01  DM-ARY               PIC X(10).
   01  LANG-NAME            PIC X(17).

   77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
   77  CS-DATE-DISPLAYMODEL PIC S9(11)  USAGE BINARY    VALUE 0.
   77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
   77  RESULT               PIC S9(11)  USAGE BINARY.

   PROCEDURE DIVISION.
   INTLCOBOL85.
       OPEN OUTPUT OUTPUT-FILE.
       PERFORM CNV-DISPLAYMODEL-COB.
       CLOSE OUTPUT-FILE.
       STOP RUN.

  ***** CNV-DISPLAYMODEL-COB ************************************
   CNV-DISPLAYMODEL-COB.
       CHANGE ATTRIBUTE FUNCTIONNAME OF "CENTRALSUPPORT" TO
"CENTRALSUPPORT".
       CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
       MOVE "ASERIESNATIVE" TO CNV-NAME.
       MOVE "ENGLISH" TO LANG-NAME.
       CALL "CNV_DISPLAYMODEL_COB OF CENTRALSUPPORT"
         USING CS-DATE-DISPLAYMODEL,
               CNV-NAME,
               LANG-NAME,
               DM-ARY
         GIVING RESULT.
       MOVE RESULT TO OF-RESULT.
       WRITE OUTPUT-RECORD FROM OF-1.
       IF RESULT IS EQUAL TO CS-DATAOKV
         THEN MOVE DM-ARY TO OF-DM-ARY
              WRITE OUTPUT-RECORD FROM OF-2.
```

Figure 16–10. Calling the CNV_DISPLAYMODEL_COB Procedure

Explanation

CS-DATE-DISPLAYMODEL

This is an integer that is passed to the procedure. It indicates whether you want the display model to be a numeric date or a numeric time. The possible values are

Value	Sample Data Item and Meaning
0	CS-DATE-DISPLAYMODEL The display model will be a numeric date.
1	CS-TIME-DISPLAYMODEL The display model will be a numeric time.

CNV-NAME

This is passed to the procedure. It contains the name of the convention from which the date or time model is retrieved. If this parameter contains all blanks or zeros, the procedure will use the hierarchy to determine the convention to be used. Refer to the *MLS Guide* for the list of convention names and the explanation of the hierarchy.

LANG-NAME

This is passed to the procedure. It contains the name of the language in which the date or time components are to be displayed. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the language to be used. Refer to the *MLS Guide* for information about determining the valid language names on your system and the explanation of the hierarchy.

DM-ARY

This is returned by the procedure. It contains the display model. The recommended size of the display model is 10 characters.

RESULT

This is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure. Possible values returned by CNV_DISPLAYMODEL_COB are as follows:

1	1001	1002	2001	2002
3000	3001	3002	3006	

For more information on the error result values, see Table 16–6 later in this section.

Sample Output

The output from Figure 16–10 is as follows:

```
RESULT =             1
DM-ARY = mm/dd/yyyy
```

CNV_FORMATDATETMP_COB

This procedure formats a date according to a template. You specify the template, date value, and language in which the date is to be displayed. The procedure then returns the formatted date. The template may be retrieved for any convention from the CNV_TEMPLATE_COB procedure or may be created by the user.

Example

Figure 16–11 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the CNV_FORMATDATETMP_COB library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example formats a date using a template provided by the calling program. The formatted date is translated to English and returned in FD-ARY. The date consists of an unabridged day of week name, abbreviated month name, numeric day of month, day of month suffix “rd,” and numeric year.

```
    IDENTIFICATION DIVISION.
    ENVIRONMENT DIVISION.
    INPUT-OUTPUT SECTION.
    FILE-CONTROL.
      SELECT OUTPUT-FILE ASSIGN TO DISK.

    DATA DIVISION.
    FILE SECTION.
    FD  OUTPUT-FILE
        LABEL RECORD IS STANDARD
        VALUE OF TITLE IS "OUT/COBOL85/CNVFMTDATETMPCOB."
        PROTECTION SAVE
        RECORD CONTAINS 80 CHARACTERS
        DATA RECORD IS OUTPUT-RECORD.

    01  OUTPUT-RECORD        PIC X(80).

  WORKING-STORAGE SECTION.
       01  OF-1.
         05  FILLER          PIC X(09)   VALUE "RESULT = ".
         05  OF-RESULT       PIC ZZZZZZZZZZZ9.
```

```
           05  FILLER           PIC X(59)   VALUE SPACE.
       01  OF-2.
           05  FILLER           PIC X(09)   VALUE "FD-ARY = ".
           05  OF-FD-ARY        PIC X(45).
           05  FILLER           PIC X(26)   VALUE SPACE.

      ****************************************************************
      ***  The following global declarations are used as parameters  ***
      ***  to the CENTRALSUPPORT procedures.                         ***
      ****************************************************************

       01  DATE-ARY             PIC X(08).
       01  FD-ARY               PIC X(45).
       01  LANG-NAME            PIC X(17).
       01  TMP-ARY              PIC X(48).

       77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
       77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
       77  RESULT               PIC S9(11)  USAGE BINARY.

       PROCEDURE DIVISION.
       INTLCOBOL85.
           OPEN OUTPUT OUTPUT-FILE.
           PERFORM CNV-FORMATDATETMP-COB.
           CLOSE OUTPUT-FILE.
           STOP RUN.

      ***** CNV-FORMATDATETMP-COB ************************************
       CNV-FORMATDATETMP-COB.
           CHANGE ATTRIBUTE FUNCTIONNAME OF "CENTRALSUPPORT" TO
    "CENTRALSUPPORT".
           CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
           MOVE "19901003" TO DATE-ARY.
           MOVE "!W!, !1N!. !DE!, !Y!" TO TMP-ARY.
           MOVE "ENGLISH" TO LANG-NAME.
           CALL "CNV_FORMATDATETMP_COB OF CENTRALSUPPORT"
             USING DATE-ARY,
                   TMP-ARY,
                   LANG-NAME,
                   FD-ARY
             GIVING RESULT.
           MOVE RESULT TO OF-RESULT.
           WRITE OUTPUT-RECORD FROM OF-1.
           IF RESULT IS EQUAL TO CS-DATAOKV
             THEN MOVE FD-ARY TO OF-FD-ARY
                  WRITE OUTPUT-RECORD FROM OF-2.
```

Figure 16–11. Calling the CNV_FORMATDATETMP_COB Procedure

Explanation

DATE-ARY

This is passed into the procedure. It contains the date to be formatted. The date must be in the form YYYYMMDD. The fields of the record have fixed positions. You must use blanks or zeros in any fields that you omit.

TMP-ARY

This is passed into the procedure. It contains the template used to format the date. The recommended size of a template is 48 characters. The template must use the control characters described in the *MLS Guide*.

LANG-NAME

This is passed into the procedure. It contains the name of the language to be used in formatting the date. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the language to be used. Refer to the *MLS Guide* for information about determining the valid language names on your system.

FD-ARY

This is returned by the procedure. It contains the formatted date. The recommended size of the formatted date is 45 characters.

RESULT

This is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure. Possible values returned by CNV_FORMATDATETMP_COB are as follows:

1	1001	1002	2001	3000	3001
3002	3011	3012	3030	3045	3046
3047	3048	3057	3058	3059	

For more information on the error result values, see Table 16–6 later in this section.

Sample Output

The output from Figure 16–11 is as follows:

```
RESULT =             1
FD-ARY = Wednesday, Oct. 3rd, 1990
```

CNV_FORMATDATE_COB

This procedure receives a date and formats it in the form you select according to the convention and language you specify.

You might use this procedure to output a date according to the Greek long-date format and Greek language, for example.

Example

Figure 16–12 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the CNV_FORMATDATE_COB library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example formats the date in numeric form using the Netherlands convention. The English language is specified.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT OUTPUT-FILE ASSIGN TO DISK.

DATA DIVISION.
FILE SECTION.
FD  OUTPUT-FILE
    LABEL RECORD IS STANDARD
    VALUE OF TITLE IS "OUT/COBOL85/CNVFMTDATECOB."
    PROTECTION SAVE
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS OUTPUT-RECORD.

01  OUTPUT-RECORD        PIC X(80).

WORKING-STORAGE SECTION.
  01  OF-1.
    05  FILLER           PIC X(09)   VALUE "RESULT = ".
    05  OF-RESULT        PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(59)   VALUE SPACE.
  01  OF-2.
    05  FILLER           PIC X(09)   VALUE "FD-ARY = ".
    05  OF-FD-ARY        PIC X(10).
    05  FILLER           PIC X(61)   VALUE SPACE.
```

```
 ****************************************************************
 ***  The following global declarations are used as parameters  ***
 ***  to the CENTRALSUPPORT procedures.                         ***
 ****************************************************************

  01  CNV-NAME            PIC X(17).
  01  DATE-ARY            PIC X(08).
  01  FD-ARY              PIC X(10).
  01  LANG-NAME           PIC X(17).

  77  CS-DATAOKV          PIC S9(11)  USAGE BINARY    VALUE 1.
  77  CS-FALSEV           PIC S9(11)  USAGE BINARY    VALUE 0.
  77  CS-NDATEV           PIC S9(11)  USAGE BINARY    VALUE 2.
  77  RESULT              PIC S9(11)  USAGE BINARY.

  PROCEDURE DIVISION.
  INTLCOBOL85.
      OPEN OUTPUT OUTPUT-FILE.
      PERFORM CNV-FORMATDATE-COB.
      CLOSE OUTPUT-FILE.
      STOP RUN.
   ***** CNV-FORMATDATE-COB ***************************************
  CNV-FORMATDATE-COB.
      CHANGE ATTRIBUTE FUNCTIONNAME OF "CENTRALSUPPORT" TO
"CENTRALSUPPORT".
      CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
      MOVE "17760704" TO DATE-ARY.
      MOVE "Netherlands" TO CNV-NAME.
      MOVE "ENGLISH" TO LANG-NAME.
      CALL "CNV_FORMATDATE_COB OF CENTRALSUPPORT"
        USING CS-NDATEV,
              DATE-ARY,
              CNV-NAME,
              LANG-NAME,
              FD-ARY
        GIVING RESULT.
      MOVE RESULT TO OF-RESULT.
      WRITE OUTPUT-RECORD FROM OF-1.
      IF RESULT IS EQUAL TO CS-DATAOKV
        THEN MOVE FD-ARY TO OF-FD-ARY
             WRITE OUTPUT-RECORD FROM OF-2.
```

Figure 16–12. Calling the CNV_FORMATDATE_COB Procedure

Explanation

CS-NDATEV

This is an integer passed by reference to the procedure. It indicates which of the following three formats will be used to format the date:

Value	Sample Data Item and Meaning
0	LONG-DATE-V Use the long date format.
1	SHORT-DATE-V Use the short date format.
2	NUMERIC-DATE-V Use the numeric date format.

DATE-ARY

This is passed to the procedure. It contains the date to be formatted. The date must be in the form YYYYMMDD, left justified. The fields of the array have fixed positions. You must use blanks or zeros in any fields that you omit.

CNV-NAME

This is passed to the procedure. It contains the name of the convention to be used to format the date value. If this parameter contains all blanks or 17 character zeros, the procedure uses the hierarchy to determine the convention to be used. Refer to the *MLS Guide* for the list of convention names and the explanation of the hierarchy.

LANG-NAME

This is passed to the procedure. It contains the language to be used in formatting the date. If this parameter contains all blanks or 17 character zeros, the procedure uses the hierarchy to determine the language to be used. Refer to the *MLS Guide* for information about determining the valid language names on your system and the explanation of the hierarchy.

FD-ARY

This is returned by the procedure. It contains the formatted date. The recommended length of a formatted date is 45 characters.

RESULT

This is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure. Possible values returned by CNV_FORMATDATE_COB are as follows:

1	2	1001	2001	2002	3000
3001	3002	3006	3012	3045	3046
3047	3048	3057	3058	3059	

For more information on the error result values, see Table 16–6 later in this section.

Sample Output

Sample output from Figure 16–12 follows:

```
RESULT =            1
FD-ARY = 4.7.76
```

CNV_FORMATTIMETMP_COB

This procedure formats a time according to a template. You specify the template, time value, and language in which the time is to be displayed. The procedure then returns the formatted time. The template may be retrieved for any convention from the CNV_TEMPLATE_COB procedure or may be created by the user.

With this procedure, if the time template is !0t!:!0m!:0s!, the language is English, and the input time is 1255016256, the numeric time is formatted as 12:55:01.

Example

Figure 16–13 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the CNV_FORMATTIMETMP_COB library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example formats a caller-supplied time using a template also passed in by the calling program. Alphabetic time components are translated to English and returned in FT-ARY.

```
IDENTIFICATION DIVISION.
   ENVIRONMENT DIVISION.
   INPUT-OUTPUT SECTION.
   FILE-CONTROL.
     SELECT OUTPUT-FILE ASSIGN TO DISK.

   DATA DIVISION.
   FILE SECTION.
   FD  OUTPUT-FILE
       LABEL RECORD IS STANDARD
       VALUE OF TITLE IS "OUT/COBOL85/CNVFMTTIMETMPCOB."
       PROTECTION SAVE
       RECORD CONTAINS 80 CHARACTERS
       DATA RECORD IS OUTPUT-RECORD.
     01  OUTPUT-RECORD        PIC X(80).

   WORKING-STORAGE SECTION.
     01  OF-1.
       05  FILLER           PIC X(09)   VALUE "RESULT = ".
       05  OF-RESULT        PIC ZZZZZZZZZZ9.
       05  FILLER           PIC X(59)   VALUE SPACE.
     01  OF-2.
       05  FILLER           PIC X(13)   VALUE "FT-ARY = ".
       05  OF-FT-ARY        PIC X(30).
       05  FILLER           PIC X(37)   VALUE SPACE.
```

```
 ****************************************************************
 ***  The following global declarations are used as parameters  ***
 ***  to the CENTRALSUPPORT procedures.                         ***
 ****************************************************************

  01  FT-ARY              PIC X(30).
  01  LANG-NAME           PIC X(17).
  01  TIME-ARY            PIC X(10).
  01  TMP-ARY             PIC X(48).

  77  CS-DATAOKV          PIC S9(11)  USAGE BINARY    VALUE 1.
  77  CS-FALSEV           PIC S9(11)  USAGE BINARY    VALUE 0.
  77  RESULT              PIC S9(11)  USAGE BINARY.

  PROCEDURE DIVISION.
  INTLCOBOL85.
      OPEN OUTPUT OUTPUT-FILE.
      PERFORM CNV-FORMATTIMETMP-COB.
      CLOSE OUTPUT-FILE.
      STOP RUN.

 ***** CNV-FORMATTIMETMP-COB ************************************
  CNV-FORMATTIMETMP-COB.
      CHANGE ATTRIBUTE FUNCTIONNAME OF "CENTRALSUPPORT" TO
"CENTRALSUPPORT".
      CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
      MOVE "114958" TO TIME-ARY.
      MOVE "!T! !I! !M! !K! !S! !R!" TO TMP-ARY.
      MOVE "ENGLISH" TO LANG-NAME.
      CALL "CNV_FORMATTIMETMP_COB OF CENTRALSUPPORT"
        USING TIME-ARY,
              TMP-ARY,
              LANG-NAME,
              FT-ARY
        GIVING RESULT.
      MOVE RESULT TO OF-RESULT.
      WRITE OUTPUT-RECORD FROM OF-1.
      IF RESULT IS EQUAL TO CS-DATAOKV
        THEN MOVE FT-ARY TO OF-FT-ARY
             WRITE OUTPUT-RECORD FROM OF-2.
```

Figure 16–13. Calling the CNV_FORMATTIMETMP_COB Procedure

Explanation

TIME-ARY

This is passed to the procedure. You specify the time to be formatted in the form HHMMSSPPPP. The partial seconds field, PPPP, is optional. The fields of the array have fixed positions. You must use blanks or zeros in any fields that you omit.

TMP-ARY

This is passed to the procedure. You specify the template to be used to format the time in this parameter. The recommended length of a template is 48 characters. Refer to the *MLS Guide* for information about creating a template.

LANG-NAME

This is passed to the procedure. You specify the language to be used in formatting the time in this parameter. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the language to be used. Refer to the *MLS Guide* for information about determining the valid language names on your system and the explanation of the hierarchy.

FT-ARY

This is returned by the procedure. It contains the time value formatted according to the template and language you designated. The recommended length of a formatted time is 45 characters.

RESULT

This is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure.

Possible values returned by CNV_FORMATTIMETMP_COB are as follows:

1	1001	1002	2001	3000	3001
3002	3011	3013	3030	3051	3052
3053	3054	3055			

For more information on the error result values, see Table 16–6 later in this section.

Sample Output

The output from Figure 16–13 is as follows:

```
RESULT =            1
FT-ARY =     11 hours 49 minutes 58 seconds
```

CNV_FORMATTIME_COB

This procedure formats a user-supplied time according to the convention, language, and type of time that you specify.

Example

Figure 16–14 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the CNV_FORMATTIME_COB library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example formats the time in numeric form using the Belgium convention. The formatted time is returned in FT-ARY.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT OUTPUT-FILE ASSIGN TO DISK.

DATA DIVISION.
FILE SECTION.
FD  OUTPUT-FILE
    LABEL RECORD IS STANDARD
    VALUE OF TITLE IS "OUT/COBOL85/CNVFMTTIMECOB."

    PROTECTION SAVE
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS OUTPUT-RECORD.

01  OUTPUT-RECORD        PIC X(80).

WORKING-STORAGE SECTION.
  01  OF-1.
    05  FILLER           PIC X(09)   VALUE "RESULT = ".
    05  OF-RESULT        PIC ZZZZZZZZZZ9.
    05  FILLER           PIC X(59)   VALUE SPACE.
01  OF-2.
    05  FILLER           PIC X(09)   VALUE "FT-ARY = ".
    05  OF-FT-ARY        PIC X(30).
    05  FILLER           PIC X(41)   VALUE SPACE.
```

```
 ****************************************************************
 ***  The following global declarations are used as parameters  ***
 ***  to the CENTRALSUPPORT procedures.                         ***
 ****************************************************************

  01  CNV-NAME            PIC X(17).
  01  FT-ARY              PIC X(30).
  01  LANG-NAME           PIC X(17).
  01  TIME-ARY            PIC X(10).

  77  CS-DATAOKV          PIC S9(11)  USAGE BINARY    VALUE 1.
  77  CS-FALSEV           PIC S9(11)  USAGE BINARY    VALUE 0.
  77  CS-NTIMEV           PIC S9(11)  USAGE BINARY    VALUE 4.
  77  RESULT              PIC S9(11)  USAGE BINARY.

  PROCEDURE DIVISION.
  INTLCOBOL85.
      OPEN OUTPUT OUTPUT-FILE.
      PERFORM CNV-FORMATTIME-COB.
      CLOSE OUTPUT-FILE.
      STOP RUN.

 ***** CNV-FORMATTIME-COB ****************************************
  CNV-FORMATTIME-COB.
      CHANGE ATTRIBUTE FUNCTIONNAME OF "CENTRALSUPPORT" TO
"CENTRALSUPPORT".
      CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
      MOVE "114958" TO TIME-ARY.
      MOVE "Belgium" TO CNV-NAME.
      MOVE "ENGLISH" TO LANG-NAME.
      CALL "CNV_FORMATTIME_COB OF CENTRALSUPPORT"
        USING CS-NTIMEV,
              TIME-ARY,
              CNV-NAME,
              LANG-NAME,
              FT-ARY
        GIVING RESULT.
      MOVE RESULT TO OF-RESULT.
      WRITE OUTPUT-RECORD FROM OF-1.
      IF RESULT IS EQUAL TO CS-DATAOKV
        THEN MOVE FT-ARY TO OF-FT-ARY
             WRITE OUTPUT-RECORD FROM OF-2.
```

Figure 16–14. Calling the CNV_FORMATTIME_COB Procedure

Explanation

CS-NTIMEV

This is passed by reference to the procedure. It indicates which of the following two formats will be used to edit the time:

Value	Sample Data Item and Meaning
3	LONG-TIME-V Use the long time format.
4	NUMERIC-TIME-V Use the numeric time format.

TIME-ARY

This is passed to the procedure. It contains the time to be formatted in the form HHMMSSPPPP, left justified. The partial seconds field, PPPP, is optional. The fields of the record have fixed positions. You must use blanks or zeros in any fields that you omit.

CNV-NAME

This is passed to the procedure. It contains the name of the convention to be used to edit the time value. If this parameter contains all blanks or zeros, the procedure will use the hierarchy to determine the convention to be used. Refer to the *MLS Guide* for the list of convention names and the explanation of the hierarchy.

LANG-NAME

This is passed into the procedure. It contains the language to be used in formatting the time. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the language to be used. Refer to the *MLS Guide* for information about determining the valid language names on your system and the explanation of the hierarchy.

FT-ARY

This is returned by the procedure. It contains the formatted time value. The recommended length of the formatted time is 45 characters.

RESULT

This is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure. Possible values returned by CNV_FORMATTIME_COB are as follows:

1	1001	1002	2001	2002	3000
3001	3002	3006	3011	3013	3051
3052	3053	3054	3055		

For more information on the error result values, see Table 16–6 later in this section.

Sample Output

The sample output from Figure 16–14 is as follows:

```
RESULT =           1
 FT-ARY =  11:49:58
```

CNV_FORMSIZE

This procedure returns the default lines-per-page and default characters-per-line values from the specified convention. Each convention provides these values to be used with printed output.

Example

Figure 16–15 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the CNV_FORMSIZE library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example obtains paper dimensions (lines per page and characters per line) from the Denmark convention.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT OUTPUT-FILE ASSIGN TO DISK.

DATA DIVISION.
FILE SECTION.
FD  OUTPUT-FILE
    LABEL RECORD IS STANDARD
    VALUE OF TITLE IS "OUT/COBOL85/CNVFORMSIZE."
    PROTECTION SAVE
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS OUTPUT-RECORD.

01  OUTPUT-RECORD        PIC X(80).

WORKING-STORAGE SECTION.

01  OF-1.
    05  FILLER           PIC X(22)
        VALUE "RESULT              = ".
    05  OF-RESULT        PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(46)   VALUE SPACE.
01  OF-2.
    05  FILLER           PIC X(22)
        VALUE "Lines per Page      = ".
    05  OF-LPP           PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(46)   VALUE SPACE.
```

```
01  OF-3.
       05  FILLER           PIC X(22)
           VALUE "Characters per Line = ".
       05  OF-CPL           PIC ZZZZZZZZZZ9.
       05  FILLER           PIC X(46)   VALUE SPACE.

  ****************************************************************
  ***  The following global declarations are used as parameters  ***
  ***  to the CENTRALSUPPORT procedures.                         ***
  ****************************************************************

   01  CNV-NAME             PIC X(17).

   77  CPL                  PIC S9(11)  USAGE BINARY.
   77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
   77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
   77  LPP                  PIC S9(11)  USAGE BINARY.
   77  RESULT               PIC S9(11)  USAGE BINARY.

   PROCEDURE DIVISION.
   INTLCOBOL85.
       OPEN OUTPUT OUTPUT-FILE.
       PERFORM CNV-FORMSIZE.
       CLOSE OUTPUT-FILE.
       STOP RUN.

  ***** CNV-FORMSIZE ***********************************************
   CNV-FORMSIZE.
       CHANGE ATTRIBUTE FUNCTIONNAME OF "CENTRALSUPPORT" TO
"CENTRALSUPPORT".
       CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
       MOVE "Denmark" TO CNV-NAME.
       CALL "CNV_FORMSIZE OF CENTRALSUPPORT"
         USING CNV-NAME,
               LPP,
               CPL
         GIVING RESULT.
       MOVE RESULT TO OF-RESULT.
       WRITE OUTPUT-RECORD FROM OF-1.
       IF RESULT IS EQUAL TO CS-DATAOKV
         THEN MOVE SPACE TO OUTPUT-RECORD
              MOVE LPP TO OF-LPP
              WRITE OUTPUT-RECORD FROM OF-2
              MOVE CPL TO OF-CPL
              WRITE OUTPUT-RECORD FROM OF-3.
```

Figure 16–15. Calling the CNV_FORMSIZE Procedure

Explanation

CNV-NAME

This is passed to the procedure. It contains the name of the convention to be used to specify the default printer form sizes. If this parameter contains all blanks or zeros, the procedure will use the hierarchy to determine the convention to be used. Refer to the *MLS Guide* for the list of convention names and the explanation of the hierarchy.

CPL

This is returned by the procedure. It contains the default number of characters per line specified by the convention you referenced.

LPP

This is returned by the procedure. It contains the default number of lines per page specified by the convention you referenced.

RESULT

This is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure. Possible values returned by CNV_FORMSIZE are as follows:

1	1001	1002	2002	3000

For more information on the error result values, see Table 16–6 later in this section.

Sample Output

The output from Figure 16–15 is as follows:

```
RESULT =                          1
Lines per Page      =            70
Characters per Line =            82
```

CNV_NAMES

This procedure returns a list of convention names and the total number of convention that are available on the system. The first name is the system default name.

You might use this procedure to obtain the name of a convention to be used as input to another procedure.

Example

Figure 16–16 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the CNV_NAMES library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example obtains the names of conventions currently available on the system. Note that this is an arbitrary list that may differ from system to system.

```
     IDENTIFICATION DIVISION.
     ENVIRONMENT DIVISION.
     INPUT-OUTPUT SECTION.
     FILE-CONTROL.
         SELECT OUTPUT-FILE ASSIGN TO DISK.

     DATA DIVISION.
     FILE SECTION.
     FD  OUTPUT-FILE
         LABEL RECORD IS STANDARD
         VALUE OF TITLE IS "OUT/COBOL85/CNVNAMES."
         PROTECTION SAVE
         RECORD CONTAINS 80 CHARACTERS
         DATA RECORD IS OUTPUT-RECORD.

     01  OUTPUT-RECORD        PIC X(80).

     WORKING-STORAGE SECTION.

     01  OF-1.
         05  FILLER           PIC X(09)   VALUE "RESULT = ".
         05  OF-RESULT        PIC ZZZZZZZZZZZ9.
         05  FILLER           PIC X(59)   VALUE SPACE.
     01  OF-2.
         05  FILLER           PIC X(16)   VALUE "Convention Names".
         05  FILLER           PIC X(64)   VALUE SPACE.
   01  OF-3.
         05  FILLER           PIC X(16)   VALUE ALL "-".
         05  FILLER           PIC X(64)   VALUE SPACE.
     01  OF-4.
```

```
           05  OF-NAMES-ELEM    PIC X(17).
           05  FILLER           PIC X(63)   VALUE SPACE.
      ****************************************************************
      ***  The following global declarations are used as parameters  ***
      ***  to the CENTRALSUPPORT procedures.                         ***
      ****************************************************************

       01  NAMES-ARY.
           05  NAMES-ELEM       PIC X(17)   OCCURS 80 TIMES.

       77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
       77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
       77  RESULT               PIC S9(11)  USAGE BINARY.
       77  SUB                  PIC S9(11)  USAGE BINARY.
       77  TOTAL                PIC S9(11)  USAGE BINARY.

       PROCEDURE DIVISION.
       INTLCOBOL85.
           OPEN OUTPUT OUTPUT-FILE.
           PERFORM CNV-NAMES.
           CLOSE OUTPUT-FILE.
           STOP RUN.
      ***** CNV-NAMES ************************************************
       CNV-NAMES.
           CHANGE ATTRIBUTE FUNCTIONNAME OF "CENTRALSUPPORT" TO
    "CENTRALSUPPORT".
           CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
           CALL "CNV_NAMES OF CENTRALSUPPORT"
             USING TOTAL,
                   NAMES-ARY
             GIVING RESULT.
           MOVE RESULT TO OF-RESULT.
           WRITE OUTPUT-RECORD FROM OF-1.
           IF RESULT IS EQUAL TO CS-DATAOKV
             THEN MOVE SPACE TO OUTPUT-RECORD
                  WRITE OUTPUT-RECORD
                  WRITE OUTPUT-RECORD FROM OF-2
                  WRITE OUTPUT-RECORD FROM OF-3
                  MOVE 1 TO SUB
                  PERFORM DISPLAYNAMESARY UNTIL SUB IS GREATER THAN TOTAL.

      ***** DISPLAYNAMESARY ******************************************
       DISPLAYNAMESARY.
           MOVE NAMES-ELEM(SUB) TO OF-NAMES-ELEM.
           WRITE OUTPUT-RECORD FROM OF-4.
           ADD 1 TO SUB.
```

Figure 16–16. Calling the CNV_NAMES Procedure

Explanation

TOTAL

This is returned by the procedure. It contains the total number of conventions that reside on the system.

NAMES-ARY

This is returned by the procedure. Each element of the record contains the name of a convention defined in the SYSTEM/CONVENTIONS file. Each name uses one element of NAMES-ARY. The record can hold up to 20 names. Each name can be up to 17 characters long and is left justified in the field. If there are less than 17 characters in the name, the field is filled on the right with blanks.

RESULT

This is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure. Possible values returned by CNV_NAMES are as follows:

1	1001	1002	3001

For more information on the error result values, see Table 16–6 later in this section.

Sample Output

The output from Figure 16–16 is as follows:

```
RESULT =            1

Convention Names
----------------
ASERIESNATIVE
Netherlands
Denmark
UnitedKingdom1
Turkey
Norway
Sweden
Greece
FranceListing
FranceBureautique
EuropeanStandard
Belgium
Spain
Switzerland
Zimbabwe
Italy
UnitedKingdom2
KENYA
NIGERIA
SOUTHAFRICA
CYRILLIC
BRAZIL
NEWZEALAND
STNDYUGOSLAVIAN
FRENCHCANADA
ARGENTINA
CHILE
COLOMBIA
COSTARICA
MEXICO
PERU
VENEZUELA
AUSTRALIA
EGYPT
ENGLISHCANADA
Japan1
Japan2
```

CNV_SYMBOLS

This procedure returns a list of numeric and monetary symbols defined for a specified convention.

Example

Figure 16–17 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the CNV_SYMBOLS library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example obtains monetary and numeric symbols, monetary and numeric grouping specifications, and international currency notation defined for the Norway convention.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT OUTPUT-FILE       ASSIGN TO DISK.
DATA DIVISION.
FILE SECTION.
FD  OUTPUT-FILE
    LABEL RECORD IS STANDARD
    VALUE OF TITLE IS "OUT/COBOL85/CNVSYMBOLS."
    VALUE OF PROTECTION IS SAVE
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS OUTPUT-RECORD.
01  OUTPUT-RECORD        PIC X(80).
WORKING-STORAGE SECTION.
01  OF-1.
    05  OF1-TITLE        PIC X(9).
    05  OF1-RESULT       PIC Z(11)9.
    05  FILLER           PIC X(59).
01  DF-1 REDEFINES OF-1.
    05  DF1-NAME         PIC X(32).
    05  FILLER           PIC X(5).
    05  DF1-LENGTH       PIC Z(11)9.
    05  FILLER           PIC XXX.
    05  DF1-SYMBOL1      PIC X(12).
    05  DF1-SYMBOL2      PIC X(12).
    05  FILLER           PIC X(4).
```

```
01  DF-NAME-DEFN.
       05  FILLER           PIC X(32) VALUE
           "International Currency Notation:".
       05  FILLER           PIC X(32) VALUE
           "National Currency Notation:".
       05  FILLER           PIC X(32) VALUE
           "Monetary Thousands Separator:".
       05  FILLER           PIC X(32) VALUE
           "Monetary Decimal Symbol:".
       05  FILLER           PIC X(32) VALUE
           "Monetary Positive Symbol:".
       05  FILLER           PIC X(32) VALUE
           "Monetary Negative Symbol:".
       05  FILLER           PIC X(32) VALUE
           "Monetary Left Enclosure Symbol:".
       05  FILLER           PIC X(32) VALUE
           "Monetary Right Enclosure Symbol:".
       05  FILLER           PIC X(32) VALUE
           "Numeric Thousands Separator:".
       05  FILLER           PIC X(32) VALUE
           "Numeric Decimal Symbol:".
       05  FILLER           PIC X(32) VALUE
           "Numeric Positive Symbol:".
       05  FILLER           PIC X(32) VALUE
           "Numeric Negative Symbol:".
       05  FILLER           PIC X(32) VALUE
           "Numeric Left Enclosure Symbol:".
       05  FILLER           PIC X(32) VALUE
           "Numeric Right Enclosure Symbol:".
       05  FILLER           PIC X(32) VALUE
           "Monetary Grouping Specification:".
       05  FILLER           PIC X(32) VALUE
           "Numeric Grouping Specification:".
 01  DF-NAME-ARY REDEFINES DF-NAME-DEFN.
     05  DF-NAME-ENTRY    PIC X(32)   OCCURS 16 TIMES.
*****************************************************************
*** The following global declarations are used as parameters ***
*** to the CENTRALSUPPORT procedures.                         ***
*****************************************************************
   01  CNV-NAME             PIC X(17).
 01  SYM-ARY.
     05  SYM-ELEM         PIC X(12)   OCCURS 18 TIMES.
 01  SYMLEN-ARY                       USAGE BINARY.
     05  SYMLEN-ELEM      PIC S9(11)  OCCURS 16 TIMES.

 77  CS-DATAOKV           PIC S9(11)  USAGE BINARY   VALUE 1.
 77  CS-FALSEV            PIC S9(11)  USAGE BINARY   VALUE 0.
 77  MAX                  PIC S9(11)  USAGE BINARY   VALUE 16.
 77  RESULT               PIC S9(11)  USAGE BINARY.
 77  SUB1                 PIC S9(11)  USAGE BINARY.
 77  SUB2                 PIC S9(11)  USAGE BINARY.
 77  TOTAL                PIC S9(11)  USAGE BINARY.
```

```
   PROCEDURE DIVISION.
   INTLCOBOL85.
       OPEN OUTPUT OUTPUT-FILE.
       PERFORM CNV-SYMBOLS.
       CLOSE OUTPUT-FILE.
       STOP RUN.

  **** CNV-SYMBOLS ******************************************************
   CNV-SYMBOLS.
       CHANGE ATTRIBUTE FUNCTIONNAME OF "CENTRALSUPPORT" TO
"CENTRALSUPPORT".
       CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
       MOVE "Norway" TO CNV-NAME.
       CALL "CNV_SYMBOLS OF CENTRALSUPPORT"
           USING CNV-NAME,
                 TOTAL,
                 SYMLEN-ARY,
                 SYM-ARY
           GIVING RESULT.
       MOVE SPACES TO OF-1.
       MOVE "Result = " TO OF1-TITLE.
       MOVE RESULT TO OF1-RESULT.
       WRITE OUTPUT-RECORD FROM OF-1.
       IF RESULT = CS-DATAOKV THEN
          MOVE SPACES TO OUTPUT-RECORD
          WRITE OUTPUT-RECORD
          MOVE "Field Meaning              Symbols Length
  -           "  Convention Symbols" TO OUTPUT-RECORD
          WRITE OUTPUT-RECORD
          MOVE "----------               ----------
  -           "  ------------------" TO OUTPUT-RECORD
          WRITE OUTPUT-RECORD
          MOVE SPACES TO OF-1
          MOVE 1 TO SUB1, SUB2.
          PERFORM DISPLAY-ARY UNTIL SUB1 > MAX.

  **** DISPLAY-ARY******************************************************
   DISPLAY-ARY.
       MOVE DF-NAME-ENTRY (SUB1) TO DF1-NAME.
       MOVE SYMLEN-ELEM (SUB1) TO DF1-LENGTH.
       MOVE SYM-ELEM (SUB2) TO DF1-SYMBOL1.
       IF SUB2 > 14 THEN
          ADD 1 TO SUB2
          MOVE SYM-ELEM (SUB2) TO DF1-SYMBOL2.
       WRITE OUTPUT-RECORD FROM DF-1.
       ADD 1 TO SUB1, SUB2.
```

Figure 16–17. Calling the CNV_SYMBOLS Procedure

Explanation

CNV-NAME

This is passed to the procedure. It contains the name of the convention to be used to retrieve the monetary and numeric symbols. If this parameter contains all blanks or zeros, the procedure will use the hierarchy to determine the convention to be used. Refer to the *MLS Guide* for the list of convention names and the explanation of the hierarchy.

TOTAL

This is returned by the procedure. It contains the total number of symbols returned.

SYMLEN-ARY

This is returned by the procedure. It contains the lengths of all symbols being returned in SYM-ARY. The recommended length of SYM-ARY is 16 words. Table 16–5 shows the offset in words of the fields in the record SYMLEN-ARY, which contains the symbol lengths for the monetary and numeric symbols.

SYM-ARY

This is returned by the procedure. Each element of the record contains a symbol defined in the monetary and numeric template for the specified convention. The corresponding entry in SYMLEN-ARY contains the length of each symbol. The maximum length of SYM-ARY is 216 bytes. Table 16–6 shows the monetary and numeric symbols that are returned in the record SYM-ARY and the offset in bytes of the field in which the symbol is returned.

SYMLEN-ARY and SYM-ARY are parallel records. Each entry in SYMLEN-ARY specifies the number of characters the corresponding entry in SYM-ARY has. If an entry in SYMLEN-ARY is 0 (zero), it indicates that the symbol is not defined and the corresponding entry in SYM-ARY is filled with blanks. If an entry in SYMLEN-ARY is not 0 (zero), but the corresponding entry in SYM-ARY is all blanks, then the number of blanks specified by the SYMLEN-ARY entry forms the symbol.

The procedure returns the monetary and numeric templates defined by the convention in fixed-length fields. Each field is 12 bytes long except where noted.

MAX

This is not a parameter but a constant with the value of 16. This constant ensures that SUB1, a subscript of the SYMLEN-ELEM array, does not exceed 16. This constant indirectly ensures that SUB2, a subscript of the SYM-ELEM array, does not exceed 18.

RESULT

This is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure. Possible values returned by CNV-SYMBOLS are as follows:

1	2	1001	1002	2002
2004	3000	3001	3002	3011

For more information on the error result values, see Table 16–6 later in this section.

Table 16–5. Symbols and Offsets Returned in the SYM-ARY Record

Monetary Symbol	Offset	Numeric Symbol	Offset in Integers	Offset in Words
International currency notation	0	Thousands separator	96	8
National currency symbol	12	Decimal symbol	108	9
Thousands separator	24	Positive sign	120	10
Decimal symbol	36	Negative sign	132	11
Positive sign	48	Left enclosure	144	12
Negative sign	60	Right enclosure	156	13
Left enclosure	72	Monetary grouping	168	14
Right enclosure	84	Numeric grouping	192	15

The monetary and numeric grouping each occupy two adjacent fields (24 bytes) in SYM_ARY. The monetary and numeric groupings, when present, are character strings consisting of unsigned integers separated by commas. The integers specify the number of digits in each group and appear exactly as declared in the monetary and numeric templates including embedded commas.

Sample Output

The output from Figure 16–17 is as follows:

```
RESULT =           1

Field Meaning                     Symbols Length   Convention Symbols
-------------                     --------------   ------------------
International Currency Notation:               3   NKR
National Currency Notation:                    3   KR.
Monetary Thousands Separator:                  1
Monetary Decimal Symbol:                       1   ,
Monetary Positive Symbol:                      0
Monetary Negative Symbol:                      1   -
Monetary Left Enclosure Symbol:                0
Monetary Right Enclosure Symbol:               0
Numeric Thousands Separator:                   1
Numeric Decimal Symbol:                        1   ,
Numeric Positive Symbol:                       0
Numeric Negative Symbol:                       1   -
Numeric Left Enclosure Symbol:                 0
Numeric Right Enclosure Symbol:                0
Monetary Grouping Specification:               1   3
Numeric Grouping Specification:                1   3
```

CNV_SYSTEMDATETIMETMP_COB

This procedure formats the system date, the system time, or both according to a template and language that you supply. The system obtains the date and time from a single function call to avoid the possibility of splitting the date and time across a day boundary. The template may be retrieved for any convention from the CNV_TEMPLATE_COB procedure or may be created by the user.

Example

Figure 16–18 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the CNV_SYSTEMDATETIMETMP_COB library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example formats system date and time according to a template provided by the calling program in TMP-ARY. The formatted date and time are translated to English and returned in SDT-ARY. DTEMP-LEN is set to the length of the date template in TMP-ARY.

```
IDENTIFICATION DIVISION.
   ENVIRONMENT DIVISION.
   INPUT-OUTPUT SECTION.
   FILE-CONTROL.
     SELECT OUTPUT-FILE ASSIGN TO DISK.

   DATA DIVISION.
   FILE SECTION.
   FD  OUTPUT-FILE
       LABEL RECORD IS STANDARD
       VALUE OF TITLE IS "OUT/COBOL85/CNVSYSDATETIMETMP."
       PROTECTION SAVE
       RECORD CONTAINS 80 CHARACTERS
       DATA RECORD IS OUTPUT-RECORD.

   01  OUTPUT-RECORD        PIC X(80).

   WORKING-STORAGE SECTION.

   01  OF-1.
       05  FILLER           PIC X(09)   VALUE "RESULT = ".
       05  OF-RESULT        PIC ZZZZZZZZZZZ9.
       05  FILLER           PIC X(59)   VALUE SPACE.
   01  OF-2.
       05  FILLER           PIC X(10)   VALUE "SDT-ARY = ".
       05  OF-SDT-ARY       PIC X(40).
       05  FILLER           PIC X(30)   VALUE SPACE.
```

```
 ****************************************************************
 ***  The following global declarations are used as parameters  ***
 ***  to the CENTRALSUPPORT procedures.                         ***
 ****************************************************************

  01  LANG-NAME           PIC X(17).
  01  SDT-ARY             PIC X(40).
  01  TMP-ARY             PIC X(48).

  77  CS-DATAOKV          PIC S9(11)  USAGE BINARY    VALUE 1.
  77  CS-FALSEV           PIC S9(11)  USAGE BINARY    VALUE 0.
  77  DTEMP-LEN           PIC S9(11)  USAGE BINARY.
  77  RESULT              PIC S9(11)  USAGE BINARY.

  PROCEDURE DIVISION.
  INTLCOBOL85.
      DISPLAY "***  INTL_COBOL85: CNV_SYSTEMDATETIMETMP_COB".
      OPEN OUTPUT OUTPUT-FILE.
      PERFORM CNV-SYSTEMDATETIMETMP-COB.
      CLOSE OUTPUT-FILE.
      STOP RUN.

 ***** CNV-SYSTEMDATETIMETMP-COB *********************************
  CNV-SYSTEMDATETIMETMP-COB.
      CHANGE ATTRIBUTE FUNCTIONNAME OF "CENTRALSUPPORT" TO
"CENTRALSUPPORT".
      CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
      MOVE "!W!, !N! !D!, !YYYY! !OT!:!OM!:!OS!" TO TMP-ARY.
      MOVE 21 TO DTEMP-LEN.
      MOVE "ENGLISH" TO LANG-NAME.
      CALL "CNV_SYSTEMDATETIMETMP_COB OF CENTRALSUPPORT"
        USING TMP-ARY,
              LANG-NAME,
              DTEMP-LEN,
              SDT-ARY
        GIVING RESULT.
      MOVE RESULT TO OF-RESULT.
      WRITE OUTPUT-RECORD FROM OF-1.
      IF RESULT IS EQUAL TO CS-DATAOKV
        THEN MOVE SDT-ARY TO OF-SDT-ARY
             WRITE OUTPUT-RECORD FROM OF-2.
```

Figure 16–18. Calling the CNV_SYSTEMDATETIMETMP_COB Procedure

Explanation

TMP-ARY

This is passed to the procedure. It contains the template you specify, left-justified in the field. The recommended length of a template is 48 characters. If both date and time templates are present, the date template must appear first. Refer to the *MLS Guide* for information about creating a template.

LANG-NAME

This is passed to the procedure. It contains the name of the language to be used in formatting the date, the time value or both. If this parameter contains all blanks or zeros, the procedure uses the hierarchy to determine the language to be used. Refer to the *MLS Guide* for information about determining the valid language names on your system and the explanation of the hierarchy.

DTEMP-LEN

This is an integer passed by reference to the procedure. It specifies the length of the date template in TMP-ARY. If DTEMP-LEN is 0 (zero), it indicates there is no date template in TMP-ARY. If you specify both a date and time template, then the date template must appear first in TMP-ARY. The date and time are formatted if both date and time templates are present, the date is formatted if only date template is present, and the time is formatted if only time template is present.

SDT-ARY

This is returned by the procedure. It contains the formatted date, formatted time, or both. The recommended length of the formatted value is 45 characters.

RESULT

This is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure. Possible values returned by CNV_SYSTEMDATETIMETMP_COB are as follows:

1	2	1001	1002	2001	2002
3000	3001	3002	3011	3045	3046
3047	3048	3051	3052	3053	3054
3055	3057				

For more information on the error result values, see Table 16–6 later in this section.

Sample Output

The output from Figure 16–18 is as follows:

```
RESULT =              1
SDT-ARY = Thursday, March 7, 1991 18:31:23
```

CNV_SYSTEMDATETIME_COB

This procedure formats the system date, the system time, or both according to the specified convention. It translates the date and time components to the natural language you specify. The system computes both the date and time from the result of a single function call. Thus, the possibility that the date and time are split across midnight does not exist.

You might use this procedure to output the system date and time in the Spain convention and the Spanish language, for example.

Example

Figure 16–19 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the CNV_SYSTEMDATETIME_COB library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example formats the system date and time according to formatting definitions in the ASeriesNative convention. The form of date and time is specified by CS-LDATENTIMEV (long date and numeric time). Formatted date and time are translated to English and returned in SDT-ARY.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT OUTPUT-FILE ASSIGN TO DISK.

DATA DIVISION.
FILE SECTION.
FD  OUTPUT-FILE
    LABEL RECORD IS STANDARD
    VALUE OF TITLE IS "OUT/COBOL85/CNVSYSDATETIME."
    PROTECTION SAVE
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS OUTPUT-RECORD.

01  OUTPUT-RECORD        PIC X(80).
```

```
  WORKING-STORAGE SECTION.

  01  OF-1.
      05  FILLER           PIC X(09)   VALUE "RESULT = ".
      05  OF-RESULT        PIC ZZZZZZZZZZZ9.
      05  FILLER           PIC X(59)   VALUE SPACE.
  01  OF-2.
      05  FILLER           PIC X(10)   VALUE "SDT-ARY = ".
      05  OF-SDT-ARY       PIC X(40).
      05  FILLER           PIC X(30)   VALUE SPACE.

 ****************************************************************
 ***  The following global declarations are used as parameters  ***
 ***  to the CENTRALSUPPORT procedures.                         ***
 ****************************************************************
  01  CNV-NAME             PIC X(17).
  01  LANG-NAME            PIC X(17).
  01  SDT-ARY              PIC X(40).

  77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
  77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
  77  CS-LDATENTIMEV       PIC S9(11)  USAGE BINARY    VALUE 6.
  77  RESULT               PIC S9(11)  USAGE BINARY.

  PROCEDURE DIVISION.
  INTLCOBOL85.
      OPEN OUTPUT OUTPUT-FILE.
      PERFORM CNV-SYSTEMDATETIME-COB.
      CLOSE OUTPUT-FILE.
      STOP RUN.
  ***** CNV-SYSTEMDATETIME-COB ************************************
  CNV-SYSTEMDATETIME-COB.
      CHANGE ATTRIBUTE FUNCTIONNAME OF "CENTRALSUPPORT" TO
"CENTRALSUPPORT".
      CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
      MOVE "ASERIESNATIVE" TO CNV-NAME.
      MOVE "ENGLISH" TO LANG-NAME.
      CALL "CNV_SYSTEMDATETIME_COB OF CENTRALSUPPORT"
        USING CS-LDATENTIMEV,
              CNV-NAME,
              LANG-NAME,
              SDT-ARY
        GIVING RESULT.
      MOVE RESULT TO OF-RESULT.
      WRITE OUTPUT-RECORD FROM OF-1.
      IF RESULT IS EQUAL TO CS-DATAOKV
        THEN MOVE SDT-ARY TO OF-SDT-ARY
             WRITE OUTPUT-RECORD FROM OF-2.
```

Figure 16–19. Calling the CNV_SYSTEMDATETIME_COB Procedure

Explanation

CS-LDATENTIMEV

This is passed to the procedure. It indicates one of the following formats is used when the date and time are returned:

Value	Sample Data Item and Meaning
0	CS-LDATEV Long date format
1	CS-SDATEV Short date format
2	CS-NDATEV Numeric date format
3	CS-LTIMEV Long time format
4	CS-NTIMEV Numeric time format
5	CS-LDATELTIMEV Long date and long time
6	CS-LDATENTIMEV Long date and numeric time
7	CS-SDATELTIMEV Short date and long time
8	CS-SDATENTIMEV Short date and numeric time
9	CS-NDATELTIMEV Numeric date and long time
10	CS-NDATENTIMEV Numeric date and numeric time

CNV-NAME

This is passed to the procedure. It contains the name of the convention to be used to edit the date and time value. If this parameter contains all blanks or zeros, the procedure will use the hierarchy to determine the convention to be used. Refer to the *MLS Guide* for the list of convention names and the explanation of the hierarchy.

LANG-NAME

This is passed to the procedure. It contains the language to be used in formatting the date and time value. If this parameter contains all blanks or zeros, the procedure will use the hierarchy to determine the language to be used. Refer to the *MLS Guide* for information about determining the valid language names on your system and the explanation of the hierarchy.

SDT-ARY

This is returned by the procedure. It contains the formatted date, formatted time, or both. The recommended length of the formatted value is 40 characters.

RESULT

This is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure. Possible values returned by CNV_SYSTEMDATETIME_COB are as follows:

1	2	1001	1002	2001	2002
2004	3000	3001	3006	3011	3045
3046	3047	3048	3051	3052	3053
3054	3055	3057			

For more information on the error result values, see Table 16–6 later in this section.

Sample Output

The output from Figure 16–19 is as follows:

```
RESULT =             1
SDT-ARY = Wednesday, November 7, 1990 17:14:58
```

CNV_TEMPLATE_COB

This procedure returns the requested format template for a designated convention.

You might want to use this procedure to improve the performance of your program. By retrieving and storing a template that you want to use in many places, you can improve the performance of your program by eliminating the calls to the CENTRALSUPPORT library.

Example

Figure 16–20 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the CNV_TEMPLATE_COB library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example retrieves a monetary editing template from the Turkey convention. The template is returned in TMP-ARY.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT OUTPUT-FILE ASSIGN TO DISK.

DATA DIVISION.
FILE SECTION.
FD  OUTPUT-FILE
    LABEL RECORD IS STANDARD
    VALUE OF TITLE IS "OUT/COBOL85/CNVTEMPLATECOB."
    PROTECTION SAVE
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS OUTPUT-RECORD.

01  OUTPUT-RECORD        PIC X(80).

WORKING-STORAGE SECTION.

01  OF-1.
    05  FILLER           PIC X(09)   VALUE "RESULT = ".
    05  OF-RESULT        PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(59)   VALUE SPACE.
01  OF-2.
    05  FILLER           PIC X(10)   VALUE "TMP-ARY = ".
    05  OF-TMP-ARY       PIC X(48).
    05  FILLER           PIC X(22)   VALUE SPACE.
```

```
 ****************************************************************
 ***  The following global declarations are used as parameters  ***
 ***  to the CENTRALSUPPORT procedures.                         ***
 ****************************************************************

  01  CNV-NAME             PIC X(17).
  01  TMP-ARY              PIC X(48).

  77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
  77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
  77  CS-MONETARY-TEMPV    PIC S9(11)  USAGE BINARY    VALUE 5.
  77  RESULT               PIC S9(11)  USAGE BINARY.

  PROCEDURE DIVISION.
  INTLCOBOL85.
      OPEN OUTPUT OUTPUT-FILE.
      PERFORM CNV-TEMPLATE-COB.
      CLOSE OUTPUT-FILE.
      STOP RUN.

 ***** CNV-TEMPLATE-COB ******************************************
  CNV-TEMPLATE-COB.
      CHANGE ATTRIBUTE FUNCTIONNAME OF "CENTRALSUPPORT" TO
"CENTRALSUPPORT".
      CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
      MOVE "Turkey" TO CNV-NAME.
      CALL "CNV_TEMPLATE_COB OF CENTRALSUPPORT"
        USING CS-MONETARY-TEMPV,
              CNV-NAME,
              TMP-ARY
        GIVING RESULT.
      MOVE RESULT TO OF-RESULT.
      WRITE OUTPUT-RECORD FROM OF-1.
      IF RESULT IS EQUAL TO CS-DATAOKV
        THEN STRING TMP-ARY DELIMITED BY @OO@,
                    @OO@ DELIMITED BY SIZE
                    INTO OF-TMP-ARY
             WRITE OUTPUT-RECORD FROM OF-2.
```

Figure 16–20. Calling the CNV_TEMPLATE_COB Procedure

Explanation

CS-MONETARY-TEMPV

This is passed to the procedure. It specifies the type of template to be returned. This parameter can have the following values:

Value	Sample Data Item and Template to be Retrieved
0	CS-LONGDATE-TEMPV Long date
1	CS-SHORTDATE-TEMPV Short date
2	CS-NUMDATE-TEMPV Numeric date
3	CS-LONGTIME-TEMPV Long time
4	CS-NUMTIME-TEMPV Numeric time
5	CS-MONETARY-TEMPV Monetary template
6	CS-NUMERIC-TEMPV Numeric template

CNV-NAME

This is passed to the procedure. It contains the name of the convention that you specify. If this parameter contains all blanks or zeros, the procedure will use the hierarchy to determine the convention to be used. Refer to the *MLS Guide* for the list of convention names and the explanation of the hierarchy.

TMP-ARY

This is returned by the procedure. It contains the requested template. The recommended length of a template is 48 characters.

RESULT

This is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure.

Possible values returned by CNV_TEMPLATE_COB are as follows:

1	1001	1002	2002
3000	3001	3002	3006

For more information on the error result values, see Table 16–6 later in this section.

Sample Output

The output from Figure 16–20 is as follows:

```
RESULT =              1
TMP-ARY = !T[.:0,3]D[,]#N[-]C[T]!
```

CNV_VALIDATENAME

This procedure returns a value in the procedure result that indicates whether the convention name you specified is currently defined on the host system.

You might use this procedure to ensure that a convention used as an input parameter exists on the system on which your program is running.

Example

Figure 16–21 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the CNV_VALIDATENAME library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example determines whether or not a convention named Sweden is currently available on the system.

```
IDENTIFICATION DIVISION.
   ENVIRONMENT DIVISION.
   INPUT-OUTPUT SECTION.
   FILE-CONTROL.
     SELECT OUTPUT-FILE ASSIGN TO DISK.

   DATA DIVISION.
   FILE SECTION.
   FD  OUTPUT-FILE
       LABEL RECORD IS STANDARD
       VALUE OF TITLE IS "OUT/COBOL85/VALIDATENAME."
       PROTECTION SAVE
       RECORD CONTAINS 80 CHARACTERS
       DATA RECORD IS OUTPUT-RECORD.
   01  OUTPUT-RECORD        PIC X(80).
   WORKING-STORAGE SECTION.
   01  OF-1.
       05  FILLER           PIC X(09)   VALUE "RESULT = ".
       05  OF-RESULT        PIC ZZZZZZZZZZZ9.
       05  FILLER           PIC X(59)   VALUE SPACE.
```

```
 ****************************************************************
 ***  The following global declarations are used as parameters  ***
 ***  to the CENTRALSUPPORT procedures.                         ***
 ****************************************************************
  01  CNV-NAME             PIC X(17).
  77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
  77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
  77  RESULT               PIC S9(11)  USAGE BINARY.
  PROCEDURE DIVISION.
  INTLCOBOL85.
      OPEN OUTPUT OUTPUT-FILE.
      PERFORM CNV-VALIDATENAME.
      CLOSE OUTPUT-FILE.
      STOP RUN.

 ***** CNV-VALIDATENAME ******************************************
  CNV-VALIDATENAME.
      CHANGE ATTRIBUTE FUNCTIONNAME OF "CENTRALSUPPORT" TO
"CENTRALSUPPORT".
      CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
      MOVE "Sweden" TO CNV-NAME.
      CALL "CNV_VALIDATENAME OF CENTRALSUPPORT"
        USING CNV-NAME
        GIVING RESULT.
      MOVE RESULT TO OF-RESULT.
      WRITE OUTPUT-RECORD FROM OF-1.
```

Figure 16–21. Calling the CNV_VALIDATENAME Procedure

Explanation

CNV-NAME

This is passed to the procedure. It contains the name of the convention that is to be checked. If this parameter contains all blanks or nulls, the RESULT parameter returns a value of 0 (zero) or FALSE. Refer to the *MLS Guide* for the list of convention names and the explanation of the hierarchy.

RESULT

This is an integer that is returned by the procedure. It contains the procedure result. The possible values for RESULT and their meanings are as follows:

Value	Condition-Name and Meaning
0	CS-FALSEV The convention name is not valid.
1	CS-DATAOKV The convention name is valid.

Sample Output

The output from Example 16–23 is as follows:

```
RESULT =            0
```

GET_CS_MSG

This procedure returns message text associated with the designated message number. The message number is obtained as the result value returned from a call to any of the CENTRALSUPPORT procedures.

When calling the GET_CS_MSG procedure, you can designate the language to which the message is to be translated and the desired length of the returned message. If the returned text is shorter than the length specified, the procedure pads the remaining portion of the record with blanks.

An entire message consists of the following three parts:

- The header, which comprises the first 80 characters of the message text returned by the MSG parameter. The text in the header provides the message number and a concise text description.
- The short description, which comprises the second 80 character of the message text returned by the MSG parameter. If space is a consideration, you might want to limit the description of the message to the header and short description.
- The long description, which comprises the remaining characters of the message text returned by the MSG parameter. The long description provides a complete explanation of the message that was returned.

Part or all of the message text can be returned. Note that the header part starts at offset 0 (zero), the short description at offset 80, and the long description at offset 160. For example, if you specify the MSG-LEN parameter to be equal to 200 characters, then the MSG parameter would contain the header message padded with blanks to offset 80, if necessary, followed by the short description padded with blanks to offset 160, if necessary, followed by the first 40 characters of the long description.

The message length should be at least 80 characters, equal to one line of text. Anything less results in an incomplete message. Using a value of either 80, 160, or 999 is recommended. The value of 999 causes the entire message to be returned.

You might want to use this procedure to retrieve the text of an error message so that it can be displayed by your program.

Example

Figure 16–22 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the GET_CS_MSG library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example illustrates how to get the message text associated with a CENTRALSUPPORT error message. Assume that the sample call to VALIDATE_NAME_RETURN_NUM returns the error 3004 (The requested name was not found.). When the error is returned, this example gets the first 160 characters (2 lines) of the message text for the error.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT OUTPUT-FILE ASSIGN TO DISK.

DATA DIVISION.
FILE SECTION.
FD OUTPUT-FILE
  LABEL RECORD IS STANDARD
  VALUE OF TITLE IS "OUT/COBOL85/GETCSMSG."
  PROTECTION SAVE
  RECORD CONTAINS 80 CHARACTERS
  DATA RECORD IS OUTPUT-RECORD.

01 OUTPUT-RECORD   PIC X(80).

WORKING-STORAGE SECTION.

01 OF-1.
   05 FILLER       PIC X(39)
      VALUE "RESULT FROM VALIDATE_NAME_RETURN_NUM = ".
   05 OF-RESULT1   PIC ZZZZZZZZZZ9.
   05 FILLER       PIC X(23)  VALUE SPACE.
01 OF-2.
   05 FILLER       PIC X(39)
      VALUE "RESULT FROM GET_CS_MSG        = ".
   05 OF-RESULT2   PIC ZZZZZZZZZZ9.
   05 FILLER       PIC X(23)  VALUE SPACE.
01 OF-3.
   05 FILLER       PIC X(06)  VALUE "MSG = ".
   05 FILLER       PIC X(74)  VALUE SPACE.
01 OF-4.
   05 OF-MSG       PIC X(80).

****************************************************************
*** The following global declarations are used as parameters ***
*** to the CENTRALSUPPORT procedures.             ***
****************************************************************

 01 LANG-NAME       PIC X(17).
 01 MSG.
    05 MSG-ELEM     PIC X(80)  OCCURS 2 TIMES.
 01 NAME-ARY        PIC X(17).

 77 CS-CHARACTER-SETV  PIC S9(11) USAGE BINARY  VALUE 0.
```

```
 77 CS-DATAOKV         PIC S9(11) USAGE BINARY  VALUE 1.
 77 CS-FALSEV          PIC S9(11) USAGE BINARY  VALUE 0.
 77 MSG-LEN            PIC S9(11) USAGE BINARY.
 77 NUM                PIC S9(11) USAGE BINARY.
 77 RESULT1            PIC S9(11) USAGE BINARY.
 77 RESULT2            PIC S9(11) USAGE BINARY.

 PROCEDURE DIVISION.
 INTLCOBOL85.
    DISPLAY "*** INTL_COBOL85: GET_CS_MSG".
    OPEN OUTPUT OUTPUT-FILE.
    PERFORM GET-CS-MSG.
    CLOSE OUTPUT-FILE.
    STOP RUN.

***** GET-CS-MSG ************************************************
 GET-CS-MSG.
    CHANGE ATTRIBUTE FUNCTIONNAME OF "CENTRALSUPPORT" TO "CENTRALSUPPORT".
    CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
    MOVE "BADNAME" TO NAME-ARY.
    CALL "VALIDATE_NAME_RETURN_NUM OF CENTRALSUPPORT"
     USING CS-CHARACTER-SETV,
        NAME-ARY,
        NUM
     GIVING RESULT1.
    MOVE RESULT1 TO OF-RESULT1.
    WRITE OUTPUT-RECORD FROM OF-1.
    IF RESULT1 IS NOT EQUAL TO CS-DATAOKV
     THEN MOVE 160 TO MSG-LEN
       CALL "GET_CS_MSG OF CENTRALSUPPORT"
        USING RESULT1,
           LANG-NAME,
           MSG,
           MSG-LEN
        GIVING RESULT2
       MOVE RESULT2 TO OF-RESULT2
       WRITE OUTPUT-RECORD FROM OF-2
       WRITE OUTPUT-RECORD FROM OF-3
       MOVE MSG-ELEM(1) TO OF-MSG
       WRITE OUTPUT-RECORD FROM OF-4
       MOVE MSG-ELEM(2) TO OF-MSG
       WRITE OUTPUT-RECORD FROM OF-4.
```

Figure 16–22. Calling the GET_CS_MSG Procedure

Explanation

NUM

This is passed to the procedure. It contains the number of the message for which you want the text. These values are returned by calls on other CENTRALSUPPORT procedures. The message numbers and their meanings are listed at the end of this section.

LANG-NAME

This is passed to the procedure. It specifies the language in which the message is to be displayed. The maximum length of a language name is 17 characters. If this parameter contains all blanks or zeros, the procedure uses the default language hierarchy to determine the language to be used. Refer to the *MLS Guide* for details about determining the valid language names on the system and for the explanation of the default language hierarchy.

MSG

This is returned by the procedure. It contains the message text associated with the specified message number. It is recommended that the size of this record be at least 80 characters.

MSG_LEN

This is passed to the procedure. For an output parameter, MSG_LEN contains an update value. For input, it specifies the maximum length of the message to be returned. If MSG_LEN is equal to 0 (zero), one line of text (80 characters) is returned. If MSG_LEN is between 1 and 79, then only a partial message is returned. MSG_LEN should not be greater than the size of the MSG record. Recommended values for MSG_LEN are 80, 160, or a large number that returns all of the message. For output, MSG_LEN specifies the actual length of the message returned by the procedure.

NAME-ARY

This is passed to the procedure. It contains the coded character set or ccsversion name for which a message number is being requested. The name can be up to 17 characters long. If this parameter contains zeros or blanks, the procedure uses the hierarchy to determine the ccsversion or character set to be used. If there is no system default, the procedure returns an error in RESULT.

CS-CHARACTER-SETV

This is passed to the procedure. If this flag represents 0 (zero), the coded character set is being checked. If it represents 1 (one), the ccsversion is being checked.

RESULT1

This is passed to the procedure. It contains the number of the message for which you want the text. These values are returned by calls on other CENTRALSUPPORT procedures. The message numbers and their meanings are listed at the end of this section. In Example
16–19, the RESULT1 field is from an executed VALIDATE_NAME_RETURN_NUM procedure that requested a ccsversion number for the name BADNAME.

RESULT2

This is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure.

Possible values returned by GET_CS_MSG are as follows:

1	1001	1002	2004
3000	3001	3002	3003

For more information on the error result values, see Table 16–6 later in this section.

Sample Output

The output from Figure 16–22 is as follows:

```
RESULT FROM VALIDATE_NAME_RETURN_NUM =        3004
RESULT FROM GET_CS_MSG               =           1
MSG =
>>> CENTRALSUPPORT INTERFACE ERROR (#3004) <<<
INVALID CHARACTER SET OR CCSVERSION NAME
```

MCP_BOUND_LANGUAGES

This procedure returns the names of languages that are currently bound to the operating system. For information about binding a language to the operating system, refer to the *MLS Guide*.

Example

Figure 16–23 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the MCP_BOUND_LANGUAGES library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example returns the languages bound by the operating system. Assume for this example that the bound language is English.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT OUTPUT-FILE ASSIGN TO DISK.

DATA DIVISION.
FILE SECTION.
FD  OUTPUT-FILE
    LABEL RECORD IS STANDARD
    VALUE OF TITLE IS "OUT/COBOL85/MCPBOUNDLANGUAGES."
    PROTECTION SAVE
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS OUTPUT-RECORD.
  01  OUTPUT-RECORD        PIC X(80).
  WORKING-STORAGE SECTION.
  01  OF-1.
    05  FILLER           PIC X(09)   VALUE "RESULT = ".
    05  OF-RESULT        PIC ZZZZZZZZZZ9.
    05  FILLER           PIC X(59)   VALUE SPACE.
01  OF-2.
    05  FILLER           PIC X(09)   VALUE "Languages".
    05  FILLER           PIC X(71)   VALUE SPACE.
01  OF-3.
    05  FILLER           PIC X(09)   VALUE ALL "-".
    05  FILLER           PIC X(71)   VALUE SPACE.
01  OF-4.
    05  OF-LANG-ELEM     PIC X(17).
    05  FILLER           PIC X(63)   VALUE SPACE.
```

```
 ****************************************************************
 ***  The following global declarations are used as parameters  ***
 ***  to the CENTRALSUPPORT procedures.                         ***
 ****************************************************************
  01  LANGUAGES-ARY.
      05  LANGUAGES-ELEM   PIC X(17)   OCCURS 20 TIMES.
  01  SUB                  PIC 9(02).

  77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
  77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
  77  RESULT               PIC S9(11)  USAGE BINARY.
  77  TOTAL                PIC S9(11)  USAGE BINARY.

  PROCEDURE DIVISION.
  INTLCOBOL85.
      OPEN OUTPUT OUTPUT-FILE.
      PERFORM MCP-BOUND-LANGUAGES.
      CLOSE OUTPUT-FILE.
      STOP RUN.

 ***** MCP-BOUND-LANGUAGES ***************************************
  MCP-BOUND-LANGUAGES.
      CHANGE ATTRIBUTE FUNCTIONNAME OF "CENTRALSUPPORT" TO
"CENTRALSUPPORT".
      CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
      CALL "MCP_BOUND_LANGUAGES OF CENTRALSUPPORT"
        USING TOTAL,
              LANGUAGES-ARY
        GIVING RESULT.
      MOVE RESULT TO OF-RESULT.
      WRITE OUTPUT-RECORD FROM OF-1.
      IF RESULT IS EQUAL TO CS-DATAOKV
        THEN MOVE SPACE TO OUTPUT-RECORD
             WRITE OUTPUT-RECORD
             WRITE OUTPUT-RECORD FROM OF-2
             WRITE OUTPUT-RECORD FROM OF-3
             MOVE 1 TO SUB
             PERFORM DISPLAYLANGUAGESARY
               UNTIL SUB IS GREATER THAN TOTAL.

 ***** DISPLAYLANGUAGESARY ***************************************
  DISPLAYLANGUAGESARY.
      MOVE LANGUAGES-ELEM(SUB) TO OF-LANG-ELEM.
      WRITE OUTPUT-RECORD FROM OF-4.
      ADD 1 TO SUB.
```

Figure 16–23. Calling the MCP_BOUND_LANGUAGES Procedure

Explanation

TOTAL

This is an integer returned by the procedure. It contains the total number of languages that are bound to the operating system.

LANGUAGES-ARY

This is returned by the procedure. It contains the names of the languages bound to the operating system. The maximum length of each name is 17 characters, and the names are left justified. For any name that is less than 17 characters, the field is filled on the right with blanks. In the example, the size of the record is 84 characters; a record of that size holds 5 names.

RESULT

This is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure. Possible values returned by this procedure are as follows:

1	1001	1002	3000	3001

For more information on the error result values, see Table 16–6 later in this section.

Sample Output

The output from Figure 16–23 is as follows:

```
RESULT =            1
Languages
---------
ENGLISH
```

VALIDATE_NAME_RETURN_NUM

This procedure examines a coded character set or ccsversion name to determine if it resides in the file SYSTEM/CCSFILE. The first parameter specifies whether you want to examine a coded character set or ccsversion. The next parameter specifies the name to be validated. The procedure returns the number of the coded character set or ccsversion in the last parameter.

You might use this procedure to obtain the ccsversion number needed as a parameter to other CENTRALSUPPORT library procedures.

Example

Figure 16–24 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the VALIDATE_NAME_RETURN_NUM library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example checks to see if a ccsversion named CanadaGP is valid. Assume for this example that CanadaGP is valid and its associated number is 75.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT OUTPUT-FILE ASSIGN TO DISK.

DATA DIVISION.
FILE SECTION.
FD  OUTPUT-FILE
    LABEL RECORD IS STANDARD
    VALUE OF TITLE IS "OUT/COBOL85/VALIDATENAMERTRN."
    PROTECTION SAVE
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS OUTPUT-RECORD.

01  OUTPUT-RECORD        PIC X(80).
```

```
 WORKING-STORAGE SECTION.

 01  OF-1.
     05  FILLER           PIC X(09)   VALUE "RESULT = ".
     05  OF-RESULT        PIC ZZZZZZZZZZ9.
     05  FILLER           PIC X(59)   VALUE SPACE.
 01  OF-2.
     05  FILLER           PIC X(09)   VALUE "NUM    = ".
     05  OF-NUM           PIC ZZZZZZZZZZ9.
     05  FILLER           PIC X(59)   VALUE SPACE.

***************************************************************
***  The following global declarations are used as parameters  ***
***  to the CENTRALSUPPORT procedures.                         ***
***************************************************************

 01  NAME-ARY             PIC X(17).

 77  CS-CCSVERSIONV       PIC S9(11)  USAGE BINARY    VALUE 1.
 77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
 77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
 77  NUM                  PIC S9(11)  USAGE BINARY.
 77  RESULT               PIC S9(11)  USAGE BINARY.

 PROCEDURE DIVISION.
 INTLCOBOL85.
     OPEN OUTPUT OUTPUT-FILE.
     PERFORM VALIDATE-NAME-RETURN-NUM.
     CLOSE OUTPUT-FILE.
     STOP RUN.

***** VALIDATE-NAME-RETURN-NUM *********************************
 VALIDATE-NAME-RETURN-NUM.
     CHANGE ATTRIBUTE FUNCTIONNAME OF "CENTRALSUPPORT" TO
"CENTRALSUPPORT".
     CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
     MOVE "CanadaGP" TO NAME-ARY.
     CALL "VALIDATE_NAME_RETURN_NUM OF CENTRALSUPPORT"
       USING CS-CCSVERSIONV,
             NAME-ARY,
             NUM
       GIVING RESULT.
     MOVE RESULT TO OF-RESULT.
     WRITE OUTPUT-RECORD FROM OF-1.
     IF RESULT IS EQUAL TO CS-DATAOKV
       THEN MOVE NUM TO OF-NUM
            WRITE OUTPUT-RECORD FROM OF-2.
```

Figure 16–24. Calling the VALIDATE_NAME_RETURN_NUM Procedure

Explanation

CS-CCSVERSIONV

This is passed to the procedure. It indicates whether the entry specified in NAME is a coded character set or ccsversion name. The allowable values are as follows:

Value	Sample Data Item and Meaning
0	CS-CHARACTER-SET-V Coded character set name
1	CS-CCSVERSION-V Ccsversion name

NAME-ARY

This is passed to the procedure. It contains the coded character set or ccsversion name for which a number is being requested. The name can be up to 17 characters long. If this parameter contains zeros or blanks and type is equal to 1, the procedure validates the system default ccsversion.

NUM is returned by the procedure. It contains the coded character set or ccsversion number requested.

RESULT

This is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure.

Possible values returned by VALIDATE_NAME_RETURN_NUM are as follows:

1 1001 1002 3000 3002 3004 3006

For more information on the error result values, see Table 16–6 later in this section.

Sample Output

The output from Figure 16–24 is as follows:

```
RESULT =             1
NUM    =            75
```

VALIDATE_NUM_RETURN_NAME

This procedure examines the number of a coded character set or ccsversion to determine if it resides in the SYSTEM/CCSFILE. The first parameter designates whether a coded character set or ccsversion is to be examined. The second parameter specifies the number to be validated. The procedure then returns the name of the given character set or ccsversion number. Refer to the *MLS Guide* for the list of numbers for coded character sets and ccsversions.

You might use this procedure to display the name of the coded character set or the ccsversion being used.

Example

Figure 16–25 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the VALIDATE_NUM_RETURN_NAME library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example checks to see if the ccsversion number 75 is valid. Assume for this example that 75 is valid and its associated name is CanadaGP.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT OUTPUT-FILE ASSIGN TO DISK.

DATA DIVISION.
FILE SECTION.
FD  OUTPUT-FILE
    LABEL RECORD IS STANDARD
    VALUE OF TITLE IS "OUT/COBOL85/VALIDATENUMRTRN."
    PROTECTION SAVE
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS OUTPUT-RECORD.

01  OUTPUT-RECORD        PIC X(80).
```

```
 WORKING-STORAGE SECTION.

 01  OF-1.
     05  FILLER           PIC X(09)   VALUE "RESULT = ".
     05  OF-RESULT        PIC ZZZZZZZZZZZ9.
     05  FILLER           PIC X(59)   VALUE SPACE.

 01  OF-2.
     05  FILLER           PIC X(11)   VALUE "NAME-ARY = ".
     05  OF-NAME-ARY      PIC X(17).
     05  FILLER           PIC X(52)   VALUE SPACE.

***************************************************************
***  The following global declarations are used as parameters  ***
***  to the CENTRALSUPPORT procedures.                          ***
***************************************************************

 01  NAME-ARY             PIC X(17).

 77  CS-CCSVERSIONV       PIC S9(11)  USAGE BINARY    VALUE 1.
 77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
 77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
 77  NUM                  PIC S9(11)  USAGE BINARY.
 77  RESULT               PIC S9(11)  USAGE BINARY.

 PROCEDURE DIVISION.
 INTLCOBOL85.
     OPEN OUTPUT OUTPUT-FILE.
     PERFORM VALIDATE-NUM-RETURN-NAME.
     CLOSE OUTPUT-FILE.
     STOP RUN.

***** VALIDATE-NUM-RETURN-NAME **********************************
 VALIDATE-NUM-RETURN-NAME.
     CHANGE ATTRIBUTE FUNCTIONNAME OF "CENTRALSUPPORT" TO
"CENTRALSUPPORT".
     CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
     MOVE 75 TO NUM.
     CALL "VALIDATE_NUM_RETURN_NAME OF CENTRALSUPPORT"
       USING CS-CCSVERSIONV,
             NUM,
             NAME-ARY
       GIVING RESULT.
     MOVE RESULT TO OF-RESULT.
     WRITE OUTPUT-RECORD FROM OF-1.
     IF RESULT IS EQUAL TO CS-DATAOKV
       THEN MOVE NAME-ARY TO OF-NAME-ARY
            WRITE OUTPUT-RECORD FROM OF-2.
```

Figure 16–25. Calling the VALIDATE_NUM_RETURN_NAME Procedure

Explanation

CS-CCSVERSIONV

This is passed to the procedure. It indicates whether the value specified in NUM is a coded character set number or a ccsversion number. The valid values are as follows:

Value	Sample Data Item and Meaning
0	CS-CHARACTER-SET-V Coded character set number
1	CS-CCSVERSION-V Ccsversion number

NUM

This is passed by reference to the procedure. It contains the number of the coded character set or ccsversion for which the name is being requested. If you supply the value –2 in the NUM parameter when you are checking a ccsversion, the procedure returns the name of the system default ccsversion. Refer to the *MLS Guide* for more information about the hierarchy.

NAME-ARY

This is returned by the procedure. It contains the coded character set or ccsversion name. The recommended length of the name is 17 characters.

RESULT

This is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure.

Possible values returned by VALIDATE_NUM_RETURN_NAME are as follows:

1 1001 1002 3000 3001 3003 3006

For more information on the error result values, see Table 16–6 later in this section.

Sample Output

The output from Figure 16–25 is as follows:

```
RESULT =           1
NAME-ARY = CANADAGP
```

VSNCOMPARE_TEXT

This procedure compares two records, using one of three comparison methods. The comparison is specified as one of the following types:

- A binary comparison, which is based on the hexadecimal code values of the characters
- An equivalent comparison, which is based on the ordering sequence values (OSVs) of the characters
- A logical comparison, which is based on the ordering sequence values (OSVs) plus the priority sequence values (PSVs) of the characters

The procedure retrieves the OSVs and PSVs from the file SYSTEM/CCSFILE based on the specified ccsversion.

Example

Figure 16–26 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the VSNCOMPARE_TEXT library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example compares two strings using the CanadaEBCDIC ccsversion. The first string is "hotel" and the second string is "hôtel." Assume the following ordering values for the characters:

Character	Ordering Sequence Value (OSV)	Priority Sequence Value (PSV)
e	69	2
h	72	2
l	76	2
o	79	2
t	84	2
ô	79	8

The compare relation is CsCmpEql (=) to determine if "hotel" is equal to "hôtel" using a logical comparison. You can use the *MLS Guide* to determine that the ccsversion number for CanadaEBCDIC is 74. You can also retrieve this number by calling the procedure VALIDATE_NAME_RETURN_NUM with the name CanadaEBCDIC.

```
IDENTIFICATION DIVISION.
 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
   SELECT OUTPUT-FILE ASSIGN TO DISK.

 DATA DIVISION.
 FILE SECTION.
 FD  OUTPUT-FILE
     LABEL RECORD IS STANDARD
     VALUE OF TITLE IS "OUT/COBOL85/VSNCOMPARETEXT."
     PROTECTION SAVE
     RECORD CONTAINS 80 CHARACTERS
     DATA RECORD IS OUTPUT-RECORD.

 01  OUTPUT-RECORD        PIC X(80).

 WORKING-STORAGE SECTION.

 01  OF-1.
     05  FILLER           PIC X(09)   VALUE "RESULT = ".
     05  OF-RESULT        PIC ZZZZZZZZZZZ9.
     05  FILLER           PIC X(59)   VALUE SPACE.

****************************************************************
***  The following global declarations are used as parameters  ***
***  to the CENTRALSUPPORT procedures.                         ***
****************************************************************

 01  TEXT1-TEXT           PIC X(05).
 01  TEXT2-TEXT           PIC X(05).

 77  CS-CMPEQLV           PIC S9(11)  USAGE BINARY    VALUE 2.
 77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
 77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
 77  CS-LOGICALV          PIC S9(11)  USAGE BINARY    VALUE 2.
 77  TEXT1-START          PIC S9(11)  USAGE BINARY.
 77  TEXT2-START          PIC S9(11)  USAGE BINARY.
 77  COMPARE-LEN          PIC S9(11)  USAGE BINARY.
 77  RESULT               PIC S9(11)  USAGE BINARY.
 77  VSN-NUM              PIC S9(11)  USAGE BINARY.
```

```
 PROCEDURE DIVISION.
 INTLCOBOL85.
     OPEN OUTPUT OUTPUT-FILE.
     PERFORM VSNCOMPARE-TEXT.
     CLOSE OUTPUT-FILE.
     STOP RUN.

***** VSNCOMPARE-TEXT ******************************************
 VSNCOMPARE-TEXT.
     CHANGE ATTRIBUTE FUNCTIONNAME OF "CENTRALSUPPORT" TO CENTRALSUPPORT".
     CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
     MOVE 74 TO VSN-NUM.
     MOVE 5 TO COMPARE-LEN.
     MOVE "hotel" TO TEXT1-TEXT.
     MOVE "hôtel" TO TEXT2-TEXT.
     CALL "VSNCOMPARE_TEXT OF CENTRALSUPPORT"
       USING VSN-NUM,
             TEXT1-TEXT,
             TEXT1-START,
             TEXT2-TEXT,
             TEXT2-START,
             COMPARE-LEN,
             CS-CMPEQLV,
             CS-LOGICALV
       GIVING RESULT.
     MOVE RESULT TO OF-RESULT.
     WRITE OUTPUT-RECORD FROM OF-1.
```

Figure 16–26. Calling the VSNCOMPARE_TEXT Procedure

Explanation

VSN-NUM

This is passed to the procedure. It contains the number of the ccsversion that is used to compare the text records. You can obtain the number by referring to the *MLS Guide*. Valid values are as follows:

If the value is . . .	Then . . .
Greater than or equal to 0 (zero)	Designate a ccsversion.
–2	Use the system default ccsversion. If the system default ccsversion is not available, the procedure returns an error in RESULT.

TEXT1-TEXT

This is passed to the procedure. It contains the first record of text to be compared. You determine the size of the record.

TEXT1-START

This is passed by reference to the procedure. It contains the byte offset in TEXT1-TEXT, relative to 0 (zero), at which the comparison begins.

TEXT2-TEXT

This is passed to the procedure. It contains the second record of text to be compared. You determine the size of the record.

TEXT2-START

This is passed to the procedure. It contains the byte offset in TEXT2-TEXT, relative to 0 (zero), at which the comparison begins.

COMPARE-LEN

This is passed by reference to the procedure. It contains the number of characters to compare. If COMPARE-LEN is larger than the number of characters in the strings, then the procedure might be comparing invalid data. The value of COMPARE-LEN should not exceed the bounds of either TEXT1-TEXT or TEXT2-TEXT.

The strings should be of equal size or padded with blanks up to the value of COMPARE-LEN. If all pairs of characters compare equally, the strings are considered equal. Otherwise, the first pair of unequal characters encountered is compared to determine their relative ordering. The string that contains the character with the higher ordering (higher PSV and higher OSV) is considered to be the string with the greater value. If substitution forms strings of unequal length, the comparison proceeds as if the shorter string were padded with blanks on the right. This padding ensures that the strings are of equal length.

CS-CMPEQLV

This is passed by reference to the procedure. It indicates the relational operator of the comparison. The valid values are as follows:

Value	Sample Value Name	Meaning
0	CS-CMPLSSV	TEXT1-TEXT is less than TEXT2-TEXT.
1	CS-CMPLEQV	TEXT1-TEXT is less than or equal to TEXT2-TEXT.
2	CS-CMPEQLV	TEXT1-TEXT is equal to TEXT2-TEXT.
3	CS-CMPGTRV	TEXT1-TEXT is greater than TEXT2-TEXT.
4	CS-CMPGEQV	TEXT1-TEXT is greater than or equal to TEXT2-TEXT.
5	CS-CMPNEQV	TEXT1-TEXT is not equal to TEXT2-TEXT.

CS-LOGICALV

This is passed by reference to the procedure. It indicates the type of comparison to be performed by the procedure. The valid values are as follows:

Value	Sample Value Name	Meaning
0	BINARY-V	Perform a binary comparison.
1	EQUIVALENT-V	Perform an equivalent comparison
2	LOGICAL-V	Perform a logical comparison

RESULT

This is returned as the value of the procedure. It contains the procedure result or indicates that an error occurred during the execution of the procedure. The possible values for RESULT and their meanings are shown in the following table.

Value	Condition-name	Meaning
0	CS-FALSEV	No error and the condition is FALSE
1	CS-DATAOKV	No error and the condition is TRUE

Other possible values returned by the procedure are as follows:

1000 1001 1002 3003 3006

For more information on the error result values, see Table 16–6 later in this section.

Sample Output

The output from Figure 16–26 is as follows:

```
RESULT =              0
```

VSNESCAPEMENT

This procedure takes the input text and rearranges it according to the escapement rules of the ccsversion. Both the character advance direction and the character escapement direction are used. If the character advance direction is positive, then the starting position for the escapement process is the leftmost position of the text in the DEST-TEXT parameter. If the character advance direction is negative, then the starting position for the escapement process is the rightmost position of the text in the DEST-TEXT parameter. From that point on, the character advance direction value and the character escapement direction values, in combination, control where each character should be placed in relation to the previous character.

Example

Figure 16–27 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the VSNESCAPEMENT library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation that follows the example, the parameters are explained using the names given to them in the example. When writing your program, choose parameter names that are appropriate for your use.

This example takes the string *ABCDEFG* and rearranges it according to the escapement rules of the ccsversion. Assume for this example a ccsversion number of 999 with a character advance direction of plus (+, left to right) and with the following character escapements:

Character	Escapement	Meaning
A	+	Left to right.
B	—	Right to left.
C	—	Right to left.
D	—	Right to left.
E	+	Left to right.
F	+	Left to right.
G	Blank	User character advance direction value.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT OUTPUT-FILE ASSIGN TO DISK.

DATA DIVISION.
FILE SECTION.
FD  OUTPUT-FILE
    LABEL RECORD IS STANDARD
    VALUE OF TITLE IS "OUT/COBOL85/VSNESCAPEMENT."
    PROTECTION SAVE
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS OUTPUT-RECORD.

01  OUTPUT-RECORD        PIC X(80).

WORKING-STORAGE SECTION.

01  OF-1.
    05  FILLER           PIC X(09)   VALUE "RESULT = ".
    05  OF-RESULT        PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(59)   VALUE SPACE.
01  OF-2.
    05  FILLER           PIC X(12)   VALUE "DEST-TEXT = ".
    05  OF-DEST-TEXT     PIC X(07).
    05  FILLER           PIC X(61)   VALUE SPACE.

****************************************************************
***  The following global declarations are used as parameters  ***
***  to the CENTRALSUPPORT procedures.                         ***
****************************************************************

01  DEST-TEXT            PIC X(07).
01  SOURCE-TEXT          PIC X(07).

77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
77  SOURCE-START         PIC S9(11)  USAGE BINARY.
77  TRANS-LEN            PIC S9(11)  USAGE BINARY.
77  RESULT               PIC S9(11)  USAGE BINARY.
77  VSN-NUM              PIC S9(11)  USAGE BINARY.
```

```
PROCEDURE DIVISION.
   INTLCOBOL85.
       OPEN OUTPUT OUTPUT-FILE.
       PERFORM VSNESCAPEMENT.
       CLOSE OUTPUT-FILE.
       STOP RUN.


***** VSNESCAPEMENT ******************************************
   VSNESCAPEMENT.
       CHANGE ATTRIBUTE FUNCTIONNAME OF "CENTRALSUPPORT" TO
"CENTRALSUPPORT".
       CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
       MOVE 999 TO VSN-NUM.
       MOVE "ABCDEFG" TO SOURCE-TEXT.
       MOVE 7 TO TRANS-LEN.
       CALL "VSNESCAPEMENT OF CENTRALSUPPORT"
         USING VSN-NUM,
               SOURCE-TEXT,
               SOURCE-START,
               DEST-TEXT,
               TRANS-LEN
         GIVING RESULT.
       MOVE RESULT TO OF-RESULT.
       WRITE OUTPUT-RECORD FROM OF-1.
       IF RESULT IS EQUAL TO CS-DATAOKV
         THEN MOVE DEST-TEXT TO OF-DEST-TEXT
              WRITE OUTPUT-RECORD FROM OF-2.
```

Figure 16–27. Calling the VSNESCAPEMENT Procedure

Explanation

VSN-NUM

This is passed by reference to the procedure. It specifies the ccsversion to be used. The ccsversion contains the escapement rules. Valid values for VSN-NUM are as follows:

Value	Meaning
Greater than or equal to 0	Specifies a ccsversion. The numbers of the ccsversions are listed in the *MLS Guide*.
–2	Specifies the system default ccsversion. If the system default ccsversion is not available, an error is returned.

SOURCE-TEXT

This is passed to the procedure. It contains the text to be arranged according to the escapement rules. You must determine the size of the record.

SOURCE-START

This is passed by reference to the procedure. It specifies where in SOURCE-TEXT the procedure is to begin rearranging the text.

DEST-TEXT

This is returned by the procedure. It contains the rearranged text. The length of the SOURCE-TEXT parameter and the DEST-TEXT parameter should be the same.

TRANS-LEN

This is passed by reference to the procedure. It specifies the number of characters to rearrange, beginning at SOURCE-START.

RESULT

This is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure.

Possible values returned by the procedure are as follows:

1	1001	1002	3000	3002	3003

For more information on the error result values, see Table 16–6 later in this section.

Sample Output

The output from Figure 16–27 is as follows:

```
RESULT =            1
DEST-TEXT = ADCBEFG
```

VSNGETORDERINGFOR_ONE_TEXT

This procedure returns the ordering information for a specified input text. The ordering information determines how the input text is collated. It includes the ordering sequence values (OSVs) and optionally the priority sequence values (PSVs) of the characters. It always includes any substitution of characters to be made when the input text is sorted. You can choose one of the following types of ordering information:

If the ordering type is . . .	Then the DEST parameter consists of . .
Equivalent	A sequence of OSVs.
Logical	A sequence of OSVs followed by PSVs.

Example

Figure 16–28 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the VSNGETORDERINGFOR_ONE_TEXT library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example obtains the OSVs and PSVs for the input text string "ABCæÆ." The ccsversion is CanadaEBCDIC. You can use the *MLS Guide* to determine that the ccsversion for CanadaEBCDIC is 74. You can also retrieve this number by calling the procedure VALIDATE_NAME_RETURN_NUM with the name CanadaEBCDIC. This example requests logical ordering information, so both the OSVs and PSVs are returned. This example also allows for maximum substitution, so the parameter max_osvs is equal to *itext_len * 3* and the parameter total_storage is equal to *max_osvs + round(max_osvs/2.0).*

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT OUTPUT-FILE ASSIGN TO DISK.

DATA DIVISION.
FILE SECTION.
FD  OUTPUT-FILE
    LABEL RECORD IS STANDARD
    VALUE OF TITLE IS "OUT/COBOL85/VSNGETORDONETEXT."
    PROTECTION SAVE
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS OUTPUT-RECORD.

01  OUTPUT-RECORD        PIC X(80).
```

```
   WORKING-STORAGE SECTION.

   01  OF-1.
       05  FILLER           PIC X(09)   VALUE "RESULT = ".
       05  OF-RESULT        PIC ZZZZZZZZZZZ9.
       05  FILLER           PIC X(59)   VALUE SPACE.
   01  OF-2.
       05  FILLER           PIC X(12)   VALUE "DEST-TEXT = ".
       05  OF-DEST-TEXT     PIC X(51).
       05  FILLER           PIC X(17)   VALUE SPACE.

  ****************************************************************
  ***  The following global declarations are used as parameters  ***
  ***  to the CENTRALSUPPORT procedures.                         ***
  ****************************************************************

   01  DEST-TEXT            PIC X(51).
   01  SOURCE-TEXT          PIC X(51).

   77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
   77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
   77  CS-LOGICALV          PIC S9(11)  USAGE BINARY    VALUE 2.
   77  DEST-START           PIC S9(11)  USAGE BINARY.
   77  ITEXT-LEN            PIC S9(11)  USAGE BINARY.
   77  MAX-OSVS             PIC S9(11)  USAGE BINARY.
   77  RESULT               PIC S9(11)  USAGE BINARY.
   77  SOURCE-START         PIC S9(11)  USAGE BINARY.
   77  TOTAL-STORAGE        PIC S9(11)  USAGE BINARY.
   77  VSN-NUM              PIC S9(11)  USAGE BINARY.

PROCEDURE DIVISION.
   INTLCOBOL85.
     OPEN OUTPUT OUTPUT-FILE.
     PERFORM VSNGETORDERINGFOR-ONE-TEXT.
     CLOSE OUTPUT-FILE.
     STOP RUN.

  ***** VSNGETORDERINGFOR-ONE-TEXT *********************************
   VSNGETORDERINGFOR-ONE-TEXT.
       CHANGE ATTRIBUTE FUNCTIONNAME OF "CENTRALSUPPORT" TO
"CENTRALSUPPORT".
       CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
       MOVE 74 TO VSN-NUM.
       MOVE 5 TO ITEXT-LEN.
       COMPUTE MAX-OSVS = ITEXT-LEN * 3.
       COMPUTE TOTAL-STORAGE = MAX-OSVS + MAX-OSVS / 2.
       MOVE "ABCæÆ" TO SOURCE-TEXT.
       CALL "VSNGETORDERINGFOR_ONE_TEXT OF CENTRALSUPPORT"
         USING VSN-NUM,
               SOURCE-TEXT,
               SOURCE-START,
```

```
        ITEXT-LEN,
        DEST-TEXT,
        DEST-START,
        MAX-OSVS,
        TOTAL-STORAGE,
        CS-LOGICALV
  GIVING RESULT.
MOVE RESULT TO OF-RESULT.
WRITE OUTPUT-RECORD FROM OF-1.
IF RESULT IS EQUAL TO CS-DATAOKV
  THEN MOVE DEST-TEXT TO OF-DEST-TEXT
       WRITE OUTPUT-RECORD FROM OF-2.
```

Figure 16–28. Calling the VSNGETORDERINGFOR_ONE_TEXT Procedure

Explanation

VSN-NUM

This is passed to the procedure. It contains the number of the ccsversion that is used. You can obtain the number by calling the CENTRALSTATUS procedure or by referring to the *MLS Guide*. The valid values are as follows:

If the value is . . .	Then . . .
Greater than or equal to 0	Specifies a ccsversion. The numbers of the ccsversions are listed in the *MLS Guide*.
–2	Use the system default ccsversion. If the system default ccsversion is not available, an error is returned.

SOURCE-TEXT

This record is passed to the procedure. It contains the text for which the ordering information is requested.

SOURCE-START

This is passed by reference to the procedure. It contains the offset of the location where the translation is to begin.

ITEXT-LEN

This is passed by reference to the procedure. It contains the length of the text that is to be translated.

DEST-TEXT

This is a record returned by the procedure. It contains the ordering information for the input text.

DEST-START

This is returned by the procedure. It designates the starting offset at which the result values are placed.

MAX-OSVS

This is an integer passed by reference to the procedure. It designates the maximum number of storage bytes to be used to store the ordering sequence values.

The value of MAX-OSVS should be the length of the input text. In the case when substitution is required, the MAX-OSVS value might need to be more than the length of the input text. The maximum substitution length defined for any ccsversion is 3; therefore, to allow for substitution for every character, the value of MAX-OSVS is as follows:

```
(length of source text in bytes) * 3
```

If the number of OSVs returned is less than MAX-OSVS, then the alphanumeric record is packed with the ordering sequence value for blank.

TOTAL-STORAGE

This is passed by reference to the procedure. It defines the maximum number of bytes needed to store the complete ordering information for the text. If you request equivalent ordering information, TOTAL-STORAGE and MAX-OSVS should be set the same. If you request logical ordering information, you must provide space for the four-bit priority values in addition to the space allowed for the OSVs. Each OSV has one PSV, and one byte can hold two PSVs. Therefore, the space allowed for PSVs *MAX-OSVS/2*, and the value of TOTAL-STORAGE should be set as follows:

```
MAX-OSVS + (MAX-OSVS)/2
```

When the ordering information is returned by the procedure, all the OSVs are listed first, followed by all the PSVs.

CS-LOGICALV

This is an integer passed by reference to the procedure. It indicates the type of ordering information you want, as follows:

Value	Sample Value Name	Meaning
1	CS-EQUIVALENTV	OSVs only
2	CS-LOGICALV	PSVs only

RESULT

This is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure. Possible values returned by this procedure are as follows:

0	1	1000	1001	1002	3000
3001	3002	3003	3006	3008	

For more information on the error result values, see Table 16–6 later in this section.

Sample Output

The output from Figure 16–28 is as follows:

```
RESULT = CS_DATAOKV
DEST-TEXT = 414243414541 454040404040 404040111221 111111111111
```

Based on the values of DEST-TEXT, the OSVs are 65, 66, 67, 65, 69, 65, and 69. The PSVs are 1, 1, 1, 2, 2, 1, and 1.

VSNINSPECT_TEXT

This procedure searches a specified text for characters that are present or not present in a requested data class. The SCANNED-CHARS parameter is an integer that represents the number of characters that were searched when the criteria specified in the CS_NOT_INTSETV parameter were met. If SCANNED-CHARS is equal to INSPECT-LEN, then all the characters were searched but none met the criteria. Otherwise, adding the TEXT-START value to the RESULT value gives the location of the character, from the start of the array, that met the search criteria.

Example

Figure 16–29 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the VSNINSPECT_TEXT library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example examines a record that contains two fields, a name and a phone number. The name is verified to contain only alphabetic characters as defined by the France ccsversion. You can use the *MLS Guide* to determine that the ccsversion number for France is 35. You can also retrieve this number by calling the procedure CCSVSNNUM with the name France.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT OUTPUT-FILE ASSIGN TO DISK.
DATA DIVISION.
FILE SECTION.
FD  OUTPUT-FILE
    LABEL RECORD IS STANDARD
    VALUE OF TITLE IS "OUT/COBOL85/VSNINSPECTTEXT."
    PROTECTION SAVE
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS OUTPUT-RECORD.
  01  OUTPUT-RECORD        PIC X(80).
  WORKING-STORAGE SECTION.
  01  OF-1.
    05  FILLER           PIC X(09)   VALUE "RESULT = ".
    05  OF-RESULT        PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(59)   VALUE SPACE.
01  OF-2.
    05  FILLER           PIC X(16)   VALUE "SCANNED-CHARS = ".
    05  OF-SCANNED-CHARS PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(52)   VALUE SPACE.
```

```
 ****************************************************************
 ***  The following global declarations are used as parameters  ***
 ***  to the CENTRALSUPPORT procedures.                         ***
 ****************************************************************

  01  SOURCE-TEXT          PIC X(41).

  77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
  77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
  77  CS-NOT-INTSETV       PIC S9(11)  USAGE BINARY    VALUE 0.
  77  CS-NUMERICSV         PIC S9(11)  USAGE BINARY    VALUE 13.
  77  ID-LEN               PIC S9(11)  USAGE BINARY    VALUE 10.
  77  INSPECT-LEN          PIC S9(11)  USAGE BINARY.
  77  NAME-LEN             PIC S9(11)  USAGE BINARY    VALUE 30.
  77  SCANNED-CHARS        PIC S9(11)  USAGE BINARY.
  77  SOURCE-START         PIC S9(11)  USAGE BINARY.
  77  RESULT               PIC S9(11)  USAGE BINARY.
  77  VSN-NUM              PIC S9(11)  USAGE BINARY.

  PROCEDURE DIVISION.
  INTLCOBOL85.
      OPEN OUTPUT OUTPUT-FILE.
      PERFORM VSNINSPECT-TEXT.
      CLOSE OUTPUT-FILE.
      STOP RUN.

 ***** VSNINSPECT-TEXT ******************************************
  VSNINSPECT-TEXT.
      CHANGE ATTRIBUTE FUNCTIONNAME OF "CENTRALSUPPORT" TO
"CENTRALSUPPORT".
      CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
      MOVE 35 TO VSN-NUM.
      MOVE NAME-LEN TO INSPECT-LEN.
      MOVE "7775961089John Alan Smith               " TO SOURCE-TEXT.
      CALL "VSNINSPECT_TEXT OF CENTRALSUPPORT"
        USING VSN-NUM,
              SOURCE-TEXT,
              SOURCE-START,
              INSPECT-LEN,
              CS-NUMERICSV,
              CS-NOT-INTSETV,
              SCANNED-CHARS
        GIVING RESULT.
 MOVE RESULT TO OF-RESULT.
      WRITE OUTPUT-RECORD FROM OF-1.
      IF RESULT IS EQUAL TO CS-DATAOKV AND
         SCANNED-CHARS IS EQUAL TO ID-LEN
        THEN MOVE SCANNED-CHARS TO OF-SCANNED-CHARS
             WRITE OUTPUT-RECORD FROM OF-2.
```

Figure 16–29. Calling the VSNINSPECT_TEXT Procedure

Explanation

VSN-NUM

This is passed by reference to the procedure. It specifies the ccsversion to be used. The ccsversion contains the rules for applying a truthset. The valid values for VSN-NUM are as follows:

If the value is . . .	Then the parameter specifies . . .
Greater than or equal to 0	Specifies a ccsversion. The numbers of the ccsversions are listed in the *MLS Guide*.
–2	Specifies the system default ccsversion. If the system default ccsversion is not available, an error is returned.

SOURCE-TEXT

This is passed to the procedure. The record is searched for a character using the requested truthset and type of search. You determine the size of the record.

SOURCE-START

This is passed by reference to the procedure. It contains the byte offset in SOURCE-TEXT, relative to 0 (zero), at which the search begins.

ID-LEN

This is passed by reference to the procedure. It specifies the length of the inspected test; that is, the number of characters found to be numeric.

INSPECT-LEN

This is passed by reference to the procedure. It specifies the number of characters to be searched beginning at SOURCE-START. It specifies that maximum length of the search.

NAME-LEN

This passed by reference to the procedure. It specifies the length of the name to be inspected.

CS-NUMERICSV

This is passed to the procedure. It indicates the type of truthset to be used for the search. The valid values for CS-NUMERICSV and their meanings are as follows:

Value	Sample Value Name and Meaning
12	CS-ALPHAV Alphabetic truthset. It identifies the characters defined as alphabetic in the specified ccsversion.
13	CS-NUMERICSV Numeric truthset. It identifies the characters defined as numeric in the specified ccsversion.
14	CS-PRESENTATIONV Presentation truthset. It identifies the characters in the ccsversion that can be represented on a presentation device, such as a printer.
15	CS-SPACESV Spaces truthset. It identifies the characters defined as spaces in the specified ccsversion.
16	CS-LOWERCASEV Lowercase truthset. It identifies the characters defined as lowercase alphabetic in the specified ccsversion.
17	CS-UPPERCASEV Uppercase truthset. It identifies the characters defined as uppercase alphabetic in the specified ccsversion.

A ccsversion is not required to have a definition for each of these truthsets. Some of the truthsets, such as 16 and 17, are optional. A result of 4002 might be returned if the truthset was not defined for the ccsversion. The input text remains unchanged.

CS-NOT-INSETV

This parameter is passed to the procedure, and indicates the type of search to be performed. The valid values for this parameter and their meanings are as follows:

Value	Sample Data Name	Meaning
0	CS-NOTINTSETV	Search the text until a character is found that is not in the requested truthset.
1	CS-INTSETV	Search the text until a character is found that is in the requested truthset.

SCANNED-CHARS

This integer is returned by the procedure. It contains the number of characters, relative to 0 (zero), that were scanned when the search criteria was met.

RESULT

This parameter is the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure. Possible values returned by VSNINSPECT_TEXT are as follows:

1	1000	1001	1002	3000
3001	3003	3006	3007	4002

For more information on the error result values, see Table 16–6 later in this section.

Sample Output

The output from Figure 16–29 is as follows:

```
RESULT =            1
SCANNED-CHARS =            10
```

VSNTRANS_TEXT

This procedure applies a specified mapping table to the source text and places the result into the destination parameter. You can use the same record for both the source and destination text.

You might use this procedure to translate alternative digits received as data into numeric digits for arithmetic processing.

Example

Figure 16–30 shows the parameter declarations and the PROCEDURE DIVISION syntax required to call the VSNTRANS_TEXT library procedure. The declarations identify the category of data-item required for parameter matching. For example, numeric items must be declared PIC S9(11) USAGE BINARY.

In the explanation following the example, the parameters are explained using the names given to them in the example. In your program, choose parameter names that are appropriate for your use.

This example translates a string in lowercase letters to uppercase letters using the CanadaEBCDIC ccsversion. The input string is "pæan." You can use the *MLS Guide* to determine that the ccsversion number for CanadaEBCDIC is 74. You can also retrieve this number by calling the procedure CCSVSNNUM with the name CanadaEBCDIC.

```
IDENTIFICATION DIVISION.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT OUTPUT-FILE ASSIGN TO DISK.

DATA DIVISION.
FILE SECTION.
FD  OUTPUT-FILE
    LABEL RECORD IS STANDARD
    VALUE OF TITLE IS "OUT/COBOL85/VSNTRANSTEXT."
    PROTECTION SAVE
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS OUTPUT-RECORD.
  01  OUTPUT-RECORD        PIC X(80).

WORKING-STORAGE SECTION.
  01  OF-1.
    05  FILLER           PIC X(09)   VALUE "RESULT = ".
    05  OF-RESULT        PIC ZZZZZZZZZZZ9.
    05  FILLER           PIC X(59)   VALUE SPACE.
01  OF-2.
    05  FILLER           PIC X(12)   VALUE "DEST-TEXT = ".
    05  OF-DEST-TEXT     PIC X(07).
    05  FILLER           PIC X(61)   VALUE SPACE.
```

```
 ****************************************************************
 ***  The following global declarations are used as parameters  ***
 ***  to the CENTRALSUPPORT procedures.                         ***
 ****************************************************************

  01  DEST-TEXT            PIC X(07).
  01  SOURCE-TEXT          PIC X(07).

  77  CS-DATAOKV           PIC S9(11)  USAGE BINARY    VALUE 1.
  77  CS-FALSEV            PIC S9(11)  USAGE BINARY    VALUE 0.
  77  CS-LOWTOUPCASEV      PIC S9(11)  USAGE BINARY    VALUE 7.
  77  DEST-START           PIC S9(11)  USAGE BINARY.
  77  SOURCE-START         PIC S9(11)  USAGE BINARY.
  77  RESULT               PIC S9(11)  USAGE BINARY.
  77  TRANS-LEN            PIC S9(11)  USAGE BINARY.
  77  VSN-NUM              PIC S9(11)  USAGE BINARY.

  PROCEDURE DIVISION.
  INTLCOBOL85.
      OPEN OUTPUT OUTPUT-FILE.
      PERFORM VSNTRANS-TEXT.
      CLOSE OUTPUT-FILE.
      STOP RUN.

 ***** VSNTRANS-TEXT ********************************************
  VSNTRANS-TEXT.
      CHANGE ATTRIBUTE FUNCTIONNAME OF "CENTRALSUPPORT" TO
"CENTRALSUPPORT".
      CHANGE ATTRIBUTE LIBACCESS OF "CENTRALSUPPORT" TO BYFUNCTION.
      MOVE 74 TO VSN-NUM.
      MOVE 4 TO TRANS-LEN.
      MOVE "pæan" TO SOURCE-TEXT.
      CALL "VSNTRANS_TEXT OF CENTRALSUPPORT"
        USING VSN-NUM,
              SOURCE-TEXT,
              SOURCE-START,
              DEST-TEXT,
              DEST-START,
              TRANS-LEN,
              CS-LOWTOUPCASEV
        GIVING RESULT.
      MOVE RESULT TO OF-RESULT.
      WRITE OUTPUT-RECORD FROM OF-1.
      IF RESULT IS EQUAL TO CS-DATAOKV
        THEN MOVE DEST-TEXT TO OF-DEST-TEXT.
             WRITE OUTPUT-RECORD FROM OF-2.
```

Figure 16–30. Calling the VSNTRANS_TEXT Procedure

Explanation

VSN-NUM

This is an integer passed by reference to the procedure. It contains the number of the ccsversion to be used. The ccsversion contains the rules for translation of text. Refer to the *MLS Guide* for a list of the ccsversion numbers. The valid values for VSN-NUM and their meanings are as follows:

If the value is . . .	Then . . .
Greater than or equal to 0	Use the specified ccsversion number.
–2	Use the system default ccsversion. If the system default ccsversion is not available, an error is returned.

SOURCE-TEXT

This is passed to the procedure. It contains the data to translate. You should determine the size of this record.

SOURCE-START

This is passed to the procedure. It designates the byte offset, relative to 0 (zero), in SOURCE-TEXT at which translation is to begin.

DEST-TEXT

This is returned by the procedure. It contains the translated text. This record and the record in the SOURCE-TEXT parameter should be the same size.

DEST-START

This is passed to the procedure. It indicates the offset in the DEST-TEXT parameter where the translated text is to be placed.

TRANS-LEN

This is passed to the procedure. It designates the number of characters in the SOURCE-TEXT parameter to translate, beginning at SOURCE-START.

CS-LOWTOUPCASEV

This is passed to the procedure. It designates the type of translation requested.

The valid values for CS-LOWTOUPCASEV and their meanings are as follows:

Value	Sample Value Name and Meaning
5	CS-NUMTOALTDIGV Translate numbers 0 through 9 to alternate digits specified in the ccsversion.
6	CS-ALTDIGTONUMV Translate alternate digits to numbers 0 through 9.
7	CS-LOWTOUPCASEV Translate all characters from lowercase to uppercase.
8	CS-UPTOLOWCASEV Translate all character from uppercase to lowercase.
9	CS-ESCMENTPERCHARV Translate a character to its escapement value.

A ccsversion is not required to have a definition for each of these tables. Some of the tables, such as 5, 6, 7, and 8, are optional. A result of 4002 might be returned if the table was not defined for the ccsversion. The input text remains unchanged.

RESULT

This is returned as the value of the procedure. It indicates whether an error occurred during the execution of the procedure. Values greater than or equal to 1000 indicate an error. An explanation of the error result values can be found at the end of this section. You should check the procedure result whenever you use this procedure. Possible values returned by VSNTRANS_TEXT are as follows:

1	1000	1001	1002	3000
3001	3002	3003	3006	4002

For more information on the error result values, see Table 16–6 later in this section.

Sample Output

The output from Figure 16–30 is as follows:

```
RESULT =           1
DEST-TEXT = PÆAN
```

Errors

All of the procedures in the CENTRALSUPPORT library return integer results to indicate the success or failure of the procedure. Figure 16–31 shows a sample set of declarations for the message values.

```
01  ERROR-VALUES       PIC S9(11)  USAGE BINARY.
      88  CS-FILE-ACCESS-ERRORV        VALUE  1000.
      88  CS-FAULTV                    VALUE  1001.
      88  CS-SOFTERRV                  VALUE  1002.
      88  LANGUAGE-NOT-FOUNDV          VALUE  2001.
      88  CONVENTION-NOT-FOUNDV        VALUE  2002.
      88  INCOMPLETE-DATAV             VALUE  2004.
      88  INCOMPLETE-CHARV             VALUE  2005.
      88  BAD-ARRAY-DESCRIPTIONV       VALUE  3000.
      88  ARRAY-TOO-SMALLV             VALUE  3001.
      88  BAD-DATA-LENV                VALUE  3002.
      88  NO-NUM-FOUNDV                VALUE  3003.
      88  NO-NAME-FOUNDV               VALUE  3004.
      88  NO-MSGNUM-FOUNDV             VALUE  3005.
      88  BAD-TYPE-CODEV               VALUE  3006.
      88  BAD-FLAGV                    VALUE  3007.
      88  BAD-TEXT-PARAMV              VALUE  3008.
      88  BAD-TEMPCHARV                VALUE  3011.
      88  BAD-DATEINPUTV               VALUE  3012.
      88  BAD-TIMEINPUTV               VALUE  3013.
      88  CNV-EXISTS-ERRV              VALUE  3014.
      88  BAD-MAXDIGITSV               VALUE  3015.
      88  BAD-FRACDIGITSV              VALUE  3016.
      88  BAD-ALTFRACDIGITSV           VALUE  3017.
      88  BAD-LDATETEMPV               VALUE  3018.
      88  BAD-SDATETEMPV               VALUE  3019.
      88  BAD-NDATETEMPV               VALUE  3020.
      88  BAD-LTIMETEMPV               VALUE  3021.
      88  BAD-NTIMETEMPV               VALUE  3022.
      88  BAD-MONTEMPV                 VALUE  3023.
      88  BAD-NUMTEMPV                 VALUE  3024.
      88  BAD-LPPV                     VALUE  3027.
      88  BAD-CPLV                     VALUE  3028.
      88  REQSYMBOLV                   VALUE  3029.
      88  BAD-TEMPLENV                 VALUE  3030.
      88  MUTUAL-EXCLUSIVEV            VALUE  3031.
      88  BAD-MINDIGITSV               VALUE  3032.
      88  MISSING-RBRACKETV            VALUE  3033.
      88  MISSING-TCCOLONV             VALUE  3034.
      88  BAD-INPUTVALV                VALUE  3035.
      88  CNV-NOTAVAILV                VALUE  3036.
      88  CNVFILE-NOTPRESENTV          VALUE  3037.
      88  BAD-PRECISIONV               VALUE  3038.
      88  NO-CNVNAMEV                  VALUE  3039.
      88  DEL-PERMANENTCNV-ERRV        VALUE  3040.
      88  NO-HEXCODE-DELIMV            VALUE  3041.
```

```
88  BAD-HEXCODEV                VALUE  3042.
88  NO-ALTCURR-DELIMV           VALUE  3043.
88  DATA-NOT-FOUNDV             VALUE  4002.
88  COMPLEX-TRAN-REQV           VALUE  4004.
```

Figure 16–31. Sample Declarations for Message Values

Explanation of Error Values

The following table explains the general meaning of various ranges of error messages.

Messages in the range . . .	Indicate . . .
1000 through 1999	A system software error.
2000 through 2999	That the caller passed invalid data to a procedure, but the CENTRALSUPPORT library was able to return some valid data.
3000 through 3999	That the caller passed invalid data to a CENTRALSUPPORT procedure, and the CENTRALSUPPORT library was unable to return any valid data.
4000 through 4999	That the caller passed some sort of data for which the CENTRALSUPPORT library could find no return information. CENTRALSUPPORT completed the request, but no data was returned.

Table 16–6 lists the error numbers that can be returned for internationalization and the specific descriptions of the error messages that you can have your program display.

For information about the message parts, refer to GET_CS_MSG earlier in this section.

For a list of the complete error messages and for information about the corrective actions to be taken if an error occurs, refer to the *MLS Guide*.

Table 16–6. Error Result Values

Error Value	Meaning
1000	An error occurred while accessing the SYSTEM/CCSFILE or the SYSTEM/CONVENTIONS file.
1001	An unexpected fault occurred in CENTRALSUPPORT and a program dump might occur. Your request cannot be processed at this time.
1002	A CENTRALSUPPORT software error was detected and a program dump might occur. Your request cannot be processed at this time.
2001	The data is not in the requested language. It is in MYSELF.LANGUAGE or the SYSTEM LANGUAGE or the first available LANGUAGE.
2002	The data is not in the requested convention; it is in MYSELF.CONVENTION or the SYSTEM CONVENTION.
2004	Only partial data is being returned. There was insufficient space in the output array.
2005	Incomplete data is being returned for a multibyte stream.
3000	A parameter was incorrectly specified as less than or equal to 0.
3001	The output array size is smaller than the length of the data it is supposed to contain.
3002	At least one array length is invalid or the offset + length is greater than the size of the array.
3003	The requested number was not found.
3004	The requested name was not found.
3005	The requested number was not found.
3006	The type code specified is out of the acceptable range.
3007	The flag specified is out of the acceptable range.
3008	The space for OSVs or total storage allocated in OUTPUT is not big enough for OSVs and/or PSVs.
3009	The absolute value of AMT is greater than the maximum double-precision integer or the AMT value is not an integer.
3011	An invalid control character was detected in the template.
3012	The input date contains illegal characters.
3013	The input time contains illegal characters.
3014	An attempt was made to add a new convention with the name of an existing convention.
3015	The maximum digits value is either missing or out of range.
3016	The fractional digits value is either missing or out of range.

Table 16–6. Error Result Values

Error Value	Meaning
3017	The international fractional digits value is either missing or out of range.
3018	The long date template is either missing or contains invalid information.
3019	The short date template is either missing or it contains invalid information.
3020	The numeric date template is either missing or contains invalid information.
3021	The long time template is either missing or it contains invalid information.
3022	The numeric time template is either missing or contains invalid information.
3023	The monetary template is either missing or it contains invalid information.
3024	The numeric template is either missing or it contains invalid information.
3027	The lines per page value is either missing or it is out of range.
3028	The characters per line value is either missing or it is out of range.
3029	A required symbol in either the monetary or the numeric template is missing.
3030	An invalid template length value was encountered.
3031	A mutually exclusive combination of control characters has been encountered in a monetary or numeric template.
3032	The mindigits field in a "t" control character in a monetary or numeric template is out of range.
3033	A right bracket "]" is required to terminate a "t" control character symbol definition list.
3034	An expected colon ":" is missing from the "t" control character in a monetary or numeric template.
3035	The input value did not contain digits or an expected symbol was missing.
3036	Specified convention does not exist and cannot be retrieved, modified, or deleted.
3037	A convention definition cannot be added, modified, or deleted.
3038	The "PRECISION" value is not in the range of 0 to 9.
3039	A required convention name was not provided.
3040	The named convention is a standard convention and cannot be modified or deleted.

Table 16–6. Error Result Values

Error Value	Meaning
3041	A hexadecimal value representing a symbol in a monetary or numeric template is missing a required delimiter.
3042	An invalid character was encountered in a hex value representing a symbol in a monetary or numeric template.
3043	The international currency notation is missing a required terminating delimiter.
3044	The date components are separated by an invalid character.
3045	The year component exceeds 2 digits.
3046	A nonzero value is required for the year component.
3047	The month value is outside of the valid range. Acceptable values are in the range 1 through 12.
3048	The day value is outside of the valid range. Acceptable value ranges for the months January through December are as follows: January—1 through 31 February—1 through 28 (29 in a leap year) March—1 through 31 Apri1—through 30 May—1 through 31 June—1 through 30 July—1 through 31 August—1 through 31 September—1 through 30 October—1 through 31 November—1 through 30 December—1 through 31
3049	An input date is required but missing
3050	Time components are separated by an invalid character.
3051	The hour value is outside of the valid range for the 24-hour clock. Acceptable values are in the range 0 through 23.
3052	The hour value is outside of the valid range for the 12-hour clock. Acceptable values are in the range 1 through 12.
3053	The minute value is outside the valid range. Acceptable values are in the range 0 through 59.
3054	The second value is outside of the valid range. Acceptable values are in the range 0 through 59.
3055	The partial second value contains invalid characters.
3056	An input time is required but missing.
3057	The month value is required but missing.

Table 16–6. Error Result Values

Error Value	Meaning
3058	The day of year value is outside the valid range. Acceptable values are in the range 1 through 365 (1 through 366 for a leap year).
3059	The day of year value cannot be calculated because a date component (year, month, or day) is missing.
4002	The requested data was not found.
4004	The translation must be performed by calling the CCSTOCCS_TRANS_TEXT_COMPLEX procedure.

Using the Properties File

To save time with coding your programs for internationalization, the release media includes a file that contains the predefined syntax for a variety of CENTRALSUPPORT data items and procedure declarations. You can copy specific segments of this file into your program by using the COPY statement with a specified sequence number range as shown in Figure 16–31. The complete name of the properties file is SYMBOL/INTL/COBOL85/PROPERTIES, and the content of the file is as follows:

```
000010*COPYRIGHT* * * * * * * * * * * * * * * * * * * * * * * * * * * * * **
000011**                                                                    **
000012*T         TITLE: A SERIES SYSTEM SOFTWARE RELEASE 45.1               **
000013**                                                                    **
000014*F           FILE ID: SYMBOL/INTL/COBOL85/PROPERTIES                  **
000015**                                                                    **
000016*C         COPYRIGHT (C) 1997 1998 UNISYS CORPORATION                 **
000017**                  ALL RIGHTS RESERVED                               **
000018**                  UNISYS PROPRIETARY                                **
000019**                                                                    **

000020**   THIS  MATERIAL  IS  PROPRIETARY  TO  UNISYS  CORPORATION         **
000021**   AND IS  NOT TO BE REPRODUCED, USED OR  DISCLOSED  EXCEPT         **
000022**   IN ACCORDANCE  WITH  PROGRAM  LICENSE  OR  UPON  WRITTEN         **
000023**   AUTHORIZATION OF UNISYS CORPORATION.                              **
000024**                                                                    **
000025*** * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * **
000026**                                                                    **
000027**   THE WITHIN INFORMATION IS NOT INTENDED TO BE NOR  SHOULD         **
000028**   SUCH   BE  CONSTRUED  AS   AN   AFFIRMATION   OF   FACT,         **
000029**   REPRESENTATION  OR  WARRANTY  BY  UNISYS  CORPORATION OF         **
000030**   ANY  TYPE,  KIND OR CHARACTER.  ANY PRODUCT AND  RELATED         **
000031**   MATERIALS DISCLOSED HEREIN IS  ONLY  FURNISHED  PURSUANT         **
000032**   AND  SUBJECT  TO  THE  TERMS  AND  CONDITIONS OF A  DULY         **
000033**   EXECUTED LICENSE AGREEMENT.  THE ONLY WARRANTIES MADE BY         **
000034**   UNISYS  WITH RESPECT TO THE  PRODUCTS DESCRIBED  IN THIS         **
000035**   MATERIAL ARE SETFORTH IN THE ABOVE MENTIONED AGREEMENT.          **
000036**                                                                    **
000037**   THE CUSTOMER SHOULD EXERCISE CARE TO ASSURE THAT USE  OF         **
000038**   THE SOFTWARE WILL BE IN FULL COMPLIANCE WITH LAWS, RULES         **
000039**   AND REGULATIONS OF THE  JURISDICTIONS  WITH  RESPECT  TO         **
000040**   WHICH IT IS USED.                                                **
000041**                                                                    **
000042*COPYRIGHT* * * * * * * * * * * * * * * * * * * * * * * * * * * * * **
000092$$ VERSION 45.204.0003
000100
000200*
000300* ----------------------------------------------------------------
000400* NOTE: Each procedure declared in this file has been assigned a
000500*       static sequence number range.  DO NOT resequence any
000600*       procedure declaration outside of the range it currently
000700*       resides.  If new procedures need to be added, use the end
```

```
000800*       the current declarations and sequence accordingly.
000900*       Software relies on the sequence ranges to be able to
001000*       $INCLUDE only those procedure that are needed.  Disruption
001100*       of this sequencing convention will cause compilation
001200*       errors in the software.
001300* ---------------------------------------------------------------
001400*
001500* Portions of this file are not $INCLUDEd by host software.  Do
001600* not add program information to the sequence ranges 000000-099999
001700* and 900000 through 999999.  These areas are intended for COBOL85
001800* symbol to make this file compileable in its entirety.  Follow
001900* the table below when adding or changing declarations in the
002000* $INCLUDE file:
002100*
002200* Sequence range        Description
002300* ------------------    ----------------------------------------
002400* 100000 - 199999       WORKING-STORAGE section
002500* 200000 - 299999       LOCAL-STORAGE section
002600* 300000 - 399999       *** nothing ***
002700* 500000 - 599999       Character Set Procedure Declarations
002800* 600000 - 699999       Conventions Procedure Declarations
002900* ---------------------------------------------------------------
003000*
003100
003200 IDENTIFICATION DIVISION.
003300 ENVIRONMENT DIVISION.
003400 DATA DIVISION.
100000 WORKING-STORAGE SECTION.
100100
100200*
100300* Result value information
100400*
100500
100600 77  CS-FIRST-ERRORV      PIC S9(11)  USAGE BINARY   VALUE  1000.
100700*    The lowest error number returned.
100800
100900 77  CS-LAST-MSGV         PIC S9(11)  USAGE BINARY   VALUE  9999.
101000
101100 77  CS-FALSEV            PIC S9(11)  USAGE BINARY   VALUE  0.
101200*    No errors; FALSE.
101300
101400 77  CS-DATAOKV           PIC S9(11)  USAGE BINARY   VALUE  1.
101500*    No errors; TRUE.
101600
101700* Global TYPE values:
101800
101900 77  CS-INPUTV            PIC S9(11)  USAGE BINARY   VALUE  0.
102000
102100 77  CS-OUTPUTV           PIC S9(11)  USAGE BINARY   VALUE  1.
102200
102300 77  CS-CHARACTER-SETV    PIC S9(11)  USAGE BINARY   VALUE  0.
102400
```

```
102500 77  CS-CCSVERSIONV       PIC S9(11)  USAGE BINARY   VALUE  1.
102600
102700 77  CS-NUMTOALTDIGV      PIC S9(11)  USAGE BINARY   VALUE  5.
102800*    translate the nums 0 - 9 to alternate digits that were
102900*    specified in the ccsversion
103000
103100 77  CS-ALTDIGTONUMV      PIC S9(11)  USAGE BINARY   VALUE  6.
103200
103300 77  CS-LOWTOUPCASEV      PIC S9(11)  USAGE BINARY   VALUE  7.
103400*    translates all lowercase chars to uppercase
103500
103600 77  CS-UPTOLOWCASEV      PIC S9(11)  USAGE BINARY   VALUE  8.
103700*    translates all uppercase chars to lowercase
103800
103900 77  CS-ESCMENTPERCHARV   PIC S9(11)  USAGE BINARY   VALUE  9.
104000*    given a character translates it to its associated
104100*    escapement value, +, -, 0, blank, or *
104200
104300 77  CS-ALPHAV            PIC S9(11)  USAGE BINARY   VALUE  12.
104400*    set has all alphabetic chars specified in the ccsversion
104500
104600 77  CS-NUMERICSV         PIC S9(11)  USAGE BINARY   VALUE  13.
104700*    set has all numeric chars
104800
104900 77  CS-PRESENTATIONV     PIC S9(11)  USAGE BINARY   VALUE  14.
105000*    set has all displayable chars
105100
105200 77  CS-SPACESV           PIC S9(11)  USAGE BINARY   VALUE  15.
105300*    set has all space characters
105400
105500 77  CS-LOWERCASEV        PIC S9(11)  USAGE BINARY   VALUE  16.
105600*    set has lowercase characters
105700
105800 77  CS-UPPERCASEV        PIC S9(11)  USAGE BINARY   VALUE  17.
105900*    set has uppercase characters
106000
106100* Types of ordering which may be specified.  Substitution applies
106200* to both EQUIVALENT and LOGICAL.
106300
106400 77  CS-BINARYV           PIC S9(11)  USAGE BINARY   VALUE  0.
106500*    No translation is to be performed.
106600
106700 77  CS-EQUIVALENTV       PIC S9(11)  USAGE BINARY   VALUE  1.
106800*    Translate data to OSVs only.
106900
107000 77  CS-LOGICALV          PIC S9(11)  USAGE BINARY   VALUE  2.
107100*    Translate data to OSVs and PSVs.
107200
107300* FLAG values
107400
107500 77  CS-NOTINTSETV        PIC S9(11)  USAGE BINARY   VALUE  0.
107600*    search while not in truthset
```

```
107700
107800 77  CS-INTSETV           PIC S9(11)  USAGE BINARY   VALUE  1.
107900*    search while in truthset
108000
108100 77  CS-VSN-NOT-SPECIFIEDV PIC S9(11)  USAGE BINARY  VALUE  -2.
108200*    Version is intentionally not specified.
108300
108400* Comparison types
108500
108600 77  CS-CMPLSSV           PIC S9(11)  USAGE BINARY   VALUE  0.
108700
108800 77  CS-CMPLEQV           PIC S9(11)  USAGE BINARY   VALUE  1.
108900
109000 77  CS-CMPEQLV           PIC S9(11)  USAGE BINARY   VALUE  2.
109100
109200 77  CS-CMPGTRV           PIC S9(11)  USAGE BINARY   VALUE  3.
109300
109400 77  CS-CMPGEQV           PIC S9(11)  USAGE BINARY   VALUE  4.
109500
109600 77  CS-CMPNEQV           PIC S9(11)  USAGE BINARY   VALUE  5.
109700
109800* Date and Time type values:
109900*
110000
110100 77  CS-LDATEV            PIC S9(11)  USAGE BINARY   VALUE  0.
110200*    Long date only
110300
110400 77  CS-SDATEV            PIC S9(11)  USAGE BINARY   VALUE  1.
110500*    Short date only
110600
110700 77  CS-NDATEV            PIC S9(11)  USAGE BINARY   VALUE  2.
110800*    Numeric date only
110900
111000 77  CS-LTIMEV            PIC S9(11)  USAGE BINARY   VALUE  3.
111100*    Long time only
111200
111300 77  CS-NTIMEV            PIC S9(11)  USAGE BINARY   VALUE  4.
111400*    Numeric time only
111500
111600 77  CS-LDATELTIMEV       PIC S9(11)  USAGE BINARY   VALUE  5.
111700*    Long date and long time
111800
111900 77  CS-LDATENTIMEV       PIC S9(11)  USAGE BINARY   VALUE  6.
112000*    Long date and numeric time
112100
112200 77  CS-SDATELTIMEV       PIC S9(11)  USAGE BINARY   VALUE  7.
112300*    Short date and long time
112400
112500 77  CS-SDATENTIMEV       PIC S9(11)  USAGE BINARY   VALUE  8.
112600*    Short date and numeric time
112700
112800 77  CS-NDATELTIMEV       PIC S9(11)  USAGE BINARY   VALUE  9.
```

```
112900*    Numeric date and long time
113000
113100 77  CS-NDATENTIME        PIC S9(11)  USAGE BINARY   VALUE  10.
113200*    Numeric date and numeric time
113300
113400* Type values for use with template procedures:
113500*
113600
113700 77  CS-LONGDATE-TEMPV    PIC S9(11)  USAGE BINARY   VALUE  0.
113800
113900 77  CS-SHORTDATE-TEMPV   PIC S9(11)  USAGE BINARY   VALUE  1.
114000
114100 77  CS-NUMDATE-TEMPV     PIC S9(11)  USAGE BINARY   VALUE  2.
114200
114300 77  CS-LONGTIME-TEMPV    PIC S9(11)  USAGE BINARY   VALUE  3.
114400
114500 77  CS-NUMTIME-TEMPV     PIC S9(11)  USAGE BINARY   VALUE  4.
114600
114700 77  CS-MONETARY-TEMPV    PIC S9(11)  USAGE BINARY   VALUE  5.
114800
114900 77  CS-NUMERIC-TEMPV     PIC S9(11)  USAGE BINARY   VALUE  6.
115000
115100
115200* Date and time display model type values:
115300
115400 77  CS-DATE-DISPLAYMODEL PIC S9(11)  USAGE BINARY   VALUE  0.
115500
115600 77  CS-TIME-DISPLAYMODEL PIC S9(11)  USAGE BINARY   VALUE  1.
115700
115800* EXPAND values:
115900
116000 77  CS-EXPAND-HEXV          PIC S9(11)  USAGE BINARY  VALUE  0.
116100
116200 77  CS-EXPAND-HEXTOEBCDICV  PIC S9(11)  USAGE BINARY  VALUE  1.
116300
116400
116500* Option Parameter Values for CCSTOCCS-TRANS-TEXT-COMPLEX
116600
116700 77  CS-OPT-INITIAL-COMPLETEV PIC S9(11)  USAGE BINARY  VALUE 0.
116800 77  CS-OPT-INITIAL-HEADV     PIC S9(11)  USAGE BINARY  VALUE 2.
116900 77  CS-OPT-HEADV             PIC S9(11)  USAGE BINARY  VALUE 7.
117000 77  CS-OPT-COMPLETEV         PIC S9(11)  USAGE BINARY  VALUE 5.
117100 77  CS-OPT-MIDDLEV           PIC S9(11)  USAGE BINARY  VALUE 3.
117200 77  CS-OPT-TAILV             PIC S9(11)  USAGE BINARY  VALUE 1.
117300*
200000 LOCAL-STORAGE SECTION.
200100
200200 LD LD-NATL-PARAMS.
200300
200400 01  CSERRORVALUES        PIC S9(11)  USAGE BINARY.
200500
200600* The range 1000 - 1999 is reserved for category 1 error messages.
```

```
200700* These represent internal errors in CentralSupport.
200800
200900     88  CS-FILE-ACCESS-ERRORV        VALUE  1000.
201000*        An error occurred while accessing the CCSFILE or the
201100*        CONVENTIONS file.
201200
201300     88  CS-FAULTV                    VALUE  1001.
201400*        An unexpected fault occurred during procedure execution.
201500*        This error might occur if array parameters passed to the
201600*        procedure are not of the required length.
201700
201800     88  CS-SOFTERRV                  VALUE  1002.
201900*        A software error was detected.  There is an error in the
202000*        procedure implementation.
202100
202200* The range 2000 - 2999 is reserved for category 2 error messages.
202300* These represent errors in which the caller passed some sort of
202400* invalid data to a CentralSupport procedure, but the
202500* CentralSupport library was able to return some valid data.
202600
202700*    VALUE 2000 IS RESERVED
202800
202900     88  CS-LANGUAGE-NOT-FOUNDV       VALUE  2001.
203000*        The data is not in the requested language; it is in
203100*        the LANGUAGE attribute for the program task or the
203200*        SYSTEMLANGUAGE.
203300

203400     88  CS-CONVENTION-NOT-FOUNDV     VALUE  2002.
203500*        The data is not in the requested convention; it is in
203600*        the CONVENTION attribute for the program task or the
203700*        SYSTEM CONVENTION.
203800
203900     88  CS-FIELD-TRUNCATEDV          VALUE  2003.
204200
204300     88  CS-INCOMPLETE-DATAV          VALUE  2004.
204600
204610     88  CS-INCOMPLETE-CHARV          VALUE  2005.
204620*        The source data terminated without proper bracketing, in
204630*        the middle of a multibyte character, or after a single
204640*        shift character but without a data character.
204650
204700* The range 3000 - 3999 is reserved for category 3 error messages.
204800* These represent errors in which the caller passed invalid data
204900* to a CentralSupport procedure, and the CentralSupport library
205000* was unable to return any valid data.
205100
205200     88  CS-BAD-ARRAY-DESCRIPTIONV    VALUE  3000.
205300*        A parameter was incorrectly specified as less than or
205400*        equal to 0.
205500
205600     88  CS-ARRAY-TOO-SMALLV          VALUE  3001.
```

```
205700*        The array size is smaller than the length of the data
205800*        it is supposed to contain
205900
206000     88  CS-BAD-DATA-LENV             VALUE  3002.
206100*        The length is not valid or the offset + length is
206200*        greater than the size of at least one array
206300
206400     88  CS-NO-NUM-FOUNDV             VALUE  3003.
206500
206600     88  CS-NO-NAME-FOUNDV            VALUE  3004.
206700
206800     88  CS-NO-MSGNUM-FOUNDV          VALUE  3005.
206900
207000     88  CS-BAD-TYPE-CODEV            VALUE  3006.

207100
207200     88  CS-BAD-FLAGV                 VALUE  3007.
207300
207400     88  CS-BAD-TEXT-PARAMV           VALUE  3008.
207500*        For at least one text item the space allocated for
207600*        OSVs in the output or the total storage allocated in the
207700*        output is not large enough to hold the necessary OSVs
207800*        and/or PSVs.
208300
208400     88  CS-BAD-TEMPCHARV             VALUE  3011.
208500*        An invalid control character was detected in the
208600*        template.
208700
208800     88  CS-BAD-DATEINPUTV            VALUE  3012.
208900*        The date component specifies a value out of range.
209000
209100     88  CS-BAD-TIMEINPUTV            VALUE  3013.
209200*        The time component specifies a value out of range.
209300
209400     88  CS-CNV-EXISTS-ERRV           VALUE  3014.
209500*        An attempt was made to add a new convention with the name
209600*        of an existing convention.
209700
209800     88  CS-BAD-MAXDIGITSV            VALUE  3015.
209900*        The "maximum digits" value is either missing or out of
210000*        range.
210100
210200     88  CS-BAD-FRACDIGITSV           VALUE  3016.
210300*        The "fractional digits" value is either missing or out of
210400*        range.
210500
210600     88  CS-BAD-ALTFRACDIGITSV        VALUE  3017.
210700*        The "alternate fractional digits" value is either missing
210800*        or out of range.
210900
211000     88  CS-BAD-LDATETEMPV            VALUE  3018.
211100*        The long date template is either missing or contains
```

```
211200*        invalid information.
211300
211400     88  CS-BAD-SDATETEMPV            VALUE  3019.
211500*        The short date template is either missing or contains
211600*        invalid information.
211700
211800     88  CS-BAD-NDATETEMPV            VALUE  3020.
211900*        The numeric date template is either missing or it
212000*        contains invalid information.
212100
212200     88  CS-BAD-LTIMETEMPV            VALUE  3021.
212300*        The long time template is either missing or it
212400*        contains invalidate information.
212500
212600     88  CS-BAD-NTIMETEMPV            VALUE  3022.
212700*        The numeric time template is either missing or
212800*        contains invalid information.
212900
213000     88  CS-BAD-MONTEMPV              VALUE  3023.
213100*        The monetary template is either missing or it
213200*        contains invalid information.
213300
213400     88  CS-BAD-NUMTEMPV              VALUE  3024.
213500*        The numeric template is either missing or it
213600*        contains invalid information.
213700
213800     88  CS-BAD-DDMODELV              VALUE  3025.
213900*        The date display model is either missing or it
214000*        contains invalid information.
214100
214200     88  CS-BAD-TDMODELV              VALUE  3026.
214300*        The time display model is either missing or it
214400*        contains invalid information.
214500
214600     88  CS-BAD-LPPV                  VALUE  3027.
214700*        The "lines per page" value is either missing or it
214800*        is out of range.
214900
215000     88  CS-BAD-CPLV                  VALUE  3028.
215100*        The "characters per line" value is either missing or it
215200*        is out of range.
215300
215400     88  CS-REQSYMBOLV                VALUE  3029.
215500*        A required symbol in either monetary or numeric
215600*        template is missing.
215700
216000     88  CS-BAD-TEMPLENV              VALUE  3030.
218000
220000     88  CS-MUTUAL-EXCLUSIVEV         VALUE  3031.
222000
224000     88  CS-BAD-MINDIGITSV            VALUE  3032.
226000
```

```
228000     88  CS-MISSING-RBRACKETV         VALUE  3033.
230000
232000     88  CS-MISSING-TCCOLONV          VALUE  3034.
234000
236000     88  CS-BAD-INPUTVALV             VALUE  3035.
238000
240000     88  CS-CNV-NOTAVAILV             VALUE  3036.
242000
244000     88  CS-CNVFILE-NOTPRESENTV       VALUE  3037.
246000
246020     88  CS-BAD-PRECISIONV            VALUE  3038.
246040
246060     88  CS-NO-CNVNAMEV               VALUE  3039.
246080
246100     88  CS-DEL-PERMANENT-CNV-ERRV    VALUE  3040.
246102
246104     88  CS-NO-HEXCODE-DELIMV         VALUE  3041.
246106
246108     88  CS-BAD-HEXCODEV              VALUE  3042.
246110
246112     88  CS-NO-ALTCURR-DELIMV         VALUE  3043.
246120
246200* The range 4000 - 4999 is reserved for category 4 error messages.
246400* These are messages in which the caller passed some sort of data
246600* for which the CentralSupport library could find no return
246800* information.  CentralSupport completed the request, but no data
247000* was returned.
247200
247400     88  CS-DATA-NOT-FOUNDV           VALUE  4002.
247600
247800     88  CS-NO-MATCH-FOUNDV           VALUE  4003.
248000*        No match could be found in the code format specified for
248200*        the character set or ccsversion number.
248400
248500     88  CS-COMPLEX-TRANS-REQV        VALUE  4004.
248600*        The standard ALGOL type translate table mapping is not
248700*        available, but CCSTOCCS-TRANS-TEXT-COMPLEX will map
248800*        the data.
248900
249990     88  CS-LAST-MSGV                 VALUE  9999.
250000
250100 77  DATETIMETYPE         PIC S9(11)  USAGE BINARY.
250200 77  AMT                  REAL.
250250 77  DAMT                 PIC S9(23)  USAGE BINARY.
250260 77  PRECISION            PIC S9(11)  USAGE BINARY.
250300 77  CCSFROM              PIC S9(11)  USAGE BINARY.
250400 77  CCSTO                PIC S9(11)  USAGE BINARY.
250500 77  CCSVSNCODE           PIC S9(11)  USAGE BINARY.
250600 77  COMPARE-LEN          PIC S9(11)  USAGE BINARY.
250700 77  DEST-START           PIC S9(11)  USAGE BINARY.
250800 77  FLAG                 PIC S9(11)  USAGE BINARY.
250900 77  INSPECT-LEN          PIC S9(11)  USAGE BINARY.
```

```
251000 77  NUM                  PIC S9(11)  USAGE BINARY.
251100 77  RSLT                 PIC S9(11)  USAGE BINARY.
251200 77  SOURCE-START         PIC S9(11)  USAGE BINARY.
251300 77  TEXT-START           PIC S9(11)  USAGE BINARY.
251400 77  TEXT1-START          PIC S9(11)  USAGE BINARY.
251500 77  TEXT2-START          PIC S9(11)  USAGE BINARY.
251600 77  TOTAL                PIC S9(11)  USAGE BINARY.
251700 77  TRANS-LEN            PIC S9(11)  USAGE BINARY.
251800 77  TSETTYPE             PIC S9(11)  USAGE BINARY.
251900 77  TTABLETYPE           PIC S9(11)  USAGE BINARY.
252000 77  VSNNUM               PIC S9(11)  USAGE BINARY.
252100 77  SCANNED-CHARS        PIC S9(11)  USAGE BINARY.
252200 77  RLTN                 PIC S9(11)  USAGE BINARY.
252300 77  ORD-TYPE             PIC S9(11)  USAGE BINARY.
252400 77  TEMPLATE-TYPE        PIC S9(11)  USAGE BINARY.
252500 77  LINES-PER-PAGE       PIC S9(11)  USAGE BINARY.
252600 77  CHARACTERS-PER-LINE  PIC S9(11)  USAGE BINARY.
252700 77  DISPLAYMODEL-TYPE    PIC S9(11)  USAGE BINARY.
252800 77  DATE-TEMPLATE-LEN    PIC S9(11)  USAGE BINARY.
252900 77  MSG-LEN              PIC S9(11)  USAGE BINARY.
253000 77  TOTAL-STORAGE        PIC S9(11)  USAGE BINARY.
253100 77  ITEXT-LEN            PIC S9(11)  USAGE BINARY.
253200 77  MAX-OSVS             PIC S9(11)  USAGE BINARY.
253300
253400 01  CNVNAME           PIC X(17).
253500 01  SYSINFO           PIC X(51).
253600 01  CONTROLINFO-ARRAY    USAGE BINARY.
253700     05  CONTROLINFO-ELEMENT PIC S9(11)  OCCURS 8 TIMES.
253800 01  DATETIME          PIC X(100).
253900 01  DATE-ARRAY        PIC X(100).
254000 01  TIME-ARRAY        PIC X(100).
254100 01  DEST-TEXT         PIC X(100).
254200 01  LANGNAME          PIC X(17).
254300 01  NAME              PIC X(17).
254400 01  NAMES-ARRAY.
254500     05  NAMES-ARRAY-ELEMENT PIC X(18)
254600                                    OCCURS 40 TIMES.
254700 01  NUMS-ARRAY           USAGE BINARY.
254800     05  NUMS-ARRAY-ELEMENT  PIC S9(11) OCCURS 40 TIMES.
254900 01  FORMATTED-AMT        PIC X(100).
255000 01  SOURCE-TEXT          PIC X(100).
255100 01  MSG                  PIC X(100).
255200 01  TEXT0                PIC X(100).
255300 01  TEXT1                PIC X(100).
255400 01  TEXT2                PIC X(100).
255500 01  TEMPLATE             PIC X(100).
255600 01  SYMBOLS              PIC X(100).
255700 01  SYMLEN-ARRAY         USAGE BINARY.
255800     05  SYMLEN-ELEMENT   PIC S9(11)  OCCURS 256 TIMES.
255900 01  DISPLAYMODEL         PIC X(100).
256000 01  FORMATTED-DATE       PIC X(100).
256100 01  FORMATTED-TIME       PIC X(100).
```

```
256200
256300 LD LD-CCSINFO.
256400 77  LD-CCSINFO-CCS-NUM          PIC S9(11)   BINARY CONTENT.
256500 01  LD-CCSINFO-ARY              REAL.
256600     05  LD-CCSINFO-ARY-WRD       REAL         OCCURS 30 TIMES.
256700 77  LD-CCSINFO-RSLT             PIC S9(11)   BINARY.
256800
256900 LD LD-CCSTOCCS-TRANS-TEXT-COMPLEX.
257000 77  LD-COMPLEX-CCS-FROM         PIC S9(11)   BINARY CONTENT.
257100 77  LD-COMPLEX-CCS-TO           PIC S9(11)   BINARY CONTENT.
257200 01  LD-COMPLEX-SOURCE-TEXT      PIC X(100).
257300 77  LD-COMPLEX-SOURCE-START     PIC S9(11)   BINARY REFERENCE.
257400 77  LD-COMPLEX-SOURCE-BYTES     PIC S9(11)   BINARY CONTENT.
257500 01  LD-COMPLEX-DEST-TEXT        PIC X(100).
257600 77  LD-COMPLEX-DEST-START       PIC S9(11)   BINARY REFERENCE.
257700 77  LD-COMPLEX-DEST-BYTES       PIC S9(11)   BINARY CONTENT.
257800 01  LD-COMPLEX-STATE            REAL.
257900     05 LD-COMPLEX-STATE-WRD     REAL         OCCURS 10 TIMES.
258000 77  LD-COMPLEX-OPTION           PIC S9(11)   BINARY CONTENT.
258100 77  LD-COMPLEX-RSLT             PIC S9(11)   BINARY.
258200
500000*
500100* CentralSupport procedure declarations
500200*
500300 PROGRAM-LIBRARY SECTION.
500400 LB CENTRALSUPPORT IMPORT
500500    ATTRIBUTE
500600       FUNCTIONNAME IS "CENTRALSUPPORT"
500700       LIBACCESS IS BYFUNCTION.
500800*
500900    ENTRY PROCEDURE GET-CS-MSG
501000       FOR "GET_CS_MSG"
501100       WITH LD-NATL-PARAMS
501200       USING  NUM,
501300              LANGNAME,
501400              MSG,
501500              MSG-LEN
501600       GIVING RSLT.
501700*
501800    ENTRY PROCEDURE CENTRALSTATUS
501900       WITH LD-NATL-PARAMS
502000       USING  SYSINFO,
502100              CONTROLINFO-ARRAY
502200       GIVING RSLT.
502300*
502400    ENTRY PROCEDURE VALIDATE-NAME-RETURN-NUM
502500       FOR "VALIDATE_NAME_RETURN_NUM"
502600       WITH LD-NATL-PARAMS
502700       USING  CCSVSNCODE,
502800              NAME,
502900              NUM
503000       GIVING RSLT.
```

```
503100*
503200    ENTRY PROCEDURE VALIDATE-NUM-RETURN-NAME
503300       FOR "VALIDATE_NUM_RETURN_NAME"
503400       WITH LD-NATL-PARAMS
503500       USING  CCSVSNCODE,
503600              NUM,
503700              NAME
503800       GIVING RSLT.
503900*
504000    ENTRY PROCEDURE CCSVSN-NAMES-NUMS
504100       FOR "CCSVSN_NAMES_NUMS"
504200       WITH LD-NATL-PARAMS
504300       USING  CCSVSNCODE,
504400              TOTAL,
504500              NAMES-ARRAY,
504600              NUMS-ARRAY
504700       GIVING RSLT.
504800*
504900    ENTRY PROCEDURE CCSTOCCS-TRANS-TEXT
505000       FOR "CCSTOCCS_TRANS_TEXT"
505100       WITH LD-NATL-PARAMS
505200       USING  CCSFROM,
505300              CCSTO,
505400              SOURCE-TEXT,
505500              SOURCE-START,
505600              DEST-TEXT,
505700              DEST-START,
505800              TRANS-LEN
505900       GIVING RSLT.
506000*
506100    ENTRY PROCEDURE VSNTRANS-TEXT
506200       FOR "VSNTRANS_TEXT"
506300       WITH LD-NATL-PARAMS
506400       USING  VSNNUM,
506500              SOURCE-TEXT,
506600              SOURCE-START,
506700              DEST-TEXT,
506800              DEST-START,
506900              TRANS-LEN,
507000              TTABLETYPE
507100       GIVING RSLT.
507200*
507300    ENTRY PROCEDURE VSNINSPECT-TEXT
507400       FOR "VSNINSPECT_TEXT"
507500       WITH LD-NATL-PARAMS
507600       USING  VSNNUM,
507700              TEXTO,
507800              TEXT-START,
507900              INSPECT-LEN,
508000              TSETTYPE,
508100              FLAG,
508200              SCANNED-CHARS
```

```
508300       GIVING RSLT.
508400*
508500    ENTRY PROCEDURE VSNCOMPARE-TEXT
508600       FOR "VSNCOMPARE_TEXT"
508700       WITH LD-NATL-PARAMS
508800       USING  VSNNUM,
508900              TEXT1,
509000              TEXT1-START,
509100              TEXT2,
509200              TEXT2-START,
509300              COMPARE-LEN,
509400              RLTN,
509500              ORD-TYPE
509600       GIVING RSLT.
509700*
509800    ENTRY PROCEDURE VSNGETORDERINGFOR-ONE-TEXT
509900       FOR "VSNGETORDERINGFOR_ONE_TEXT"
510000       WITH LD-NATL-PARAMS
510100       USING  VSNNUM,
510200              SOURCE-TEXT,
510300              SOURCE-START,
510400              ITEXT-LEN,
510500              DEST-TEXT,
510600              DEST-START,
510700              MAX-OSVS,
510800              TOTAL-STORAGE,
510900              ORD-TYPE
511000       GIVING RSLT.
511100
511200    ENTRY PROCEDURE VSNESCAPEMENT
511300       WITH LD-NATL-PARAMS
511400       USING  VSNNUM,
511500              SOURCE-TEXT,
511600              SOURCE-START,
511700              DEST-TEXT,
511800              TRANS-LEN
511900       GIVING RSLT.
512000*
512100    ENTRY PROCEDURE CCSTOCCS-TRANS-TEXT-COMPLEX
512200       FOR "CCSTOCCS_TRANS_TEXT_COMPLEX"
512300       WITH LD-CCSTOCCS-TRANS-TEXT-COMPLEX
512400       USING
512500          LD-COMPLEX-CCS-FROM,
512600          LD-COMPLEX-CCS-TO,
512700          LD-COMPLEX-SOURCE-TEXT,
512800          LD-COMPLEX-SOURCE-START,
512900          LD-COMPLEX-SOURCE-BYTES,
513000          LD-COMPLEX-DEST-TEXT,
513100          LD-COMPLEX-DEST-START,
513200          LD-COMPLEX-DEST-BYTES,
513300          LD-COMPLEX-STATE,
513400          LD-COMPLEX-OPTION
```

```
513500       GIVING LD-COMPLEX-RSLT.
513600*
513700    ENTRY PROCEDURE CCSINFO
513800       FOR "CCSINFO"
513900       WITH LD-CCSINFO
514000       USING
514100          LD-CCSINFO-CCS-NUM,
514200          LD-CCSINFO-ARY
514300       GIVING LD-CCSINFO-RSLT.
514400
600000*
600100* CentralSupport Conventions procedures
600200*
600300    ENTRY PROCEDURE CNV-NAMES
600400       FOR "CNV_NAMES"
600500       WITH LD-NATL-PARAMS
600600       USING  TOTAL,
600700              NAMES-ARRAY
600800       GIVING RSLT.
600900*
601000    ENTRY PROCEDURE CNV-TEMPLATE
601100       FOR "CNV_TEMPLATE_COB"
601200       WITH LD-NATL-PARAMS
601300       USING  TEMPLATE-TYPE,
601400              CNVNAME,
601500              TEMPLATE
601600       GIVING RSLT.
601700*
601800    ENTRY PROCEDURE CNV-FORMSIZE
601900       FOR "CNV_FORMSIZE"
602000       WITH LD-NATL-PARAMS
602100       USING  CNVNAME,
602200              LINES-PER-PAGE,
602300              CHARACTERS-PER-LINE,
602400       GIVING RSLT.
602500*
602600    ENTRY PROCEDURE CNV-SYMBOLS
602700       FOR "CNV_SYMBOLS"
602800       WITH LD-NATL-PARAMS
602900       USING  CNVNAME,
603000              TOTAL,
603100              SYMLEN-ARRAY,
603200              SYMBOLS
603300       GIVING RSLT.
603400*
603500    ENTRY PROCEDURE CNV-DISPLAYMODEL
603600       FOR "CNV_DISPLAYMODEL_COB"
603700       WITH LD-NATL-PARAMS
603800       USING  DISPLAYMODEL-TYPE,
603900              CNVNAME,
604000              LANGNAME,
604100              DISPLAYMODEL
```

```
604200       GIVING RSLT.
604300*
604400    ENTRY PROCEDURE CNV-SYSTEMDATETIME
604500       FOR "CNV_SYSTEMDATETIME_COB"
604600       WITH LD-NATL-PARAMS
604700       USING  DATETIMETYPE,
604800              CNVNAME,
604900              LANGNAME,
605000              DATETIME
605100       GIVING RSLT.
605200*
605300    ENTRY PROCEDURE CNV-SYSTEMDATETIMETMP
605400       FOR "CNV_SYSTEMDATETIMETMP_COB"
605500       WITH LD-NATL-PARAMS
605600       USING  TEMPLATE,
605700              LANGNAME,
605800              DATE-TEMPLATE-LEN,
605900              DATETIME
606000       GIVING RSLT.
606100*
606200    ENTRY PROCEDURE CNV-FORMATDATE
606300       FOR "CNV_FORMATDATE_COB"
606400       WITH LD-NATL-PARAMS
606500       USING  DATETIMETYPE,
606600              DATE-ARRAY,
606700              CNVNAME,
606800              LANGNAME,
606900              FORMATTED-DATE
607000       GIVING RSLT.
607100*
607200    ENTRY PROCEDURE CNV-FORMATDATETMP
607300       FOR "CNV_FORMATDATETMP_COB"
607400       WITH LD-NATL-PARAMS
607500       USING  DATE-ARRAY,
607600              TEMPLATE,
607700              LANGNAME,
607800              FORMATTED-DATE
607900       GIVING RSLT.
608000*
608100    ENTRY PROCEDURE CNV-FORMATTIME
608200       FOR "CNV_FORMATTIME_COB"
608300       WITH LD-NATL-PARAMS
608400       USING  DATETIMETYPE,
608500              TIME-ARRAY,
608600              CNVNAME,
608700              LANGNAME,
608800              FORMATTED-TIME
608900       GIVING RSLT.
609000*
609100    ENTRY PROCEDURE CNV-FORMATTIMETMP
609200       FOR "CNV_FORMATTIMETMP_COB"
609300       WITH LD-NATL-PARAMS
```

```
609400       USING  TIME-ARRAY,
609500              TEMPLATE,
609600              LANGNAME,
609700              FORMATTED-TIME
609800       GIVING RSLT.
609900*
610000    ENTRY PROCEDURE CNV-CURRENCYEDIT-DOUBLE
610100       FOR "CNV_CURRENCYEDIT_DOUBLE_COB"
610200       WITH LD-NATL-PARAMS
610300       USING  DAMT,
610400              PRECISION,
610500              CNVNAME,
610600              FORMATTED-AMT
610700       GIVING RSLT.
610800*
610900    ENTRY PROCEDURE CNV-CURRENCYEDITTMP-DOUBLE
611000       FOR "CNV_CURRENCYEDITTMP_DOUBLE_COB"
611100       WITH LD-NATL-PARAMS
611200       USING  DAMT,
611300              PRECISION,
611400              TEMPLATE,
611500              CNVNAME,
611600              FORMATTED-AMT
611700       GIVING RSLT.
611800*
611900    ENTRY PROCEDURE CNV-VALIDATENAME
612000       FOR "CNV_VALIDATENAME"
612100       WITH LD-NATL-PARAMS
612200       USING  CNVNAME
612300       GIVING RSLT.
```

Example of Calling Procedures in the CENTRALSUPPORT Library

Figure 16–32 shows how you can use explicit library calls to access procedures in the CENTRALSUPPORT library.

```
  000100   IDENTIFICATION DIVISION.
  000200   PROGRAM-ID. EXPLICIT-CENTRALSUPPORT-EXMPL.
  000300   ENVIRONMENT DIVISION.
  000400   INPUT-OUTPUT SECTION.
  000500   FILE-CONTROL.
  000600       SELECT RESULTS-FILE           ASSIGN TO DISK.
  000700   DATA DIVISION.
  000800   FILE SECTION.
  000900   FD  RESULTS-FILE.
  001000   01  RESULTS-RCD                   PIC X(80).
  001100   WORKING-STORAGE SECTION.
  001200   77  SUB                           PIC S9(11) USAGE BINARY.
  001300   77  RESULT                        PIC S9(11) USAGE BINARY.
  001400   77  RSLT-TOTAL                    PIC S9(11) USAGE BINARY.
  001500   77  CS-MSG-LEN                    PIC S9(11) USAGE BINARY.
  001600   77  CS-MSG-NBR                    PIC S9(11) USAGE BINARY.
  001700   77  CNV-NAME                      PIC S9(17).
  001800   77  LANG-NAME                     PIC S9(17).
  001900   77  DATE-TIME                     PIC S9(80).
  002000   01  NAMES-ARY.
  002100       05  NAMES-ENTRY               PIC X(17) OCCURS 80 TIMES.
  002200   01  CS-MSG-ARY.
  002300       05  CS-MSG                    PIC X(80) OCCURS 2 TIMES.
  002400   01  RSLT-RCD1.
  002500       05  RR1-ALPHA                 PIC X(20).
  002600       05  FILLER                    PIC X(60).
  002700   01  RSLT-RCD2 REDEFINES RSLT-RCD1.
  002800       05  RR2-NAME                  PIC X(13).
  002900       05  RR2-VALUE                 PIC -Z(11).
  003000       05  FILLER                    PIC X(55).
  003100***  CentralSupport Values ***
  003200       COPY "SYMBOL/INTL/COBOL85/PROPERTIES"
  003300           FROM 101400 THRU 101500.
1:101400   77  CD-DATAOKV                    PIC S9(11) USAGE BINARY  VALUE
1.
1:101500*    No errors; TRUE.
  003400       COPY "SYMBOL/INTL/COBOL85/PROPERTIES"
  003500           FROM 111600 THRU 111700.
1:111600   77  CS-LDATELTIMEV                PIC S9(11) USAGE BINARY  VALUE
5.
1:111700*    Long date and long time
  003600   LOCAL-STORAGE SECTION.
  003700       COPY "SYMBOL/INTL/COBOL85/PROPERTIES"
  003800           FROM 200200 THRU 200200.
```

```
1:200200   LD  LD-NATL-PARAMS.
  003900*
  004000**** For convenience, when calling several CENTRALSUPPORT    ****
  004100**** procedures, include the sequence number range of 250000 ****
  004200**** through 299999.                                        ****
  004300*
  004400       COPY "SYMBOL/INTL/COBOL85/PROPERTIES"
  004500            FROM 250100 THRU 250100.
1:250100   77  DATETIMETYPE                  PIC S9(11) USAGE BINARY.
  004600       COPY "SYMBOL/INTL/COBOL85/PROPERTIES"
  004700            FROM 251000 THRU 251100.
1:251000   77  NUM                           PIC S9(11) USAGE BINARY.
1:251100   77  RSLT                          PIC S9(11) USAGE BINARY.
  004800       COPY "SYMBOL/INTL/COBOL85/PROPERTIES"
  004900            FROM 251600 THRU 251600.
1:251600   77  TOTAL                         PIC S9(11) USAGE BINARY.
  005000       COPY "SYMBOL/INTL/COBOL85/PROPERTIES"
  005100            FROM 252700 THRU 252900.
1:252700   77  DISPLAYMODEL-TYPE             PIC S9(11) USAGE BINARY.
1:252800   77  DATE-TEMPLATE-LEN             PIC S9(11) USAGE BINARY.
1:252900   77  MSG-LEN                       PIC S9(11) USAGE BINARY.
  005200       COPY "SYMBOL/INTL/COBOL85/PROPERTIES"
  005300            FROM 253400 THRU 253400.
1:253400   01  CNVNAME                       PIC X(17).
  005400       COPY "SYMBOL/INTL/COBOL85/PROPERTIES"
  005500            FROM 253800 THRU 253800.
1:253800   01  DATETIME                      PIC X(100).
  005600       COPY "SYMBOL/INTL/COBOL85/PROPERTIES"
  005700            FROM 254200 THRU 254600.
1:254200   01  LANGNAME                      PIC X(17).
1:254300   01  NAME                          PIC X(17).
1:254400   01  NAMES-ARRAY.
1:254500       05  NAMES-ARRAY-ELEMENT       PIC X(18)
1:254600                                              OCCURS 40 TIMES.
  005800       COPY "SYMBOL/INTL/COBOL85/PROPERTIES"
  005900            FROM 255100 THRU 255100.
1:255100   01  MSG                           PIC X(100).
  006000       COPY "SYMBOL/INTL/COBOL85/PROPERTIES"
  006100           FROM 500000 THRU 501700.
1:500000*
1:500100* CentralSupport procedure declarations
1:500200*
1:500300   PROGRAM-LIBRARY SECTION.
1:500400   LB CENTRALSUPPORT IMPORT
1:500500      ATTRIBUTE
1:500600         FUNCTIONNAME IS "CENTRALSUPPORT"
1:500700         LIBACCESS IS BYFUNCTION.
1:500800*
1:500900      ENTRY PROCEDURE GET-CS-MSG
1:501000         FOR "GET_CS_MSG"
1:501100         WITH LD-NATL-PARAMS
1:501200         USING NUM,
```

```
1:501300              LANGNAME,
1:501400              MSG,
1:501500              MSG-LEN,
1:501600         GIVING RSLT.
1:501700*
  006200         COPY "SYMBOL/INTL/COBOL85/PROPERTIES"
  006300           FROM 600300 THRU 600900.
1:600300      ENTRY PROCEDURE CNV-NAMES
1:600400         FOR "CNV_NAMES"
1:600500         WITH LD-NATL-PARAMS
1:600600         USING TOTAL,
1:600700               NAMES-ARRAY
1:600800         GIVING RSLT.
1:600900*
  006400       COPY "SYMBOL/INTL/COBOL85/PROPERTIES"
  006500           FROM 604400 THRU 605200.
1:604400     ENTRY PROCEDURE CNV-SYSTEMDATETIME
1:604500        FOR "CNV_SYSTEMDATETIME_COB"
1:604600        WITH LD-NATL-PARAMS
1:604700        USING DATETIMETYPE,
1:604800              CNVNAME,
1:604900              LANGNAME,
1:605000              DATETIME
1:605100        GIVING RSLT.
1:605200*
  006600   PROCEDURE DIVISION.
  006700   MAIN-PARA.
  006800       OPEN OUTPUT RESULTS-FILE.
  006900       PERFORM LONG-DATE-TIME.
  007000       PERFORM CONVENTION-NAMES.
  007100       CLOSE RESULTS-FILE WITH SAVE.
  007200       STOP RUN.
  007300   LONG-DATE-TIME.
  007400       CALL CNV-SYSTEMDATETIME OF CENTRALSUPPORT
  007500         USING CS-LDATELTIMEV,
  007600               CNV-NAME,
  007700               LANG-NAME,
  007800               DATE-TIME
  007900         GIVING RESULT.
  008000       IF RESULT = CS-DATAOKV THEN
  008100           MOVE SPACES TO RSLT-RCD1
  008200           MOVE DATE-TIME TO RSLT-RCD1
  008300           WRITE RESULTS-RCD FROM RSLT-RCD1
  008400           MOVE SPACES TO RESULTS-RCD
  008500           WRITE RESULTS-RCD
  008600     ELSE
  008700         PERFORM CS-ERROR-MSG.
  008800   CONVENTION-NAMES.
  008900       CALL CNV-NAMES OF CENTRALSUPPORT
  009000         USING RSLT-TOTAL,
  009100               NAMES-ARY
  009200         GIVING RESULT.
```

```
009300       IF RESULT = CS-DATAOKV THEN
009400           MOVE SPACES TO RSLT-RCD1
009500           MOVE "Convention Names" TO RR1-ALPHA
009600           WRITE RESULTS-RCD FROM RSLT-RCD1
009700       MOVE "----------" TO RR1-ALPHA
009800           WRITE RESULTS-RCD FROM RSLT-RCD1
009900           MOVE 1 TO SUB
010000           PERFORM DISPLAY-NAMES UNTIL SUB > RSLT-TOTAL
010100       ELSE
010200           PERFORM CS-ERROR-MSG.
010300   DISPLAY-NAMES.
010400       MOVE SPACES TO RSLT-RCD1.
010500       MOVE NAMES-ENTRY (SUB) TO RR1-ALPHA.
010600       WRITE RESULTS-RCD FROM RSLT-RCD1.
010700       ADD 1 TO SUB.
010800   CS-ERROR-MSG.
010900       MOVE SPACES TO RSLT-RCD2.
011000       MOVE "BAD RESULT = " TO RR2-NAME.
011100       MOVE RESULT TO RR2-VALUE.
011200       WRITE RESULTS-RCD FROM RSLT-RCD2.
011300       MOVE RESULT TO CS-MSG-NBR.
011400       MOVE SPACES TO CS-MSG-ARY.
011500       CALL GET-CS-MSG OF CENTRALSUPPORT
011600           USING CS-MSG-NBR,
011700            LANG-NAME,
011800                 CS-MSG-ARY,
011900            CS-MSG-LEN
012000           GIVING RESULT.
012100       IF RESULT NOT = CS-DATAOKV THEN
012200           MOVE RESULT TO RR2-VALUE
012300           WRITE RESULTS-RCD FROM RSLT-RCD2
012400       ELSE
012500           MOVE CS-MSG (1) TO RSLT-RCD1
012600           WRITE RESULTS-RCD FROM RSLT-RCD1
012700           IF CS-MSG (2) NOT = SPACES THEN
012800               MOVE CS-MSG (2) TO RSLT-RCD1
012900               WRITE RESULTS-RCD FROM RSLT-RCD1.
013000       MOVE SPACES TO RESULTS-RCD.
013100       WRITE RESULTS-RCD.
```

Figure 16–32. Calling Procedures in the CENTRALSUPPORT Library

Output

The output from Figure 16–32 is as follows:

```
Friday, May 20, 1994 11:19:46.0000

Convention Names
---------------
ASERIESNATIVE
Netherlands
Denmark
UnitedKingdom1
StandardTurkey
Norway
Sweden
Greece
FranceListing
FranceBureautique
EuropeanStandard
Belgium
Spain
Switzerland
Zimbabwe
Italy
UnitedKingdom2
KENYA
NIGERIA
SOUTHAFRICA
CYRILLIC
BRAZIL
NEWZEALAND
YUGOSLAVIA
FRENCHCANADA
ARGENTINA
CHILE
COLOMBIA
COSTARICA
MEXICO
PERU
VENEZUELA
AUSTRALIA
EGYPT
ENGLISHCANADA
Japan1
Japan2
CHINA
HONGKONG
MALAYSIA
PHILIPPINES
TAIWAN
CZECHOSLOVAKIA
UAE
```

```
HUNGARY
JORDAN
KUWAIT
LEBANON
CROATIA
POLANDCROATIA
POLAND
```

Appendix A
Output Messages

Note: *This appendix provides the normal and abnormal compiler output messages, and the run-time output messages generated by the COBOL85 compiler. Although this section is not highlighted, all of the information is relative to Unisys extensions to COBOL ANSI-85.*

Normal Compiler Output Messages

The normal compiler output messages listed in this section are divided into two categories:

- Numerical compiler output messages
- Non-numerical compiler output messages

Numerical Compiler Output Messages

The following list arranges the output messages in numerical order based on the message number that is displayed immediately preceding the output message. Each output message is accompanied by a brief explanation of its meaning.

001 : THIS INDEX IS EITHER NOT ASSOCIATED WITH THIS TABLE OR IS ASSOCIATED WITH A DIFFERENT DIMENSION OF THIS TABLE

This index is not defined to be associated with this table or is defined to be associated with a different dimension of this table.

002 : FILE IS INVALID FOR USE WITH BUFFERSHARING. FILE MUST BE AN ORGANIZATION SEQ, RANDOM ACCESS DISK FILE

A file used for buffersharing has an organization other than sequential and access other than random.

003 : A DATA ITEM AND ITS REDEFINITION MUST START ON THE SAME ALIGNMENT BOUNDARY

A data item starts at an odd digit offset and its redefinition requires a character alignment, or a data item starts at a character offset and its redefinition requires a word or double alignment

004 : ALPHABET NAME REQUIRED

An alphabet name is required in the current context.

005 : ONLY A DISPLAY ITEM OR LITERAL OF 1 TO 6 CHARACTERS IS ALLOWED

This error occurs when the size of the SERIALNO attribute value specified in a CHANGE or FD statement exceeds six characters.

006 : ALPHANUMERIC DATA ITEM REQUIRED

An alphanumeric data item is required in the current context.

007 : ALPHANUMERIC OPERAND EXPECTED

The operand in your syntax was not of the alphanumeric class. To belong to the alphanumeric class, the operand must be a data item of the alphanumeric, alphanumeric-edited, or numeric-edited category. The category of a data item is declared in the PICTURE clause. Refer to Section 4 for details.

008 : ALPHANUMERIC SENDING FIELD REQUIRED

An alphanumeric sending field is required in the current context.

010 : ALTER VERB INCORRECT USAGE

An attempt was made to use the ALTER statement in an incorrect manner or an invalid context.

011 : UNABLE TO CHANGE THE PARAMETER DEFAULT FOR BOUND PROCEDURE, MUST SPECIFY BY CONTENT OR BY REFERENCE ON FORMAL PARAMETER

The program has altered the expected default value of one or more formal parameters. The program must explicitly specify either BY CONTENT or BY REFERENCE in the program.

012 : ARITHMETIC EXPRESSION EXPECTED

An arithmetic expression was expected in the syntax. For the syntax and rules for creating arithmetic expressions, refer to Section 5.

018 : THE ITEMS, ELEMENTARY ITEMS, GROUPS AND GROUP ITEMS DISPLAY OPTIONS MAY ONLY BE APPLIED TO IDENTIFIERS

Declaration information was requested for a display item that is not a declared data-item.

The purpose of the ITEMS, GROUPS, ELEMENTARY ITEMS, and GROUP ITEMS display options is to enable you to request declaration information about a data-item and its subordinate items. Because not all display items are declared, you cannot get declaration information for all display items. For example, figurative constants and literals are not declared data items. This is a TADS error message.

019 : INVALID DMSTATUS SPECIFIED

The attribute name specified in the DMSTATUS entry is invalid.

025 : AN ATTRIBUTE MNEMONIC VALUE MUST ONLY BE COMPARED TO AN ATTRIBUTE-NAME

Use of an attribute mnemonic value independent of an attribute-name is not allowed.

028 : BLOCK SIZE MUST BE MULTIPLE OF RECORD SIZE

The BLOCKSIZE attribute of a file must be a multiple of the MAXRECSIZE attribute of the file.

035 : OCCURS DEPENDING CLAUSE CAN BE FOLLOWED ONLY BY SUBORDINATE ITEMS

A data item with an OCCURS DEPENDING clause can be followed, within that record description only, by data description entries that are subordinate to it.

036 : CANNOT BE VARIABLE SIZE

An attempt was made to use a variable-size data item in an invalid context.

037 : CANNOT CHAIN REDEFINES

An attempt was made to redefine a data item by reference to another redefined data item.

038 : THE BLOCK SIZE FOR A FILE MUST NOT EXCEED 65,535

Reduce the block size of the subject file to 65,535 bytes or less and recompile the program.

039 : CANNOT CONTAIN VARIABLE OCCURRENCE

You specified the OCCURS clause for a data item that cannot have an OCCURS clause. The OCCURS clause cannot appear in a data description entry that has either of the following:

- A level-number of 01, 66, 77, or 88
- A variable-occurrence data item subordinate to the entry

040 : CURRENCYSIGN COMPILER CONTROL OPTION IGNORED

042 : A DOUBLE-WORD DATA ITEM MAY NOT BE USED AS THE INTERNAL LENGTH FIELD OF A VARIABLE LENGTH RECORD

When you use the RECORD...VARYING...DEPENDING form of the record clause to provide for variable-length records in a file, the data name of the DEPENDING phrase cannot be a double-word field that is defined in the record description of the file.

043 : ATTRIBUTES OF THIS TYPE CAN NOT BE DISPLAYED

Some file, library, and task attributes cannot be displayed due to the type of the attribute.

044 : THIS CONSTRUCT IS NOT ALLOWED IN A DEFINITION PROGRAM

045 : CANNOT START IN MARGIN A

This item cannot begin in area A (columns 8 through 11) of the source program. Only the following items can begin in area A:

- Division, section, and paragraph headers
- The keywords DECLARATIVES and END DECLARATIVES
- The level indicators FD and SD
- The level numbers 01 and 77

You must place sequence numbers in columns 1 through 6. Line indicator symbols and compiler control options belong in column 7. All other items must begin in columns 12 through 72 (area B).

046 : EXCEPTION/NOT EXCEPTION CLAUSE NOT ALLOWED WITH THIS TYPE OF CALL

The EXCEPTION/NOT EXCEPTION clause is permitted only with the Format 2 CALL statement.

If the CALL statement is nested and the EXCEPTION/NOT EXCEPTION clause is intended for the statement that the CALL is nested within, then add an END-CALL scope delimiter.

047: OVERFLOW CLAUSE NOT ALLOWED WITH THIS TYPE OF CALL

The OVERFLOW clause is permitted only with the Format 1 CALL statement.

If the CALL statement is nested and the OVERFLOW clause is intended for the statement that the call is nested within, then add an END-CALL scope delimiter.

049 : CLAUSE MAY NOT BE USED WITH INDEXED FILE

An invalid file description entry clause was encountered in the file description entry for an indexed file.

050 : CLAUSE COMBINATIONS NOT PERMISSIBLE IN FORMAT 3 ENTRIES

Clause combinations are not permissible in Format 3 Report-Group Description Entries. Refer to Table 12-2 for more information.

051 : CLOSE OPTION NOT MEANINGFUL FOR FILE KIND

052 : THIS PROGRAM MUST BE A DEFINITION PROGRAM

053 : BINDSTREAM OPTION MUST BE SET

054 : COLON EXPECTED

A colon character was expected, but was not found.

055 : THIS CAN NOT BE A DEFINITION PROGRAM

056 : NESTED PROGRAM NOT ALLOWED IN A BINDSTREAM

057 : COMP OR DISPLAY EXPECTED

In this context, the data description entry must include a USAGE IS COMP or USAGE IS DISPLAY clause.

060 : COMPILATION TERMINATED

The compilation of the program is terminated as a result of a previous error.

061 : COMPILER ERROR

063 : NO TWO OCCURENCES OF CURRENCY PICTURE SYMBOL MAY BE THE SAME VALUE

064 : CONDITION NAME EXPECTED

065 : CONFLICTING CLASS

The use of a data item conflicts with its declared class.

066 : CONSTRUCT IGNORED

067 : CONSTRUCT NOT IMPLEMENTED

068 : THE PROGRAM BEING CALLED IS NOT IN SCOPE

Refer to "Nested Source Programs" in Section 10 for more information on rules for nested program calls.

069 : THE PROGRAM BEING CALLED WAS NOT FOUND

070 : TOO MANY CALLS WITH $CALLNESTED IN THIS PROGRAM

The table limit has been exceeded for the table containing the nested cells.

071 : $CALLNESTED IS NOT ALLOWED IN SUB PROGRAMS

The dollar option CALLNESTED is ignored for programs compiled at lex level 3 or greater.

072 : DATA BASE ATTRIBUTE EXPECTED

073 : CURRENCY PICTURE SYMBOL LIMITED TO ONE CHARACTER

074 : DATA BASE OPERAND EXPECTED

075 : INVALID CURRENCY STRING

076 : INVALID CURRENCY PICTURE SYMBOL

077 : DATA NAME EXPECTED

078 : CANNOT SPECIFY ALPHANUMERIC COLLATING SEQUENCE NAME

The alphanumeric alphabet name in the PROGRAM COLLATING SEQUENCE clause cannot be specified as a national alphabet name in the ALPHABET... FOR NATIONAL clause.

079 : DATA SET INVALID WITH KEY CONDITION SPECIFIED

081 : CANNOT SPECIFY NATIONAL COLLATING SEQUENCE NAME

The national alphabet name in the PROGRAM COLLATING SEQUENCE clause cannot be specified as an alphanumeric alphabet name in the ALPHABET... FOR ALPHANUMERIC clause.

082 : MUST BE BOOLEAN DATA ITEM OR BOOLEAN EXPRESSION

083 : SYMBOL MAY CONFLICT WITH FREE OPTION

084 : BOOLEAN DATA ITEMS OF LENGTH GREATER THAN ONE NOT IMPLEMENTED

085 : BOOLEAN DATA ITEMS OF USAGE DISPLAY OR NATIONAL NOT IMPLEMENTED

086 : DECLARED AS RECEIVED BUT NOT IN PARAMETER LIST

089 : DELIMITED BY EXPECTED

090 : MUST BE BOOLEAN DATA ITEM

091 : DICTIONARY BEGIN-SESSION ERROR FOLLOWS

097 : DICTIONARY ENTITY NOT FOUND

098 : DICTIONARY ENTITY NOT RECORD

099 : DICTIONARY ENTITY NOT FILE

103 : THIS OBSOLETE LANGUAGE FEATURE WILL BE DELETED IN THE NEXT ANSI COBOL REVISION

An obsolete language feature was encountered by the compiler. The language feature will be deimplemented in the next revision of the ANSI COBOL standard.

104 : THIS OBSOLETE LANGUAGE FEATURE WILL BE DELETED IN THE NEXT ANSI COBOL REVISION HOWEVER UNISYS WILL CONTINUE TO SUPPORT THIS FEATURE AS AN EXTENSION TO THE COBOL LANGUAGE

An obsolete language feature was encountered by the compiler. The language feature will be deimplemented in the next revision of the ANSI COBOL standard. Unisys, however, will continue to support this feature as an extension to the COBOL language.

105 : THE INTERFACE TO THE DICTIONARY HAS FAILED

Check that the function name of the dictionary library exists. The function name for the dictionary is specified in the SPECIAL-NAMES paragraph.

106 : DICTIONARY INVALID ENTITY TYPE

107 : INVALID CCSVERSION SPECIFIED IN THE ALPHABET CLAUSE

An invalid CCSVERSION name is specified in the ALPHABET... FOR NATIONAL clause.

108 : CENTRALSUPPORT INTERFACE ERROR OCCURRED

An error occurred while calling a procedure in the CENTRALSUPPORT library.

109 : DICTIONARY NOT APPLICABLE WITH REPLACING

110 : DICTIONARY ONLY 01 LEVEL ALLOWED

115 : BOOLEAN LITERALS OF LENGTH GREATER THAN ONE NOT IMPLEMENTED

116 : DICTIONARY RECORD/FILE RELATIONSHIP MISMATCH

118 : CCSVERSION PHRASE NOT ALLOWED IN NESTED PROGRAM

The ALPHABET clause specified with the CCSVERSION collating sequence is not allowed in a nested program.

119 : INVALID BOOLEAN LITERAL

120 : BOOLEAN ITEM OF LENGTH GREATER THAN 1 NOT ALLOWED

121 : ILLEGAL COMPARISON OPERATOR USED WITH BOOLEAN ITEM

122 : ALIGNED AND SYNCHRONIZED CLAUSES ARE MUTUALLY EXCLUSIVE

123 : DICTIONARY TRACKING ENFORCED

131 : DISPLAY ITEM OR NON-NUMERIC LITERAL 6 LONG REQUIRED

132 : DUPLICATE ALIGNED SPECIFIED

133 : ILLEGAL VALUE FOR BOOLEAN DATA ITEMS

137 : THIS CLAUSE OF THE DICTIONARY STATEMENT HAS ALREADY BEEN SPECIFIED

A duplicate clause has been found in the DICTIONARY statement of the SPECIAL-NAMES paragraph. Refer to Volume 2 for information about using the ADDS program interface and the DICTIONARY statement.

138 : THE PROGRAM'S DEFAULT COMPUTATIONAL SIGN HAS ALREADY BEEN SPECIFIED

A duplicate DEFAULT COMPUTATIONAL SIGN clause has been found in the SPECIAL-NAMES paragraph.

139 : THE PROGRAM'S CURRENCY SIGN HAS ALREADY BEEN SPECIFIED

A duplicate CURRENCY SIGN clause has been found in the SPECIAL-NAMES paragraph.

140 : THE PROGRAM'S DECIMAL-POINT HAS ALREADY BEEN SPECIFIED

A duplicate DECIMAL-POINT clause has been found in the SPECIAL-NAMES paragraph.

141 : THE PROGRAM'S DEFAULT DISPLAY SIGN HAS ALREADY BEEN SPECIFIED

A duplicate DEFAULT DISPLAY SIGN clause has been found in the SPECIAL-NAMES paragraph.

142 : DUPLICATE ACCESS MODE SPECIFIED

The ACCESS MODE clause of the SELECT clause was specified more than once for the same file.

143 : DUPLICATE ACTUAL KEY SPECIFIED

The ACTUAL KEY clause o-of the SELECT clause was specified more than once for the same file.

144 : DUPLICATE BLOCK SIZE SPECIFIED

The BLOCK CONTAINS clause of a file description entry was specified more than once for the same file.

145 : DUPLICATE DEVICE SPECIFIED

The ASSIGN clause of the SELECT clause was specified more than once for the same file.

146 : DUPLICATE EXTERNAL RECORD

The EXTERNAL clause was specified more than once for the same item.

147 : DUPLICATE GLOBAL RECORD

The GLOBAL clause was specified more than once for the same item.

148 : DUPLICATE LABEL SPECIFIED

The LABEL clause was specified more than once for the same file.

149 : DUPLICATE ORGANIZATION SPECIFIED

The ORGANIZATION clause of the SELECT clause was specified more than once for the same file.

150 : DUPLICATE PADDING CHARACTERS

The PADDING CHARACTERS clause of the SELECT clause was specified more than once for the same file.

151 : DUPLICATE RECORD DELIMITER

The RECORD DELIMITER clause of the SELECT clause was specified more than once for the same file.

153 : DUPLICATE RECORD KEY SPECIFIED

The RECORD KEY clause of a file description entry was specified more than once for the same file.

154 : DUPLICATE RECORD SIZE SPECIFIED

The RECORD clause of a file description entry was specified more than once for the same file.

156 : DUPLICATE RESERVED AREA SPECIFIED

The RESERVE clause of the SELECT clause was specified more than once for the same file.

157 : DUPLICATE STATUS VARIABLE SPECIFIED

The FILE STATUS clause of the SELECT clause was specified more than once for the same file.

158 : DUPLICATE DMERROR USE PROCEDURE

More than one USE ON DMERROR procedure has been specified in the declaratives section of the program.

160 : DUPLICATE FILE CONTROL ENTRY

A File Control entry already exists for this file.

161 : DUPLICATE OR INCOMPATIBLE CLAUSE

162 : DUPLICATE PICTURE CLAUSE

The PICTURE clause was specified more than once for the same item.

163 : DUPLICATE PROCEDURE DIVISION USING PARAMETER

A duplicate USING clause was encountered in the Procedure Division header. Only one USING clause is allowed.

164 : EITHER ALL OF THE DIGIT POSITIONS AFTER THE DECIMAL POINT MUST BE REPRESENTED IN THE PICTURE BY THE SYMBOL '9', OR ALL OF THEM MUST BE REPRESENTED BY THE FLOATING INSERTION OR ZERO SUPPRESSION SYMBOL; THEY MAY NOT BE MIXED IN THAT CONTEXT.

For example, while 'PICTURE +++.99' and 'PICTURE +++.++' are both permitted according to this rule, 'PICTURE +++.+9' is not.

165 : ILLEGAL USE OF CONDITION-NAME

A condition-name cannot be used as an operand in a relation condition.

167 : ELEMENTARY ITEM MUST HAVE SIZE

A size must be associated with an elementary data item.

171 : END OF STATEMENT

172 : MISSING PROGRAM-ID FOR $CALLMODULE PROGRAM.

The PROGRAM-ID clause was expected but was not found. The PROGRAM-ID is required for a program that has the CALLMODULE compiler control option set.

173 : THE "END-OF-PAGE" CONDITION IS NOT REPORTED UNLESS THE ASSOCIATED FILE HAS A LINAGE CLAUSE

174 : PROGRAM INFORMATION DID NOT REACH DICTIONARY

178 : EVENT NAME OPERAND EXPECTED

179 : A LOCAL ARRAY TEMPORARY HAS BEEN GENERATED FOR THIS STATEMENT, WHICH MAY CAUSE AN INITIAL PBIT TO OCCUR. TO AVOID A PERFORMANCE PROBLEM CAUSED BY THE PBIT, RESET THE LOCALTEMP CCI OR MODIFY THE STATEMENT.

A local array temporary was allocated for the statement. A Presence Bit interrupt is generated to access the local array temporary when the statement is executed.

180 : ALIGNED AND SYNCHRONIZED CLAUSES CANNOT BE USED ON BIT BOOLEAN ITEMS WITH AN OCCURS CLAUSE

185 : NONCONFORMING STANDARD—REPORT WRITER FEATURE

A Report Writer feature was encountered.

188 : NONCONFORMING NONSTANDARD—EXTENSION EXCEEDS FIPS HIGH LEVEL

An extension to ANSI-85 COBOL was encountered that exceeded the FIPS high level.

189 : NONCONFORMING STANDARD—FEATURE EXCEEDS MINIMUM LEVEL

The FIPS minimum level was exceeded.

190 : NONCONFORMING STANDARD—FEATURE EXCEEDS FIPS INTERMEDIATE LEVEL

The FIPS intermediate level was exceeded.

192 : FILE ALREADY NAMED IN SAME RECORD AREA CLAUSE

A SAME AREA clause of the I-O CONTROL paragraph references a file-name previously declared.

195 : FILE CONTROL EXPECTED

A FILE CONTROL paragraph was expected, but was not found.

196 : FILE DECLARATION MUST CONTAIN ASSIGN CLAUSE

The ASSIGN clause is required in the SELECT clause of a FILE-CONTROL paragraph.

197 : FILE EXPECTED

A file was expected, but was not found.

199 : FILE INFO OVERFLOW

203 : FILLER ADDED

204 : FILLER AND EXTERNAL CONFLICT

A data-item was declared to be both EXTERNAL and FILLER.

205 : A FLOATING POINT QUOTIENT IS NOT ALLOWED FOR A 'DIVIDE' STATEMENT THAT HAS A 'REMAINDER' PHRASE

The DIVIDE statement with the REMAINDER phrase which results in a floating point quotient is disallowed.

206 : THIS FLOATING POINT LITERAL IS TOO LARGE FOR THE DESTINATION

This error can be due to various reasons. One example is if the MOVE of a large floating point literal to a numeric data item (BINARY, DISPLAY or COMP) that contains a fraction in its definition and the result is a value that exceeds 23 digits. This error can also be attributed to moves from double precision floating point numbers to single precision destinations.

209 : FOR EXPECTED

211 : FORM RECORD LIBRARY NAME EXPECTED

213 : FORMAL PARAMETER ERROR

The definition of a formal parameter to a procedure or program does not comply with the declaration rules for formal parameters.

216 : FORWARD DECLARATION MISMATCH

217 : FORWARD REFERENCE TABLE OVERFLOW

219 : FROM EXPECTED

220 : GENERAL STACK OVERFLOW

221 : GIVING EXPECTED

223 : COMMON AND OWN CANNOT BOTH BE SPECIFIED

An attempt was made to set both the COMMON and OWN compiler options to TRUE. The COMMON and OWN compiler options are mutually exclusive. The first option encountered is set to TRUE, and the second option encountered is ignored.

224 : COMMON OR OWN CANNOT BE DECLARED AT LEVEL 2

The COMMON and OWN compiler options cannot be declared at lexicographical level 2.

225 : COMMON OR OWN DATA NAMES CANNOT BE PARAMETERS

Data items declared to be COMMON or OWN through the use of the COMMON or OWN clause, or through the use of the COMMON or OWN compiler options, cannot be used as parameters in calls to other procedures.

226 : COMMON OR RECEIVED BY REFERENCE CLAUSE EXPECTED

The reserved word COMMON or a RECEIVED BY REFERENCE clause was expected, but was not found.

228 : GROUP COMP ITEMS CANNOT BE DISPLAYED

229 : GROUP ITEM IS SPECIFIED AS ONE OF THE OPERANDS IN MOVE STATEMENT; RESULTS OF MOVE SHOULD BE EXAMINED TO ENSURE THAT THEY MATCH THE EXPECTATIONS OF THE PROGRAMMER

The MOVE statement involves an operation between a group item and an elementary numeric item. The result of the MOVE should be examined to ensure that they match your expectations.

231 : GROUP ITEM CONTAINS PIC CLAUSE

An attempt was made to specify a PICTURE clause for a group data item. The PIC clause is valid only for elementary data items.

232 : GROUP ITEM REQUIRED

A group data item is required in the current context.

233 : HEADER EXPECTED

234 : HERE EXPECTED

237 : IDENTIFIER EXCEEDS 30 CHARACTERS

An identifier exceeds the maximum length of a user-defined word.

238 : IDENTIFIER EXPECTED

An identifier was expected, but was not found.

241 : IDENTIFIER OR LITERAL EXPECTED

An identifier or a literal was expected, but was not found.

242 : IDENTIFIER SHOULD HAVE KEY CLAUSE

246 : ILLEGAL CARD SIZE

An invalid record size was declared for a file assigned to READER.

247 : INVALID SYNTAX; COMPILING SUSPENDED HERE

The compiler has encountered a series of unrecognizable syntactical elements. The compiler will attempt to skip these elements and resume compilation at the next recognizable syntactical element.

249 : ILLEGAL CHANNEL SPECIFICATION

The CHANNEL clause of the SPECIAL-NAMES paragraph contains an invalid channel specification.

250 : THE "LOCK" PHRASE OF THE "OPEN" STATEMENT IS INCOMPATIBLE WITH "OPEN OUTPUT"

251 : ILLEGAL NATIONAL CHARACTER LITERAL

The national character literal you specified is not valid. A national literal is a character-string containing 1 to 79 characters (2 to 158 bytes). The character-string must be delimited on the left by N" and on the right by quotation marks ("). For details about national literals, refer to Section 1.

252 : ILLEGAL CLAUSE USED WITH INDEX USAGE

A data description entry for a data item declared as USAGE IS INDEX contains a clause that is invalid. The BLANK WHEN ZERO, JUSTIFIED, PICTURE, SYNCHRONIZED, and VALUE clauses must not be specified for USAGE IS INDEX data items.

254 : ILLEGAL CODE LITERAL OR COLUMN NUMBER

255 : ILLEGAL COMPARISON OF TWO OPERANDS

For the details and rules regarding the comparison of operands, refer to Section 5.

257 : ILLEGAL DEPENDING ON CLAUSE

The DEPENDING ON clause of the OCCURS clause of a data description entry is invalid for the current context.

258 : ILLEGAL DEPENDING ON RANGE

The DEPENDING ON clause of the OCCURS clause of a data description entry specifies an invalid range.

259 : ILLEGAL DEVICE NAME

262 : ILLEGAL DUPLICATE NAME

An attempt was made to declare a data item or a file connector using a previously declared user-defined word.

263 : ILLEGAL DUPLICATE TYPE GROUP

269 : ILLEGAL HARDWARE TYPE

The ASSIGN clause of the SELECT clause specified an invalid hardware device as the storage medium to be associated with the file.

270 : ILLEGAL HARDWARE TYPE FOR ACTUAL KEY CLAUSE

The ACTUAL KEY clause was specified for a file that has been assigned to a hardware type that does not permit the use of the ACTUAL KEY element.

271 : ILLEGAL INVOKE

272 : ILLEGAL KEY ITEM

An attempt was made to associate a RECORD KEY or a RELATIVE KEY with a file declared to be ORGANIZATION IS SEQUENTIAL.

275 : ILLEGAL REPLACING PHRASE

The REPLACING phrase of the statement is invalid.

276 : ILLEGAL LINE VALUE OR LINE CLAUSE OMITTED

277 : UNDIGIT LITERAL FOR NATIONAL CHARACTER ITEM MUST BE MOD 4

281 : ILLEGAL PADDING CHARACTER

The PADDING CHARACTER clause of a SELECT clause is invalid.

283 : ILLEGAL REDEFINE

An attempt to redefine a data item is not allowed.

288 : ILLEGAL SUBSCRIPT

The specified subscript is invalid or inappropriate in the current context.

290 : ILLEGAL USE OF COMMON OR OWN CLAUSE

The COMMON or OWN clause of a data description entry is invalid or inappropriate in the current context.

291 : ILLEGAL USE OF EXTERNAL CLAUSE

The EXTERNAL clause of a data description entry is invalid or inappropriate in the current context.

292 : ILLEGAL USE OF LOWER BOUNDS CLAUSE

The WITH LOWER BOUNDS clause of a data description entry is invalid or inappropriate in the current context.

295 : ILLEGAL USE OF SAME RECORD AREA

The SAME RECORD AREA clause of an I-O CONTROL entry is invalid or inappropriate to the current context.

298 : SUBSCRIPTED COMP ITEM INACCESSIBLE DUE TO HARDWARE LIMITATIONS

Unless the TARGET compiler option is LEVEL4 or a subsequent machine family, the maximum size of a COMP item is 524,287 bytes.

303 : INCONSISTENT GLOBAL SPECIFICATION

The GLOBAL clause of a file description entry is inconsistent with the current context.

310 : INDEXED FILE MUST HAVE RECORD KEY CLAUSE

The RECORD KEY clause was not found in the file description entry for an indexed file. The file description entry for an indexed file must contain the RECORD KEY clause.

311 : INFLEXIBLE COMPILER LIMIT EXCEEDED

A compiler limit has been exceeded.

313 : INFO TABLE OVERFLOW

Too many names are specified in the source program. Names include program names, file names, section names, paragraph names, and compiler-generated names.

324 : INVALID ACCESS

The ACCESS MODE clause of a SELECT entry is invalid.

325 : INVALID EXTERNAL LEVEL

The EXTERNAL clause was encountered in a data description entry with a level number other than 01. Only 01-level data items can be declared EXTERNAL data items.

326 : INVALID EXTERNAL SECTION

The EXTERNAL clause was encountered in a section other than the Working-Storage Section. Only 01-level data items in the Working-Storage Section can be declared EXTERNAL data items.

327 : INVALID GLOBAL LEVEL

The GLOBAL clause was encountered in a data description entry with a level number other than 01. Only 01-level data items can be declared as GLOBAL data items.

328 : INVALID GLOBAL SECTION

The GLOBAL clause was encountered in a section other than the Working-Storage Section. Only 01-level data items in the File Section or Working-Storage Section can be declared GLOBAL data items.

333 : INVALID NUMBER OF AREAS

The RESERVE clause of a SELECT clause specified an invalid number of input-output areas.

334 : INVALID ORGANIZATION

The ORGANIZATION clause of a SELECT clause is invalid or inappropriate in the current context.

346 : INVALID BOUND

The number of occurrences of a data item specified by the OCCURS clause is invalid or out of range.

347 : INVALID CHARACTER IN UNDIGIT LITERAL

An invalid character was found in an undigit literal. Undigit literals are restricted to the hexadecimal character set (digits 0 through 9, letters A through F).

348 : INVALID CLASS

The CLASS clause of the SPECIAL-NAMES paragraph is invalid.

349 : INVALID CLASS LITERAL

A literal specified in the CLASS clause of the SPECIAL-NAMES paragraph is invalid.

350 : INVALID CONDITION PHRASE

The specified condition is invalid.

351 : THE CONTROL VARIABLE FOR THE 'PERFORM VARYING' CLAUSE IS INVALID

The identifier that is a reference to a data item in the VARYING phrase must be numeric.

352 : INVALID ENTITY REFERENCE DECLARATION

A declaration has been encountered in an inappropriate section.

359 : INVALID LEVEL

360 : INVALID LEVEL NUMBER

The specified level number is invalid or inappropriate in the current context.

361 : INVALID LEXICAL LEVEL

The lexicographical level specified by the LEVEL compiler option is invalid for the current compilation.

362 : INVALID LIBRARY NAME IN COPY STATEMENT

The library-name specified in a COPY statement is invalid.

363 : INVALID NESTED USAGE

An attempt was made to specify a different USAGE clause for a data item subordinated to a group data item containing a USAGE clause. The USAGE clause can be specified for subordinate data items beneath a group data item containing a USAGE clause only if the same USAGE is specified.

366 : INVALID NUMBER OF SORT TAPES

The number of tapes specified in a SELECT clause for a sort file is invalid. The number of tapes must be in the range of 3 thorough 8.

367 : INVALID OCCURS LEVEL

An OCCURS clause has been encountered at an invalid level of a data description entry. An OCCURS clause cannot appear in a data description entry with a level number of 01, 66, 77, or 88, or in a data item that contains a variable occurrence data item subordinate to it.

368 : INVALID OPERAND FOR CALL SYSTEM

An invalid procedure or option was encountered in a CALL SYSTEM statement.

371 : INVALID PARAMETER

376 : INVALID SECTION FOR 01 REDEFINE

The REDEFINES clause was encountered in a data description entry outside the File Section. The REDEFINES clause is not permitted in Working-Storage Section, Linkage Section, or Local-Storage Section data description entries.

382 : INVALID TEXT NAME IN COPY STATEMENT

The text-name specified in a COPY statement is invalid.

385 : INVALID VALUE CLAUSE IN LINKAGE

The VALUE clause was encountered in a Linkage Section data description entry describing a data item that is not a condition-name. The VALUE clause can be used in the Linkage Section only for condition-name data items.

386 : I-O PROCEDURE ILLEGAL FOR TAGSORT

An attempt was made to specify an INPUT PROCEDURE or an OUTPUT PROCEDURE for a TAG-KEY or TAG-SEARCH sort. A SORT statement containing either the TAG-KEY or TAG-SEARCH phrases cannot contain either an INPUT PROCEDURE or OUTPUT PROCEDURE specification.

390 : ITEM CANNOT BE ZERO SIZE

An attempt was made to specify the size of a data item as zero.

391 : ITEM MUST BE DECLARED IDENTIFIER

A declared identifier was expected in the current context.

398 : KEY EXPECTED

The reserved word KEY was expected, but was not found.

400 : KEY LIMIT EXCEEDED

The KEY specified in a SORT statement or a MERGE statement exceeds the maximum size permitted for key data items.

401 : KEY NOT ALLOWED

402 : A SINGLE PRECISION KEY IS EXPECTED

The ACTUAL KEY or RELATIVE KEY must not exceed a size of PICTURE 9(11) nor be declared as DOUBLE.

409 : LEVEL NUMBERS MUST BE IDENTICAL

An attempt was made to REDEFINE a data item by reference to a data item with a different level number.

410 : LEVEL-77 NOT ALLOWED IN THIS SECTION

A level-77 data item describes a noncontiguous data item. A level-77 data item can be defined only in the Working-Storage section and the Linkage section.

423 : LITERAL SIZE EXCEEDS DECLARED SIZE

An attempt was made to store a literal whose length is greater than the declared size of the receiving field.

434 : MASS STORAGE MUST HAVE LABEL

The LABEL RECORD(S) OMITTED clause of a file description entry is invalid for files assigned to DISK.

435 : MAXIMUM ITEM SIZE OF 65535 EXCEEDED

An attempt was made to specify a record whose length was greater than 65535. The record size is limited to a maximum value of 65535 bytes.

437 : MAXIMUM RECORD SIZE GREATER THAN BLOCK SIZE

The MAXRECSIZE attribute of a file has been assigned a value greater than the BLOCKSIZE attribute of the file. The MAXRECSIZE attribute must be less than the BLOCKSIZE attribute for a given file.

451 : MINIMUM RECORD SIZE MUST BE LESS THAN MAXIMUM SIZE

The MINRECSIZE attribute of a file has been assigned a value less than the MAXRECSIZE attribute of the file. The MINRECSIZE attribute must be less than or equal to the MAXRECSIZE attribute for a given file.

454 : MISSING BY IN REPLACING PHRASE

The required reserved word BY was not found in the REPLACING phrase of a COPY statement or a REPLACE statement.

460 : MISSING ENDING QUOTE

An ending quote character was expected, but was not found.

461 : MISSING FILE DESCRIPTION

A file description entry in the Data Division for a file declared in the Environment Division was expected, but was not found.

464 : MISSING QUALIFICATION

465 : MISSING SUBJECT OF RELATION

The first operand in the relation condition, the subject, is not present. The subject can be an identifier, a literal, an index-name, or an arithmetic expression. Refer to Section 5 for the syntax for a relation condition.

466 : MNEMONIC VALUE CANNOT BE QUALIFIED

An attempt was made to qualify the value assigned to a mnemonic attribute. Mnemonic attribute values cannot be qualified.

467 : MNEMONIC VALUE EXPECTED

A mnemonic value was expected, but was not found.

470 : MOVE SENDING FIELD OPERAND EXPECTED

The operand preceding the word TO (the "sending" field) in the MOVE Statement is missing. You must specify either an identifier, a literal, a file-attribute identifier, or a task-attribute identifier in the sending field. For the syntax of the MOVE statement, refer to Section 6.

479 : QUALIFIER MUST BE THE CURRENT REPORT NAME

When qualified, the data name used in the UPON phrase must be qualified by the current report name.

480 : MUST BE DATA ITEM

A user-defined word that references a data item was expected, but was not found.

481 : MUST BE DATA NAME

485 : MUST BE ELEMENTARY ITEM

A field parameter in CALL USING cannot be a group item; it can only be an elementary item.

486 : MUST BE FILE

487 : MUST BE INTEGER

An integer valued data item or literal was expected, but was not found.

488 : MUST BE LD NAME

A local-name previously declared in the Local-Storage Section was expected in the WITH clause, but was not found.

489 : MUST BE LESS THAN 12 DIGITS

490 : MUST BE LEVEL 1

A level-1 data item was expected in the current context, but was not found.

491 : MUST BE LEVEL-1 OR LEVEL-77 ITEM

492 : MUST BE LEVEL-77 ITEM

A level-1 or level-77 data item was expected in the current context, but was not found.

493 : IDENTIFIER MUST BE A DETAIL GROUP NAME

The data name used in the UPON phrase must be a DETAIL report group name described in the same report as the CONTROL FOOTING report group in which the SUM clause appears.

494 : MUST BE NONEDITED NUMERIC ITEM

A nonedited numeric data item was expected in the current context, but was not found.

495 : MUST BE NUMERIC ITEM

A numeric data item was expected in the current context, but was not found.

496 : MUST BE NUMERIC

A numeric literal was expected in the current context, but was not found.

497 : MUST BE YYYYMMDD

The qualifier "YYYYMMDD" was expected, but was not found.

498 : MUST BE YYYYDDD

The qualifier "YYYYDDD" was expected, but was not found.

499 : MUST BE MMDDYYYY

The qualifier "MMDDYYYY" was expected, but was not found.

501 : MUST BE SINGLE ALPHA CHARACTER

A single alphabetic character was expected in the current context, but was not found.

503 : MUST BE UNSIGNED DISPLAY NUMERIC

An unsigned display numeric data item was expected in the current context, but was not found.

504 : MUST BE WITHIN CURRENT LD DESCRIPTION

A formal parameter was not found in the local-storage description entry specified by the WITH clause of the Program-Library Section or specified in the Declarative section.

505 : MUST START IN MARGIN A

The following items must begin in area A:

- Division, section, and paragraph headers
- The keywords DECLARATIVES and END DECLARATIVES
- The level indicators FD and SD
- The level numbers 01 and 77

You must place sequence numbers in columns 1 through 6. Line indicator symbols and compiler control options belong in column 7. All other items must begin in columns 12 through 72 (area B).

512 : NEED DATA BASE TITLE

514 : NESTED COPY STATEMENTS ARE NOT ALLOWED

An attempt was made to copy text that contained a COPY statement from another COBOL source program, or to initiate a COPY statement within a COPY statement. The text to be copied into the current COBOL source program cannot contain a COPY statement.

543 : NOT ALLOWED FOR SEQUENTIAL ORGANIZATION FILE

An attempt was made to use an invalid clause in the SELECT clause for a file declared with an ORGANIZATION of SEQUENTIAL.

547 : NUMBER GREATER THAN 23 DIGITS

An attempt was made to process a number greater than 23 digits in length.

549 : NUMERIC DATA NAME OPERAND EXPECTED

A data name representing a numeric operand was expected, but was not found.

551 : NUMERIC LITERAL MUST BE IN RANGE OF 1 TO 256

552 : MAXIMUM RECORD SIZE OF 16777215 BYTES EXCEEDED

This record exceeds the maximum size for machines at the lower range of the LEVEL4 group.

553 : RECORD SIZE MAY EXCEED LIMITS FOR SOME MACHINES OF THIS TARGET FAMILY

A target group or a secondary target was specified and this record exceeds the limits of some of those target machines. The record might run successfully on higher-range machines and fail at run time on lower-range machines.

554 : MAXIMUM RECORD SIZE OF 268435455 BYTES EXCEEDED

This record exceeds the maximum size for machines at the higher range of the LEVEL4 group and all LEVEL5 machines.

555 : NUMERIC RECEIVING FIELD OPERAND EXPECTED

The GIVING clause of a library declaration or a CALL statement did not reference a numeric data item. When the GIVING clause is used to declare or call a procedure that returns a value, it must reference a numeric data item.

556 : NUMERIC SENDING FIELD OPERAND EXPECTED

557 : MAXIMUM RECORD SIZE OF 134217727 BYTES EXCEEDED

This record exceeds the maximum size for machines at the lower range of the LEVEL5 group.

558 : MAXIMUM NUMERIC ITEM SIZE OF 99999 EXCEEDED

559 : MAXIMUM RECORD SIZE OF 1048572 BYTES EXCEEDED

The record size exceeds the maximum byte limit of 1048572.

560 : OCCURS NOT ALLOWED FOR TARGET

The OCCURS clause was encountered in a data description entry that contained the REDEFINES clause. The OCCURS clause and the REDEFINES clause are mutually exclusive.

561 : ILLEGAL USE OF LONG NUMERIC ITEM

562 : UNSUPPORTED MOVE USING LONG NUMERIC OPERAND

563 : COMPARES OF LONG NUMERIC ITEMS MUST MATCH IN TYPE, SIZE AND USAGE

564 : ILLEGAL ITEM TYPE COMPARED WITH LONG NUMERIC ITEM

565 : ILLEGAL COMPARISON OPERATOR USED WITH LONG NUMERIC ITEM

566 : ILLEGAL COMPARISON OPERATOR FOR UNDIGIT LITERAL AND NUMERIC

567 : ONLY UNSIGNED AND INTEGER NUMERIC ITEMS MAY BE COMPARED WITH UNDIGIT LITERALS

568 : SIZES DO NOT MATCH IN COMPARE OF UNDIGIT LITERAL AND NUMERIC ITEM

569 : LONG NUMERIC MUST BE UNSIGNED INTEGER

570 : ONLY ONE ODT INPUT PRESENT ALLOWED

571 : ILLEGAL TO COMPARE WITH UNDIGIT LITERAL

576 : USAGE MUST BE DISPLAY OR NATIONAL

585 : PAGE LIMIT CLAUSE REQUIRED IN RD ENTRY

586 : THIS PROCEDURE USED A DIFFERENT NUMBER OF PARAMETERS IN A PRIOR CALL

This call has a different number of parameters than the previous call. Compare all calls with the formal parameter list and modify the call appropriately.

587 : PARAMETER TYPE NOT ALLOWED FOR LIBRARIES

588 : PARAMETER WILL BE PASSED BY REFERENCE

589: THIS PROCEDURE USED A SHORTER PARAMETER ON A PRIOR CALL. ALL EXCESS DATA WILL BE TRUNCATED.

A CALL statement is using a subordinate data item as a parameter, which cannot be passed directly; a copy of the item is created and passed. The first CALL to the same procedure (as seen by the compiler) used a shorter parameter. The subordinate item used by this CALL is truncated to fit the formal parameter that was created for the library template built for the prior CALL. To avoid the extra processing required by subordinate items, use 01 or 77 level data items as parameters. For more information, refer to "Library Concepts" in Section 1, Program Structure and Language Elements.

592 : PERIOD EXPECTED

A period was expected, but was not found.

593 : A COMMA MUST BE USED FOR THE DECIMAL POINT BECAUSE THE DECIMAL-POINT IS COMMA CLAUSE WAS SPECIFIED IN THE SPECIAL-NAMES SECTION.

A period instead of a comma was used as a decimal point. When the DECIMAL-POINT IS COMMA clause is specified in the SPECIAL NAMES SECTION, a comma—not a period—must be used for a decimal point.

595 : PICTURE CLAUSE IS REQUIRED IN FORMAT 3 ENTRIES

A Format 3 Report-Group Description Entry must have a PICTURE clause.

596 : THE CHARACTER STRING IN THE PICTURE CLAUSE IS INVALID, OR IT IS INCOMPATIBLE WITH THE DECLARED USAGE OF THE ASSOCIATED DATA ITEM

597 : PICTURE STRING EXCEEDS 30 CHARACTERS

An attempt was made to declare a PICTURE character-string greater than 30 characters in length. The maximum number of characters allowed in a PICTURE character-string is 30.

600 : PORT ACCESS IGNORED

The ACCESS MODE clause was specified for a file declared as a port file. The ACCESS MODE clause has no effect on port files.

601 : PORT ORGANIZATION IGNORED

The ORGANIZATION clause was specified for a file declared as a port file. The ORGANIZATION clause has no effect on port file.

602 : PORT RESULT IGNORED

The RESERVE clause was specified for a file declared as a port file. The RESERVE clause has no effect on port file.

605 : THE LENGTH OF THE SENDING ITEM IS GREATER THAN THE LENGTH OF THE RECEIVING ITEM. ALL EXCESS DATA WILL BE TRUNCATED AFTER THE RECEIVING ITEM HAS BEEN FILLED.

This message is intended for your information and does not indicate an error condition.

610 : DATA TYPE MISMATCH MAY CAUSE A LOSS OF PRECISION OF THE VALUE STORED IN THE DESTINATION.

The characteristics of the destination field in this statement (BINARY, DISPLAY or COMP) are such that digits that might be significant in the source field (REAL or DOUBLE), or in the arithmetic result, might be lost when the value is stored in the destination.

611 : PROGRAM NESTED TOO DEEP

The compiler has encountered a source program that is too deeply nested within other source programs. Programs can be nested to a depth of 15.

613 : PSEUDO TEXT 1 MAY NOT BE NULL

The syntactical element pseudo-text-1 of the REPLACING phrase of a COPY statement or a REPLACE statement must contain one or more text words. Pseudo-text-1 cannot be null.

614 : QUALIFICATION AMBIGUOUS

The compiler is unable to determine the referent of a user-defined word, because the user-defined word is not qualified appropriately. Additional qualification of the user-defined word is required to remove ambiguity.

618 : QUALIFICATION NOT COMPLETE

The qualification of a user-defined word is incomplete.

619 : QUALIFIER OR NAME HAS NOT APPEARED BEFORE

A qualifier or name that is unknown to the compiler has been encountered.

620 : THIS RECORD DESCRIPTION CONTAINS MORE CHARACTERS THAN THE MAXIMUM SPECIFIED WITHIN THE “RECORD” CLAUSE OF THE FILE DESCRIPTION

This message is intended for your information and does not indicate an error condition.

621 : THIS RECORD DESCRIPTION CONTAINS FEWER CHARACTERS THAN THE MINIMUM SPECIFIED WITHIN THE “RECORD” CLAUSE OF THE FILE DESCRIPTION

This message is intended for your information and does not indicate an error condition.

622 : READ ONLY ATTRIBUTE

623 : REAL OR DOUBLE ITEM CANNOT HAVE PICTURE OR SIZE

A data item declared with USAGE IS REAL or USAGE IS DOUBLE contains a PICTURE clause or a SIGN clause. Both the PICTURE clause and the SIGN clause are invalid for data items declared as REAL or DOUBLE.

624 : REAL OR DOUBLE LITERAL TOO LARGE

625 : RECORD DELIMITER ONLY FOR VARIABLE LENGTH RECORD

627 : RECEIVING FIELD MAY NOT BE SIGNED, SCALED, OR EDITED

628 : VALUE ENTERED AS MAXIMUM RECORD LENGTH WITHIN THE RECORD CONTAINS CLAUSE DOES NOT MATCH THE LARGEST SUBORDINATE RECORD SIZE

This message is intended for your information and does not indicate an error condition.

629 : VALUE ENTERED AS MINIMUM RECORD LENGTH WITHIN THE RECORD CONTAINS CLAUSE DOES NOT MATCH THE SMALLEST SUBORDINATE RECORD SIZE

This message is intended for your information and does not indicate an error condition.

630 : RECORD DESCRIPTIONS NOT ALLOWED FOR REPORT FILES

A file description entry for a report file is not allowed by record description entries.

631 : RECORD EXPECTED

The reserved word RECORD was expected, but was not found.

632 : RECORD KEY MUST BE WITHIN RECORD AREA

The RECORD KEY or ALTERNATE RECORD KEY defined for an indexed file must refer to data items defined within the record area for the file.

634: REDEFINED AREA MUST BE GREATER OR SAME SIZE

636 : REDEFINED FORMLIB MUST BE INVOKED WITH SAME

637 : REDEFINES AND EXTERNAL CONFLICT

Both the REDEFINES clause and the EXTERNAL clause were encountered in the data description entry of a data item. The REDEFINES clause and the EXTERNAL clause must not be specified in the same data description entry.

639 : REDEFINES CANNOT HAVE VALUE

Both the REDEFINES clause and the VALUE clause were encountered in the data description entry of a data item. The REDEFINES clause and the VALUE clause must not be specified in the same data description entry.

645 : THE MAXIMUM NUMBER OF RETURN STATEMENTS AND OUTPUT PROCEDURES HAS BEEN EXCEEDED

The compiler is capable of managing up to a total of 500 return statements and output procedure declarations.

646 : RELEASE MUST APPEAR IN INPUT PROCEDURE

647 : THE MAXIMUM NUMBER OF RELEASE STATEMENTS ALLOWED IN A PROGRAM (400) HAS BEEN EXCEEDED

The compiler is capable of managing up to 400 release statements. It is recommended that the release statement be placed in a PERFORM function.

648 : THE MAXIMUM NUMBER OF RETURN STATEMENTS ALLOWED IN A PROGRAM (400) HAS BEEN EXCEEDED

The compiler is capable of managing up to 400 return statements.

649 : RENAMED CANNOT BE VARIABLE SIZE

An attempt was made to rename a variable-occurrence elementary data item or a group of elementary data items containing one or more variable occurrence data items.

650 : RENAMED ITEM CANNOT HAVE OCCURS

An attempt was made to rename an elementary data item or a group of elementary data items that contained an OCCURS clause.

651 : RENAMED LEVEL NUMBER IS ILLEGAL

An attempt was made to rename a data item with a level number of 01, 66, 77, or 88.

652 : RENAMES DATANAME-3 PRIOR TO DATANAME- 2

A level-number 66 RENAMES entry specified the ending data item before specifying the beginning data item of the renamed elementary items.

654 : RENAMES NOT NEXT TO RENAMED AREA

A level-number 66 RENAMES entry did not follow immediately after the last data description entry of the associated record description entry.

656 : REPORT FAILS TO SPECIFY A CODE LITERAL

If you specify the CODE clause for any report in a file, it must be specified for all reports in that file.

658 : REPORT NAME OR DETAIL GROUP NAME EXPECTED

A report name or detail group name was expected in the current context but was not found.

663 : RESERVE AREA LIMIT EXCEEDED

The RESERVE clause of a SELECT clause specifies an invalid number of input-output areas. The maximum number of input-output areas that can be declared is 63.

664 : RESERVED WORD CANNOT BE USER DEFINED

An attempt was made to define a reserved word as a user-defined word.

665 : RETURN MUST APPEAR IN OUTPUT PROCEDURE

666 : MISSING LIBRARY TITLE OR FUNCTIONNAME

The library title or library function name is expected to follow the "entry-procedure-name IN . . . " phrase in the CALL statement syntax.

667 : BYFUNCTION IS NOT PERMITTED WITH A LIBRARY TITLE

The BYFUNCTION clause is not valid when a library title is specified in the CALL statement syntax.

668 : INVALID LIBRARY ACCESS, BYTITLE OR BYFUNCTION EXPECTED—BYTITLE ASSUMED

You specified an invalid library access mode following the library title or function name. Only BYTITLE or BYFUNCTION are allowed.

669 : FAMILYNAME EXPECTED

A family name was expected following the word ON in the library title specification.

672 : SCOPE STACK OVERFLOW

696 : SET NAME EXPECTED

699 : SIGN CLAUSE MUST BE USED WITH SIGNED NUMERIC ITEM

Every numeric data description entry whose PICTURE contains the character S is considered to be a signed numeric data item. The S indicates only the presence of the operational sign. You must specify the position and mode of representation of the sign by including the SIGN clause in the data description entry.

700 : SIGN TYPE CONFLICT

A SIGN clause was encountered in an inappropriate data description entry. The SIGN clause is valid only in numeric data description entries whose PICTURE clause contains the character S, or a group item containing at least one such numeric data description entry.

705 : SDO OR EDO EXPECTED

You specified a national literal but omitted the control characters SDO and EDO, which enable the compiler to distinguish national characters from alphanumeric characters. The control character SDO (for "start of double octet") must follow the first quotation mark that begins the national literal. The control character EDO (for "end of double octet") must precede the quotation mark that ends the literal. For more information, refer to the discussion of national literals in Section 1.

717 : STATISTICS ILLEGAL AFTER LEVEL 2

722 : STRING CONTINUATION INCORRECT

The continuation of a string is incorrect. The continuation line of a string must begin in area B.

723 : STRING OR FIGURATIVE MUST FOLLOW ALL

724 : NATIONAL CHARACTER LITERAL EXCEEDS 80 CHARACTERS

A national literal can contain a maximum of 79 characters for a maximum of 158 bytes. The remaining character (2 bytes) is occupied by the control characters SDO (for "start of double octet" and EDO for "end of double octet"). Each control character occupies 1 byte.

727 : SUBSCRIPTS REQUIRED

A subscripted data item is required in the current context.

728 : SYMBOL TABLE OVERFLOW

Too many names were specified in the same program, including program names, file names, data names, section names, paragraph names, and compiler generated names.

730 : RUN TIME ERROR POSSIBLY AN UNINITIALIZED VARIABLE

745 : TAGSORT NOT IMPL

An attempt was made to specify multiple input files for a SORT statement that contained the TAGSEARCH option. The TAGSEARCH option does not support more than one input file.

753 : THIS LEVEL NUMBER SHOULD BE AT AREA A

A 01 or 77 level data description entry should begin in Margin A.

757 : TO EXPECTED

761 : TOO MANY FILES

The compiler has attempted to open more files than are permitted. The maximum number of files for each compilation is 255.

764 : TOO MANY PROGRAMS

765 : TOO MANY RECORDS

766 : TOO MANY STATEMENT BLOCKS

767 : TOO MANY STATEMENTS

768 : TOO MANY TWIGS

778 : TRUNCATION OF NON-ZERO DIGITS

An action directed by the source program has resulted in the truncation of nonzero digits.

781 : UNDEFINED FORWARD IDENT

Identifiers referenced in the Environment Division or the Data Division were not previously declared.

783 : UNDIGIT LENGTH MUST BE EVEN

784 : UNDIGIT LITERALS NOT ALLOWED IN THIS CONTEXT

The undigit literal specified is invalid or inappropriate in the current context.

790 : UNRECOGNIZED CONSTRUCT

791 : UNRESOLVED QUALIFIER POOL OVERFLOW

794 : USAGE CONFLICT BETWEEN LEVELS

An attempt was made to specify a different USAGE clause for a data item subordinated to a group data item containing a USAGE clause. The USAGE clause can be specified for subordinate data items beneath a group data item that contains a USAGE clause only if the same USAGE is specified.

795 : USAGE CONFLICT

800 : USING EXPECTED

802 : VALID FOR SEQUENTIAL ORGANIZATION

A clause that is valid only for sequential organization files was encountered in a SELECT clause defining a relative, indexed, or sort or merge file.

805 : VALUE CLAUSE LIMIT EXCEEDED

An attempt was made to assign an initial value that exceeded the size of the data item.

806 : VALUE CLAUSE NOT ALLOWED FOR COMMON OR OWN ITEM

The VALUE clause was encountered in a data description entry that also contained the COMMON clause or the OWN clause.

807 : VALUE NOT ALLOWED FOR ITEM WITHIN OCCURS

An attempt was made to assign an initial value that exceeded the size of the data item.

808 : VALUE NOT ALLOWED IN FILE SECTION

809 : VALUE NOT ALLOWED

The VALUE clause is invalid or inappropriate in the current context.

810 : VALUE PHRASES OVERFLOW

811 : VALUE SPECIFIED IN PRIOR LEVEL

The VALUE clause was encountered in a data item subordinated to a group item that contained a VALUE clause.

814 : DEIMPLEMENTED DOLLAR OPTION TREATED AS A USER-DEFINED BOOLEAN OPTION

815 : THERE ARE TOO MANY PROGRAMS REQUIRING A CODE POINTER INDEX

The program has exceeded a compiler limit on the number of called programs. The program contains too many IPC CALL statements identifying different procedures to be called.

816 : THERE ARE TOO MANY LONG PROGRAM-ID NAMES, THE RUN TIME COBOL85SUPPORT INTERFACE HAS BEEN EXCEEDED

The program has exceeded a compiler limit on the size of PROGRAM-ID names. The program contains too many nested programs with long PROGRAM-ID names.

817 : THERE ARE TOO MANY COMMON NESTED PROGRAMS, THE RUN TIME COBOL85SUPPORT INTERFACE HAS BEEN EXCEEDED

The program has exceeded a compiler limit on the number of allowable COMMON nested programs. The program contains too many nested programs that are allowed to be called by other programs. The maximum number of COMMON programs that the compiler permits depends on how deeply the programs are nested and on how many program are nested inside a given program.

818 : THERE ARE TOO MANY NESTED PROGRAMS, THE RUN TIME COBOL85SUPPORT INTERFACE HAS BEEN EXCEEDED

The program has exceeded a compiler limit on the number of nested programs. The limit was exceeded because the program contains one of the following conditions:

- There are too many nested programs for the run-time COBOL85SUPPORT interface.
- There are too many nested programs with a FILE SECTION because the local file requires an implicit CLOSE procedure at the EXIT PROGRAM statement.

820 : TYPE CLAUSE OVERRIDES THE EDITED PICTURE CLAUSE

The edited PICTURE clause is ignored if it is used with the TYPE clause.

821 : THIS DECLARATION IS NOT ALLOWED WITH TYPE CLAUSE

The PICTURE and USAGE clauses are the only clauses that can be used with the TYPE clause. The USAGE clause can be only DISPLAY or NATIONAL.

822 : THE ONLY VALID TYPES ARE SHORT-DATE, LONG-DATE, NUMERIC-DATE, NUMERIC-TIME AND LONG-TIME

A data description entry contains an invalid value for the TYPE clause. The only valid values for the TYPE clause are SHORT-DATE, LONG-DATE, NUMERIC-DATE, NUMERIC-TIME, and LONG-TIME.

823 : A CONVENTION OR LANGUAGE CLAUSE IS EXPECTED

A data description entry contains an invalid value for the USING clause. The only valid values for the USING clause are CONVENTION and LANGUAGE.

826 : WRONG TYPE ENTITY

827 : INVALID SYNTAX

The specified syntax is invalid or inappropriate in the current context.

828 : USAGE MUST BE DISPLAY

The USAGE of the current data item must be DISPLAY.

830 : NOT ALLOWED FOR RELATIVE ORGANIZATION FILE

The attempted action is invalid or inappropriate for files declared with ORGANIZATION IS RELATIVE.

831 : RELATIVE FILES MUST BE ASSIGNED TO MASS STORAGE

An attempt was made to ASSIGN a file declared as ORGANIZATION IS RELATIVE to a hardware type other than DISK.

832 : RELATIVE KEY EXPECTED

The RELATIVE KEY clause is expected for a relative file in random access mode.

833 : INDEXED FILES MUST BE ASSIGNED TO MASS STORAGE

An attempt was made to ASSIGN a file declared as ORGANIZATION IS INDEXED to a hardware type other than DISK.

834 : ILLEGAL USE OF SAME SORT AREA

The files specified in a SAME SORT AREA were not sort or merge files.

835 : FILE ALREADY NAMED IN SAME AREA CLAUSE

A SAME AREA clause or a SAME RECORD AREA clause referenced a file that has already appeared in a SAME AREA clause or SAME RECORD AREA clause. A file cannot appear in more than one SAME AREA clause or more than one SAME RECORD AREA clause.

836 : SORT FILE CANNOT HAVE SAME AREA

A SAME AREA clause referenced a file previously declared as a sort or merge file.

837 : INVALID SINGLE UNIT OPTION SPECIFIED

An attempt was made to use an option which is available only for files declared as disk files.

838 : INVALID ALLOCATION SPECIFIED

An attempt was made to specify a file allocation which is available only for files declared as disk files.

840 : STANDARD RECORD DELIMITER ONLY FOR MAG TAPE

841 : ILLEGAL BLOCK SIZE

The size specified by the BLOCK CONTAINS clause of a file description entry is invalid.

842 : ILLEGAL RECORD SIZE

The size specified by the RECORD clause of a file description entry is invalid.

843 : ILLEGAL VARIABLE LENGTH RECORD SPECIFIED

The variable-length record specified by the RECORD clause of a file description entry is invalid.

844 : ILLEGAL CHARACTER IN SPELLING OF IDENTIFIER

A user-defined word contained an invalid character.

845 : NUMERIC LITERAL CONTAINS UNDIGIT

A user-defined word contained an undigit character.

846 : LITERAL EXCEEDS MAXIMUM ALLOWED

A literal exceeds the maximum length allowed for that particular type of literal.

847 : UNEXPECTED SYMBOL

An unexpected character, literal, or user-defined word has been encountered.

848 : ITEM MUST BE LABEL

A section-name or paragraph-name was expected by the current context, but was not found.

849 : NON-EXECUTABLE STATEMENT ENCOUNTERED

Any statements following this GO TO, STOP RUN, or EXIT PROGRAM statement and before the next section or paragraph header are not accessible.

850 : RECEIVING FIELD INCOMPATIBLE WITH SENDING FIELD

The receiving field of a data manipulation operation is incompatible with the sending field.

851 : GO TO STATEMENT WITH NO LABEL MUST BE IN A PARAGRAPH BY ITSELF

If a GO TO statement appears without a paragraph-name or section-name, it must be in a paragraph by itself.

852 : GO TO STATEMENT MUST BE THE LAST IN AN IMPERATIVE SENTENCE

If a GO TO statement appears in a consecutive sequence of imperative sentences, it must appear as the last statement in that sequence.

853 : STOP RUN STATEMENT MUST BE THE LAST IN AN IMPERATIVE SENTENCE

If a STOP RUN statement appears in a consecutive sequence of imperative sentences, it must appear as the last statement in that sequence.

854 : VALUE OUT OF RANGE

The specified value is out of the acceptable range for the current context.

855 : NUMBER OF SUBSCRIPTS DOES NOT MATCH DECLARATION

The number of subscripts specified for a data item does not match the number declared by the OCCURS clause in the data description entry for the data item.

856 : MUST BE MNEMONIC FOR ODT

An ACCEPT or DISPLAY statement specified a name other than an acceptable mnemonic-name for the operator's display terminal (ODT). Acceptable mnemonic-names for the ODT are specified with the ODT clause of the SPECIAL-NAMES paragraph.

857 : ITEM TO BE SEARCHED MUST BE A TABLE

A SEARCH statement specifies a data item that is not a table as the element to be searched. Only table data items can be searched using the SEARCH statement.

858 : TABLE TO BE SEARCHED MUST HAVE INDEXED BY CLAUSE

A SEARCH statement specifies a table that is not an indexed table as the element to be searched. Only tables that contain the INDEXED BY clause can be searched by using the SEARCH statement.

859 : ITEM TO BE SEARCHED MUST HAVE KEY IS CLAUSE

A binary SEARCH statement specifies a table that does not contain the KEY IS clause in its data description.

860 : NEXT SENTENCE AND END-SEARCH MUST NOT BOTH APPEAR

If the END-SEARCH clause appears as part of the SEARCH statement, the NEXT SENTENCE clause is invalid.

861 : ILLEGAL INDEX-VARIABLE FOR VARYING IDENT

The index variable specified by the VARYING phrase of the SEARCH statement is illegal or invalid.

862 : CONDITION NAME IN THIS USAGE MUST HAVE ONLY 1 ASSOCIATED VALUE

Condition names used in the WHEN phrase of a SEARCH statement must have only a single value.

863 : MUST BE RECORD KEY

The data-name used as the source comparison in a binary search (specified in the WHEN phrase of a SEARCH statement) must be referred to in the KEY IS phrase in the OCCURS clause for the table to be searched.

864 : MUST NOT BE RECORD KEY

The data-name used as the target comparison in a binary search (specified in the WHEN phrase of a SEARCH statement) must not be referred to in the KEY IS phrase in the OCCURS clause for the table to be searched.

865 : MUST BE RECORD KEY OR BE SUBSCRIPTED BY INDEX

The data-name used as the source comparison in a binary search (specified in the WHEN phrase of a SEARCH statement) must be referred to in the KEY IS phrase in the OCCURS clause for the table to be searched, or subscripted by the first index-name associated with the table to be searched.

866 : CURRENCY SIGN WITHOUT PICTURE SYMBOL LIMITED TO ONE CHARACTER

The literal specified as the currency symbol by use of the CURRENCY SIGN clause, without the PICTURE SYMBOL phrase, is greater than one character in length.

867 : DASH AT END

A hyphen character was encountered at the end of a number or user-defined word. The compiler ignored the hyphen character.

868 : THE LIBRARYLOCK OPTION MAY NOT BE USED WITH AN EXPLICITLY DEFINED COBOL85 LIBRARY - OPTION IGNORED

The LIBRARYLOCK compiler control option has no effect when used in a COBOL85 library.

869 : A LIBRARY CANNOT BE COMPILED ABOVE LEX LEVEL 2

The LEVEL compiler control option was set higher than 2 when the library was compiled. Remove the LEVEL option, set it to 2, or change the program to no longer be a library.

870 : DUPLICATE SYMBOLIC CHARACTER

An attempt was made to define a symbolic character more than once in the SYMBOLIC CHARACTERS clause of the SPECIAL-NAMES paragraph.

871 : SYMBOLIC CHARACTER UNEQUAL LIST SIZE

The number of symbolic characters does not match the number of integers specified in the SYMBOLIC CHARACTERS clause of the SPECIAL-NAMES paragraph.

872 : INTEGER MUST BE A VALID ORDINAL POSITION IN THE NATIVE CHARACTER SET

The integer specified in the SYMBOLIC CHARACTERS clause of the SPECIAL-NAMES paragraph is not a valid ordinal position in the native character set.

873 : A SWITCH REQUIRES A MNEMONIC NAME

A valid mnemonic-name was expected in the SET statement, but was not found.

874 : SORT FILE EXPECTED

A valid sort file name was expected, but was not found.

875 : INVALID SORT KEY

The KEY specified in a SORT statement is invalid.

876 : INVALID ALPHABET NAME

The alphabet-name specified as the COLLATING SEQUENCE is invalid.

877 : SAME SORT AREA OR SAME SORT-MERGE AREA NOT ALLOWED

The SAME SORT AREA or SAME SORT-MERGE AREA is invalid or inappropriate to the current context.

878 : SAME CLAUSE NOT ALLOWED

The SAME clause is invalid or inappropriate in the current context.

879 : SORT FILE RECORD EXPECTED

The record specified in a RELEASE statement was not a sort file record. Only sort file records can be specified in the RELEASE statement.

880 : SORT FILE RECORD AREA AND FROM/INTO IDENTIFIER MUST NOT BE THE SAME STORAGE AREA

The same storage area is used by the data item specified by the FROM phrase, and the sort-merge record.

881 : THE NUMBER OF SELECTION SUBJECTS DOES NOT MATCH THE NUMBER OF SELECTION OBJECTS

The number of data items specified in the WHEN phrase of an EVALUATE statement do not match.

882 : IDENTIFICATION DIVISION OR END OF PROGRAM EXPECTED

The reserved words IDENTIFICATION DIVISION or END OF PROGRAM were expected, but were not found.

883 : END OF PROGRAM IDENTIFIER DOES NOT MATCH THE CURRENT SCOPE

The program-id specified in an END OF PROGRAM statement does not match the current scope of the program.

884 : TOO MANY LITERALS

A compiler limit on the number of literals that can be defined has been exceeded.

885 : AN INDEXED DATA ITEM CANNOT BE A CONDITIONAL VARIABLE

An attempt was made to use a data item declared as an index variable as a condition.

886 : A REDEFINED ITEM CANNOT BE OF VARIABLE SIZE

An attempt was made to REDEFINE a variable size data item.

887 : CANNOT BE SUBSCRIPTED

A subscript variable was encountered that was invalid or inappropriate to the current context.

888 : AN EXTERNAL ITEM CANNOT HAVE A VALUE

Both the VALUE clause and the EXTERNAL clause were encountered in the data description entry of a data item. The VALUE clause and the EXTERNAL clause must not be specified in the same data description entry.

889 : STATEMENTS IN DECLARATIVES MUST NOT REFERENCE A NON-DECLARATIVE PROCEDURE

A GOTO statement or a PERFORM statement in the declaratives section of the program referenced a nondeclarative procedure.

890 : IF ONE PROCEDURE IS DECLARATIVE THEN BOTH PROCEDURES MUST BE DECLARATIVE

The PERFORM THRU statement in the declaratives section of the program specified a procedure that is not in the declaratives section.

891 : ILLEGAL BRANCH TO A DECLARATIVE PROCEDURE FROM OUTSIDE THE DECLARATIVE SECTION

An attempt was made to branch to a procedure declared in the declaratives section from outside the declaratives section.

892 : THIS 'SET' STATEMENT SPECIFIES A VALUE THAT CAUSES THE ASSOCIATED INDEX TO BE OUTSIDE THE BOUNDARIES OF THE TABLE WITH WHICH IT IS ASSOCIATED.

You tried to set an index to a value that was larger or smaller than the limits established in the OCCURS clause.

893 : INVALID SEQUENCE RANGE IN COPY STATEMENT

The sequence range for a library specified in a COPY statement is invalid.

894 : TRUNCATED LIBRARY NAME IN COPY STATEMENT

The library-name specified in a COPY statement was too long causing the library name to be truncated.

895 : REPLACING PHRASE NOT USED: CHECK SPELLING

The REPLACING phrase of a COPY statement or a REPLACE statement was not used. A possible spelling error might have caused the replacement search to fail.

896 : SYNTAX ERROR, TOKEN RESPELLED TO BE

The compiler has encountered an unknown or inappropriate token. The compiler has changed the spelling of the token and will attempt to continue.

897 : SYNTAX ERROR, TOKEN INSERTED

The compiler has encountered a syntax error. The compiler has inserted a token and will attempt to continue.

898 : SYNTAX ERROR; TOKEN DELETED

The compiler has encountered an unexpected token. The compiler has deleted the token and will attempt to continue.

899 : SYNTAX ERROR; TOKEN REPLACED BY

The compiler has encountered an unexpected token. The compiler has replaced the token and will attempt to continue.

900 : PARSER RESUMED AT

The compiler was unable to resolve a syntax error. The source record or source records in error were skipped, and compilation was resumed.

901 : IRRECONCILABLE PARSER ERROR; TOKEN EXPECTED

The compiler has encountered unexpected syntactical elements. The token expected is displayed, and the compilation is resumed with the next identifiable syntax element.

902 : PARSER RESUMED AT END OF DIVISION

The compiler was unable continue parsing the syntactical elements of the current division. The remainder of the division has been skipped, and parsing has been resumed at the end of the division.

903 : MUST BE NUMERIC INTEGER OR INDEX

904 : MUST BE NUMERIC OR INDEX IDENT

905 : MUST BE INDEX NAME OR DATA NAME

906 : MUST BE INDEX NAME

907 : INVALID USING CLAUSE

The USING clause is invalid or inappropriate in the current context.

908 : DMS KEY LIMIT EXCEEDED

A compiler limit on the number of database key items that can be defined has been exceeded.

909 : DMS INFO LIMIT EXCEEDED

A compiler limit on the number of database items has been exceeded.

910 : NAME TRUNCATED TO 17 CHARACTERS

Specified database name exceeds 17 characters. Database names are limited to 17 characters in length.

911 : UNDEFINED DATA NAME

An attempt was made to reference an undefined data item.

913 : DIVIDE REMAINDER STACK OVERFLOW

The data item specified by the REMAINDER phrase of a DIVIDE statement was not large enough to contain the remainder.

914 : ARITHMETIC STACK OVERFLOW

The evaluation of an arithmetic expression has produced a resultant value that is too large for the data item or temporary variable.

916 : CANNOT BE EXPLICIT LIBRARY

The LIBRARYPROG or LEVEL compiler option was encountered. The presence of either or both of these options prohibit the program from being an explicit library program.

917 : INVALID PROGRAM ATTRIBUTE

An error was encountered in the IS clause of the PROGRAM-ID paragraph.

918 : VALUE NOT ALLOWED IN LD

An attempt was made to use the VALUE clause in data description entries the Local-Storage section. The VALUE clause is invalid in the Local-Storage section.

919 : MUST BE A LIBRARY PROGRAM

An attempt was made to use syntactical elements that are valid only in explicit library programs.

920 : INTNAME MUST BE PROGRAM-ID

The library-name specified in the library EXPORT definition of the Program-Library section must match the program-name specified in the PROGRAM-ID paragraph of the program.

921 : MUST BE IN THE OUTER BLOCK

The library EXPORT definition of the Program-Library section and the CALL SYSTEM FREEZE statement are valid only in the outer block of an explicit library program.

922 : MUST BE WITH IMPORT SYNTAX

An attempt was made to use syntactical elements that are valid only in library IMPORT definitions of the Program-Library section.

923 : MUST BE WITH EXPORT SYNTAX

An attempt was made to use syntactical elements that are valid only in library EXPORT definitions of the Program-Library section.

924 : INVALID LIBRARY ATTRIBUTE

An attempt was made to specify an invalid library attribute.

925 : INVALID LIBRARY ENTRY

The declared library entry point is invalid or inappropriate to the current context.

926 : LD NAME ALREADY SPECIFIED

Only one local-name area can be specified in the WITH clause.

927 : USAGE OF THIS ITEM SHOULD BE EITHER BINARY OR REAL

928 : EXPRESSION SHOULD BE EITHER BINARY OR REAL

929 : INVALID LENGTH IN BIT MOVE

930 : IDENTIFIER USED IN A WAIT STATEMENT; INVALID P PICTURE CLAUSE

931 : MORE THAN 250 EVENTS SPECIFIED IN WAIT STATEMENTS

932 : TYPE MISMATCH ON FILE ATTRIBUTE ASSIGNMENT

933 : ATTRIBUTE MAY NOT BE PARAMETERIZED

934 : WRITE ONLY ATTRIBUTE

935 : ATTRIBUTE MUST BE ASSIGNED TO

937 : ILLEGAL MNEMONIC VALUE FOR THE ATTRIBUTE

939 : FILE ATTRIBUTE DEIMPLEMENTED

940 : UNRECOGNIZED FILE ATTRIBUTE

941 : NUMERIC VALUE OF ATTRIBUTE IS OUT OF RANGE

The specified value of a file attribute is out of range.

942 : ATTRIBUTE MAY HAVE ONLY ONE PARAMETER

The file attribute was given more than one parameter. Valid number of parameters is 0 or 1.

943 : FILE ATTRIBUTE CANNOT HAVE TWO PARAMETERS

944 : FILE ATTRIBUTE MUST HAVE ONE OR TWO PARAMETERS

945 : FILE ATTRIBUTE MUST HAVE ONE PARAMETER

946 : NUMERIC ATTRIBUTE EXPECTED

947 : IMPROPER DESIGNATOR NAME

948 : IMPROPER NAME SPECIFIED FOR REPLACEMENT

949 : INPUT HEADER NAME EXPECTED

950 : OUTPUT HEADER NAME EXPECTED

951 : FILE ATTRIBUTE MAY HAVE ONLY ONE OR TWO PARAMETERS

The file attribute was given more than two parameters. Valid number of parameters is 0, 1, or 2.

953 : THE CCSVERSION OF THE DATABASE ITEM DOES NOT MATCH THE CCSVERSION OF THE PROGRAM DATA ITEM

The CCSVERSION specified for the database item does not match the CCSVERSION specified for the program data item.

955 : UNDEFINED SERVICE FUNCTION

956 : MISSING PROGRAM-ID FOR SEQUENTIAL PROGRAMS

The PROGRAM-ID clause was expected but was not found. The PROGRAM-ID clause is required when a single source file contains more than one source program at the outermost level.

957 : MISSING PROGRAM-ID FOR NESTED PROGRAMS

The PROGRAM-ID clause was expected but was not found. The PROGRAM-ID clause is required when source programs are nested within other source programs.

958 : CLAUSE VALID IN LOCAL-STORAGE OR LINKAGE SECTIONS ONLY

A clause was encountered that was invalid in the current context. Use of the data declaration clauses IS INTEGER, IS STRING, BY REFERENCE, and BY CONTENT is restricted to the Local-Storage Section and the Linkage Section.

959 : SYNTAX VALID ONLY WITH STATIONLIST ATTRIBUTE

960 : FREE ONLY FREES DATASETS AND GLOBAL ITEMS

961 : IDENTIFIER MUST BELONG TO A DATABASE

962 : CATEGORY MAY NOT BE REPEATED WITHIN REPLACING PHRASE

963 : INVALID INITIALIZE IDENTIFIER

964 : ILLEGAL CHARACTER

966 : PARAMETER WITH REDEFINES IS ILLEGAL

A parameter cannot have a redefines specification.

967 : STRING OR INTEGER PARAMETER IS ILLEGAL

In CALL USING or DATA DIVISION, an INTEGER or STRING is illegal on this parameter.

968 : BY CONTENT OR BY REFERENCE IS ILLEGAL

In CALL USING or DATA DIVISION, a BY CONTENT or BY REFERENCE is illegal on this parameter.

969 : PARAGRAPH NAME DOES NOT EXIST

A paragraph name was expected but was not found.

970 : SECTION NAME DOES NOT EXIST

A section name was expected but was not found.

971 : DUPLICATE PARAGRAPH NAMES FOUND

Two paragraphs have the same name. The paragraph name must be unique.

972 : IF A USAGE COMP OR USAGE INDEX IS THE FIRST DATA ITEM REFERENCED IN A RENAMES CLAUSE, THAT DATA ITEM MUST BE POSITIONED ON AN 8-BIT CHARACTER BOUNDARY WITHIN THE RECORD.

When data-name-3 is specified and data-name-2 is an elementary COMPUTATIONAL or INDEX data item in a RENAMES clause, it must be positioned to begin at an 8-bit character boundary.

973 : IF A USAGE COMP OR USAGE INDEX APPEARS AS THE LAST DATA ITEM IN A RENAMES DECLARATION, THAT DATA ITEM MUST BE POSITIONED SUCH THAT IT ENDS ON AN 8-BIT CHARACTER BOUNDARY

When data-name-3 is specified and is an elementary COMPUTATIONAL or INDEX data item in a RENAMES clause, it must be positioned to end at the end of an 8-bit character boundary.

974 : OPT1 TABLE OVERFLOW

The total number of data items to be optimized with the OPT1 option cannot exceed 256.

975 : UNIMPLEMENTED USAGE OF IDENTIFIER CONTAINING OPT1

Only numeric items in the Working Storage Section that are DISPLAY, COMPUTATIONAL, or PACKED-DECIMAL can be optimized with the OPT1 option.

976 : THE TADS ON COMMAND RESTRICTS VARIABLES TO A MAXIMUM LENGTH OF 30 CHARACTERS. TO MONITOR LONGER STRING VARIABLES, PLEASE USE THE <SUBSTRING EXPRESSION> CONSTRUCT TO FOCUS ON THE AREA OF INTEREST

977 : PLEASE BE AWARE THAT DURING TADS SESSIONS, OPTIMIZATIONS USED AS A RESULT OF THE OPTIMIZE COMPILER CONTROL OPTION MAY REORDER OR ELIMINATE EXECUTION OF CERTAIN STATEMENTS OR REMOVE POTENTIAL STATEMENT BREAKPOINTS

978 : APPLICATION OF THE OPTIMIZE COMPILER CONTROL OPTION MAY CAUSE UNEXPECTED RESPONSES: STATEMENTS AND THEIR BREAKPOINTS MAY BE MOVED OUT OF ORDER OR EVEN REMOVED

979 : THE SOURCE MUST BE ONE CHARACTER IN LENGTH WHEN THE CHARACTERS BY PHRASE IS USED

The REPLACING CHARACTERS BY option of the INSPECT clause requires that the size of the object of the BY phrase be one character in size.

980 : THE LENGTH OF THE SOURCE MUST BE EQUAL TO THE LENGTH OF THE TARGET

The item data that is being replaced by the INSPECT clause must be the same size as the data item with which it is being replaced.

981 : UNDEFINED CONTROL NAME

The control name referenced in the CONTROL clause is not defined.

982 : REPORT ITEM NOT ALLOWED FOR CONTROL

The control name must not be defined in the Report Section.

983 : UNSIGNED INTEGER EXPECTED

An unsigned integer literal is expected here.

984 : REPORT GROUP TYPE UNDEFINED

The TYPE clause specified the particular type of report group and it must be specified for the Format 1 report group.

985 : FORMAT 1 REPORT GROUP NOT COMPLETED

A Format 1 report group description entry is expected in this context.

986 : FORMAT 2 REPORT GROUP EXPECTED

A Format 2 report group description entry is expected in this context.

987 : FORMAT 3 REPORT GROUP EXPECTED

A Format 3 report group description entry is expected in this context.

988 : DUPLICATE SOURCE CLAUSE FOUND

A Format 3 report group description entry cannot have more than one SOURCE clause.

989 : DUPLICATE VALUE CLAUSE FOUND

A Format 3 report group description entry cannot have more than one VALUE clause.

990 : REPORT GROUP INDICATE NOT ALLOWED

The GROUP INDICATE clause is not allowed for Format 1 or Format 2 report group description entry.

991 : SUM CLAUSE ALLOWED ONLY FOR CONTROL FOOTING REPORT GROUP

The SUM clause is not allowed for Format 1 or Format 2 report group description entry.

992 : REPORT NAME EXPECTED

The report name must be a subject specified in the report description entry.

993 : DUPLICATE LINE CLAUSE

A report group description entry cannot have more than one LINE clause specified.

994 : LINE NUMBER OUT OF RANGE

The line number specified in the LINE clause must not be less than zero (0) and cannot exceed three significant digits in length.

995 : LINE CLAUSE ALREADY DEFINED IN GROUP

A report group description entry cannot have more than one LINE clause specified.

996 : ABSOLUTE LINE NOT ALLOWED WITHOUT PAGE LIMIT

If the PAGE clause is omitted from a given report description entry, only relative LINE NUMBER clauses can be specified in any report group description entry within that report.

997 : ABSOLUTE LINE NOT ALLOWED AFTER RELATIVE

Within a given report group description entry, all absolute LINE NUMBER clauses must precede all relative LINE NUMBER clauses.

998 : LINE NUMBER OUT OF REGION

The LINE NUMBER clause cannot be specified to cause any line of a report group to be presented outside the vertical subdivision of the page designated for that report group type, as defined by the PAGE clause.

999 : LINE NUMBER NOT IN ASCENDING ORDER

Within a given report group description entry, successive absolute LINE NUMBER CLAUSES must specify integers that are in ascending order.

1000 : RELATIVE LINE NOT ALLOWED WITH NEXT PAGE

A LINE NUMBER clause with the NEXT PAGE phrase can appear only in the description of body groups and in a report footing report group.

1001 : NEXT GROUP NOT ALLOWED IN ELEMENTARY ITEM OR IN PH AND RF

The NEXT GROUP clause must not be specified in a report page heading (PH) report group or a report footing (RF) report group.

1002 : COLUMN NUMBER NOT IN ASCENDING ORDER

Within a given print line, the printable items must be defined in ascending column number order such that each printable item defined occupies each unique sequence of contiguous character positions.

1003 : DUPLICATE USING CLAUSE

A data description entry contains two CONVENTION or two LANGUAGE clauses. Only one CONVENTION clause and one LANGUAGE clause are allowed in each data description entry.

1004 : COLUMN NUMBER EXCEED LIMIT

The number specified in the COLUMN NUMBER clause must be greater than or equal to zero and cannot exceed 255.

1005 : DUPLICATE USE BEFORE REPORTING

A report group cannot be referenced in more than one USE statement.

1006 : REPORT GROUP EXPECTED

A report group data item is expected in this context.

1007 : STATEMENT NOT IN USE-BEFORE-REPORTING PROCEDURE

This statement can appear only in a USE BEFORE REPORTING procedure.

1008 : STATEMENT NOT ALLOWED IN USE-BEFORE-REPORTING PROCEDURE

This statement cannot appear in a USE BEFORE REPORTING procedure.

1009 : REPORT CANNOT CONTAIN MORE THAN ONE DETAIL REPORT GROUP

A report name referenced in a GENERATE statement cannot have more than one detail report group.

1010 : MISSING CONTROL CLAUSE IN THE REPORT ENTRY DESCRIPTION

A report name referenced in a GENERATE statement must have a CONTROL clause in the corresponding report description entry.

1011 : INVALID DICTIONARY NAME

1012 : NAME EXCEEDS MAXIMUM ALLOWED FOR DICTIONARY

1013 : UNRECOGNIZED KEYWORD IN DICTIONARY SPECIFICATION

1014 : INVALID DICTIONARY VERSION

1015 : INVOKE DATA NOT ALLOWED

1016 : COMPILER TEMPLATE ERROR

1017 : DEBUGGING OPTION IS NOT VALID UNLESS $FREE IS RESET

1018 : INLINEPERFORM IGNORED DUE TO RESET OPTIMIZE

The INLINEPERFORM compiler control option has been set around a PERFORM statement, but the OPTIMIZE compiler control option has not been set.

1019 : THE LABEL RECORDS ARE OMITTED PHRASE IS NOT APPLICABLE TO DISK OR REMOTE FILES. THE DEFAULT VALUE OF LABEL RECORDS ARE STANDARD WILL BE ASSUMED.

The LABELRECORDS ARE OMITTED phrase is not allowed for DISK or REMOTE files.

1020 : SUM COUNTER EXPECTED

If the identifier referenced in the SUM clause is defined in the Report Section, then the identifier must reference a SUM counter.

1021 : THE POINTER IDENTIFIER MUST BE OF SUFFICIENT SIZE TO CONTAIN A VALUE EQUAL TO 1 PLUS THE SIZE OF THE SOURCE

The data item referenced by this identifier in the WITH POINTER clause must be described as an elementary numeric integer of sufficient size to contain a value equal to 1 plus the size of the data item referenced by identifier-1, which follows the UNSTRING verb.

1022 : THE OPEN STATEMENT FOR A REPORT FILE MUST CONTAIN ONLY THE OUTPUT PHRASE OR THE EXTEND PHRASE

The OPEN statement for a report file must contain only the OUTPUT phrase or the EXTEND phrase.

1023 : THE CLOSE STATEMENT FOR A REPORT FILE CAN ONLY TERMINATE THE PROCESSING OF REEL/UNITS AND FILES WITH OPTIONAL REWIND AND/OR LOCK OR REMOVAL WHERE APPLICABLE

The CLOSE statement for a report file can only terminate the processing of reel/units and files with optional rewind and/or lock or removal where applicable.

1024 : THE DEPENDING ON PHRASE IN THE RECORD CLAUSE IS NOT ALLOWED WITH RELATIVE OR INDEXED FILES UNLESS THE ANSI CCI OPTION HAS BEEN SET PRIOR TO THE FD FOR THIS FILE

An attempt was made to specify a variable length record file for an Indexed or Relative file by using the DEPENDING ON phrase in the record clause. Variable length record files are allowed only for Indexed or Relative files when the ANSI CCI option is set.

1025 : IF A 'USAGE COMP' OR 'USAGE INDEX' IS THE FIRST DATA ITEM REFERENCED IN A RENAMES CLAUSE, THAT DATA ITEM MUST BE POSITIONED ON AN 8-BIT CHARACTER BOUNDARY WITHIN THE RECORD

1026 : IF A 'USAGE COMP' OR 'USAGE INDEX' IS THE LAST DATA ITEM IN A RENAMES DECLARATION, THAT DATA ITEM MUST BE POSITIONED SUCH THAT IT ENDS ON AN 8-BIT CHARACTER BOUNDARY

1027 : ILLEGAL DATA NAME

This data name does not conform to the rules for a data name.

1028 : EDITED ITEM IS NOT ALLOWED

The alphanumeric item or the numeric-edited item specified is invalid in the current context.

1029 : JUSTIFIED ITEM IS NOT ALLOWED

The item with the JUSTIFIED clause specified is invalid in the current context.

1030 : EXTERNAL ITEMS OF THIS USAGE ARE NOT IMPLEMENTED

External variables must be declared with a usage of DISPLAY. Any other usage is not allowed for external variables.

1031 : DUPLICATE USAGE CLAUSE

A data description entry contains two USAGE clauses. Only one USAGE clause is allowed in each data description entry.

1032 : DUPLICATE SIGN CLAUSE

A data description entry contains two SIGN clauses. Only one SIGN clause is allowed in each data description entry.

1033 : DUPLICATE SYNCHRONIZED CLAUSE

A data description entry contains two SYNCHRONIZED clauses. Only one SYNCHRONIZED clause is allowed in each data description entry.

1034 : DUPLICATE JUSTIFIED CLAUSE

A data description entry contains two JUSTIFIED clauses. Only one JUSTIFIED clause is allowed in each data description entry.

1035 : DUPLICATE BLANK WHEN ZERO CLAUSE

A data description entry contains two BLANK WHEN ZERO clauses. Only one BLANK WHEN ZERO clause is allowed in each data description entry.

1036 : DUPLICATE INITIAL VALUE CLAUSE

A data description entry contains two INITIAL VALUE clauses. Only one INITIAL VALUE clause is allowed in each data description entry.

1037 : DUPLICATE EXTERNAL CLAUSE

A data description entry contains two EXTERNAL clauses. Only one EXTERNAL clause is allowed in each data description entry.

1038 : DUPLICATE GLOBAL CLAUSE

A data description entry contains two GLOBAL clauses. Only one GLOBAL clause is allowed in each data description entry.

1039 : DUPLICATE LOWER BOUND CLAUSE

A data description entry contains two LOWER BOUNDS clauses. Only one LOWER BOUNDS clause is allowed in each data description entry.

1040 : NO DATA BASE HAS BEEN INVOKED

1041 : SYNTAX 'KANJI' WILL BE DEIMPLEMENTED—USE 'NATIONAL'

KANJI is being replaced by the word NATIONAL in the USAGE clause of a data description entry. You should change any USAGE IS KANJI clause in a data description entry to USAGE IS NATIONAL to avoid compiler errors in the future when the word KANJI is deimplemented.

1042 : SYNTAX 'NC' WILL BE DEIMPLEMENTED—USE 'N'

The beginning delimiter for a KANJI character, NC, is being replaced by the delimiter N. You should change all NC delimiters to N to avoid compiler errors in the future when NC is deimplemented.

1043 : USING DISPLAY AND NATIONAL CANNOT BE MIXED IN THIS CONTEXT

1044 : NATIONAL LITERAL IS NOT ALLOWED

1045 : ALPHANUMERIC OR NATIONAL DATA ITEM REQUIRED

1046 : 'DELIMITED IGNORED' WILL BE DEIMPLEMENTED—USE 'EXTERNAL-FORMAT FOR NATIONAL' IN ASSIGN CLAUSE

1047 : INVALID SYNTAX: USING

1048 : CANNOT SPECIFY A FILE WITH EXTERNAL-FORMAT FOR NATIONAL OPTION

1049 : NATIONAL CHARACTER ACTUAL OR FORMAL PARAMETER EXPECTS 16 BIT DATA; OTHERWISE THE RESULT IS UNDEFINED

1050 : TASKS CANNOT HAVE NESTED OCCURS

A task variable that includes an OCCURS clause cannot have a subordinate task variable that includes an OCCURS clause. (A task variable is a data description entry declared with the USAGE IS TASK clause.)

1051 : TASKS, EVENTS, AND LOCKS CANNOT BE REDEFINED BY ITEMS OF DIFFERENT USAGE

1052 : TASKS, EVENTS AND LOCKS CANNOT HAVE PICTURE CLAUSES

The PICTURE clause is not valid with data items that have a usage of TASK, EVENT, or LOCK.

1053 : ILLEGAL CLAUSE IN USE WITH TASK, EVENT, OR LOCK IDENTIFIER

1054 : MUST BE LEVEL-1 AND DISPLAY ITEM

1055 : TASK NAME, MYSELF OR MYJOB EXPECTED

You specified incorrect syntax following the word OF in the task-attribute-identifier. Only the task-name, or the words MYSELF or MYJOB are allowed.

1056 : INVALID TASK ATTRIBUTE

The attribute you specified is not a valid task attribute. For a list of valid task attributes, refer to the Task Attributes Programming Reference Manual.

1057 : INVALID TASK ATTRIBUTE MNEMONIC

1058 : MUST BE PORT, REMOTE, OR PRINTER FILE

1059 : RENAMES DATA NOT ALLOWED

1060 : ILLEGAL USAGE FOR 01 RECORD IN FD

1061 : THIS LANGUAGE FEATURE IS NOT IMPLEMENTED—IT WILL BE IGNORED

1062 : A REDEFINED PARAMETER IS ILLEGAL

To prevent data corruption, a redefined data item cannot be used as a parameter for identifier-2 in the CALL statement with the ON OVERFLOW option. A redefined data item is a data item declared in the File Section with a REDEFINES clause (an explicit redefinition), or a subordinate of a data item declared with a REDEFINES clause (an implicit redefinition).

1063 : A PARAMETER FROM AN IMPLICIT REDEFINITION IS ILLEGAL

To prevent data corruption, a redefined data item cannot be used as a parameter for identifier-2 in the CALL statement with the ON OVERFLOW option. This rule includes implicit redefinitions. An implicit redefinition occurs when the first data item declared in the File Section is followed by subsequent level-01 items. The subsequent level-01 items are considered to be implicit redefinitions of the first item.

1065 : ILLEGAL USE OF EXTERNAL CLAUSE

1066 : BYFUNCTION IS NOT PERMITTED WITH A LIBRARY TITLE

BYFUNCTION is not permitted in the CALL statement literal when a file title is specified.

1067 : INVALID LIBRARY ACCESS, BYTITLE OR BYFUNCTION EXPECTED—BYTITLE ASSUMED

BYTITLE or BYFUNCTION is expected following the library-name in the CALL statement. BYTITLE is assumed.

1068 : FAMILYNAME EXPECTED

A valid family name was not specified following the "ON" clause of the library file title in the CALL literal statement.

1069 : MISSING LIBRARY TITLE OR FUNCTIONNAME

A library title or function name was expected following the "entrypoint IN/OF" clause in the CALL literal statement.

1070 : THE 'AT END' OR 'NOT AT END' PHRASE IS EXPECTED

Either the "AT END" or the "NOT AT END" phrase is expected for sequential access mode.

1071 : THE 'INVALID KEY' OR 'NOT INVALID KEY' CONDITION IS EXPECTED

Either the "INVALID KEY" or "NOT VALID KEY" phrase is expected for random access mode.

1072 : INCORRECT NUMBER OF PARAMETERS SPECIFIED FOR THE INTRINSIC FUNCTION

Refer to Section 9 for the syntax of the intrinsic functions.

1073 : INVALID INTRINSIC FUNCTION

The intrinsic function you specified is not valid. For a list of intrinsic functions and their syntax, refer to Section 9.

1074 : CLASS OF PARAMETER IS UNDETERMINED

1075 : CLASS OF PARAMETER MUST BE ALPHABETIC

The parameter in your syntax must be of the alphabetic class. To belong to the alphabetic class, the parameter must be a data item with the letter A in its PICTURE clause. Refer to Section 4 for details about classes of data items and for a discussion of the PICTURE clause.

1076 : CLASS OF PARAMETER MUST BE ALPHANUMERIC

The operand in your syntax was not of the alphanumeric class. To belong to the alphanumeric class, the parameter must be a data item of the alphanumeric, alphanumeric-edited, or numeric-edited category. The category of a data item is declared in the PICTURE clause. Refer to Section 4 for details.

1077 : CLASS OF PARAMETER MUST BE NATIONAL

The parameter in your syntax was not of the national class. To belong to the national class, the parameter must be a data item with the letter N in its PICTURE clause. Refer to Section 4 for details.

1078 : CLASS OF PARAMETER MUST BE NUMERIC

The parameter in your syntax was not of the numeric class. To belong to the numeric class, the parameter must be a data item with only the symbols 9, P, S, and V in its PICTURE clause. Refer to Section 4 for details.

1079 : CLASS OF PARAMETER MUST BE ALPHABETIC, ALPHANUMERIC, OR NATIONAL

1080 : CLASS OF PARAMETER MUST BE ALPHABETIC OR ALPHANUMERIC

1081 : PARAMETER MUST BE ONE CHARACTER IN LENGTH

1082 : NON-SEQUENTIAL FILE OPENED WITH REVERSE

The REVERSED option of the OPEN statement is valid only for files with sequential organization.

1083 : NEXT NOT ALLOWED

1084 : MUST BE UNSIGNED DISPLAY NUMERIC, FIRST 4 BYTES

1085 : FILE HAS NO LINAGE COUNT

1086 : INVALID CASE VALUE

1087 : INVALID WRITE OPTION

1088 : THIS RECORD-NAME IS ILLEGAL

1089 : MISSING RECORD DESCRIPTION

1090 : ELEMENTARY TASK DATA ITEM REQUIRED

1091 : ALPHANUMERIC OR ELEMENTARY TASK DATA ITEM REQUIRED

1093 : FILE, TASK, MYSELF OR MYJOB EXPECTED

1095 : EXTERNAL FORMAT FOR NATIONAL MUST BE SPECIFIED FOR THE WRITE FILE OPTION

The IS EXTERNAL FORMAT FOR NATIONAL clause must be specified in the file description entry of the Environment Division for the file you want to write with the WRITE FILE statement. The IS EXTERNAL FORMAT FOR NATIONAL clause is discussed in Section 3. The WRITE statement is described in Section 8.

1100 : NUMERIC LITERAL MUST BE BETWEEN 1 AND 65535

A numeric literal is smaller or larger than the allowed limits.

1101 : THIS COBOL85 CONSTRUCT NOT ALLOWED IN A COBOL85 TADS SESSION

Only certain COBOL85 constructs can be entered during a TADS session. Consult the COBOL85 language documentation or the COBOL85 TADS documentation.

1102 : INVALID ERROR MESSAGE NUMBER

The most likely cause of this message is a version mismatch between the COBOL85 compiler and the SLICESUPPORT library. This indicates some problem with installing COBOL85, because SIMPLEINSTALL prevents such a mismatch.

1103 : OPT1 GROUP DATA ITEM CANNOT BE REDEFINED

1104 : ATTRIBUTE BUFFERSHARING MAY ONLY BE SPECIFIED FOR ORGANIZATION SEQUENTIAL FILE

1105 : CANNOT CHANGE FILE ATTRIBUTES TO TASK ATTRIBUTES AND VICE VERSA

1106 : CALLMODULE CCI NOT SET, NOT CODE GENERATED FOR EXIT MODULE

1107 : UNSUPPORTED INTMODE VALUE

Typically this would be caused by declaring the first 01 record under an FD as a DOUBLE or BINARY item large enough that it would be stored in a double operand.

1108 : TOO MANY DISPLAY OPTIONS

The DISPLAY command contained too many options. You must specify only one of the HEX, DECIMAL, or EBCDIC options. This is a TADS error message.

1109 : DUPLICATE GROUP/ITEM SPECIFICATION

The DISPLAY command contained a duplicate option specification. One of the options ITEMS, GROUP ITEMS, ELEMENTARY ITEMS, GROUPS, HEX, EBCDIC, or DECIMAL appears more than once in the command specification. This is a TADS error message.

1110 : MISSING WORD "ITEMS"

The DISPLAY command is missing the word ITEMS in the command specification. You may specify GROUPS or GROUP ITEMS, but not GROUP alone. This is a TADS error message.

1111 : INVALID DISPLAY OPTION

The DISPLAY command contains an invalid DISPLAY option. The valid display options include ITEMS, GROUP ITEMS, ELEMENTARY ITEMS, GROUPS, HEX, EBCDIC, and DECIMAL. This is a TADS error message.

1112: THIS COBOL85 TADS CONSTRUCT IS NOT ALLOWED IN A COBOL85 PROGRAM

The compiler encountered the options of DISPLAY command that are valid only from a TADS session at compile time. The options ITEMS, GROUP ITEMS, ELEMENTARY ITEMS, GROUPS, HEX, EBCDIC, and DECIMAL are not allowed outside of the TADS environment.

1113 : INVALID USE OF INDEX

Index names cannot be mixed with integers or data names in an expression that is being used in an arithmetic statement. When used in a subscript, an index name must be in the form: “index-name [+/- integer]”.

1114 : THE GROUPS/GROUP ITEMS DISPLAY OPTION MAY NOT BE APPLIED TO ELEMENTARY ITEMS

The GROUPS and GROUP ITEMS clauses do not apply to elementary items. By definition, elementary items contain no group leveled items. As such, any attempt to apply these options to elementary items is considered to be an error.

1115 : THE DECIMAL DISPLAY OPTION MAY NOT BE APPLIED TO NONNUMERIC DATA ITEMS

The DISPLAY option only applies to numeric data items.

1116 : UNABLE TO COMPLETE THE REQUESTED DISPLAY, THE OPTIONS NAMED GENERATE TOO MANY DISPLAY ELEMENTS

The requested display exceeded a software boundary. Request a display that generates fewer display elements.

1117 : UNABLE TO COMPLETE THE REQUESTED DISPLAY, THE NAMED ITEM HAS TOO MANY SUBORDINATE ITEMS

The limit for subordinate items is 4095. Display subordinate items separately.

1118 : TRAVERSAL OF SUBSCRIPTED GROUP LEVEL ITEMS IS NOT SUPPORTED

This error message occurs in response to a statement similar to the following:

DISPLAY <identifier> AS ITEMS (or GROUPS, GROUP ITEMS, ELEMENTARY ITEMS) where <identifier> is a well-formed, subscripted, group-level data item.

Non-numerical Compiler Output Messages

The following is an alphabetical listing of the non-numerical output messages that are generated by the COBOL compiler. An explanation of each message and, where applicable, the corrective action you should take, are provided.

$ MERGE option may only occur once.

- You can use the MERGE compiler control option only once in a compilation.

$ option name is limited to 31 characters.

- A user-defined compiler control option can contain a maximum of 31 characters. Extra characters are truncated.

$ option not allowed inside INITIALCCI file.

- The compiler control option in error is not allowed inside an INITIALCCI file.

$BINDSTREAM must be set

- The BINDSTREAM control option must be set in the program.

A fault occurred while scanning the display form title—<filename>

- The specified file name is not valid.
- Correct the file name and recompile the program.

A name node was expected between or after slashes in the filename—<filename>.

- The file name is invalid either because it contains two consecutive slashes (//), or it ends with a slash (/).
- Correct the file name and recompile the program.

A non-alphanumeric character was found in the familyname for the filename—<filename>

- A family name must contain only uppercase alphanumeric characters.
- Correct the file name and recompile the program.

A null quoted string is illegal as an identifier in the filename—<filename>

- The file name might contain invalid characters. A filename node can include hyphens (-) and underscores (_). Other nonalphanumeric characters can be included if they are enclosed in quotation marks (").
- Correct the file name and recompile the program.

A right parenthesis was expected after the usercode for the filename—<filename>.

- An identifier that represents a usercode must be enclosed in parentheses. The usercode can contain hyphens (-) and underscores (_). Other nonalphanumeric characters can be included if they are enclosed in quotation marks.
- Correct the file name and recompile the program.

A slash was expected between successive identifiers in the filename—<filename>.

- The file name might contain invalid characters. A filename node can include hyphens (-) and underscores (_). Other nonalphanumeric characters can be included if they are enclosed in quotation marks (").
- Correct the file name and recompile the program.

A usercode was expected after the left parenthesis in the filename—<filename>.

- An identifier that represents a usercode must be enclosed in parentheses. The usercode can contain hyphens (-) and underscores (_). Other nonalphanumeric characters can be included if they are enclosed in quotation marks (").
- Correct the file name and recompile the program.

An identifier that contains a non-alphanumeric character was not enclosed in quotation marks for the filename—<filename>.

- The file name might contain invalid characters. A filename node can include hyphens (-) and underscores (_). Other nonalphanumeric characters can be included if they are enclosed in quotation marks (").
- Correct the file name and recompile the program.

Bound codefiles are not TADS capable.

- The source file specified both the TADS compiler control option and the BINDINFO, LEVEL, or LIBRARY compiler control option.
- This warning indicates that you can initiate a TADS session on an object file with bind information (BINDINFO, LEVEL, or LIBRARY), but you cannot initiate a TADS session on an object file generated by the Binder.

Cannot be referenced as a subordinated option.

- Class options are the only options that can have suboptions. The compiler control option in error is not a Class option. Therefore, it has no suboptions and cannot be referenced as a subordinated option.

Cannot be used within the OPTION (...) clause.

- Only user-defined Boolean options can be specified with the OPTION compiler control option.

Cannot optimize with $OPTIMIZE (LEVEL) > 5 when DMS statements exist. Compilation will continue with $OPTIMIZE (LEVEL) = 5.

- When $OPTIMIZE (LEVEL) is greater than 5, the optimizer attempts to merge the offset of the beginning of the array with the offset within the array. This cannot be accomplished when DMS statements exist, thus the $OPTIMIZE (LEVEL) is set to level 5.

CODE VERSION MISMATCH in the following software.

- The version of the listed software does not match the version of the software currently running. Using different versions together could produce unwanted results.
- Ensure that both versions of the software match, and then restart the compilation.

COMPILER ERROR <sequence number> (<internal number>): LIBRARY_INFO table size of <number> exceeded.

- The source file contains too many library declarations, too many procedures declared as library entry points, or too many characters in strings associated with the libraries (titles, intnames, and so on).
- Reduce the above limits, or split the source file into smaller files.

COMPILER ERROR <sequence number> (<internal number>): LIBRARY_PARAMS table size of <number> exceeded at <function name>.

- The source file contains too many parameters in the procedures declared as library entry points.
- Reduce the number of parameters or the complexity of the parameters.

COMPILER ERROR <sequence number> (<internal number>): LIBRARY template array size of <number> exceeded for library at Address Couple = (<stack address>).

- The source file contains too many library declarations, too many procedures declared as library entry points, or too many characters in strings associated with the libraries (titles, intnames, and so on).
- Reduce the above limits, or split the source file into smaller files.

COMPILER ERROR <sequence number> (<internal number>): Maximum number of declared_temp entries (<number>) exceeded.

- The statement, possibly with its adjoining statements, requires too many temporaries to be compiled.
- Simplify the statement by moving subexpressions out as separate statements.

COMPILER ERROR <sequence number> (<internal number>): Maximum number of info entries (<number>) exceeded.

- The source file has too many identifiers to be compiled.
- Reduce the number of identifiers, or split the file into smaller files.

COMPILER ERROR <sequence number> (<internal number>): Maximum number of label table entries exceeded.

- The generated code has too many branches.
- Reduce the complexity, or split the source file into smaller files.

COMPILER ERROR <sequence number> (<internal number>): Maximum number of real_set entries (<number>) exceeded.

- The statement, possibly with its adjoining statements, requires too many temporaries for the statement to be compiled.
- Simplify the statement by moving subexpressions out as separate statements.

COMPILER ERROR <sequence number> (<internal number>): Maximum number of str table entries exceeded.

- Too many string constants exist in the source file.
- Reduce the number of string constants by combining string constants together or by splitting the source file.

COMPILER ERROR <sequence number> (<internal number>): Maximum number of symbol entries (<number>) exceeded.

- The source file has too many characters in its identifiers for the file to be compiled.
- Reduce the length of the identifiers, reduce the number of identifiers, or split the file into smaller files.

COMPILER ERROR <sequence number> (<internal number>): Maximum number of twig table entries exceeded.

- The source file contains too many occurrences of variables, constants, or operators.
- Reduce the number of variables, constants, or operators; or split the source file into smaller files.

COMPILER ERROR <sequence number> (<internal number>): Maximum number of txt table entries exceeded.

- Too many characters exist in all the string constants in the source file.
- Reduce the number of characters by removing string constants, making the strings shorter, or splitting the source file.

COMPILER ERROR <sequence number> (<internal number>): Too many library entry points (<number>) in library at Address Couple = (<stack address>).

- The source file contains too many procedures declared as library entry points.
- Reduce the number of procedures, or split the source file into smaller files.

COMPILER ERROR <sequence number> (<internal number>): Too much statistics information needed.

- The $$ STATISTICS compiler control option is set for too many procedures or blocks for the file to be compiled.
- Reduce the number of procedures or blocks that have the $$ STATISTICS compiler control option set, or split the source file into smaller files.

COMPILER ERROR <sequence number> (<internal number>): Type Stack Overflow—Probably a reentrant expression tree.

- The source file contains an expression that is too complex.
- Split the expression into smaller expressions. Each smaller expression should be a separate statement.
- Specify the SHARING option as PRIVATE, SHAREDBYRUNUNIT, or DONTCARE.

Conflicts with prior $BINDER_MATCH option with same first string but different second string.

- You specified two $BINDER_MATCH options, using the same name for the first strings but different values for the second strings.
- You can correct this error by ensuring either that both the values are identical or that the names are different.

Dag_Value table maximum entries exceeded

- The program is too large to compile with the OPTIMIZE compiler control option set.
- Remove $SET OPTIMIZE from the user program.

DELETE or VOIDT may only be used on $ cards from the CARD file.

- The DELETE and VOIDT options are valid only within the primary input file (CARD).
- Remove all occurrences of DELETE and VOIDT outside the primary input file (CARD), and recompile.

$ELSE or $END seen, but no $IF has been seen.

- The compiler control option $ELSE or $END was encountered with no previous $IF. The control option will be ignored.

Enumerated option must be compared equal (= or ==) or not equal (!= or ^= or <>) to enumerated constant.

- Within a Boolean expression in a compiler control image, an enumerated option can be compared only for equality or inequality with one of its enumerated constants.

ERROR LIMIT OF <error limit> HAS BEEN EXCEEDED. COMPILATION WILL BE TERMINATED.

- Either the error limit specified by the ERRORLIMIT option or the default error limit set by the compiler has been exceeded.
- You can increase or decrease the number of errors allowed before a compilation is terminated by using the ERRORLIMIT option.

GET_CONSTANT only handles logical, integer, and real types.

- An invalid parameter was passed to an internal compiler procedure.
- You might be able to resolve this problem by correcting any errors that occurred and then restarting the compiler.

GET_CONSTANT parameter not constant.

- An internal compiler procedure received an invalid parameter instead of a constant parameter.
- You can resolve this problem by correcting any errors that occurred and then restarting the compiler.

Illegal compatibility TARGET list or missing parenthesis. Code will be produced for TARGET = LEVEL0.

- The target list that specifies the secondary system is missing a comma or a right parenthesis, or contains an invalid target identifier.

Illegal enumeration value.

- The enumeration value for this compiler control option is either invalid or should be a value less than one.

Illegal option in $ card.

- No information was found within the parentheses used with the current option.
- Either include a value within the parentheses or delete the parentheses, and restart the compilation.

Illegal or unrecognized $ option in expression.

- Either an illegal compiler control option was found within an expression or the members of the expression are not valid.
- Correct the syntax and restart the compilation.

Illegal use of OPTION declarator.

- You incorrectly specified the OPTION compiler control option.
- Check the OPTION syntax in your COBOL manual. Then correct your file and restart the compiler.

Illegal INCLUDE syntax.

- The INCLUDE compiler control option is either incomplete or incorrect.
- Check the INCLUDE syntax in your COBOL manual. Then correct your file and restart the compiler.

Illegal TARGET value in $ card.

- The value you used in the TARGET option is not recognized by the current system.
- Check the TARGET syntax in your COBOL manual. Then correct your file and restart the compiler.

Illegal VERSION option syntax in $ card.

- The VERSION option syntax is incomplete or incorrect.
- Check the VERSION syntax in your COBOL manual. Then correct your file and restart the compiler.

In-line of PERFORM not done.

- A PERFORM statement was worded as an in-line PERFORM, but the statement's logic did not allow it to be performed in line.

  A PERFORM statement cannot be in-lined unless all possible paths of execution from the first statement being performed end up at the last statement being performed. For example, if a GO TO statement in a group of statements being performed specifies a paragraph that is out of the range of the PERFORM statement, the PERFORM statement will not be in-lined.

  In addition, a nested PERFORM statement prevents an outer PERFORM statement from being in-lined. However, the Optimizer will attempt to automatically in-line any nested PERFORM statements if you directly or indirectly specify the $INLINEPERFORM option for the outer PERFORM statement.
- Check the PERFORM statement syntax in your COBOL manual. Then correct your file and restart the compiler.

Integer option must be compared (= , != or ^= or <>, <, <= , >, >=) to integer constant or option.

- Within a Boolean expression in a compiler control image, an integer option can be compared only with another integer option or an unsigned integer constant. Comparison can be for the following:
  - Equality (=)
  - Inequality (^= or != or <>)
  - Less than (<)
  - Less than or equal to (<=)
  - Greater than (>)
  - Greater than or equal to (>=)

Internal assertion is false.

- Unexpected results were discovered internally in the compiler.
- Correct any errors and restart the compiler. If this message appears again, contact your customer support representative.

<sequence number> Invalid OPTION Mnemonic: <name>.

- The source file specifies an invalid mnemonic for the OPTION task attribute.
- You should specify a correct mnemonic for the OPTION task attribute. Refer to the Task Attributes Programming Reference Manual for a list of valid mnemonics for the OPTION task attribute.

Invalid TARGET value in $ card. Code will be produced for TARGET = LEVEL0.

- The processor specified in the TARGET compiler control option is not a recognized processor. The compilation is continuing with the default value of TARGET=LEVEL0.

It is illegal to have nested parenthesized option lists.

- Multiple subordinate options or strings are not allowed within a list that is in parentheses.

Line <sequence number>: badly formed library name or title.

- The source file contains a library declaration in which the library name or title used is not a legal name or title. A legal library name is one that is 17 letters or digits long, or less. A legal library title is a legal file title.

Line <sequence number>: deimplemented attribute.

- The source file specifies an attribute that is no longer implemented.
- Remove the specified attribute, and if possible, replace it with a newer attribute.

Line <sequence number>: the code segment has exceeded the maximum row size of the CODE file. The AREASIZE attribute of the CODE file should be increased to at least <number>.

- The source file needs a larger AREASIZE attribute specified for the CODE file for the source file to compile.
- To specify the AREASIZE attribute, use the CANDE COMPILE command or the WFL COMPILE statement.

Line <sequence number>: the data segment has exceeded the maximum row size of the CODE file. The AREASIZE attribute of the maximum row size should be increased to at least <number>.

- The source file needs a larger AREASIZE attribute specified for the CODE file to compile.
- To specify the AREASIZE attribute, use the CANDE COMPILE command or the WFL COMPILE statement.

Line <sequence number>: the program has exceeded the maximum D<number> size of <number>.

- Too many variables, procedures, or constants exist in the source file. If the D level is 1, then too much code or too many constants exist. If the D level is 2, then too many global variables or procedures, or both, exist. If the D level is 3 or greater, too many local variables or procedures, or both, exist.
- Reduce the number, or split the source file into smaller files.

Maximum number of characters in $BINDER_MATCH options exceeded.

- The number of characters in the string parameters of the $BINDER_MATCH option exceeds the internal limit of 10,000 characters.
- Either remove some of the options or shorten some of the strings.

Maximum number of $BINDER_MATCH options exceeded.

- The number of $BINDER_MATCH options exceeded the internal limit of 200.
- Remove some of the $BINDER_MATCH options.

Missing code file title in the LIBRARY option

- The code file title must be specified in the LIBRARY control option.

Missing comma ',' in $ option expression.

- A comma is the next token required, but it was not found.
- Check the syntax of the $BINDER_MATCH option, ensuring that each string is separated by a comma.

Missing left parenthesis '(' in $ option expression.

- A left parenthesis is the next token required, but it was not found.
- Check the syntax of the $BINDER_MATCH option, ensuring that a left parenthesis precedes the first string and a right parenthesis follows the last string.

Missing right parenthesis after subfile specification in INCLUDE.

- You omitted the right parenthesis that must follow the specification of a subfile name in an INCLUDE option.

Missing right parenthesis ')' in $ option expression.

- A left parenthesis was found in the compiler control record expression, but no matching right parenthesis was found.

More than 12 nodes occurred in the filename—<filename>.

- The file name is invalid because it contains more than 12 name nodes.
- Correct the file name and recompile the program.

More than 17 characters occurred in a name node for filename—<filename>.

- A name node cannot contain more than 17 characters. Be sure to separate each node with a slash (/).
- Correct the file name and recompile the program.

NEWSOURCE sequence error.

- The sequence of the NEW file being created is out of order. The record being processed has a sequence value less than that of the last record placed in the NEW file.

No identifiers were found in the filename—<filename>.

- The file name might contain invalid characters. A filename node can include hyphens (-) and underscores (_). Other nonalphanumeric characters can be included if they are enclosed in quotation marks (").
- Correct the file name and recompile the program.

Non-fatal COMPILER ERROR <sequence number> (<internal number>): Cannot find library directory. No functions will be unrolled. This may be caused by the inability to decode D2 stack building code. Stopped reading code at (<stack address>) (#<number>).

- The $$ OPTIMIZE (SET UNRAVEL) compiler control option was set to request that certain function calls be replaced by in-line code. The function that is indicated cannot be regenerated in-line because the support library is not in its normal form.
- This is a warning only and can be ignored.

Non-fatal COMPILER ERROR <sequence number> (<internal number>):<function name> cannot be unrolled. Cheap blocks cannot be unrolled.

- The $$ OPTIMIZE (SET UNRAVEL) compiler control option was set to request that certain function calls be replaced by in-line code. The function that is indicated cannot be regenerated in-line because it contains a "cheap block."
- This is a warning only and can be ignored.

Non-fatal COMPILER ERROR (sequence number> (<internal number>):<function name> cannot be unrolled; missing from Support Library.

- The $$ OPTIMIZE (SET UNRAVEL) compiler control option was set to request that certain function calls be replaced by in-line code. The function that is indicated cannot be regenerated in-line because it is not found in the support library.
- This is a warning only and can be ignored.

Non-fatal COMPILER ERROR <sequence number> (<internal number>):<function name> cannot be unrolled. Only functions can be unrolled.

- The $$ OPTIMIZE (SET UNRAVEL) compiler control option was set to request that certain function calls be replaced by in-line code. The function that is indicated cannot be regenerated in-line because it is not a normal function.
- This is a warning only and can be ignored.

Non-fatal COMPILER ERROR <sequence number> (<internal number>):<function name> cannot be unrolled. Only Level 3 : Libraries may be unrolled. This Support Library is level <number>.

- The $$ OPTIMIZE (SET UNRAVEL) compiler control option was set to request that certain function calls be replaced by in-line code. The function indicated cannot be regenerated in-line because the support library is not in its normal form.
- This is a warning only and can be ignored.

Non-fatal COMPILER ERROR <sequence number> (<internal number>):<function name> cannot be unrolled. This function has too many locals.

- The OPTIMIZE (SET UNRAVEL) compiler option was set to request that certain function calls be replaced by in-line code. The function that is indicated cannot be regenerated in-line because it has too many local declarations.
- This is a warning only and can be ignored.

Non-fatal COMPILER ERROR <sequence number> (<internal number>): Compiler inconsistency while unrolling <function name>.

- The $$ OPTIMIZE (SET UNRAVEL) compiler control option was set to request that certain function calls be replaced by in-line code. The function that is indicated cannot be regenerated in-line because it contains unrecognized code.
- This is a warning only and can be ignored.

Number of file fragments in virtual input file exceeds the maximum allowed.

- An internal compiler limit has been exceeded.
- You can resolve this error by resequencing the source program or by lowering the number of INCLUDE files.

Numbers are limited to 11 digits on $ cards.

- A maximum of 11 digits is allowed in a compiler control record.
- Eliminate extra digits and restart the compilation.

OPTIMIZER ERROR --- A fixed limit prevents optimization of this program. Recompile with $OPTIMIZE(LEVEL=0), or without $OPTIMIZE

- The program is too large to compile with the OPTIMIZE compiler control option set.
- Remove $SET OPTIMIZE from the user program.

OPTIMIZER ERROR --- Block table maximum entries exceeded

- The program is too large to compile with the OPTIMIZE compilier control option set.
- Remove $SET OPTIMIZE from the user program.

OPTIMIZER ERROR --- Dag table maximum entries exceeded

- The program is too large to compile with the OPTIMIZE compiler control option set.
- Remove $SET OPTIMIZE from the user program.

OPTIMIZER ERROR --- Dependency table maximum entries exceeded

- The program is too large to compile with the OPTIMIZE compiler control option set.
- Remove $SET OPTIMIZE from the user program.

OPTIMIZER ERROR --- Expensive_Data table maximum entries exceeded

- The program is too large to compile with the OPTIMIZE compiler control option set.
- Remove $SET OPTIMIZE from the user program.

OPTIMIZER ERROR --- Induc_Var table maximum entries exceeded

- The program is too large to compile with the OPTIMIZE compiler control option set.
- Remove $SET OPTIMIZE from the user program.

OPTIMIZER ERROR --- Info table maximum entries exceeded

- The program is too large to compile with the OPTIMIZE compiler control option set.
- Remove $SET OPTIMIZE from the user program.

OPTIMIZER ERROR --- Linear_Exp table maximum entries exceeded

- The program is too large to compile with the OPTIMIZE compiler control option set.
- Remove $SET OPTIMIZE from the user program.

OPTIMIZER ERROR --- Link_Stack table maximum entries exceeded

- The program is too large to compile with the OPTIMIZE compiler control option set.
- Remove $SET OPTIMIZE from the user program.

OPTIMIZER ERROR --- Link table maximum entries exceeded

- The program is too large to compile with the OPTIMIZE compiler control option set.
- Remove $SET OPTIMIZE from the user program.

OPTIMIZER ERROR --- Real_Vector table maximum entries exceeded

- The program is too large to compile with the OPTIMIZE compiler control option set.
- Remove $SET OPTIMIZE from the user program.

OPTIMIZER ERROR --- Statement table maximum entries exceeded

- The program is too large to compile with the OPTIMIZE compiler control option set.
- Remove $SET OPTIMIZE from the user program.

OPTIMIZER ERROR --- There is not enough space for a new block allocated by the optimizer.

- The program is too large to compile with the OPTIMIZE compiler control option set.
- Remove $SET OPTIMIZE from the user program.

OPTIMIZER ERROR --- Twig table maximum entries exceeded

- The program is too large to compile with the OPTIMIZE compiler control option set.
- Remove $SET OPTIMIZE from the user program.

OPTIMIZER ERROR --- Vector table maximum entries exceeded

- The program is too large to compile with the OPTIMIZE compiler control option set.
- Remove $SET OPTIMIZE from the user program.

OPTIMIZER ERROR --- V_Info table maximum entries exceeded

- The program is too large to compile with the OPTIMIZE compiler control option set.
- Remove $SET OPTIMIZE from the user program.

Option can appear only as first item on $ card.

- Some compiler control cards are required to be the first option on a particular record.
- Place the option in error in a separate record as the first compiler control option.

OPTION_NAME array has become inconsistent.

- This compiler error occurred because the compiler stored an option in one of its internal tables under an unrecognized type.
- Verify the syntax of your compiler control options and correct the syntax, if necessary. If this error message appears again, contact your customer support representative.

Previously undefined $ option.

- This option is not a recognized compiler control option, so it will be treated as a user-defined option.

Prior phase must be last item on $ card.

- This compiler control option must be the only option on the last record of the $ card. No other options can follow it.

Statement linkage incorrect.

- An internal compiler procedure received incorrect data.
- Correct any errors and restart the compiler. If this message appears again, contact your customer support representative.

String assignment (=) or update (+=) operator expected after string-valued option.

- Either you used an invalid option or the option syntax is incorrect. An operator (=) or an update operator (+=) must follow a string-valued compiler control option.

String constant expected.

- When specifying the string-valued compiler control option, you either omitted the string or used an invalid string.

String too long.

- The length of one of the strings for the $BINDER_MATCH option exceeded an internal limit. The maximum length for a string is 255 characters.
- Shorten the string that exceeds 255 characters.

Subfile specification in INCLUDE must be a string or is missing.

- The symbolic name is missing from the INCLUDE option.
- Specify the symbolic name as a string following the left parenthesis in the INCLUDE option, and then restart the compiler.

"tbl" internal optimizer table maximum entries exceeded

- The program is too large to compile with the OPTIMIZE compiler control option set.
- Remove $SET OPTIMIZE from the user program.

The $ option <option name> is assumed to be a user defined option.

- This option is not a recognized compiler control option, so it will be treated as a user-defined option.

The combined space required for data items declared in <function name> (<number> words) exceeds the maximum available space of <number> words. Check MEMORY_MODEL compiler control option.

- The local variables declared in the indicated function require more space than can be supported using the current MEMORY_MODEL compiler control option.
- Reduce the size requirements of the local variables. If the memory model is tiny or large, increase the LONGLIMIT compiler control option or use a larger memory model.

The combined space required for variable length arguments to <function name> (<number> words) exceeds the maximum available space of <number> words. Check MEMORY_MODEL compiler control option.

- A call on the indicated function requires more space for the variable number of arguments than can be supported using the current MEMORY_MODEL compiler control option.
- Reduce the size of the arguments passed as the variable number of arguments. If the memory model is tiny or large, increase the LONGLIMIT compiler control option or use a larger memory model.

The combined space required for data items declared in the outer block (<number> words) exceeds the maximum available space of <number> words. Check MEMORY_MODEL compiler control option.

- The global variables require more space than can be supported using the current MEMORY_MODEL compiler control option.
- Reduce the size requirements of the global variables. If the memory model is tiny or large, increase the LONGLIMIT compiler control option or use a larger memory mode.

The compiler has faulted with fault number <fault number>.

- You can determine the cause of the error by looking up the fault number in the *ALGOL Programming Reference Manual, Volume 1.*

The cycle delta is restricted to 3 digits.

- The cycle delta that you specify with the VERSION compiler control option can be a maximum of 3 digits.
- Eliminate extra digits from the cycle delta and restart the compilation.

The cycle number is restricted to 3 digits.

- The cycle number that you specify with the VERSION compiler control option can be a maximum of 3 digits.
- Eliminate extra digits from the cycle number and restart the compilation.

The file title did not terminate with a period for the filename—<filename>.

- The file name must end with a period (.).
- Correct the file name and recompile the program.

The interface link to the dictionary has failed. FUNCTIONNAME = <library functionname>.

- Check that the function name of the dictionary library exists. The function name for the dictionary is specified in the SPECIAL-NAMES paragraph.

The LIBRARY and SEPARATE option may not be on at the same time.

- You cannot set the $LIBRARY and $SEPARATE options at the same time.
- Ensure that only one of the options is set.

The library named could not be found. The default name will be used. Library name = <library name>.

- The specified library name is not recognized as a valid string by this compiler control option.

The maximum code file size of 262,144 disk sectors (47,185,920 bytes) has been exceeded.

- The maximum allowable code file size is 262,144 disk sectors (47,185,920 bytes). If the code file cannot be reduced in size, the compiler cannot produce a code file. The size of a code file increases if the following compiler options have been set: BINDINFO, LINEINFO and TADS. Resetting one or all of these compiler options might allow the compiler to produce a code file.

The maximum number of different INCLUDE files (509) has been exceeded. INCLUDE ignored.

- The internal compiler limit of 509 files per program has been exceeded. The compiler will continue without performing the INCLUDE procedure.

The maximum number of different physical files (508) has been exceeded. MERGE ignored.

- The internal compiler limit of 508 files has been exceeded. The compiler will continue without performing the merge.

The maximum number of user defined dollar options has been exceeded.

- Reduce the number of included files and restart the compilation.

The number of NEWIDS has exceeded the limit. The last ID will be ignored.

- Eliminate some of the occurrences of the $NEWID option within the source program, and restart the compilation.

The patch delta is restricted to 4 digits.

- The patch delta that you specify with the VERSION compiler control option can contain a maximum of 4 digits.
- Eliminate extra digits from the patch delta and restart the compilation.

The patch number is restricted to 4 digits.

- The patch number that you specify with the VERSION compiler control option can contain a maximum of 4 digits.
- Eliminate extra digits from the patch number and restart the compilation.

The pointer identifier must be of sufficient size to contain a value equal to 1 plus the size of the source.

- The data item referenced by identifier-7 in the UNSTRING statement is too small. The data item must be large enough to contain a value equal to its size plus 1.
- Increase the size of the data item to be equal to its size plus 1. For more information about this data item, refer to the explanation of the UNSTRING statement syntax in Section 6.

The symbolically named region from the $ INCLUDE card could not be found.

- You omitted either the $COPYBEGIN option, which must precede the symbolically named region, or the $COPYEND option, which must follow the symbolically named region.

The value of $TADS(RESOURCE) must be between 20 and 2000 inclusive. It will be set to <integer>.

- The value of $TADS(RESOURCE) was set too high or low.

The version delta is restricted to 2 digits.

- The version delta that you specify with the VERSION compiler control option can contain a maximum of 2 digits.
- Eliminate extra digits from the version delta and restart the compilation.

The version number is restricted to 2 digits.

- The version number (or release number) that you specify with the VERSION compiler control option can contain a maximum of 2 digits.
- Eliminate extra digits from the version number and restart the compilation.

This dollar option is a recognized option and may not be changed after the first statement of the program.

- Some compiler control options must be set or reset preceding the first record of source program text. The option in error is a valid option, but it cannot be changed after the compilation has begun.

This dollar option may be set only once at the beginning of a stacked program and may not be reset afterwards.

- The LIBRARY compiler control option cannot be reset after it is set.

This INCLUDE file has been altered since its earlier use.

- The alteration date and time for the INCLUDE file is different from the date and time that the file was last used in an INCLUDE procedure. This indicates that the file has been modified since it was last used in an INCLUDE procedure.

This option cannot be used in this environment. It will be ignored.

- The string compiler control option in error has no meaning in the current compiling environment and should not be specified.

Too many bad labels declared.

- The source file contains too many bad labels. A bad label is a global label that branches from inside a nested procedure.
- Reduce the number of bad labels, or split the source file into smaller files.

Too many card image segments.

- An internal compiler procedure received data that was incorrect.
- Correct any syntax errors and restart the compiler. If this message appears again, contact your customer support representative.

Too many ordered operators (conditional-and, expression-if etc.).

- You have exceeded the number of operators that the compiler can handle.
- Eliminate some of the operators in the source program and recompile.

Total string-value-$-option pool has exceeded its maximum size.

- You have exceeded the number of string compiler control options that the compiler can handle.
- Eliminate some of the string compiler control options in the source program and recompile.

Twig Column field too small

- The program is too large to compile with the OPTIMIZE compiler control option set.

Unsigned integer constant expected.

- You did not specify an integer value for the integer compiler control option.

Update operator (+=) can only be used for string valued options.

- You incorrectly used an update operator with the NEW compiler control option.

Abnormal Compiler Output Messages

The following messages indicate that the COBOL85 compiler has detected an unexpected logic error in its own processing. Contact your customer support representative to report the problem.

Block linkage incorrect.

COMPILER ERROR <sequence number> (<sequence number>): Invalid block in subprogram.

COMPILER ERROR <sequence number> (<sequence number>): Maximum number of inspect_test entries exceeded.

Data Type of Twig is Unknown.

OPTIMIZER ERROR --- Available Definitions incorrect.

OPTIMIZER ERROR --- Block Links are incorrect.

OPTIMIZER ERROR --- Dag table incorrect.

OPTIMIZER ERROR --- Twig operator incorrect.

Too many cross referenced identifiers.

Too many Linear Sequences.

Run-Time Compiler Output Messages

The following messages indicate that the COBOL85 compiler has detected an unexpected logic error in its own processing. Contact your customer support representative to report the problem.

Abort.

Argument to CAP is not of length 1.

Argument to ICHAR is not of length 1.

Expression out of range.

Invalid assigned GO TO.

Invalid attribute value.

Missing Case.

Missing procedure: "<8-character literal>"."

Soft Stack Overflow.

String value longer than variable.

Appendix B
Reserved Words

The following is a list of reserved words. It includes all reserved words from the complete American National Standard (ANSI-85), and additions used with extensions to ANSI-85. Reserved words that are new to the ANSI-85 standard are marked with a double asterisk (**). Reserved words that are in the ANSI-85 standard but were also in the ANSI-74 standard are not marked.

The compiler does not use all of these words at the present time, but they are all in the reserved word list for the COBOL ANSI-85 compiler. The use of any of these words as a user-defined word causes an error.

A
ABORT-TRANSACTION
ACCEPT
ACCESS
ACTUAL
ADD
ADVANCING
AFTER
ALL
ALLOW
ALPHABET**
ALPHABETIC
ALPHABETIC-LOWER**
ALPHABETIC-UPPER**
ALPHANUMERIC**
ALPHANUMERIC-EDITED**
ALSO
ALTER
ALTERNATE
AND
ANY**
ARE
AREA
AREAS
AS
ASCENDING
ASCII
ASSIGN
AT
ATTACH
ATTRIBUTE
AUDIT
AUTHOR
AVAILABLE

B
BACKUP
BEFORE
BEGIN-TRANSACTION
BEGINNING
BINARY**
BLANK
BLOCK
BOTTOM
BY

C
CALL
CANCEL
CARDS
CASSETTE
CAUSE
CD
CF
CH
CHANGE
CHANNEL
CHARACTER
CHARACTERS
CLASS**
CLOCK-UNITS
CLOSE
COBOL
CODE
CODE
CODE-SET
COLLATING
COLUMN
COLUMNS
COMMA
COMMON**
COMMUNICATION
COMP
COMP-5
COMPUTATIONAL
COMPUTATIONAL-5
COMPUTE
CONFIGURATION
CONTAINS
CONTENT**
CONTINUE**
CONTROL
CONTROL-POINT
CONTROLS
CONVERSATION
CONVERTING**
COPY
CORR
CORRESPONDING

COUNT
CRCR-INPUT
CRCR-OUTPUT
CREATE
CRUNCH
CURRENCY
CURRENT
CYLINDER

D
DATA
DATA-BASE
DATE
DATE-COMPILED
DATE-WRITTEN
DAY
DAY-OF-WEEK**
DB
DE
DEADLOCK
DEBUG-CONTENTS
DEBUG-ITEM
DEBUG-LINE
DEBUG-NAME
DEBUG-SUB-1
DEBUG-SUB-2
DEBUG-SUB-3
DEBUGGING
DECIMAL-POINT
DECLARATIVES
DEFAULT
DELETE
DELIMITED
DELIMITER
DEPENDING
DESCENDING
DESTINATION
DETACH
DETAIL
DICTIONARY
DISABLE
DISALLOW
DISK
DISMISS
DISPLAY
DIVIDE
DIVISION
DMCANCEL
DMCLOSE
DMDELETE
DMERROR
DMOPEN
DMREMOVE
DMSAVE
DMSET
DMSTATUS
DMSTRUCTURE
DMTERMINATE
DOUBLE
DOWN
DUMP
DUPLICATES
DYNAMIC

E
EBCDIC
EGI
ELSE
EMI
ENABLE
END
END-ABORT-TRANSACTION
END-ADD**
END-ASSIGN
END-BEGIN-TRANSACTION
END-CALL**
END-CANCEL
END-CLOSE
END-COMPUTE**
END-CREATE
END-DELETE**
END-DIVIDE**
END-END-TRANSACTION
END-EVALUATE**
END-FIND
END-FREE
END-GENERATE
END-IF**
END-INSERT
END-LOCK
END-MODIFY
END-MULTIPLY**
END-OF-PAGE
END-OPEN
END-PERFORM**
END-READ**
END-RECEIVE**
END-RECREATE
END-REMOVE
END-RETURN**
END-REWRITE**
END-SAVE
END-SEARCH**
END-SECURE
END-SET
END-START**
END-STORE
END-STRING**
END-SUBTRACT**
END-TRANSACTION
END-UNSTRING**
END-WRITE**
ENDING
ENTER
ENTRY
ENVIRONMENT
EOP
EQUAL
ERROR
ESI
EVALUTE**
EVENT
EVERY
EXCEPTION
EXIT
EXTEND
EXTERNAL**
EXTERNAL-FORMAT

F
FALSE**
FD
FIELD
FILE
FILE-CONTROL
FILLER

FINAL
FIND
FIRST
FOOTING
FOR
FORM
FORM-KEY
FREE
FROM
FUNCTION

G
GCR
GENERATE
GIVING
GLOBAL**
GO
GREATER
GROUP

H
HEADING
HIGH-VALUE
HIGH-VALUES

I
I-O
I-O-CONTROL
IDENTIFICATION
IF
IN
INDEX
INDEXED
INDICATE
INITIAL
INITIALIZE**
INITIATE
INPUT
INQUIRY
INSERT
INSPECT
INSTALLATION
INTEGER
INTERROGATE
INTERRUPT
INTO
INVALID
INVOKE
IS

J
JUST
JUSTIFIED

K
KANJI
KEY

L
LABEL
LAST
LB
LD
LEADING
LEFT
LENGTH
LESS
LIMIT
LIMITS
LINAGE
LINAGE-COUNTER
LINE
LINE-COUNTER
LINES
LINKAGE
LOCAL
LOCAL-STORAGE
LOCK
LOCKED
LOW-VALUE
LOW-VALUES
LOWER-BOUND
LOWER-BOUNDS

M
MEMORY
MERGE
MESSAGE
MID-TRANSACTION
MODE
MODIFY
MODULE
MODULES
MOVE
MULTIPLE
MULTIPLY

N
NATIONAL
NATIONAL-EDITED
NATIVE
NEGATIVE
NEXT
NO
NO-AUDIT
NONE
NOT
NULL
NUMBER
NUMERIC
NUMERIC-EDITED**

O
OBJECT-COMPUTER
OCCURS
ODT
ODT-INPUT-PRESENT
OF
OFF
OFFER
OFFSET
OMITTED
ON
OPEN
OPTIONAL
OR
ORDER**
ORGANIZATION
OTHER**
OUTPUT
OVERFLOW
OWN

P
PACKED-DECIMAL**
PADDING**
PAGE
PAGE-COUNTER

PAPERTAPE
PERFORM
PF
PH
PHASE-ENCODED
PIC
PICTURE
PLUS
POINT
POINTER
POINTER
PORT
POSITION
POSITIVE
PRINTER
PRINTING
PRIOR
PROCEDURE
PROCEDURES
PROCEED
PROCESS
PROGRAM

PROGRAM-ID
PROGRAM-LIBRARY
PUNCH
PURGE**

Q
QUEUE
QUOTE
QUOTES

R
RANDOM
RD
READ
READ-OK
READER
REAL
RECEIVE
RECEIVED
RECORD
RECORDS
RECREATE
REDEFINES
REEL
REF
REFERENCE**
REFERENCES
RELATIVE
RELEASE
REMAINDER
REMOTE
REMOVAL
REMOVE
RENAMES
REPLACE**
REPLACING
REPORT
REPORTING
REPORTS
RERUN
RESERVE
RESET
RE-START
RETURN
REVERSED
REWIND
REWRITE
RF
RH
RIGHT
ROUNDED
RUN

S
SAME
SAVE
SD
SEARCH
SECTION
SECURE
SECURITY
SEEK
SEGMENT
SEGMENT-LIMIT
SELECT
SEND
SENTENCE
SEPARATE
SEQUENCE
SEQUENTIAL
SET
SIGN
SINGLE
SIZE
SORT
SORT-MERGE
SOURCE
SOURCE-COMPUTER
SPACE
SPACES
SPECIAL-NAMES
STACK
STANDARD
STANDARD-1
STANDARD-2**
START
STATUS
STOP
STOQ-INPUT
STOQ-OUTPUT
STORE
STRING
STRUCTURE
SUB-QUEUE-1
SUB-QUEUE-2
SUB-QUEUE-3
SUBTRACT
SUM
SUPPRESS
SW1
SW2
SW3
SW4
SW5
SW6
SW7
SW8
SYMBOLIC
SYNC
SYNCHRONIZED
SYSTEM
SYSTEMERROR

T
TABLE
TAG-KEY
TAG-SEARCH
TALLYING

TAPE
TAPES
TASK
TERMINAL
TERMINATE
TEST**
TEXT
THAN
THEN**
THROUGH
THRU
TIME
TIMER
TIMES
TO
TODAYS-DATE
TODAYS-NAME
TOP
TRACE-OFF
TRACE-ON
TRAILING
TRANSACTION
TRANSCEIVE
TRUE**
TYPE

U

UNIT
UNLOCK
UNSTRING
UNTIL
UP
UPDATE
UPON
USAGE
USE
USING

V

VALUE
VALUES
VARYING
VIA

W

WAIT
WHEN
WHERE
WITH
WORDS
WORKING-STORAGE
WRITE
WRITE-OK

Z

ZERO
ZEROES
ZEROS
ZIP

Special Characters

*
**
+
-
/
<
<=
=
>
>=

Appendix C
Interpreting General Formats

A general format is a syntax diagram that shows the arrangement of language elements in a COBOL division, section, paragraph, clause, or statement. This manual presents a general format and then describes the elements of that format. The description also includes rules for using the format. Unless stated otherwise, you must write required and optional elements of a format in the sequence in which they appear.

Within a general format, various notation conventions are used to indicate which elements are required, optional, repeatable, and so on. The notation conventions used in presenting COBOL syntax are discussed in the following paragraphs.

Uppercase Words

Uppercase words that appear in general formats have specific meanings defined by the COBOL language, and thus, are considered to be *reserved words*. Reserved words must be typed exactly as shown and cannot be used in any context other than the one shown in the general format.

When an uppercase word is underlined, it is a required part of the syntax. An underlined word is called a *keyword*. If an uppercase word is not underlined, you can omit it from the text of your program.

For a comprehensive list of COBOL reserved words, refer to Appendix B.

Example

```
PROGRAM COLLATING SEQUENCE  IS alphabet-name
```

The first four words of this example appear in uppercase characters and thus are reserved words, which means that you must type them exactly as shown. The only underlined word is SEQUENCE, and thus it is the only required word. In your program text you could use all four reserved words as in "PROGRAM COLLATING SEQUENCE IS . . . ", or you could simply use the word "SEQUENCE . . . ".

Lowercase Words

Lowercase words that appear in the general formats represent a value that you must supply. The value might be

- A user-defined word, such as a file name
- A figurative constant
- A literal
- A complete syntactical entry, such as a data description entry

When the same lowercase word appears multiple times in a general format, a number or letter appears at the end to differentiate among the like words.

For a detailed discussion of literals, figurative constants, and user-defined words, refer to Section 1. The types of user-defined words and the rules for forming them are presented in the following paragraphs.

Rules for Creating User-Defined Words

The types of user-defined words that occur in various portions of the COBOL syntax are as follows:

Alphabet-names	Library-names
Class-Names	Mnemonic-names
Computer-Names	Paragraph-names
Condition-names	Program-names
Data-names	Record-names
File-names	Section-names
Implementor-names	Text-names
Index-names	

For a description of each type of user-defined word, refer to Section 1.

Observe the following rules when you create user-defined words:

- A user-defined word can consist of 1 to 30 letters, digits, and hyphens (-).
- You cannot use reserved words as part of a user-defined word.
- The hyphen cannot appear as the first or last character of a word.
- The user-defined word must contain at least one alphabetic character, unless the word is a section-name or a paragraph-name.
- All user-defined words must be unique within a program except as specified in the rules for uniqueness of reference (refer to "Uniqueness of Reference" in Section 4). However, you can use the same name for a system name (either a computer-name or an implementor-name) and another user-defined name. The compiler can determine whether the word is to be used as a system name or another type of name by the context of the clause or phrase in which the word occurs.
- The word you assign as a data-name cannot be referenced-modified, subscripted, or qualified unless specifically permitted by the rules of the general format.

Example

```
[RECORD CONTAINS [integer-1 TO] integer-2 CHARACTERS]
```

In this example, the lowercase word *integer* appears multiple times, and so it is appended with –1 on the first occurrence and –2 on the second occurrence. Both occurrences require you to supply an integer. However, integer-1 signifies the starting number of a sequence. Integer-2 signifies the ending number of that sequence.

Brackets

Words that are enclosed in brackets ([ ]) in a general format are optional. If the brackets enclose more than one phrase stacked vertically, you can use one of the phrases or you can omit them all.

Example

```
[ RECORD CONTAINS [ integer-1 TO ] integer-2 CHARACTERS ]
```

In this example, the entire sentence is enclosed in brackets, so it is optional. If you use this sentence in your program, you can omit the [integer-1] portion, because it is enclosed in brackets.

Braces

Items enclosed in braces ({ }) in general formats indicate options. You can choose one option to include in your program syntax. If one of the options contains only reserved words that are not underlined, it is the default option. The compiler uses the default option if you do not specify an option in your program. If all the options are underlined, you must explicitly specify one of the options in your program. If the braces enclose a single phrase, you can repeat the phrase.

Example

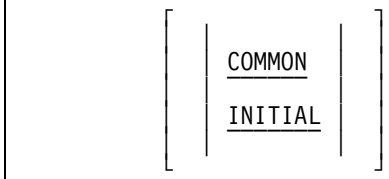
```
BLOCK CONTAINS [ integer-1 TO ]  integer-2  { RECORDS    }
                                            { CHARACTERS }
```

In this example, the braces indicate that you must choose between RECORDS and CHARACTERS. The word CHARACTERS is not underlined, so it is the default option.

Vertical Bars

When vertical bars (|) enclose a portion of a general format, you must specify one or more of the options contained within the vertical bars. You can specify each option only once.

Example



In this example, you can specify COMMON, INITIAL, or COMMON INITIAL.

Ellipses

Elements in general formats that are followed by ellipses (. . .) can be repeated. You determine the portion of the format that can be repeated in the following way. Identify the right bracket ( ]) or right brace (}) immediately to the left of the ellipses. Then find the logically matching left bracket ([ ) or left brace ({). The portion of the format between the determined pair of delimiters can be repeated.

In text other than general formats, ellipses are used in the conventional way to show omission of a word or words when such omission does not impair comprehension. The meaning of the omission becomes apparent in context.

Example

```
[SAME [ RECORD ] AREA FOR file-name-3 { ,file-name-4 } ... ] ...
```

The first set of ellipses indicates that the file-name-4 phrase can be repeated. The second set of ellipses means that the entire clause contained within brackets, from SAME through the file-name-4 phrase, can be repeated.

Punctuation Marks

A period (.) is always a required element when it appears in a general format. The comma (,) and semicolon (;) are optional.

For information on using punctuation marks in your program, refer to Table 1–1.

Example

```
[SAME [ RECORD ] AREA FOR file-name-3 { ,file-name-4 } ... ] ...
```

In the SAME RECORD AREA clause syntax, the word SAME can be preceded by a semicolon, a comma, or a space. Also, the comma that precedes file-name-4 can be a semicolon or a space.

Mathematical Symbols

When mathematical symbols appear in a general format, you must include them in your COBOL program. The following mathematical symbols are valid in the COBOL language:

Table C–1. Valid Mathematical Symbols

Symbol	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation
>	Greater than
<	Less than
=	Equal to
>=	Greater than or equal to
<=	Less than or equal to

Appendix D
Using the Checkpoint/Restart Utility

The checkpoint/restart utility can protect a program against the disruptive effects of unexpected interruptions during the execution of a program. If a halt/load or other system interruption occurs, a job is either restarted before the initiation of the task that was interrupted or, if the operator permits, it is restarted at the last checkpoint, whichever is more recent. Checkpoint information can also be retained after successful runs to enable restarting jobs to correct bad data situations.

CALLCHECKPOINT Procedure

The CALLCHECKPOINT procedure uses the CALL statement to write to a disk file the complete state of the job at a specified point. Using the disk file, the job can later be restarted from this point.

The CALLCHECKPOINT procedure can be used as an Integer function. An attempted checkpoint returns a value of success or exception. If the result is one, then either the restart flag is one on a restart or the completion code is nonzero on a checkpoint that cannot be taken. If the result is zero, then the checkpoint value is nonzero.

The following is the syntax for the CALLCHECKPOINT procedure:

```
CALL CALLCHECKPOINT IN MCPSUPPORT
USING
     CHECKPOINTDEVICE
     CHECKPOINTTYPE
     COMPLETIONCODE
     CHECKPOINTNUMBER
     RESTARTFLAG
GIVING
     RSLT.
```

Each of the five options for the CALLCHECKPOINT procedure is described in the following paragraphs.

CHECKPOINTDEVICE Option

This option enables you to specify the medium (DISK or PACK) to be used for the checkpoint files.

CHECKPOINTTYPE Option

This option enables you to specify the LOCK or PURGE option for the checkpoint files.

The PURGE option causes all checkpoint files to be removed at successful termination of the job and protects the job against system failures. The LOCK option causes all checkpoint files to be saved indefinitely and can be used to restart a job even if a job has terminated normally.

COMPLETIONCODE Option

In response to the request for a completion code, a program can receive a variety of messages. Refer to "Checkpoint/Restart Messages" in this section for a list of the completion codes and messages.

CHECKPOINTNUMBER Option

This option assigns a number to every checkpoint to distinguish between successive checkpoints.

RESTARTFLAG Option

The restart flag option is used to detect a restart and to avoid reexecuting portions of the program. When a checkpoint is invoked, the following files are created:

- The checkpoint file

  CP/<JN>/<CPN>

  where <JN> is a four-digit job number and <CPN> is a three-digit checkpoint number.

  If the PURGE option has been specified, the checkpoint number is always zero, and each succeeding checkpoint with PURGE removes the previous file. If the LOCK option is used, the checkpoint number starts with a value of one for the first checkpoint and is incremented by one for each succeeding checkpoint with LOCK. If the two types are mixed within a job, the LOCK checkpoints use the ascending numbers and the PURGE checkpoints use 0 (zero), leaving files 0 through N at the completion of the job.

- Temporary files

  CP/<JN>/T<FN>

  where <FN> is a three-digit file number beginning with one and incremented by one for each temporary disk or system resource pack file.

- The job file

  CP/<JN>/JOBFILE

  This file is created under the LOCK option only.

  The LOCK and PURGE options are also effective when the task terminates. If the task terminates abnormally and the last checkpoint has used the PURGE option, then the checkpoint file (numbered zero) is changed to have the next sequential checkpoint number, and the job file is created (if necessary). If the job terminates normally and only PURGE checkpoints have been taken, the CP/<JN> directory is removed.

Restarting a Job

A job can be restarted in one of two ways:

- After a Halt/Load

  The system automatically attempts to restart any job that was active at the time of a Halt/Load. If a checkpoint has been invoked during the execution of the interrupted task, then you are given a message requiring a response to determine whether the job should be restarted. You can respond with the system command OK (to restart at the last checkpoint), DS (to prevent a restart), or QT (to prevent a restart but save the files for later restart if the job was at a checkpoint with the PURGE option).

- By a Work Flow Language (WFL) *RERUN* statement

  A WFL job can be restarted programmatically by use of the WFL *RERUN* statement.

The following conditions can inhibit a successful restart:

- The usercode is invalid.
- The program has been recompiled since the checkpoint.
- The operating system has changed since the checkpoint.

  The restart fails if the creation timestamp of the operating system that created the checkpoint file does not match the creation timestamp of the current operating system.

- Intrinsics after the checkpoint are different from intrinsics before the checkpoint.

Refer to "Checkpoint/Restart Messages" later in this section for a list of messages that can appear as a result of an attempt to restart.

For a successful checkpoint/restart

- The task being checkpointed must have no tasks initiated through CALL or PROCESS statements.
- The task must have been initiated by a WFL job.
- The WFL job that initiates the task must not have initiated other tasks that are also running.

The following can inhibit a successful checkpoint/restart:

- Datacomm I/O (open datacomm files).
- Open Data Management System II (DMSII) sets.
- ODT files.
- Duplicated files.
- Output directly to a printer (backup files are acceptable).
- Checkpoints taken inside sort input or output procedures. The sort intrinsic provides its own restart capability. For more information on the sort intrinsic, refer to "SORT Statement" in Section 6.
- Checkpoints taken in a compile-and-go program.

If a job that produces printer backup files is restarted, the backup files can already have been printed and removed, and on restart the job requests the missing backup files. In this situation, when the backup files are requested, the operator must respond with the system command OF (Optional File). A new backup file is created. Output preceding the checkpoint is not re-created.

Checkpoint/Restart Messages

There are two categories of output messages that can appear during a checkpoint/restart procedure. The two categories are

- Output messages that appear as a result of an attempt to restart
- Output messages and completion codes that appear as a result of a checkpoint/restart procedure

Output Messages from an Attempt to Restart

The messages in the following list can appear as a result of an attempt to restart:

BAD CHECKPOINT FILE

- The checkpoint file is not valid.
- Contact your customer support representative.

BAD STACK NUMBER

- The stack number is not valid.
- Contact your customer support representative.

INVALID JOB FILE

- The job file is no longer a valid checkpoint file.
- Check to see that the job file is a valid checkpoint file.

IO ERROR DURING RESTART

- A disk error occurred.
- Try to recover and restart the program again.

MISSING CHECKPOINT FILE

- The system cannot find your checkpoint file.
- Ensure that you have entered the correct checkpoint file name; otherwise, contact your customer support representative.

MISSING CODE FILE

- The system cannot find the code file that was previously running.
- Check the version of the code file, and ensure that the program is a code file.

MISSING JOB FILE

- The system cannot find the checkpoint file that was previously created.
- Check to see that you entered the correct checkpoint file name.

NOT ABLE TO RESTART

- An internal error has occurred.
- Contact your customer support representative.

OPERATOR DSED RESTART

- The restart was discontinued by the operator. This message appears as a response to the "RESTART PENDING (RSVP)" message.
- This message is for information only and no response is necessary.

OPERATOR QTED RESTART

- The restart was postponed by the operator. This message appears as a response to the "RESTART PENDING (RSVP)" message. You can then initiate the restart later.
- This message is for information only and no response is necessary.

RESTART PENDING (RSVP)

- You restarted a job. This message appears to the operator only.
- This message is for information only and no response is necessary.

USERCODE NO LONGER VALID

- Your usercode is no longer valid.
- Contact your customer support representative.

WRONG CODE FILE

- The system cannot find the code file that was previously running.
- Check the version of the code file, and ensure that the program is a code file.

WRONG JOB FILE

- The job file is no longer a valid checkpoint file.
- Check to see that the job file is a valid checkpoint file.

WRONG MCP

- The restart was not done on the same MCP level that took the checkpoint.
- Review the input and correct the input by restarting the checkpoint using the correct MCP level.

Output Messages and Completion Codes

The following messages can appear as a result of a checkpoint/restart. Error conditions can be handled in a program by checking the completion code number and instructing the program to handle the result.

Each message in the following list includes the completion code, the message, and a description of the message

0 CHECKPOINT#nn

The checkpoint completed without any errors.

1 INVALID_AREA_IN_STACK_ERROR

Certain declarations cannot be used in a program that uses a checkpoint.

2 SYSTEM_ERROR

A internal software error has occurred. Contact your customer support representative.

3 BAD_IPC_ENVIRONMENT_ERROR

The program that is checkpointed must be a standalone job.

4 NO_USER_DISK_FOR_CP_FILE_ERROR

There is no more disk space available for your checkpoint file.

5 IO_ERROR_DURING_CHECKPOINT_ERROR

A disk error has occurred. Contact your customer support representative.

6 TOO_MANY_ROWS_IN_CP_FILE_ERROR

There is no more disk file space available for your checkpoint file.

7 DIRECT_FILE_NOT_ALLOWED_ERROR

You cannot declare a direct file in the program that is checkpointed.

8 TOO_MANY_TEMPORARY_DISK_FILES_ERROR

The system limit on temporary open files is exceeded. Delete or close unnecessary temporary files before initiating a checkpoint.

9 ILLEGAL_FILEKIND_ERROR

Datacomm files and ODT files cannot be checkpointed.

10 DUPLICATE_FILE_NOT_ALLOWED_ERROR

The DUPLICATED file attribute cannot be used with a checkpoint. Refer to the I/O Subsystem Programming Guide for information on the DUPLICATE file attribute.

11 ILLEGAL_FILE_ORGANIZATION_ERROR

The FILEORGANIZATION file attribute must have a value of NOTRESTRICTED. Refer to the I/O Subsystem Programming Guide for information on the FILEORGANIZATION file attribute.

12 INSUFFICIENT_MEMORY_TO_CHECKPOINT_ERROR

A single large area of memory is required for the checkpoint to perform successfully. Contact your customer support representative.

13 OPEN_REVERSED_TAPE_NOT_ALLOWED_ERROR

Tapes cannot be read backwards during a checkpoint.

14 ICM_AREA_IN_STACK_ERROR

A checkpoint is not allowed for a program containing a connection block (CB).

15 DMS_AREA_IN_STACK_ERROR

A checkpoint is not allowed for a data base.

16 DIRECT_ARRAY_IN_STACK_ERROR

A checkpoint is not allowed for a direct array.

17 TEMP_DISK_FILE_SECURITY_ERROR

A checkpoint cannot put temporary disk files in the disk directory for reasons of security.

19 STACKMARK_ERROR

A checkpoint cannot occur on a program that contains a stack within a program.

20 SORT_AREA_IN_STACK_ERROR

A checkpoint cannot occur during a sort.

21 IN_USE_NOT_ALLOWED_ERROR

A USEROUTINE is not allowed during a checkpoint.

22 ILLEGAL_CONSTRUCT_ERROR

An internal software error has occurred. Contact your customer support representative.

23 BDBASE_ILLEGAL_ERROR

A program cannot have an option equivalent to BDBASE. Refer to the Task Attributes Programming Reference Manual for more information on BDBASE.

24 ILLEGAL_FILESTRUCTURE_ERROR

Files in the program must have declared FILESTRUCTURE=ALIGNED180. For more information on file attributes refer to the I/O Subsystem Programming Guide.

25 MULTI_REEL_UNLABELED_TAPE_ERROR

A checkpoint cannot occur with unlabeled tapes on more than one reel.

26 SURROGATE_TASK_NOT_ALLOWED_ERROR

The parent task cannot be on a remote host.

30 ROW_SIZE_TOO_SMALL_FOR_CP_FILE_ERROR

A checkpoint performed on a structure cannot fit into the checkpoint file.

36 PRINT_DISPOSITION_EQUAL_EOT_ERROR

The file attribute PRINTDISPOSITION must not equal EOT in the checkpoint.

37 PRINT_DISPOSITION_EQUAL_CLOSE_ERROR

The file attribute PRINTDISPOSITION must not equal CLOSE in the checkpoint.

38 PRINT_DISPOSITION_EQUAL_DIRECT_ERROR

The file attribute PRINTDISPOSITION must not equal DIRECT in the checkpoint.

40 TEMPFILELIMIT_ON_CP_FILE_ERROR

The disk usage has exceeded its limit as a result of a checkpoint. Contact your customer support representative.

41 FAMILYLIMIT_ON_CP_FILE_ERROR

The disk usage has exceeded its limit as a result of a checkpoint. Contact your customer support representative.

42 INTEGRALLIMIT_ON_CP_FILE_ERROR

The disk usage has exceeded its limit as a result of a checkpoint. Contact your customer support representative.

43 UNKNOWN_STK_NUM_ERROR

An unknown stack number was detected during a checkpoint. Check the validity of the libraries.

44 ILLEGAL_CPDEVICE_ERROR

The file must checkpoint to either DISK or PACK.

45 ILLEGAL_CPTYPE_ERROR

The file must checkpoint to either PURGE or LOCK.

Locking

For jobs that take a large number of checkpoints with the LOCK option, the checkpoint number counts up to 999 and then recycles to 1 (leaving zero undisturbed). When this recycling occurs, previous checkpoint files are lost as new ones using the same numbers are created.

If a temporary disk file is open at a checkpoint, it is locked under the CP directory. If it is subsequently locked by the program, the name is changed to the current file title. As a result, at restart time the file is sought only under the CP directory, resulting in a no-file condition. To avoid this condition, all files that are to be locked eventually should be opened with the file attribute PROTECTION assigned the value SAVE. To remove such a file, it must be closed with PURGE. True temporary files, which are never locked, do not have this problem.

All data files must be on the same medium as at the checkpoint, but need not be on the same units or on the same locations on disk or pack. The files must retain the same characteristics, such as blocking. The checkpoint/restart system makes no attempt to restore the contents of a file to their state at the time of the checkpoint; the file is merely repositioned. Volume numbers are not verified.

Note: *CANDE cannot be used to run a program with checkpoints. The checkpoints are ignored if used.*

Rerunning Programs

If a rerun is initiated and the job number is in use by another job, a new job number is supplied, and the CP/<JN> directory node is changed to reflect the new job number. If a rerun is initiated, the value of the PROCESSID can be different for the restarted job. When a job is restarted at some checkpoint before the last, subsequent checkpoints taken from the restarted job continue in numerical sequence from the checkpoint used for the restart. Previous higher numbered checkpoints are lost.

CHECKPOINT Procedure Call Examples

A program that calls CHECKPOINT should be compiled with COBOL85 and the object should be saved. The program should be executed through WFL.

The first of the following examples uses the explicit library interface to access the CALLCHECKPOINT procedure. This is the preferred syntax for COBOL85. The PACK and PURGE options are shown in the example:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CHECK-POINT.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT ATTR-FILE ASSIGN TO DISK.
DATA DIVISION.
FILE SECTION.
FD ATTR-FILE.
01 ATTR-REC PIC X(80).
WORKING-STORAGE SECTION.
77 CHECKPOINTDEVICE PIC S9(11) USAGE BINARY.
77 CHECKPOINTTYPE   PIC S9(11) USAGE BINARY.
77 COMPLETIONCODE   PIC S9(11) USAGE BINARY.
77 CHECKPOINTNUMBER PIC S9(11) USAGE BINARY.
77 RESTARTFLAG      PIC S9(11) USAGE BINARY.
77 RSLT             PIC S9(11) USAGE BINARY.
77 VALUE-OF-PACK    PIC S9(11) USAGE BINARY.
77 VALUE-OF-PURGE   PIC S9(11) USAGE BINARY VALUE 0.
LOCAL-STORAGE SECTION.
LD LD-CALLCHECKPOINT.
77 CHECKPOINTDEVICE PIC S9(11) USAGE BINARY.
77 CHECKPOINTTYPE   PIC S9(11) USAGE BINARY.
77 COMPLETIONCODE   PIC S9(11) USAGE BINARY.
77 CHECKPOINTNUMBER PIC S9(11) USAGE BINARY.
77 RESTARTFLAG      PIC S9(11) USAGE BINARY.
77 RSLT             PIC S9(11) USAGE BINARY.
PROGRAM-LIBRARY SECTION.
LB MCPSUPPORT IMPORT
   ATTRIBUTE
   FUNCTIONNAME IS "MCPSUPPORT"
   LIBACCESS    IS BYFUNCTION.
ENTRY PROCEDURE CALLCHECKPOINT
   WITH   LD-CALLCHECKPOINT
   USING
          CHECKPOINTDEVICE
          CHECKPOINTTYPE
          COMPLETIONCODE
          CHECKPOINTNUMBER
          RESTARTFLAG
   GIVING
          RSLT.
```

```
PROCEDURE DIVISION.
INIT-PARA.
        CHANGE ATTRIBUTE KIND         OF ATTR-FILE
                                      TO PACK.
        MOVE   ATTRIBUTE KIND         OF ATTR-FILE
                                      TO VALUE-OF-PACK.
        PERFORM CHECKPOINT-PARA.
        STOP RUN.
CHECKPOINT-PARA.
        MOVE VALUE-OF-PACK  TO CHECKPOINTDEVICE.
        MOVE VALUE-OF-PURGE TO CHECKPOINTTYPE.
        CALL CALLCHECKPOINT
        USING
               CHECKPOINTDEVICE
               CHECKPOINTTYPE
               COMPLETIONCODE
               CHECKPOINTNUMBER
               RESTARTFLAG
        GIVING
               RSLT.
```

The next example uses the IPC *CALL* statement to access the CALLCHECKPOINT procedure. The DISK and LOCK options are shown in the example.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CHECK-POINT.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT ATTR-FILE ASSIGN TO DISK.
DATA DIVISION.
FILE SECTION.
FD ATTR-FILE.
01 ATTR-REC PIC X(80).
WORKING-STORAGE SECTION.
77 CHECKPOINTDEVICE PIC S9(11) USAGE BINARY.
77 CHECKPOINTTYPE   PIC S9(11) USAGE BINARY.
77 COMPLETIONCODE   PIC S9(11) USAGE BINARY.
77 CHECKPOINTNUMBER PIC S9(11) USAGE BINARY.
77 RESTARTFLAG      PIC S9(11) USAGE BINARY.
77 RSLT             PIC S9(11) USAGE BINARY.
77 VALUE-OF-DISK    PIC S9(11) USAGE BINARY.
77 VALUE-OF-LOCK    PIC S9(11) USAGE BINARY VALUE 1.
PROCEDURE DIVISION.
INIT-PARA.
        CHANGE ATTRIBUTE LIBACCESS    OF "MCPSUPPORT"
                                      TO  BYFUNCTION.
        CHANGE ATTRIBUTE FUNCTIONNAME OF "MCPSUPPORT"
                                      TO "MCPSUPPORT".
        CHANGE ATTRIBUTE KIND         OF ATTR-FILE
                                      TO DISK.
```

```
        MOVE   ATTRIBUTE KIND         OF ATTR-FILE
                                      TO VALUE-OF-DISK.
        PERFORM CHECKPOINT-PARA.
        STOP RUN.
CHECKPOINT-PARA.
        MOVE VALUE-OF-DISK  TO CHECKPOINTDEVICE.
        MOVE VALUE-OF-LOCK  TO CHECKPOINTTYPE.
        CALL "CALLCHECKPOINT IN MCPSUPPORT"
        USING
               CHECKPOINTDEVICE
               CHECKPOINTTYPE
               COMPLETIONCODE
               CHECKPOINTNUMBER
               RESTARTFLAG
        GIVING
               RSLT.
```

Appendix E
COBOL Binding

Note: *Although this section is not highlighted, all of the information is relative to Unisys extensions to COBOL ANSI-85.*

Program binding is the process of joining together parts of separately-compiled programs to create a single executable program. It provides COBOL85 with ALGOL-like modular programming capability, and the ability to share global data items. Programs with bound procedures may execute faster than programs that can call other programs or libraries. Creating a program through binding can offer several advantages over writing a program in a single piece:

- Small modules can be intellectually more manageable than large programs.
- Procedures in modules are granted direct access only to files and data items declared within the modules. This creates a degree of protection within the final program.
- The work of coding a large program can be subdivided among several people.
- A set of precompiled, pretested modules designed to accomplish certain functions can be established at an installation. When one of those functions is required in a program, the appropriate module can be bound into a program.
- A change to a module does not force recompilation of an entire program. Only the affected module and the main program part need be recompiled; the other precompiled modules will be bound into the final program automatically. This can reduce compilation time.

Binding is designed primarily to function as an efficiency tool, saving not only computer time in recompiling, but programmer rewriting time as well. For example, when a large program requires changes, only the portion of the program requiring changes needs to be rewritten and recompiled. The Binder is then invoked to insert the revision into the program. You do not need to rewrite or recompile the entire program to change or correct a portion of it.

Naming File Buffers

The name of the first 01 entry that is declared under an FD statement will be used as the name of the file buffer in BINDINFO and the type of the record will be EBCDIC.

Binding Example

Assume that a COBOL host program has to be bound to another COBOL subprogram. Then the two programs will appear as in the following example.

COBOL Host Program

```
$ BINDINFO
 IDENTIFICATION DIVISION.
 PROGRAM-ID.    HOST.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER.   A17.
 OBJECT-COMPUTER.     A17.
 SPECIAL-NAMES.
       "OBJECT/BIND/SOURCE/SUB" IS TO-BE-CALLED.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
      SELECT LOCAL RECEIVED BY REFERENCE FL ASSIGN TO PRINTER.
      SELECT PR-FILE ASSIGN TO PRINTER.
 DATA DIVISION.
 FILE SECTION.
 FD PR-FILE.
 01   PR-RCD               PIC X(36).
 FD FL.
 01 FL-RCD                 PIC X(36).
 WORKING-STORAGE SECTION.
 01   CO-ITEM              PIC  X(36).
 01   ORIG                 PIC  X(36).
 01   NEW                  PIC  X(36).
 LOCAL-STORAGE SECTION.
 LD PARMS.
 01  P1                    PIC  X(36)   REFERENCE.
 01  P2                    PIC  X(36)   REFERENCE.
 PROCEDURE DIVISION.
 DECLARATIVES.
 BOUND SECTION.
        USE EXTERNAL TO-BE-CALLED AS PROCEDURE
        WITH PARMS, FL
        USING P1, P2, FL.
 END DECLARATIVES.
 FIRST-PA SECTION.
 START-PA.
 OPEN OUTPUT PR-FILE.
 MOVE "THIS WILL STOP WHEN THIS LINE ENDS"
     TO ORIG.
 CALL BOUND USING ORIG, NEW, PR-FILE.
 WRITE PR-RCD FROM ORIG.
 WRITE PR-RCD FROM NEW.
 STOP RUN.
```

COBOL Subprogram

```
$ SET LEVEL = 3
 IDENTIFICATION DIVISION.
 PROGRAM-ID.   ARRAY-MIXER.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER.   A17.
 OBJECT-COMPUTER.   A17.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT RECEIVED BY REFERENCE REC-FILE
     ASSIGN TO PRINTER.
 DATA DIVISION.
 FILE SECTION.
 FD  REC-FILE.
 01  FILE-RCD  PIC  X(36).
 WORKING-STORAGE SECTION.
 01  CO-ITEM  COMMON  PIC  X(36).
 LINKAGE SECTION.
  01   X  REFERENCE.
       03   FIRSTT  PIC  X(5).
       03   SECOND  PIC  X(5).
       03   THIRD   PIC  X(5).
       03   FOURTH  PIC  X(5).
       03   FIFTH   PIC  X(5).
       03   SIXTH   PIC  X(5).
       03   SEVENTH PIC  X(5).
       03   EIGHTH  PIC  X.
  01   Y  REFERENCE.
       03   FIRS    PIC  X(5).
       03   SECON   PIC  X(5).
       03   THIR    PIC  X(5).
       03   FOURT   PIC  X(5).
       03   FIFT    PIC  X(5).
       03   SIXT    PIC  X(5).
       03   SEVENT  PIC  X(5).
       03   EIGHT   PIC  X.
  PROCEDURE DIVISION USING X Y REC-FILE.
  THAT SECTION.
  MIX.
       MOVE "IN THE BOUND PROGRAM" TO FILE-RCD.
       WRITE FILE-RCD.
       MOVE FIRSTT TO SECON.
       MOVE SECOND TO FOURT.
       MOVE THIRD TO FIRS.
       MOVE FOURTH TO THIR.
       MOVE FIFTH TO SIXT.
       MOVE SIXTH TO SEVENT.
       MOVE SEVENTH TO FIFT.
       MOVE EIGHTH TO EIGHT.
```

Once the host and subprogram are bound together, on execution the result will appear as the following:

```
IN THE BOUND PROGRAM
THIS WILL STOP WHEN THIS LINE ENDS
STOP THIS WHEN WILL ENDS THIS LINE.
```

Appendix F
Comparison of COBOL Versions

This appendix is designed to aid you in understanding the differences between COBOL68, COBOL74, and COBOL85. In particular, it describes

- The reasons for the ANSI 85 revision of COBOL
- Specific differences in the ANSI Standard between COBOL85 and previous versions of the COBOL compiler

COBOL ANSI-85 is a revision of COBOL ANSI-74. The changes have been designed to minimize the impact on existing COBOL programs and, in most cases, do not cause an incompatibility between COBOL85 and COBOL74.

The ANSI committee revised COBOL74 for the following reasons:

- To clarify unclear or ambiguous rules
- To improve the capabilities of COBOL and its ease of use
- To enhance application programming productivity
- To ease program portability
- To improve program maintenance
- To limit or remove error-prone, useless, and redundant features

After considering the benefits of COBOL85, you might want to convert your existing programs to COBOL85. For more information about migration tools and services, refer to Appendix G, COBOL Migration.

Differences Among COBOL Versions

The following paragraphs list the changes included in COBOL85. These changes include new features or changes that

- Will probably affect your programs
- Might affect your programs
- Will not impact existing programs

Changes marked as "change" are additions to the COBOL85 Standard or changes to extensions to the compiler.

Changes marked as "obsolete" will become obsolete in the next revision of standard COBOL. For this revision, they are optional elements. Consequently, if you choose not to modify your programs to either remove the obsolete elements or make them into comments, be aware that the next revision of COBOL might not handle these items.

Changes That Probably Affect Your Programs

The following paragraphs list the changes included in COBOL85 that will probably affect your programs. Also, COBOL85 has not implemented the Semantic Information Manager (SIM) or the DMSII TPS product interfaces or the ANSI74 COBOL Communications Module.

Abbreviations (Change)

COBOL85 does not allow certain abbreviations. These abbreviations must be changed as follows:

- OC changes to OCCURS
- ID changes to IDENTIFICATION
- VA changes to VALUE
- PC changes to PIC
- CMP changes to COMP

In addition, COBOL85 does not allow these conditional abbreviations:

- 'IF X = 1 2 3 AND 4' becomes 'IF X = 1 AND 2 AND 3 AND 4'
- 'IF X = 1 2 3 OR 4' becomes 'IF X = 1 OR 2 OR 3 OR 4'

ACTUAL KEY Clause (Change)

The performance of all I/O statements that act upon a sequential file declared with an actual key can be significantly improved by declaring the appropriate key as follows:

```
77 USERKEY     REAL.
```

ALL Literal and Numeric or Numeric Edited Item (Obsolete)

The figurative constant ALL literal, when associated with a numeric or numeric edited item and when the length of the literal is greater than one, has been placed in the obsolete element category.

This element is being made obsolete because the results of moving an ALL literal to a numeric data item are often unexpected.

ALPHABET-NAME Clause (Change)

The key word ALPHABET must precede alphabet-name-1 within the ALPHABET-NAME clause of the SPECIAL-NAMES paragraph.

Because system-names and user-defined words could be the same word in COBOL85, the compiler might not be able to determine which use is intended. The introduction of the key word ALPHABET in the ALPHABET-NAME clause resolves this ambiguity.

```
SPECIAL-NAMES. WORD-1 IS WORD-2.
```

In COBOL85, if WORD-1 is both an implementor-name and an alphabet-name, and WORD-2 is both a mnemonic-name and an implementor-name, then the compiler cannot distinguish whether the implementor-name clause or the ALPHABET-NAME clause was intended.

You must modify any program that contains the (now optional) SPECIAL-NAMES paragraph to use ALPHABET in front of alphabet-name-1. The preceding example would be modified as follows:

```
SPECIAL-NAMES. ALPHABET WORD-1 IS WORD-2.
```

ALTER Statement (Obsolete)

The ALTER statement has been placed in the obsolete element category. The use of the ALTER statement results in a program that could be difficult to understand and maintain because it changes the procedure referred to in a GO TO statement.

APPLY and RERUN Clause (COBOL68 Only) (Obsolete)

The APPLY and RERUN clauses of the I-O-CONTROL paragraph are not supported by COBOL74 or COBOL85. They must be deleted.

AREAS/AREASIZE (COBOL68 Only) (Change)

The AREAS and AREASIZE file attributes and INTERCHANGE option of the SELECT clause in the ENVIRONMENT DIVISION should be moved to the File Description in the DATA DIVISION.

Attributes in Conditional Expressions (COBOL68 Only) (Change)

The syntax for file and task attributes used in conditional expressions is different for COBOL68 than for COBOL74 and COBOL85.

```
IF <file-identifier> (<attribute-name>) becomes
IF ATTRIBUTE <attribute-name> OF <file-identifier>
```

AUTHOR, INSTALLATION, DATE-WRITTEN, DATE-COMPILED, and SECURITY Paragraphs (Obsolete)

The AUTHOR, INSTALLATION, DATE-WRITTEN, DATE-COMPILED, and SECURITY paragraphs in the Identification Division have been placed in the obsolete element category.

You can provide this information by using comment lines in the Identification Division. Add an asterisk (*) to column 7 to make these paragraphs comments. COBOL85 still accepts these paragraphs, but we recommend that you make them comments for the sake of clarity.

AWAIT Statement (COBOL68 Only) (Obsolete)

The AWAIT statement is not implemented in COBOL74 or COBOL85. You should replace occurrences of the AWAIT statement with the WAIT statement.

$SET BINDINFO for Binding Programs (Change)

For successful binding in COBOL85, you must set the BINDINFO compiler option in the subprogram and the host program.

The requirements for binding in COBOL68 are as follows:

- Set the LEVEL=3 compiler option in the subprogram
- Specify a USE EXTERNAL clause in the DECLARATIVES SECTION in the host program

Binding

The Binder displays the COBOL USE statement in a bound program in slightly different ways for programs compiled with COBOL85 and COBOL68 or COBOL74. The following code fragments illustrate this point:

COBOL HOST PROGRAM

```
.
.
.
FD FILE-FROM-HOST.
01 REC-FROM-HOST            PIC X(80).
.
.
.
```

ALGOL SUBPROGRAM

```
.
.
.
[
 FILE  FILE_FROM_SUB;
 EBCDIC ARRAY  REC_FROM_SUB [0:79];
]
PROCEDURE  A_SUBPROGRAM;
.
.
.
```

If the COBOL host program was compiled with COBOL68 or COBOL74, the USE statements in the bound program look like the following:

```
.
.
.
USE FILE-FROM-HOST/REC-FROM-HOST FOR REC_FROM_SUB;
.
.
.
```

If the COBOL host program was compiled with COBOL85, the USE statements in the bound program look like the following:

```
.
.
.
USE FILE-FROM-HOST FOR FILE_FROM_SUB;
USE REC-FROM-HOST FOR REC_FROM_SUB;
.
.
.
```

CALL PROGRAM DUMP (COBOL68 Only) (Change)

CALL PROGRAM DUMP requested a program dump in COBOL68. This must be changed to the COBOL74/85 CALL SYSTEM DUMP syntax.

CALL SYSTEM WITH or ZIP Statement (COBOL68 Only) (Change)

In COBOL68, you could use the CALL SYSTEM WITH <data-name or file-name> statement or the ZIP <data-name or file-name> statement to pass a control message to the operating system. COBOL85 only supports the <data-name> construct when invoking SYSTEM WFL. In COBOL85.

- CALL SYSTEM WITH <data-name> changes to CALL SYSTEM WFL USING <data-name>
- ZIP <data-name> changes to CALL SYSTEM WFL USING <data-name>

The following constructs are not supported in COBOL85:

- CALL SYSTEM WITH <file-name>
- ZIP <file-name>

CHECKPOINT Statement (COBOL68 Only) (Obsolete)

COBOL85 neither supports the CHECKPOINT statement nor provides an equivalent construct.

Class Condition (Change)

The ALPHABETIC test is true for the uppercase letters, the lowercase letters, and the space character. The ALPHABETIC-UPPER test is true for uppercase letters and the space character. The ALPHABETIC-LOWER test is true for lowercase letters and the space character.

Because lowercase letters have been added to the ALPHABETIC test, a COBOL74 program that contains this test could behave differently when compiled in COBOL85. In COBOL74, the ALPHABETIC class condition test is true only for the uppercase and space characters. If your program must distinguish between uppercase and lowercase letters, use the REPLACE statement to replace references to ALPHABETIC with ALPHABETIC-UPPER:

```
77  ALPHA-DATA         PIC X(10) VALUE "ABCDEfghij".
 .
 .
 .
    IF ALPHA-DATA IS ALPHABETIC . . .
```

The condition test in the example returns a true value in COBOL85 but a false value in COBOL74. Note that this is not a problem if your program does not need to distinguish between uppercase and lowercase letters.

CLOSE HERE Statement (COBOL68 Only) (Change)

In COBOL68, the CLOSE HERE [NO REWIND] statement enables you to write over the last portion of a tape file or to add to an existing tape file.

In COBOL85 the CLOSE HERE statement changes into a combination of CLOSE NO REWIND and OPEN OUTPUT NO REWIND statements.

CLOSE WITH LOCK (COBOL68 Only) (Change)

The CLOSE WITH LOCK statement should be replaced with the CLOSE WITH SAVE statement.

CODE SEGMENT-LIMIT Clause (Obsolete)

The CODE SEGMENT-LIMIT clause of the OBJECT-COMPUTER paragraph has been placed in the obsolete element category. The function performed by this clause is considered to be more appropriately performed by the host operating system than the individual COBOL program.

The COBOL85 compiler ignores this clause. Thus, for clarity in your program, it is recommended that you designate all occurrences of the CODE SEGMENT-LIMIT clause as comments.

COMP-2 Group Item Alignment (COBOL68 Only) (Change)

In COBOL68, group items were aligned according to their USAGE. For example, a COMP-2 group item could be aligned on a DIGIT boundary. In COBOL85, group items must be treated as USAGE IS DISPLAY items, and as such must both begin and end on a byte boundary. In COBOL68 , when the compiler control option WARNCOMP2 is set, the compiler issues a syntax warning when it encounters a COMP-2 group item that does not begin on a byte boundary. Any existing files with the old alignment must be converted.

Compiler Control Options (Obsolete)

COBOL85 does not allow several compiler option that are allowed in COBOL68. It also does not allow several compiler options that are allowed in COBOL68 and COBOL74.

The following table

- Lists the COBOL68 compiler options that are no longer available in COBOL85
- Shows the availability of the option in COBOL74
- Indicates the appropriate migration that is required for COBOL85.

COBOL68 Compiler Option	In COBOL74?	Migration to COBOL85
ANALYZE	No	Delete option
CHECK	No	Delete option
CLEAR	Yes	Delete option
COMP	No	Delete option
GLOBAL	Yes	Change to COMMON
INTRINSIC	No	Delete option
LIB$	Yes	Delete option
LIBDOLLAR	Yes	Delete option
LIST$	Yes	Change to LISTDOLLAR
LISTDELETED	Yes	Delete option
OLDNOT	No	Delete option
SECGROUP	No	Delete option
STACK	No	Change to MAP

COMPUTE with FROM or EQUALS Statement (COBOL68 Only) (Obsolete)

The COMPUTE with FROM or EQUALS statement is not implemented in COBOL74 or COBOL85. You should replace occurrences of the COMPUTE with FROM or EQUALS statement with the COMPUTE statement.

CONSTANT SECTION (COBOL68 Only) (Obsolete)

Replace the CONSTANT SECTION header of the DATA DIVISION with the $SET OPT3 compiler option. Insert the $RESET OPT3 compiler option before the next SECTION or the PROCEDURE DIVISION.

The items within the range of the OPT3 compiler option are regarded as constants by the COBOL85 compiler.

COPY Statement (Change)

When you replace a PICTURE character-string in the COPY statement, you must precede the string with the word PICTURE (or PIC). The string being replaced is represented by pseudo-text-1 in the general format of the COPY statement.

COPY. . .REPLACING Statement with Picture Character Strings (Change)

Replace Picture character strings using the COPY. . .REPLACING statement, ensuring that the word PICTURE (or PIC) precedes the replacement string.

DATA DIVISION Clauses (COBOL68 Only) (Obsolete)

DATA DIVISION clauses RANGE, RECORD AREA, SEGMENT and SIZE are not implemented in COBOL74 or COBOL85. They must be deleted.

DATA RECORDS Clause (Obsolete)

The DATA RECORDS clause of the file description entry has been placed in the obsolete element category. This clause serves only as documentation for the names of data records in their associated file. It is recommended that you delete this clause.

Debug Module (COBOL74 Only) (Obsolete)

The Debug module has been placed in the obsolete element category. To convert this feature, you can create trace statements in your program that manually test the code.

Direct I/O (COBOL68 Only) (Obsolete)

Direct I/O is not implemented in COBOL74 or COBOL85. The SELECT clause which assigns a file to DIRECT must be changed. The RECORD AREA clause for WORKING-STORAGE must also be deleted. Any of the PROCEDURE DIVISION verbs such as DEALLOCATE must be deleted.

DIV and MOD Operators (Change)

The operators MOD (remainder divide) and DIV (integer divide) are not allowed in COBOL85. The MOD and DIV features of COBOL68 should be converted to the intrinsic functions MOD and DIV. Either of these functions may use another function call as its parameter.

DIVIDE Statement (Change)

In ANSI COBOL85, any subscripts for identifier-4 in the REMAINDER phrase are evaluated after the result of the DIVIDE operation is stored in identifier-3 of the GIVING phrase.

In ANSI COBOL74, the point at which any subscript in the REMAINDER phrase is determined during the processing of the DIVIDE statement was undefined. A Series COBOL74 evaluated identifier-3 before the DIVIDE operation. For example,

```
DIVIDE THREE INTO TEN GIVING RESULTS
     REMAINDER REM(RESULTS).
```

In the preceding example, the value returned in COBOL85 for REM(RESULTS) is 1. In COBOL74, the value returned for REM(RESULTS) would be its value before the execution of the DIVIDE statement.

DIVIDE Statement with the MOD Option (COBOL68 Only) (Obsolete)

The MOD option of the DIVIDE statement is not implemented in COBOL74 or COBOL85. You should replace occurrences of the DIVIDE statement with the MOD option with FUNCTION MOD.

DMSII <ON EXCEPTION> Clause (COBOL68 Only) (Change)

In COBOL68, you can use the ELSE phrase to specify the statements to be executed when an exception does not occur. The COBOL74 compiler does not permit the ELSE clause to be used with the ON EXCEPTION clause. The COBOL85 compiler offers the NOT ON EXCEPTION clause to be used in place of the ELSE clause, which was allowed in COBOL68. For example,

In COBOL68, this code was possible:

```
FIND <set-name> WHERE <condition>
ON EXCEPTION
        <statement>
ELSE
        <statement>
```

In COBOL85, the code becomes:

```
FIND <set-name> WHERE <condition>
ON EXCEPTION
        <statement>
NOT ON EXCEPTION
        <statement>
```

DUMP Statement (COBOL68 Only) (Obsolete)

COBOL85 neither supports the DUMP statement nor provides an equivalent construct.

ELSE Phrase of Statements in PROCEDURE DIVISION (Change)

The ELSE phrase which was associated with the phrases INVALID KEY, AT END, AT END-OF-PAGE, ON EXCEPTION and ON SIZE ERROR in COBOL68 should be changed to the following syntax in COBOL85.

COBOL68 and COBOL74 Syntax	COBOL85 Syntax
ELSE phrase with ON SIZE ERROR phrase	NOT ON SIZE ERROR of COMPUTE, MULTIPLY, DIVIDE, ADD and SUBTRACT
ELSE phrase with ON EXCEPTION phrase	NOT ON EXCEPTION of DMSII statement such as FIND, LOCK, OPEN, CREATE, STORE and so on.
ELSE phrase with AT END phrase	NOT AT END of READ and RETURN
ELSE phrase with END-OF-PAGE phrase	NOT END-OF-PAGE of WRITE
ELSE phrase with INVALID KEY phrase	NOT INVALID KEY of READ and WRITE

ENTER Statement (COBOL68 Only) (Obsolete)

The ENTER statement was the predecessor of the CALL statement for the calling of external subprograms. It is not allowed in COBOL74 and has been placed in the obsolete category. The ENTER statement should be replaced with an appropriate CALL statement.

ENVIRONMENT DIVISION, Paragraphs in I-O-CONTROL (Obsolete)

Although COBOL85 supports the MULTIPLE FILE TAPE and RERUN clauses, both have been placed in the obsolete element category.

EXAMINE Statement (COBOL68 Only) (Obsolete)

The EXAMINE statement is not implemented in COBOL74 or COBOL85. You should replace occurrences of the EXAMINE statement with the INSPECT statement.

EXECUTE Statement (COBOL68 Only) (Obsolete)

The EXECUTE statement is not implemented in COBOL74 or COBOL85. You should replace occurrences of the EXECUTE statement with the RUN statement.

File Description Clauses (COBOL68 Only) (Obsolete)

Some file description clauses of COBOL68 are no longer supported. The following clauses must be changed or deleted.

- BLOCK CONTAINS which is expressed in words
- FILE CONTAINS and RECORD CONTAINS which are expressed in words
- RECORDING MODE
- SAVE-FACTOR

File with EXTMODE=HEX (Change)

A COBOL68 program can generate a file whose EXTMODE is HEX if the first 01 record entry under the file description is of USAGE COMP-2. A COBOL74 program can generate a file whose EXTMODE is HEX if the first 01 record entry under the file description is of USAGE COMP. In COBOL85, all group items are treated as EBCDIC arrays, and no way exists to manipulate a file whose EXTMODE is HEX.

You must convert an existing file whose EXTMODE is HEX into a file whose EXTMODE is EBCDIC.

FIRST DETAIL Clause (Change)

In COBOL74, if the report description (RD) includes a HEADING clause but not a FIRST DETAIL clause, the first detail is increased to the first line after the page heading. In COBOL85, under the same circumstances, the first detail is not increased to the first line after the page heading.

GLOBAL Clause (Change)

The term GLOBAL, which is used in the SELECT clause of the ENVIRONMENT DIVISION, the data-name/FILLER definitions in the DATA DIVISION and the USE PROCEDURE statement of the PROCEDURE DIVISION, is an extension to ANSI-68/74 COBOL. COBOL programs compiled at lexicographic level 3 or higher could use untyped procedures, files, and certain variables in the outer block of the host program by declaring them GLOBAL. In COBOL85, the term COMMON represents these features and the term GLOBAL is used to specify a global object in a nested source program environment. All references to GLOBAL must be changed to COMMON for COBOL85.

Hardware Names (COBOL68 Only) (Obsolete)

Hardware names in SELECT clause listed below are obsolete. They must be changed or deleted.

BACKUP DISK
BACKUP TAPE
BACKUP TAPE/DISK
CARD-PUNCH
CARD-READER
CARD-READERS
DISKPACK
DISKPACKS
DISPLAY-UNIT
KEYBOARD
MESSAGE-PRINTER
PAPER-TAPE-PUNCH
PAPER-TAPE-READER
PETAPE
PRINTER BACKUP
PUNCH BACKUP
READER
SORT-TAPE
SORT-TAPES
SPO
TAPES
TAPE7
TAPE9

HEX to EBCDIC Translation (COBOL68 Only) (Change)

A COMP-2 group item declared in a COBOL68 program is regarded as a hexadecimal array. If the group item is moved to a DISPLAY item, COBOL68 performs a HEX to EBCDIC translation.

Hexadecimal Literal Definition (COBOL68 Only) (Change)

The COBOL68 description of hexadecimal literals is not the same as that of COBOL74 and COBOL85. COBOL68 permits the following two ways to represent a hexadecimal literal:

- Specifying a non-numeric literal as a COMP-2 group item. The literal is regarded as a hexadecimal literal and is left justified.
- Delimiting both ends of the literal by the character "@", like COBOL74 or COBOL85. The feature is available only while the B2500 compiler option is set. The literal is aligned at the rightmost character position in the data item.

I-O Status (Change)

I-O status values have been added. I-O status values did not exist in COBOL68. COBOL74 implemented a limited number of I-O status values.

The additional I-O status values enable you to distinguish among many different exception conditions, which you can then treat in a variety of ways. The intention in ANSI-85 COBOL is to define status values for previously undefined I-O situations. You can check for these error conditions and take corrective action for specific error conditions where appropriate.

If your COBOL74 program checks I-O status values, add the new status values to your list.

The individual I-O status values affected are described in the following paragraphs:

- I-O status = 04. A READ statement is successfully executed, but the length of the record processed does not conform to the fixed file attributes for the file.

  Note that this situation (an attempt to write or rewrite a record that is too large or too small) cannot occur for records written by a program compiled with COBOL85.
- I-O status = 05. An OPEN statement is successfully executed, but the referenced optional file is not present at the time the OPEN statement is executed.

  This status value enables you to check if the file referred to by an OPEN statement exists before the first READ statement. Programs affected would be those that use the OPTIONAL phrase in the Sequential I-O module and then examine the I-O status for successful completion of the OPEN INPUT statement.

- I-O status = 07. The input-output statement is successfully executed. However, for a CLOSE statement with the NO REWIND, REEL/UNIT, or FOR REMOVAL phrase, or for an OPEN statement with the NO REWIND phrase, the referenced file is on a non-reel/unit medium.

  With mass-storage files, if the instances of OPEN and CLOSE mentioned in the previous paragraph are overlooked, the I-O status value makes successful completion notification possible, while preserving the information in case you wish to take a specific action.

- I-O status = 14. A sequential READ statement is attempted for a relative file, and the number of significant digits in the relative record number is larger than the size of the relative key data item described for the file.

  COBOL74 did not define the result of a Format 1 READ statement referencing a relative file when the number of significant digits of the relative record number is larger than the relative key data item. This status value defines the result.

- I-O status = 24. An attempt is made to write beyond the externally defined boundaries of a relative or indexed file; or a sequential WRITE statement is attempted for a relative file and the number of significant digits in the relative record number is larger than the size of the relative key data item described for the file.

  COBOL74 did not define the result if the number of significant digits of the relative record number is larger than the relative key data item. The ANSI COBOL85 definition of the status value of 24 has been modified to include this case.

  If your program sequentially writes more records than the maximum value allowed by the PICTURE of the relative key data item, it will be affected by this change. Otherwise, it will not.

- I-O status = 35. An OPEN statement with the INPUT phrase is attempted on a required file that is not present.

  COBOL74 did not specify what happened when a file that is not declared as optional is not present when an OPEN statement is executed. This I-O status value tests for this condition.

- I-O status = 37. An OPEN statement is attempted on a file that is required to be a mass-storage file but is not.

  COBOL74 did not specify what happened when a file that is supposed to be a mass-storage file but is not a mass-storage file is opened. This I-O status value tests for this condition.

- I-O status = 38. An OPEN statement is attempted on a file previously closed with lock.

  COBOL74 did not specify what happened if an attempt is made to reopen a file that was closed with lock during the current execution of the run unit. This I-O status value tests for this condition.

- I-O status = 39. An OPEN statement is unsuccessful because a conflict was detected between the fixed file attributes and the attributes specified for that file in the program.

  COBOL74 did not specify what happened if the fixed file attributes conflict with the attributes specified for a file in the program. Fixed file attributes include the organization, the code set, the minimum and maximum logical record size, the record type, the blocking factor, the padding character and the record delimiter. Indexed files have the additional fixed file attributes of the prime record key, the alternate record keys, and the collating sequence of the keys. This I-O status value tests for these conditions.

- I-O status = 41. An OPEN statement is attempted for a file in the open mode.

  COBOL74 did not specify what happened if you tried to OPEN an already opened file. This I-O status value tests for this condition.

- I-O status = 42. A CLOSE statement is attempted for a file not in the open mode.

  COBOL74 did not specify what happened if you tried to CLOSE an already closed file. This I-O status value tests for this condition.

- I-O status = 43. For a mass-storage file in the sequential access mode, the last input-output statement executed for the associated file before the execution of a DELETE or REWRITE statement was not a successfully executed READ statement.

  COBOL74 did not specify what happened in this case. This I-O status value tests for this condition.

- I-O status = 44. A boundary violation exists because an attempt is made to rewrite a record to one of the following types of files, and the record is not the same size as the record being replaced:

  – a sequential file

  – a relative file in level 1 of the Relative I-O mode

  – an indexed file in level 1 of the Indexed I-O module

  COBOL74 did not specify what happened when the number of character positions in the new record created by a REWRITE statement did not equal the number of character positions in the record being replaced. This I-O status value tests for these conditions.

- I-O status = 46. A sequential READ statement is attempted on a file opened in the input or I-O mode and a valid next record has not been established because either:

  – the preceding START statement was unsuccessful

  – the preceding READ statement was unsuccessful but did not cause an at end condition

  – the preceding READ statement caused an at end condition

  COBOL74 specified that in these circumstances execution of the READ statement was illegal or its execution was unsuccessful, but failed to specify a status code to indicate the situation. This I-O status value tests for these conditions. Note that I-O status value 46 can occur only if no corrective action is taken following the previous READ or START statement.

- I-O status = 47. The execution of a READ or START statement is attempted on a file not opened in the input or I-O mode.

  COBOL74 did not specify what happens if a file is not opened in the input or I-O mode at the time a READ or START statement is executed. This I-O status value tests for this condition.

- I-O status = 48. The execution of a WRITE statement is attempted on either:

  – a sequential file not opened in the output or extend mode

  – a relative or indexed file not opened in the I-O, output, or extend mode

  COBOL74 did not specify what happens if a file that is required to be opened in one of the modes specified is not. This I-O status value tests for this condition.

- I-O status = 49. The execution of a DELETE or REWRITE statement is attempted on a file not opened in the I-O mode.

  COBOL74 did not specify what happens if a file that is required to be opened in the I-O mode is not. This I-O status value tests for this condition.

Installation Intrinsic (COBOL68 Only) (Obsolete)

Neither COBOL74 or COBOL85 support installation intrinsic functions. This feature was implemented in COBOL68, ALGOL and FORTRAN to support local intrinsic functions before the library mechanism was introduced. Any installation intrinsic should be replaced with COBOL85 intrinsic functions or a run-time library.

Intrinsic Functions (COBOL68 Only) (Change)

Unlike COBOL68, the COBOL85 syntax for an intrinsic function consists of the word FUNCTION, the name of a specific predefined function, and one or more arguments. You must

- Add the word FUNCTION before each function name.
- Change the function name ARCTAN and LN of COBOL68 to ATAN and LOG, respectively.

Kanji (Change)

In COBOL74, data items and literals not in the standard American English character set were referred to as Kanji data items and literals. In COBOL85, USAGE NATIONAL replaces Kanji for data items, literals, and all other usages.

KEYEDIO (Change)

In the COBOL74 compiler, KEYEDIO was allowed for handling indexed I/O. In COBOL85, only KEYEDIOII is allowed. To access KEYEDIO data from COBOL85 programs, you must either convert the files to KEYEDIOII or set the FILEORGANIZATION file attribute to INDEXED in the program which will use them.

To convert KEYEDIO files to KEYEDIOII, use SYSTEM/KEYEDIOII/UTILITY. For details, refer to the *MCP/AS KEYEDIOII Programming Reference Manual*. To access existing KEYEDIO files without converting them in the program which accesses the files, set FILEORGANIZATION IS INDEXED in the VALUE clause of the FD for the appropriate file.

As an alternative you can use the CHANGE statement to set FILEORGANIZATION OF <file-name> TO VALUE INDEXED before you open the file.

To convert KEYEDIO files to KEYEDIOII, use SYSTEM/KEYEDIOII/UTILITY. For details, refer to the *MCP/AS KEYEDIOII Programming Reference Manual*. To access existing KEYEDIO files without converting them, in the program which accesses the files, set FILEORGANIZATION IS INDEXED in the VALUE clause of the FD for the appropriate file. As an alternative you can use the CHANGE statement to set FILEORGANIZATION OF <file-name> TO VALUE INDEXED before you open the file.

LABEL RECORDS Clause (Obsolete)

The LABEL RECORDS clause in the file description entry has been placed in the obsolete element category and has been made an optional clause.

Specifying the presence of file labels is considered a function of the operating system and as such does not belong in the COBOL program.

COBOL85 defines the LABEL RECORDS clause as optional. If the clause is present, the information will be used. If it is not present, a syntax error will not occur. For the sake of clarity, make any use of the LABEL RECORDS clause a comment.

Libraries and Interprogram Communication

The contents of identifier-1 of the CALL statement includes an important change in COBOL85 for libraries (and interprogram communication). This change provides both the new features of ANSI Standard nested programs and the continued support of explicit libraries, which is a COBOL 74 extension to the ANSI Standard.

Identifier-1 must be defined as an alphanumeric data item whose value can be a program-name. The program-name is considered a program-id for nested programs and an object-file-name for separately compiled programs. The program-name is no longer recognized as a run-time-library-id, so you need to modify references to identifier-1. The best way to do this is to change identifier-1 into a literal.

For example,

```
CALL IC208 IN OBJECT/IC208.
```

This call on a run-time library should be modified to:

```
CALL "IC208 IN OBJECT/IC208".
```

Another way to achieve the same result would be to use a regular object-file-name in place of the run-time-library-id. In either case, the run-time-library-id will no longer be recognized.

LINAGE Clause in Extend Mode (Change)

In ANSI COBOL85, files for which the LINAGE clause has been specified must not be opened in extend mode. When a file is opened in extend mode in COBOL68, the LINAGE-COUNTER data item is not set to 1; it remains at 0. The LINAGE clause should be removed from the associated file statement if the program opens the file in extend mode.

LOCK with COMP or with COMP-1 Statement (COBOL68 Only) (Obsolete)

COBOL85 neither supports the LOCK with COMP or with COMP-1 statement nor provides an equivalent construct.

LOWER-BOUND and UPPER-BOUND (Change)

When LOWER-BOUND and UPPER-BOUND reference nonnumeric data items, they must be changed to LOW-VALUES and HIGH-VALUES respectively. However, when referencing a numeric item, LOWER-BOUND should be changed to the figurative constant ZERO and UPPER-BOUND should be changed to reference the SYMBOLIC CHARACTERS named UPPER-BOUND. This SYMBOLIC CHARACTERS entry is inserted in the SPECIAL-NAMES paragraph of the converted program.

The following table shows the replacement of LOWER-BOUND and UPPER-BOUND for COBOL85:

COBOL68	COBOL85
	SPECIAL-NAMES. SYMBOLIC CHARACTERS UPPER-BOUND UPPER-BOUNDS ARE 250 250.
01 GROUPDATA. 03 DATAX PIC X(06). 03 DATA9 PIC 9(06). MOVE LOWER-BOUND TO DATAX. MOVE UPPER-BOUND TO DATAX. MOVE LOWER-BOUND TO DATA9. MOVE UPPER-BOUND TO DATA9.	01 GROUPDATA. 03 DATAX PIC X(06). 03 DATA9 PIC 9(06). MOVE LOW-VALUES TO DATAX. MOVE HIGH-VALUES TO DATAX. MOVE ZERO TO DATA9. MOVE UPPER-BOUND TO DATA9.
IF DATAX = LOWER-BOUND . . . IF DATAX = UPPER-BOUND . . . IF DATA9 = LOWER-BOUND . . . IF DATA9 = UPPER-BOUNDS . . .	IF DATAX = LOW-VALUES . . . IF DATAX = HIGH-VALUES . . . IF DATA9 = ZERO . . . IF DATA9= UPPER-BOUNDS . . .

MONITOR Statement (COBOL68 Only) (Obsolete)

COBOL85 neither supports the MONITOR statement nor provides an equivalent construct. COBOL85 Test and Debug System (TADS) provides monitoring capability.

MOVE Statement (COBOL68 Only) (Change)

In a blind MOVE statement, the sending field is DISPLAY or COMP-2 and the receiving field is COMP, COMP-1, or COMP-4 of any size within one word (48 bits). Data from the first six bytes of the sending field item is moved unchanged into the low-order six bytes of the receiving field item. This should be changed into an appropriate reference modification MOVE.

The COBOL MOVE statement in which one of the operands is a group item and the other operand is an elementary numeric item might produce a result contrary to the requirements of the COBOL-1985 standard.

The $C68MOVEWARN option can be used to issue a warning message on such a MOVE statement.

MEMORY SIZE Clause (Obsolete)

The MEMORY SIZE clause of the OBJECT-COMPUTER paragraph has been placed in the obsolete element category.

This feature is considered to be a function more appropriately controlled by the host operating system. In COBOL74, the MEMORY SIZE clause was optional. Thus, there are no standard conforming COBOL implementations that require the use of the MEMORY SIZE clause to specify the object computer memory size.

This statement can only be used with the SORT statement in COBOL85. Since the SORT statement can also specify MEMORY SIZE, and that statement takes precedence over the OBJECT-COMPUTER paragraph, you should either remove any MEMORY-SIZE statement in the OBJECT-COMPUTER paragraph or move it to the SORT statement.

If you don't use a SORT statement, and you specify the MEMORY SIZE clause in the OBJECT-COMPUTER paragraph, the specification is ignored.

MULTIPLE FILE TAPE Clause (Obsolete)

The MULTIPLE FILE TAPE clause in the I-O-CONTROL paragraph of the Environment Division has been placed in the obsolete element category.

This functionality is more appropriately provided by the operating system and not the individual COBOL program. COBOL85 provides this functionality through the use of file attributes, as did COBOL74. Therefore, this change should not materially affect your programs.

Nested Source Programs (Change)

Programs can be contained in other programs.

The constructs associated with nesting programs include:

- COMMON clause

  The COMMON clause specifies that a program is contained within another program. A common program can be called from programs other than that containing it.

- End Program header

  The End Program header indicates the end of the named source program.

- EXTERNAL clause

  The EXTERNAL clause specifies that a data item or a file connector is external and can be accessed and processed by other calling or called programs.

- GLOBAL clause

  The GLOBAL clause specifies that a data-name, a file-name, or a report-name is a global name that is available to every program which declares it.

- INITIAL clause

  The INITIAL clause specifies that the program will be in its initial state whenever the program is called.

It will not be necessary to modify your programs if you do not intend to take advantage of the ability to nest your programs. However, if you currently make extensive use of libraries or IPC, the ability to nest your programs may save you design and maintenance effort in the future. For more information about nested source programs, refer to Section 10, "Interprogram Communication."

NOTE Statement (COBOL68 Only) (Obsolete)

The NOTE statement is not implemented in COBOL74 or COBOL85. You should replace occurrences of the NOTE statement with comments (*).

OBJECT-COMPUTER Paragraph (Obsolete)

The MEMORY SIZE clause specifies the main storage requirement for the program. It is ignored in COBOL85 unless there is a SORT statement in the program. If there is a SORT statement in the program, this clause should be moved into the SORT statement. If there is not a SORT statement in the program, the clause should be marked as a comment since MEMORY SIZE clause becomes obsolete with the next ANSI standard.

The SEGMENT-LIMIT and CODE SEGMENT-LIMIT clauses cause the COBOL85 compiler to emit a warning that the feature is not implemented. These clauses should be commented out. The STACK SIZE clause is ignored by the COBOL85 compiler without any warning. It should be commented out.

OCCURS Clause at 01 Level (COBOL68 Only) (Obsolete)

The OCCURS clause on an 01 level data item is not supported by COBOL85.

OPEN with REEL–NUMBER (Format 2) Statement (COBOL68 Only) (Obsolete)

COBOL85 neither supports the OPEN with REEL–NUMBER (Format 2) statement nor provides an equivalent construct.

PICTURE Characters (COBOL68 Only) (Change)

The PICTURE characters J and K must be replaced with S.

PICTURE DEPENDING ON Clause with PICTURE Character L (COBOL68 Only) (Obsolete)

This feature is used to denote a variable length elementary item. The 'L' and the DEPENDING ON clause must be deleted.

PL/I ISAM (COBOL68 Only) (Obsolete)

The indexed sequential access method (ISAM) facility supports indexed files in COBOL68. In COBOL85, KEYEDIOII supports indexed files.

You must convert ISAM data files to KEYEDIOII data files by following this process.

- Write a COBOL68 program to create a flat file from the ISAM file.
- Write a COBOL85 program that reads records from the flat file and writes the records into a COBOL85 indexed file that has the same key declarations as the COBOL68 ISAM file.

During the reading and writing process, the COBOL85 program converts the flat file into a KEYEDIOII data file with the same access keys that existed in the COBOL68 ISAM file.

Procedure Division Header (Change)

A data item appearing in the USING phrase of the Procedure Division header must not have a REDEFINES clause in its data description entry.

Allowing an item with a REDEFINES clause to be specified in the USING phrase of the Procedure Division header could allow programming errors to remain undetected.

If your program specified a redefined item in the USING phrase of the Procedure Division header, you can convert it by substituting the redefined item, as in the following syntax:

```
      .
      .
      .
01 DUMMY.
   03 A  PIC 9(10).
   03 B  REDEFINES A PIC X(10)
     .
     .
     .
PROCEDURE DIVISION USING A.
```

Optional Procedure-name-1 in GO TO Statement (Obsolete)

The optional procedure-name-1 in the GO TO statement has been placed in the obsolete element category. Because the ALTER statement has been placed in the obsolete element category, you no longer have the option of omitting procedure-name-1 from a GO TO statement.

RELATIVE KEY (Change)

The performance of all I/O statements that act upon a relative file declared with a relative key can be significantly improved by declaring the appropriate key as follows:

```
77 USERKEY     REAL.
```

REMOTE File (COBOL68 Only) (Change)

The READ...INVALID clause for a REMOTE file must be changed to READ...AT END.

REVERSED Phrase of the OPEN Statement (Obsolete)

The REVERSED phrase has been placed in the obsolete element category.

COBOL85 will continue to support the REVERSED phrase of the OPEN statement, as defined by Standard COBOL. Note however that the next revision of Standard COBOL might not contain this phrase.

Reserved Words in COBOL74 (COBOL68 Only) (Change)

The following COBOL74 reserved words were not reserved in COBOL68:

BINARY	COMMUNICATION	COPY-NUMBER	DEBUG-CONTENTS
DEBUG-ITEM	DEBUG-LINE	DEBUG-NAME	DEBUG-SUB-1
DEBUG-SUB-2	DEBUG-SUB-3	DEBUGGING	DICTIONARY
EGI	ESI	FIELD	FORM
FORM-KEY	KANJI	OFFSET	OPTIMIZE
PROCEDURES	READ-OK	READ-WRITE	REFERENCE
RELATIVE	RIBBON	SORT-MERGE	STACK
SUB-QUEUE-1	SUB-QUEUE-2	SUB-QUEUE-3	TAG-KEY
TAG-SEARCH	THEN	TIMER	TODAYS-NAME
WRITE-OK			

More reserved words have been added to COBOL85 as follows:

ALPHABET	ALPHABETIC-LOWER	ALPHABETIC-UPPER
ALPHANUMERIC	ALPHANUMERIC-EDITED	
BINARY	CLASS	COMMON
CONTENT	CONTINUE	CONVERTING
DAY-OF-WEEK	END-ABORT-TRANSACTION	END-ADD
END-ASSIGN	END-BEGIN-TRANSACTION	END-CALL
END-CANCEL	END-CLOSE	END-COMPUTE
END-CREATE	END-DELETE	END-DIVIDE
END-EVALUATE	END-FIND	END-FREE
END-GENERATE	END-IF	END-INSERT
END-LOCK	END-MODIFY	END-MULTIPLY
END-OPEN	END-PERFORM	END-READ
END-RECEIVE	END-RECREATE	END-REMOVE
END-RETURN	END-REWRITE	END-SAVE
END-SEARCH	END-SECURE	END-SET
END-START	END-STORE	END-STRING
END-STRUCT	END-TRANSACTION	END-UNSTRING
END-WRITE	EVALUATE	EXTERNAL
FALSE	GLOBAL	NATIONAL
NUMERIC-EDITED	ORDER	OTHER
PACKED-DECIMAL	PADDING	PURGE
REFERENCE	REPLACE	Standard-2
SW1	SW2	SW3
SW4	SW5	SW6
SW7	SW8	TEST
THEN	TRUE	

If a word contains an underscore (_), the compiler converts the underscore into a hyphen.

SAME AREA/SAME RECORD AREA (Change)

COBOL85 requires that all SAME AREA/SAME RECORD AREA clauses consist of only one sentence.

A file may not appear in more than one SAME AREA clause. Also, a file may not appear in more than one SAME RECORD AREA clause. If one or more file-names of a SAME AREA clause appear in a SAME RECORD AREA clause, all of the file-names in that SAME AREA clause must appear in the SAME RECORD AREA clause.

SDF Plus Interface (Change)

Unlike DMSII and COMS, the SDF Plus interface of COBOL85 is quite different from that of COBOL74. While COBOL74 had embedded syntax such as READ FORM or WRITE FORM statements, the CALL statement, combined with the CHANGE and SDF Plus interfaces, handles the SDF Plus features in COBOL85.

SEEK with KEY CONDITION Clause Statement (COBOL68 Only) (Obsolete)

COBOL85 neither supports the SEEK with KEY CONDITION clause statement nor provides an equivalent construct.Miscellaneous SELECT clauses are obsolete and should be deleted.

SELECT Clauses (COBOL68 Only) (Obsolete)

- SINGLE
- FILE-LIMIT IS
- BY AREA
- FILE-LIMITS ARE
- BY CYLINDER
- RESERVE NO
- SAVE
- RESERVE data-name

Segmentation Module (Obsolete) (COBOL74 Only)

The Segmentation module has been placed in the obsolete element category.

The functions of the Segmentation module are provided by the operating system, external to the COBOL source code. COBOL85 will ignore segment numbers. You can retain them in your program code. However, we recommend that you make the Segmentation module into a comment for the sake of clarity.

SET Statement for Task Attributes (Obsolete)

In COBOL85, you must use the CHANGE statement to set or change task attributes. In COBOL68 and COBOL74, you can use either the SET or the CHANGE statement to set or change task attributes, although the SET statement is considered obsolete in COBOL74.

For example, in COBOL68, the syntax for changing the value of the TASKVALUE task attribute is as follows:

```
SET TASK-ID(TASKVALUE) TO 1
```

In COBOL85, the equivalent syntax is as follows:

```
CHANGE ATTRIBUTE TASKVALUE OF TASK-ID TO 1
```

SIZE DEPENDING ON Clause (COBOL68 Only) (Obsolete)

The SIZE clause without the DEPENDING ON phrase should be removed. The SIZE DEPENDING ON <size > should also be removed from WORKING-STORAGE items. All MOVE statements that reference those data items should be changed to a reference modification MOVE <dataname> (1:<size>). When the SIZE DEPENDING ON clause occurs in the FILE SECTION, replace it with the RECORD CONTAINS clause in COBOL85.

SORT Statement (Change)

In COBOL68, when MEMORY SIZE and DISK SIZE clauses were specified in a SORT statement, they had to follow the INPUT PROCEDURE and OUTPUT PROCEDURE clauses. In COBOL85, the MEMORY SIZE and DISK SIZE clauses must precede the INPUT PROCEDURE and OUTPUT PROCEDURE clauses.

STOP Literal Statement (Obsolete)

The literal variation of the STOP statement is now in the obsolete element category.

This format of the STOP statement suspends the execution of the program and displays the literal on the operator display terminal (ODT). In COBOL74, you had to reinitiate your program by typing ?OK on your terminal. Then the program would continue with the next executable statement.

In COBOL85, this format of the STOP statement will continue to work as it did in COBOL74; however, you should replace any literal variations of the STOP statement with a set of DISPLAY/ACCEPT statements:

```
PROCEDURE DIVISION.
      PARA-1.
         STOP "Error in PARA-1".
```

In this example, the STOP statement suspends the run unit. You must reinitiate the run unit by typing ?OK on your terminal.

To modify this code for COBOL85, replace the literal with a DISPLAY statement, and replace the STOP statement with an ACCEPT statement, as follows:

```
PROCEDURE DIVISION.
      PARA-1.
         DISPLAY "Error in PARA-1".
         ACCEPT keyboard-option.
```

In this example, the DISPLAY statement will display the message on the ODT, and the ACCEPT message will reinitiate the run unit.

SYNC LEFT/RIGHT (COBOL68 Only) (Change)

SYNCHRONIZED in COBOL68 can be followed optionally by LEFT or RIGHT. LEFT causes the data to start on a word boundary; RIGHT causes data to end on a word boundary. When no alignment is specified, the default is RIGHT. In many cases the data is stored in one or more words of memory, according to its size. In COBOL85 the LEFT and RIGHT words are considered commentary and have no affect on alignment. SYNCHRONIZED simply causes data to start on the natural boundary appropriate for its USAGE, occupying only as much memory as appropriate for its size.

For packed decimal and alphanumeric items, the boundary is the next byte. For boolean items or word sized numeric items, the boundary is a word. For double word sized numeric items, the boundary is the next even word. To view the effect of SYNCHRONIZED on general record layout, set the compiler options $MAP and $LIST around record definitions in WORKING-STORAGE and FILE sections. It might also be necessary to run the program far enough to install data in the records and then take a dump with ARRAYS in order to examine the actual alignment of data.

When evaluating data definitions to decide where to place FILLER, be aware that records in WORKING-STORAGE are pooled in COBOL85, but not in COBOL. While each 01 record in COBOL is a separate area, the same is not true in COBOL85. In COBOL85 the 01 records are pooled into one area until full and then a new area is created to pool subsequent records. Each record in a pooled area however, begins on a word boundary. From this you can determine the offset, alignment, and size of the data fields with respect to the start of the record. Compare this information with the offset, alignment, and size of data fields in the COBOL record and make adjustments as needed. FILLER will almost definitely be needed to get the same effect for the RIGHT option. Matching record layout is most important when moving data between storage media and program records, because data on the media is position dependent. Once data is safely in the program records, it can be reliably accessed by its named fields. If the records are not correctly aligned with the data when moving it between record and storage media, the data will be meaningless.

TIME and COMPILETIME Functions (COBOL68 Only) (Obsolete)

COBOL85 does not support the TIME and COMPILETIME functions of COBOL68. In COBOL85, a combination of intrinsic functions and Format 2 of the ACCEPT statement replace the various TIME functions in COBOL68. The task attributes ACCUMPROCTIME and ACCUMIOTIME must be used to translate the TIME(2) and TIME(3) functions. Because the time units used in TIME(2) and TIME(3) versus ACCUMPROCTIME and ACCUMIOTIME differ, some arithmetic conversions are needed. The COMPILETIME(5) and COMPILETIME(15) functions of COBOL68 must be replaced by a combination of MOVE statements and the COBOL85 intrinsic function WHEN-COMPILED.

TODAYS-DATE (COBOL68 Only) (Change)

TODAYS-DATE was allowed as the object of a TODAYS-DATE in COBOL68. This must be modified to use the ANSI-74 ACCEPT syntax.

USAGE Clauses (COBOL68 Only) (Obsolete)

The following USAGEs are not recognized by COBOL85 and must be changed:

Change . . .	To . . .
COMP	BINARY EXTENDED
COMP-1	BINARY EXTENDED
COMP-2	COMP
COMP-4	REAL
COMP-5	DOUBLE
CONTROL-POINT (CP)	TASK

The conversion of COMP and COMP-1 into BINARY EXTENDED and the conversion of COMP-5 to DOUBLE is required because these COBOL68 data types have a different meaning in COBOL85. BINARY EXTENDED is new for COBOL85.

USAGE ASCII Clause (COBOL68 Only) (Obsolete)

A data item with this clause is assumed to contain 8-bit-coded ASCII characters. Neither COBOL85 or COBOL74 support the USAGE ASCII clause.

USAGE BINARY Clause (COBOL74 Only) (Change)

In COBOL74, the TRUNCATED phrase with the USAGE BINARY clause was used to specify the contents of the PICTURE clause for truncation of higher-order digits and for SIZE ERROR determination. In COBOL85, the USAGE BINARY clause without the EXTENDED phrase behaves the same as the COBOL74 USAGE BINARY TRUNCATED clause. The EXTENDED phrase must be included with the USAGE BINARY clause in COBOL85 to achieve the same behavior as the COBOL74 USAGE BINARY clause without the TRUNCATED phrase.

In COBOL74, . . .	Is equivalent to . . . in COBOL85
USAGE BINARY	USAGE BINARY EXTENDED
USAGE BINARY TRUNCATED	USAGE BINARY

Note: *When USAGE BINARY EXTENDED items are used as destinations in arithmetic statements that do not include an ON SIZE ERROR clause, COBOL85 ensures that the result can still be represented internally as a single-precision integer or double-precision-integer (appropriate to the size of the item as declared in the PICTURE clause) so that the integrity of the data is protected. For reasons of compatibility with existing programs, COBOL74 does not provide this insurance against the data corruption that would otherwise result. INTEGER OVERFLOW terminations may thus occur with COBOL85 programs in the same circumstances in which COBOL74 programs would risk data corruption. When the ON SIZE ERROR clause is included in the statement, the SIZE ERROR condition is set in either language under these circumstances.*

USAGE INDEX FILE Clause (COBOL68 Only) (Obsolete)

The USAGE INDEX FILE clause is not supported by COBOL85. In COBOL68, this clause is permitted only for DIRECT files. It provides COBOL68 programs with a switch file that is similar to the kind available in the ALGOL compiler.

USAGE KANJI with PICTURE Character X Changed to USAGE NATIONAL (Obsolete)

In COBOL85, the term NATIONAL replaces KANJI for data items representing national characters. The usage KANJI with the picture character X should be changed to the usage NATIONAL with the picture character N. The old specification is allowed as a synonym of the new representation in COBOL85 but will be deimplemented in a future release.

USE AFTER RECORD SIZE ERROR Statement (COBOL68 Only) (Obsolete)

COBOL85 neither supports the USE AFTER RECORD SIZE ERROR statement nor provides an equivalent construct.

USE Procedure For Tape Files (Obsolete)

COBOL68 and COBOL74 support USE procedures that allow manipulation of tape label information. COBOL85 does not support this feature.

User-Defined Paragraphs (COBOL68 Only) (Obsolete)

User-defined Paragraphs in the IDENTIFICATION DIVISION are obsolete and must be commented out.

VALUE OF Clause (Obsolete)

The VALUE OF clause in the file-description entry has been placed in the obsolete element category. This clause will continue to be supported in COBOL85; however, the description of file label items is considered a function of the operating system.

WRITE DELIMITED Statement (COBOL74 Only) (Obsolete)

In COBOL74, the WRITE DELIMITED statement is used for insertion and deletion of control codes for national characters. In COBOL85, control codes are automatically inserted and deleted when the IS EXTERNAL FORMAT FOR NATIONAL clause is used with the SELECT clause in the Input-Output Section of the Environment Division. The COBOL85 compiler will ignore the WRITE DELIMITED clause and will issue a warning anytime this clause is encountered.

WRITE Statement (Change)

In COBOL85, the phrase ADVANCING PAGE and END-OF-PAGE can not be used together in a single WRITE statement. In COBOL68, it is possible to specify both of these phrases within one WRITE statement. The ADVANCING PAGE clause is processed first, then the END-OF-PAGE clause would be processed. If both phrases are present in a single WRITE statement, they should be separated into two WRITE statements or one of the phrases should be removed.

In COBOL85, the phrases ADVANCING PAGE and END-OF-PAGE cannot be used together in a single WRITE statement. In COBOL68 you can specify both of these phrases within one WRITE statement. In ANSI COBOL74, the order of precedence for the phrases was not specified. The ClearPath and A Series COBOL74 compiler processes the ADVANCING PAGE phrase first, and the END-OF-PAGE phrase second.

If both phrases are present in a single WRITE statement, you must either separate them into two WRITE statements or remove one of the phrases.

ZERO/ZEROS/ZEROES (Change)

When a programmer uses the figurative constant ZERO/ZEROS/ZEROES in a COBOL74 program, the generated object code incorrectly uses all-bits-off (analogous to LOW-VALUES) instead of EBCDIC zero characters as the documentation and the standards require. The object code that COBOL85 generates for ZERO/ZEROS/ZEROES in this context correctly uses EBCDIC zero characters. If a COBOL85 programmer wishes to obtain the same results that the COBOL74 compiler produced in this context, he or she must use LOW-VALUES instead of ZERO/ZEROS/ZEROES.

Changes That Might Affect Your Programs

The following list of changes defines those modifications to COBOL ANSI-85 that might affect your programs. The change from COBOL ANSI-74 is outlined and in some cases, specific examples of code that could be in your program are identified.

CALL "<library object title>"

COBOL85 enables the user to call a nested procedure by using syntax similar to that of a library call, such as the statement CALL "<procedure name>". As a result, COBOL85 is unable to differentiate between a nested procedure and a library call without an additional run-time code. To enable COBOL85 to recognize the statement CALL "<library object title>" as a library call, use the statement CALL "PROCEDUREDIVISION OF <library object title>" instead. This statement can significantly improve the performance of your library call.

CODE Clause (Change)

The use of the CODE Clause for a Report Description Entry of Report Writer does not automatically prefix printer files with BDREPORT as COBOL74 does. This prevents you from using the WFL PB command and the literal-1 associated with a specific report, to print an individual report. If you wish to do this, you must programmatically set ATTRIBUTE BDNAME OF MYSELF TO "BDREPORT", before opening the printer file. A side effect is that any subsequent printer files that are opened will also use the BDREPORT prefix. If you wish to avoid this, reset BDNAME to null before opening a file that should not use BDREPORT; to set BDNAME to null use CHANGE ATTRIBUTE BDNAME OF MYSELF TO ".".

Once printer files are produced with the prefix of BDREPORT, you can use either of the following WFL statements to print a specific report:

```
PB D job-number KEY REPORT EQUAL literal-1

PB D * KEY REPORT EQUAL literal-1
```

Job-number is the mix number of the job that created the report. The asterisk (*) indicates that the job-number is that of the WFL job itself. The asterisk function is useful when a PB (Printer Backup) statement is included in a WFL statement that both creates and prints the report. For further details on the use of PB, refer to Section 3, "The SYSTEM/BACKUP Utility" of the *Printing Utilities Operations Guide* (8600 0692).

Computation of Divide with a DOUBLE Data Item Result

When a COMPUTE statement contains a divide operator and the result is a DOUBLE data item, COBOL85 uses a single precision divide when both operands are single precision items. COBOL85 uses a double precision divide when either operand is a double precision item. In these situations, COBOL74 always uses a double precision divide.

CURRENCY SIGN Clause (Change)

The literal specified within the CURRENCY SIGN clause cannot be a figurative constant.

ANSI COBOL74 allowed the use of a figurative constant in the CURRENCY SIGN clause, but did not specify rules for the meaning of the use of HIGH-VALUE, LOW VALUE or ALL literal in this context. In ClearPath and A Series COBOL74, the use of HIGH-VALUE and LOW-VALUE were ignored because the representation of these literals could not be expressed in a picture clause. The use of ALL literal was a syntax error.

If your program uses HIGH-VALUE or LOW-VALUE as the literal specified in the CURRENCY-SIGN clause, replace them with true literals.

EXIT PROGRAM and PERFORM Activation (Change)

The following new rule appears for the EXIT statement: "... the ends of the ranges of all PERFORM statements executed by the called program are considered to have been reached." This situation was undefined in ANSI COBOL74.

Exponentiation (Change)

The following special cases of exponentiation are defined in ANSI COBOL85:

- If an expression having a zero value is raised to a negative or zero power, the size error condition exists.
- If the evaluation of the exponentiation yields both a positive and a negative real number, the positive number is returned.
- If a real number does not exist as a result of the evaluation, the size error condition exists.

COBOL74 did not define these special cases of exponentiation. ANSI COBOL85 clarifies them to promote program portability.

COBOL74 returned a result of 1 when an expression having a zero value was raised to a zero power (0**0). If your programs contain this statement, they will no longer return the value of 1; instead, the program will receive an ON SIZE ERROR.

If an expression with a zero value is raised to a negative power (0 ** –2), COBOL74 returned an ON SIZE ERROR. No modification to your programs is necessary in this case.

Also, COBOL74 always returned the positive number if the evaluation of the exponentiation could yield both a positive and a negative number.

Finally, COBOL74 programs that contained an expression that did not evaluate to a real number would fail with an INVALID ALOG ARGUMENT error. If your programs ever did contain such an expression, they would not have run correctly, so no modifications should be necessary to programs that do work correctly.

KEYEDIO (Change)

In the ClearPath and A Series COBOL74 compiler, KEYEDIO was allowed for handling indexed I/O. In COBOL85, only KEYEDIOII is allowed.

LINAGE Clause (Change)

In ANSI COBOL85, files for which the LINAGE clause has been specified must not be opened in the extend mode.

ANSI COBOL68 and COBOL74 do not define consistent results for a file having an associated LINAGE clause that is opened in the extend mode. For example, the value of LINAGE-COUNTER when an OPEN statement is executed is specified as 1 in ANSI COBOL74. However, if a file with an associated LINAGE-CLAUSE is opened in the extend mode, the value of LINAGE-COUNTER is unspecified.

When a file was opened in the extend mode in COBOL68 or COBOL74, LINAGE-COUNTER was not set to 1, it remained at 0. If your programs OPEN a file in the extend mode, you should remove the LINAGE clause from the associated file statements.

MERGE Statement (Change)

No two files in a MERGE statement can be specified in the SAME AREA or SAME SORT–MERGE AREA clause. The only files in a MERGE statement that can be specified in the SAME RECORD AREA clause are those associated with the GIVING phrase.

This rule did not exist in the ANSI COBOL74 Standard. If this rule were violated in COBOL74, the MERGE statement would not work properly.

ClearPath and A Series COBOL74 ignored the SAME AREA clause for those files.

PICTURE Symbol P (Change)

When a data item described by a PICTURE containing the character P is referenced, the digit positions specified by P will be considered to contain zeros in the following operations:

- Any operation requiring a numeric sending operand
- A MOVE statement where the sending operand is numeric and its PICTURE character-string contains the symbol P
- A MOVE statement where the sending operand is numeric edited and its PICTURE character-string contains the symbol P and the receiving operand is numeric or numeric edited
- A comparison operation where both operands are numeric

In ANSI COBOL74, digit positions described by a P were considered to contain zeros when used in operations involving conversion of data from one form of internal representation to another. ANSI COBOL74 did not specify what happened in operations not involving data conversion or when conversion was required.

The clarification of this issue gives consistent results for numeric and alphanumeric MOVEs where the sending item is numeric or numeric edited.

Picture Symbol * With Zero Value (Change)

In ANSI COBOL74, a data item in which all numeric positions were represented in the PICTURE clause by the asterisk character indicating leading-zero suppression, and in which the PICTURE also contained a trailing sign symbol, would reflect a positive sign in the sign symbol location when the value of the item was zero.

The ANSI COBOL74 standard, in its description of the behavior of zero-suppression editing in this particular case, does not explicitly require that such insertion editing symbols as trailing signs must be represented by asterisks in the data item.

However, the ANSI COBOL85 standard explicitly requires that all character positions in such a data item, except the decimal point, and including character positions occupied by fixed and simple insertion editing symbols in the PICTURE clause, contain asterisks when the value is zero.

In other words, if all numeric positions in a COBOL85 PICTURE are represented by asterisks, and the value is zero, every character position in the data item must contain an asterisk, including any character position(s) that would contain the sign in the event the value was non-zero. The only character position in the entire data item that would not contain an asterisk in such a case would be the decimal point, if one is explicitly called for by the appropriate symbol ('.' or ',', depending on the setting of DECIMAL-POINT IS COMMA for the program) in the PICTURE character-string.

Program Termination (Change)

COBOL74 and COBOL85 differ in their handling of attempts to execute the next executable statement in a program in which there is no next executable statement for the system to execute.

The COBOL85 compiler generates code corresponding to the end of all source images that:

- has the effect of an EXIT PROGRAM statement if encountered under control of an IPC or library CALL statement
- has the effect of an EXIT PROCEDURE statement if encountered in a bound procedure
- has the effect of a STOP RUN statement if encountered in a stand-alone program

By contrast, the code generated by COBOL74 in these circumstances:

- causes an INVALID OP fault if encountered in a program executed under control of an IPC or library CALL statement
- has the effect of an EXIT PROCEDURE statement if encountered in a bound procedure
- causes an INVALID OP fault if encountered in a stand-alone program

Users migrating from COBOL74 and expecting abnormal termination when control falls through the end of the program should consider adding logic to force an abnormal termination in a new paragraph (and section, if needed) at the end of the program to prevent the execution of the normal-termination logic supplied by the compiler.

Example

```
FALL-THROUGH SECTION.
FALL-THROUGH-PARAGRAPH.
   DISPLAY "FELL OUT THE END OF THE PROGRAM."
   CHANGE ATTRIBUTE STATUS OF MYSELF TO VALUE (TERMINATED).
```

READ Statement (Change)

The INTO phrase of the READ statement cannot be specified unless one of the following is true:

- All records associated with the file and the data item specified in the INTO phrase are group items or elementary alphanumeric items
- Only one record description is subordinate to the file description entry.

ANSI COBOL74 did not define the semantics for the move of a record to the identifier specified in the INTO phrase of the READ statement. Additionally, there was no statement as to whether any conversion of data takes place or whether a group move is performed for a file with multiple elementary records. The new rules clarify any ambiguity.

COBOL74 used the first 01 level definition to perform the data conversion according to the ANSI COBOL74 MOVE rules. If you need to convert your programs, you should use this precedence rule when dealing with multiple or conflicting receiving-field data descriptions.

RELEASE Statement (Change)

When a sortfile definition has multiple record descriptions (01), the sortfile record length is set to the largest record length defined. 0

In COBOL68 and COBOL74, a RELEASE statement of any of the SD records causes the entire record to be released to the sortfile. In COBOL85, only the specified record is released to the sortfile.

RETURN Statement (Change)

The INTO phrase of the RETURN statement cannot be specified unless:

- All records associated with the file and the data item specified in the INTO phrase are group items or elementary alphanumeric items
- Only one record description is subordinate to the sort-merge file description entry

COBOL74 did not specify the semantics for the move of a record to the identifier specified in the INTO phrase of the RETURN statement. Additionally, there was no statement as to whether any conversion of data takes place or a group move is performed for a file with multiple elementary records. The new rules disallow these potentially ambiguous situations.

COBOL74 used the first 01 level definition to perform the data conversion according to the COBOL74 MOVE rules. If conversion is required for your programs, this is the precedence rule you should use when dealing with multiple or conflicting receiving-field data descriptions.

SEARCH ALL (Change)

The dataname(s) specified by the WHEN phrase of the SEARCH ALL verb must appear directly in the KEY IS phrase of the OCCURS clause that is the object of the SEARCH ALL. The COBOL74 compiler would allow the dataname to be an item that was subordinate to an item referenced by the KEY IS phrase.

For example, the COBOL74 compiler would allow:

```
01  TABLE-AREA.
     03  TABLE-A        OCCURS 10 TIMES
                        ASCENDING KEY IS TABLE-A-KEY
                        INDEXED BY TABLE-A-INDEX.
     05  TABLE-A-KEY.
         07  TABLE-A-KEY-1            PIC 9(02).
         07  TABLE-A-KEY-2            PIC 9(02).

SEARCH ALL TABLE-A
         WHEN TABLE-A-KEY-1 = 66 AND
              TABLE-A-KEY-2 = 77
```

TABLE-A-KEY-1 and TABLE-A-KEY-2 are not specified in the KEY IS clause. The COBOL85 compiler does not allow them to be referenced in the WHEN phrase of the SEARCH ALL.

SEGMENT-LIMIT Clause (Obsolete)

The SEGMENT-LIMIT clause of the OBJECT-COMPUTER paragraph has been placed in the obsolete element category. The function performed by this clause is considered to be more appropriately performed by the host operating system than the individual COBOL program.

The COBOL85 compiler ignores this clause. Thus for clarity in your program, it is recommended that you designate all occurrences of the SEGMENT-LIMIT clause as comments.

SORT Statement (Change)

COBOL85 might give different results from other COBOL versions, when dealing with duplicate record keys. This typically occurs when more than one key is used in a sort. The sort result presents all duplicate records together, but the order in which they are arranged is not defined. Since COBOL85 and the other COBOL versions use a slightly different interface to the sort routine, the ordering of duplicate records might be slightly different in their presentation.

SUM Clause in REPORT WRITER Item Declaration

The intermediate sum counter generated when a SUM clause appears in a REPORT WRITER item is treated as if it had been declared with a numeric PICTURE whose precision and overall length are limited by the actual declared PICTURE's numeric characteristics. If the value being produced by the action of the SUM clause exceeds the capacity of the associated PICTURE, high-order digits are lost according to normal COBOL numeric truncation rules and according to the requirements of the standards. When the SUMmed item is itself used as a source field for another SUM, the amount retrieved should not, according to ANSI X3.23-1985, include high-order digits outside of the capacity of the item that is being used as a source for the SUM.

ClearPath and A Series COBOL(68) and COBOL74 would sometimes fail to truncate high-order digits in such cases.

If you don't want the COBOL85 program to truncate the value as the standards require it to, it is necessary to ensure that the PICTURE clause is large enough to contain the largest possible value. If this is impractical because of space considerations in the print line, the user may declare a sum counter item without a COLUMN clause, thus making it a hidden field, and use it both as the source for the secondary SUM clause and as an explicit SOURCE field, instead of SUM, for the original field in the report line. In this case, declaring the item with a total of 23 digits of precision—the maximum allowed on the machine—is probably appropriate to ensure that the SUM value is correctly maintained without truncation.

UNSTRING Statement (Change)

In the UNSTRING statement, any subscripting associated with the DELIMITED BY identifier, the INTO identifier, the DELIMITER IN identifier, or the COUNT IN identifier is evaluated once, immediately before the examination of the sending fields for the delimiter.

Although ANSI COBOL74 stated that any subscripting associated with the delimiters is evaluated immediately before the transfer of data into the respective data item, this is not possible. The delimiter must be known before examining the sending field. This change permits evaluation of the delimiters at the appropriate time.

ClearPath and A Series COBOL74 evaluated the subscripting before the first and after each transfer. If your program defined multiple INTO destinations, it would abort with INVALID INDEX errors.

WRITE Statement (Change)

The phrases ADVANCING PAGE and END-OF-PAGE may not both be specified in a single WRITE statement.

In ANSI COBOL74, it was possible to specify both of these phrases within one WRITE statement, but the order of precedence was not specified.

ClearPath and A Series COBOL74 would process the ADVANCING PAGE first, then the END-OF-PAGE would be processed. If your program uses both these phrases in a single WRITE statement, you should separate the statements into two WRITE statements or remove one of the phrases.

Changes that Do Not Affect Your Programs

The following new features are included in ANSI COBOL85; however, they do not affect existing programs written in previous versions of COBOL. For example, typical changes of this kind are where a new verb has been added or an additional capability for an old verb has been added.

ADD Statement (Change)

- In the format ADD identifier/literal TO identifier/literal GIVING identifier, the word TO is an optional word.
- Matching nonnumeric elementary data items in an ADD CORRESPONDING statement are ignored, just as INDEX data items are ignored.

ASSIGN Clause (Change)

The ASSIGN clause can contain a nonnumeric literal.

BLOCK CONTAINS Clause (Change)

You can omit the BLOCK CONTAINS clause if the operating environment specifies the number of records contained in a block. In previous versions of COBOL, if you omit the BLOCK CONTAINS clause, then you must designate the standard physical record size.

CALL Statement (Change)

- The BY CONTENT phrase indicates that the called program cannot change the value of a parameter in the USING phrase of the CALL statement, but the called program can change the value of the corresponding data item in the Procedure Division header of the called program.
- The BY REFERENCE phrase causes the parameter in the USING phrase of the CALL statement to be treated the same as specified in previous versions of COBOL.
- The parameters passed in a CALL statement can be other than an 01 or 77 level data item. The parameters passed in a CALL statement can be subscripted, reference modified, or both.

CANCEL Statement (Change)

The CANCEL statement now closes all open files.

The COBOL74 Standard does not define the status of files left in the open mode when the program is canceled. However, through an extension to the standard, the ClearPath and A Series COBOL74 CANCEL statement implicitly closes the file. Consequently, this change does not affect your programs.

Class Condition (Change)

Class-name is associated with a set of characters that you must specify in the CLASS clause within the SPECIAL-NAMES paragraph.

CLOSE Statement (Change)

The NO REWIND phrase cannot be specified in a CLOSE statement having either the REEL or UNIT phrase.

In the ANSI COBOL74 standard, the rules for the NO REWIND phrase and the REEL or UNIT phrase could not be processed properly. In ClearPath and A Series COBOL74, these phrases cannot be specified together in a CLOSE statement, so modification of your programs is not necessary.

CODE-SET Clause (Change)

The CODE-SET clause can be specified for all files with sequential organization. In COBOL74, the CODE-SET clause is restricted to non-mass-storage files.

Collating Sequence (Change)

The collating sequence used to access an indexed file is the collating sequence associated with the native character set that was in effect for the file at the time the file was created.

The ANSI COBOL74 standard does not state which collating sequence is to be used for the retrieving and storing of records when accessing an indexed file. ClearPath and A Series COBOL74 already behaves like COBOL85, so modification of your programs is not necessary.

Colon (:) Character (Change)

The COBOL character set has been expanded to include the colon (:) character used in reference modification.

COMMON Clause in the PROGRAM-ID Paragraph (Change)

The COMMON clause specifies a program that, despite being directly contained within another program, can be called from any program directly or indirectly contained in that other program.

Communication Description Entry (Change)

The order of clauses in the communication description entry is immaterial.

Communication Status Key (Change)

New communication status key values have been added to the Standard. Since ClearPath and A Series COBOL does not implement the communication section of the Standard, these status keys have not been implemented.

Communication Error Key (Change)

New communication error key values have been added to the Standard. Since COBOL85 does not implement the communication section of the Standard, these error keys have not been implemented.

CONTINUE Statement (Change)

The CONTINUE statement indicates that there is no executable statement present and causes an implicit transfer of control to the next executable statement.

COPY Statement (Change)

The ANSI COBOL85 standard specifies the following revisions:

- If a COPY statement appears in a comment-entry, or in the place where a comment-entry might appear, it is considered to be part of the comment-entry. In the ANSI COBOL74 Standard, specifying a COPY statement in a comment-entry is an undefined situation. The specification of this situation enhances program portability. ClearPath and A Series COBOL74 considers a COPY statement appearing as part of a comment-entry to be part of the comment-entry, so modification of your programs is not necessary.
- After all COPY statements are processed, a debugging line is considered to have all the characteristics of a comment line, if the WITH DEBUGGING MODE clause is not specified in the SOURCE-COMPUTER paragraph. The ClearPath and A Series COBOL74 compiler works the way COBOL ANSI-85 is defined, so modification of your programs is not necessary.
- Pseudo-text-1 must not consist entirely of a separator comma or a separator semicolon. The ANSI COBOL74 Standard allows pseudo-text-1 in a COPY statement to consist entirely of a separator comma or a separator semicolon but does not specify under what conditions replacement takes place. Any attempt to define the semantics in this situation would have caused a potential incompatibility. In ClearPath and A Series COBOL74, if pseudo-text-1 consists solely of either a comma or a semicolon, the compiler generates a syntax error. Modification of your programs is not necessary.

Data Division (Change)

The Data Division is optional in ANSI COBOL85.

Data Division Reference Format (Change)

The word that follows a level indicator, level-number 01, or level-number 77 on the same line can begin in area A.

DAY-OF-WEEK Phrase of ACCEPT Statement (Change)

The DAY-OF-WEEK phrase of the ACCEPT statement provides access to an integer representing the day of week; for example, 1 represents Monday, 2 represents Tuesday, and 7 represents Sunday.

DISPLAY Statement (Change)

The ANSI COBOL85 Standard specifies the following revisions:

- The figurative constant ALL literal is permitted in the DISPLAY statement. In COBOL74, the figurative constant ALL literal is not permitted in the DISPLAY statement.
- After the last operand is transferred to the hardware device, the positioning of the hardware device is reset to the leftmost position of the next line of the device. In the ANSI COBOL74 Standard, the positioning of the hardware device after the last operand is undefined. The new rule is necessary for a complete specification of the NO ADVANCING phrase. ClearPath and A Series COBOL74 already positions the hardware device to the leftmost position of the next line of the device, so modification of your programs is not necessary.

Double Character Substitution (Obsolete)

When a character set contains fewer than 51 characters, double characters must be substituted for the single characters. This feature has been placed in the obsolete element category.

The character set does not contain fewer than 51 characters. Modification is not necessary.

End Program Header (Change)

The end program header indicates the end of the named COBOL source program; the end program header can be followed by a COBOL program that is to be compiled separately in the same invocation of the compiler.

ENVIRONMENT Division (Change)

The Environment Division is optional. Within the Environment Division, the Configuration Section is optional. The SOURCE-COMPUTER paragraph and the OBJECT-COMPUTER paragraph (OBJECT-COMPUTER is described earlier under changes that probably affect your program), as well as the entries within the SOURCE-COMPUTER paragraph, OBJECT-COMPUTER paragraph, SPECIAL-NAMES paragraph, and I-O-CONTROL paragraph are also optional.

COBOL85 uses the information if it is provided, but it is not necessary. Consequently, modification of your programs is not needed.

EVALUATE Statement (Change)

The EVALUATE statement describes a structure in which multiple conditions are evaluated to determine the subsequent action of the object program.

EXIT PROGRAM Statement (Change)

The ANSI COBOL85 standard specifies the following revisions:

- The EXIT PROGRAM statement need not be the only statement in a paragraph.
- When the next executable statement in a called program does not exist, COBOL85 executes an implicit EXIT PROGRAM statement. In the ANSI COBOL74 Standard, this situation is undefined. Defining this situation in ANSI COBOL85 makes your programs more transportable. In ClearPath and A Series COBOL74, if a called program does not have an EXIT PROGRAM statement, the calling program fails when a next executable statement does not exist. Modification of your programs is not necessary.

EXTEND Phrase of the OPEN Statement (Change)

The EXTEND phrase of the OPEN statement can be used with a relative file or an indexed file.

EXTERNAL Clause (Change)

The EXTERNAL clause specifies that a data item or a file connector is external and can be accessed and processed by any program in the run unit.

Figurative Constant ZERO (Change)

The figurative constant ZERO is allowed in arithmetic expressions.

File Position Indicator (Change)

The concept of a current record pointer in COBOL74 has been changed to a file position indicator.

For combinations of update and READ NEXT statements, the current record pointer rules in ANSI COBOL74 are complex and sometimes cause unexpected results. The current record pointer rules are also poorly defined in certain cases when the record pointed to becomes inaccessible. In ANSI COBOL85, the rules based on the file position indicator are straightforward and easy to understand.

The cases in which this change in concepts can affect your programs are as follows.

- For a relative or indexed file in the dynamic access mode, execution of an OPEN I-O statement followed by one or more WRITE statements and then a READ NEXT statement causes the READ statement to access the first record in the file at the time of the execution of the READ statement.

  In the ANSI COBOL74 Standard, this sequence causes the READ statement to access the first record at the time of execution of the OPEN statement. If one of the WRITE statements inserts a record with a key or relative record number lower than that of any records previously existing in the file, a different record is accessed by the READ statement. The ClearPath and A Series COBOL74 compiler is defined to work according to the ANSI COBOL85 standard, so modification of your programs is not necessary.

- If an alternate key is the key of reference and the alternate key is changed by a REWRITE statement to a value between the current value and the next value in the file, a subsequent READ NEXT statement obtains the same record. In the ANSI COBOL74 Standard, the subsequent READ statement obtains the record with the next value for that alternate key prior to the REWRITE statement. The ClearPath and A Series COBOL74 compiler is defined to work according to the ANSI COBOL85 standard, so modification of your programs is not necessary.

FILLER Clause (Change)

The use of the word FILLER is optional for data description entries. The word FILLER can appear in a data description entry that contains a REDEFINES clause. The word FILLER can be used in a data description entry of a group item.

FOOTING Phrase (Change)

If the FOOTING phrase is not specified, an end-of-page condition independent of the page overflow condition does not exist.

In ANSI COBOL74, the specifications for the existence of the footing area are contradictory between the LINAGE clause and the WRITE statement. The ClearPath and A Series COBOL74 compiler does not provide an implicit FOOTING area if a FOOTING area is not specified in the LINAGE clause.

FOR I-O Phrase in Communication Description Entry (Change)

The FOR I-O phrase in a communication description entry provides for both input and output functions by one CD entry.

FOR REMOVAL Phrase of the CLOSE Statement (Change)

The FOR REMOVAL phrase of the CLOSE statement is allowed for a sequential single reel/unit file.

GO TO DEPENDING Statement (Change)

The number of procedure-names required in a GO TO DEPENDING statement has been reduced to one.

IF Statement (Change)

The optional word THEN has been added to the general format of the IF statement.

INITIAL Clause in PROGRAM-ID Paragraph (Change)

The INITIAL clause initializes a program to the same state as when the program was first called in the run unit, whenever the program is called by another program.

INITIALIZE Statement (Change)

The INITIALIZE statement provides the ability to set selected types of data fields to predetermined values.

INSPECT Statement (Change)

- The ALL/LEADING adjective can be distributed over multiple occurrences of identifier/literal and there can be multiple occurrences of the REPLACING CHARACTERS phrase.
- Multiple occurrences of the BEFORE/AFTER phrase permit the TALLYING/REPLACING operation to be initiated after the beginning of the inspection of the data begins and/or terminated before the end of the inspection of the data ends.
- The order of execution for evaluating subscripts in the INSPECT statement is specified. Subscripting associated with any identifier is evaluated only once as the first operation in the execution of the INSPECT statement. The order for evaluating subscripts in the INSPECT statement is undefined in ANSI COBOL74. ClearPath and A Series COBOL74 evaluates subscripts in the INSPECT statement according to the ANSI COBOL85 rules so modification of your program is not necessary.

INSPECT CONVERTING Statement (Change)

The CONVERTING phrase provides a new variation for the INSPECT statement.

I-O-CONTROL Paragraph (Change)

The order of clauses is immaterial in the I-O-CONTROL paragraph.

KEY Phrase of the ENABLE Statement (Obsolete)

The KEY phrase of the ENABLE statement has been placed in the obsolete element category and has been made an optional phrase. The ENABLE statement was not implemented in COBOL74 so modification of your program is not necessary.

LABEL RECORDS Clause (Change)

The LABEL RECORDS clause is optional; if you do not specify the LABEL RECORDS clause, the compiler assumes LABEL RECORDS ARE STANDARD.

Length of ALL Literal (Change)

When the figurative constant ALL literal is not associated with another data item, the length of the string is the length of the literal.

The rules in ANSI COBOL74 for the size of the figurative constant ALL literal differ depending on where the figurative constant has been used in the program. The rules in ANSI COBOL85 indicate that in the case of the alphabet-name clause with the figurative constant ALL literal, the length of the string is the length of the literal. COBOL74 does not allow the figurative constant ALL literal construct in the SPECIAL-NAMES clause so modification of your program is not necessary.

LINAGE Clause (Change)

Data-names within the LINAGE clause can be qualified.

LINE NUMBER Clause (Change)

You can specify the integer 0 as the relative line number in the PLUS phrase of the LINE NUMBER clause.

Lowercase Letters (Change)

When the computer character set includes lowercase letters, they can be used in character-strings. Except when used in nonnumeric literals, each is equivalent to the corresponding uppercase letter.

MERGE Statement (Change)

Multiple file-names are allowed in the GIVING phrase of the MERGE statement. A file named in either the USING or GIVING phrase of a MERGE statement can be a relative file or an indexed file.

Mixing Subscripts and Indexes (Change)

Indexes and data-name subscripts can both be written in a single set of subscripts used to reference an individual occurrence of a multidimensional table.

MOVE Statement (Change)

A numeric-edited data item can be moved to either a numeric or a numeric-edited data item. If it is moved to a numeric data item, de-editing occurs.

National Character Delimiter (Obsolete)

The characters NC, which were used to indicate the beginning of a national character string in COBOL74, are being replaced by the character N in COBOL85. Using NC in COBOL85 programs results in a warning message.

Nonnumeric Literal (Change)

A nonnumeric literal has an upper limit of 160 characters in length. The upper limit is 120 characters in ANSI COBOL74.

NOT AT END Phrase of READ Statement (Change)

The NOT AT END phrase enables you to specify procedures to be executed when the at end condition does not exist for the READ statement.

NOT AT END Phrase of RETURN Statement (Change)

The NOT AT END phrase specifies procedures to be executed when an at end condition does not exist for the RETURN statement.

NOT END-OF-PAGE Phrase of WRITE Statement (Change)

The NOT END-OF-PAGE phrase specifies procedures to be executed when an end-of-page condition does not exist for the WRITE statement.

NOT INVALID KEY Phrase of DELETE Statement (Change)

The NOT INVALID KEY phrase specifies procedures to be executed when an invalid key condition does not exist for the DELETE statement.

NOT INVALID KEY Phrase of READ Statement (Change)

The NOT INVALID KEY phrase specifies procedures to be executed when an invalid key condition does not exist for the READ statement.

NOT INVALID KEY Phrase of REWRITE Statement (Change)

The NOT INVALID KEY phrase specifies procedures to be executed when an invalid key condition does not exist for the REWRITE statement.

NOT INVALID KEY Phrase of START Statement (Change)

The NOT INVALID KEY phrase specifies procedures to be executed when an invalid key condition does not exist for the START statement.

NOT INVALID KEY Phrase of WRITE Statement (Change)

The NOT INVALID KEY phrase specifies procedures to be executed when an invalid key condition does not exist for the WRITE statement.

NOT ON OVERFLOW Phrase of STRING Statement (Change)

The NOT ON OVERFLOW phrase specifies procedures to be executed when an overflow condition does not exist for the STRING statement.

NOT ON OVERFLOW Phrase of UNSTRING Statement (Change)

The NOT ON OVERFLOW phrase specifies procedures to be executed when an overflow condition does not exist for the UNSTRING statement.

NOT ON SIZE ERROR Phrase of ADD Statement (Change)

The NOT ON SIZE ERROR phrase specifies procedures to be executed when a size error condition does not exist for the ADD statement.

NOT ON SIZE ERROR Phrase of COMPUTE Statement (Change)

The NOT ON SIZE ERROR phrase specifies procedures to be executed when a size error condition does not exist for the COMPUTE statement.

NOT ON SIZE ERROR Phrase of DIVIDE Statement (Change)

The NOT ON SIZE ERROR phrase specifies procedures to be executed when a size error condition does not exist for the DIVIDE statement.

NOT ON SIZE ERROR Phrase of MULTIPLY Statement (Change)

The NOT ON SIZE ERROR phrase specifies procedures to be executed when an on size error condition does not exist for the MULTIPLY statement.

NOT ON SIZE ERROR Phrase of SUBTRACT Statement (Change)

The NOT ON SIZE ERROR phrase specifies procedures to be executed when a size error condition does not exist for the SUBTRACT statement.

OCCURS Clause (Change)

- The data item specified in the DEPENDING ON phrase can have a zero value.
- When a receiving item is a variable length data item and contains the object of the DEPENDING ON phrase, the maximum length of the item is used. In ANSI COBOL74, the length is computed based on the value of the item before the execution of the statement. Use of the READ INTO statement can then result in a loss of data. The ClearPath and A Series COBOL74 compiler already uses the maximum length of the data item so modification of your program is not necessary.

ON EXCEPTION and NOT ON EXCEPTION Phrases of CALL Statement (Change)

The ON EXCEPTION phrase of the CALL statement is equivalent to the ON OVERFLOW phrase of the CALL statement. The NOT ON EXCEPTION phrase specifies procedures to be executed when the program specified by the CALL statement has been made available for execution.

OPTIONAL Phrase (Change)

The OPTIONAL phrase in the file control entry applies to sequential files, relative files, and indexed files opened in the input, I-O, or extend mode. In ANSI COBOL74, the OPTIONAL phrase in the file control entry applied to sequential files opened in the input mode.

Order of Execution for a Conditional Expression (Change)

Two or more conditions connected by only the logical operator AND or only the logical operator OR in a hierarchical level are evaluated in order from left to right, and evaluation of that hierarchical level terminates as soon as the compiler encounters a truth value, regardless of whether the compiler has evaluated all the constituent-connected conditions in that hierarchical level.

The only programs affected by this change will be those programs that use the ALL REFERENCES phrase in debugging declaratives. Because debugging is not implemented in COBOL85, this change does not affect your programs.

ORGANIZATION Clause (Change)

The words ORGANIZATION IS have become optional in the ORGANIZATION clause of the file control entry.

PADDING CHARACTER Clause (Change)

The PADDING CHARACTER clause in the file control entry specifies the character to be used for block padding on sequential files.

PERFORM Statement (Change)

- The order of initialization of multiple VARYING identifiers in the PERFORM statement is specified. ClearPath and A Series COBOL74 already handles the initialization of multiple VARYING identifiers in a PERFORM statement as specified in ANSI COBOL85.
- In ANSI COBOL85, the VARYING phrase of the PERFORM statement allows up to six AFTER phrases.
- Procedure-name can be omitted. This results in an in-line PERFORM of the imperative statements that precedes the END-PERFORM phrase that terminates the PERFORM statement.
- The TEST AFTER phrase causes the condition to be tested after the specified set of statements has been executed. The TEST BEFORE phrase causes the condition to be tested before the specified set of statements is executed.

PERFORM Statement, Evaluating Subscripts (Change)

The order of execution for evaluating subscripts in the PERFORM VARYING statement is specified as follows:

- For the VARYING identifier(s), subscripting is evaluated each time the identifier is set or augmented.
- For the FROM and BY identifier(s), subscripting is evaluated each time the identifier is used in a setting or augmenting operation.
- For any identifiers included in an UNTIL condition, subscripting is evaluated each time the condition is tested.

This change causes problems only when your program does all of the following:

- Uses subscripted identifiers in a PERFORM VARYING statement
- Changes the value(s) of the subscript(s) while the PERFORM statement is active
- Runs on an implementation in which subscripts are evaluated in some manner other than as defined in the new ANSI COBOL85 rules

COBOL74 evaluated subscripts in the same manner as ANSI COBOL85.

PICTURE Character-string (Change)

A PICTURE character-string can be continued between coding lines.

PICTURE Clause (Change)

The insertion character period (.) or comma (,) can be used as the last character of a PICTURE character-string, if it is immediately followed by the separator period terminating the data description entry.

Procedure Division (Change)

The Procedure Division is optional in ANSI COBOL85.

Procedure Division Header (Change)

A Linkage Section item that redefines, or is subordinate to one that redefines, an item appearing in the Procedure Division header can be referenced in the Procedure Division.

Punctuation Characters (Change)

The separators comma, semicolon, and space are interchangeable in a source program.

PURGE Statement (Change)

The PURGE statement causes the message control system (MCS) to eliminate any partial message that has been released by one or more SEND statements.

Qualification (Change)

COBOL85 can handle 50 levels of qualification in ANSI COBOL85.

READ Statement (Change)

Variable length records are allowed when the READ statement has an INTO phrase. The NEXT phrase is allowed in a READ statement referencing a file with sequential organization.

RECORD Clause (Change)

The VARYING phrase of the RECORD clause is used to specify variable length records. The DEPENDING phrase associated with the VARYING phrase specifies a data item containing the number of character positions in a record.

RECORD DELIMITER Clause (Change)

The RECORD DELIMITER clause in the file control entry indicates the method of determining the length of a variable length record on the external medium.

REDEFINES Clause (Change)

The size of the item associated with the REDEFINES clause can be less than or equal to the size of the redefined item. In COBOL74, the two items must have the same number of character positions.

REEL/UNIT Phrase of the CLOSE Statement (Change)

The REEL/UNIT phrase of the CLOSE statement can be applied to a single reel/unit file and is specifically permitted for a report file.

Reference Modification (Change)

Reference modification is a new method of referencing data by specifying a leftmost character and length for the data item.

Relational Operators (Change)

The relational operator IS GREATER THAN OR EQUAL TO (>=) is equivalent to the relational operator IS NOT LESS THAN. The relational operator IS LESS THAN OR EQUAL TO (<=) is equivalent to the relational operator IS NOT GREATER THAN.

RELATIVE KEY Phrase (Change)

The relative key data item specified by the RELATIVE KEY phrase must not contain the PICTURE symbol P.

COBOL74 did not allow a relative key item to be defined with a P.

Relative Subscripting (Change)

Relative subscripting permits a subscript to be followed by the operator + or –, which is followed by an integer.

REPLACE Statement (Change)

The REPLACE statement causes each occurrence of specified text in the source program to be replaced by the corresponding text specified in the REPLACE statement.

RETURN Statement (Change)

Variable length records are allowed when the RETURN statement has an INTO phrase.

RERUN Clause (Obsolete)

The RERUN clause of the I-O-CONTROL paragraph has been placed in the obsolete element category.

COBOL85 continues to support the form of the RERUN clause that is supported in COBOL74.

REWRITE Statement (Change)

A record of a different length can replace a record in either a relative or indexed file.

Scope Terminators (Change)

Scope terminators delimit the scope of certain procedural statements. The scope terminators include

END-ADD	END-CALL	END-COMPUTE
END-DELETE	END-DIVIDE	END-EVALUATE
END-IF	END-MULTIPLY	END-PERFORM
END-READ	END-RECEIVE	END-RETURN
END-REWRITE	END-SEARCH	END-START
END-STRING	END-SUBTRACT	END-UNSTRING
END-WRITE		

Sequence Number (Change)

The sequence number can contain any character in the computer's character set. In ANSI COBOL74 the sequence number can contain only digits.

SET Statement (Change)

Index-names and identifiers can now be mixed in ClearPath and A Series of operands preceding the word TO in a SET statement. Two new variations of the SET statement enable you to change the setting of an external switch and the value of a conditional variable.

SIGN Clause (Change)

- The SIGN clause is allowed in a report group description entry.
- Multiple SIGN clauses can be specified in the hierarchy of a data description entry; the specification at the subordinate level takes precedence over the specification at the containing group level.

SORT and MERGE Statements (Change)

The input and output procedures of a SORT or MERGE statement can contain explicit transfers of control to points outside the input or output procedure. The remainder of the Procedure Division can contain transfers of control to points inside the input or output procedure. A paragraph-name can be specified in the INPUT PROCEDURE phrase or the OUTPUT PROCEDURE phrase.

SORT Statement (Change)

Multiple file-names are allowed in the GIVING phrase of the SORT statement. A file named in a SORT statement can contain variable length records. A file named in either the USING or GIVING phrase of a SORT statement can be a relative file or an indexed file. The files named in the USING and GIVING phrases can reside on the same physical reel. If the DUPLICATES phrase is specified, records whose key values are identical remain in the same order after the sort process is completed as they were when they were input to the sort process.

SPECIAL-NAMES Paragraph (Change)

- If implementor-name is a switch, condition-name need not be specified.
- The reserved word IS has been made optional in the SPECIAL-NAMES paragraph to be consistent with the use of IS throughout the COBOL specification.

STANDARD-2 Option (Change)

The STANDARD-2 option within the ALPHABET clause of the SPECIAL-NAMES paragraph allows the specification of the ISO 7-bit character set for a character code set or collating sequence.

STOP RUN Statement (Change)

The STOP RUN statement closes all files. COBOL74 closes all files that are open at the time the STOP RUN statement is executed so modification of your program is not necessary.

STRING Statement (Change)

- The order of execution for evaluating subscripts in the STRING statement is specified. COBOL74 defines the order of execution for evaluating subscripts in the STRING statement as it is defined in COBOL85.
- The identifier in the INTO phrase of the STRING statement can be a group item.

Subscripting (Change)

A table can have up to seven dimensions.

SUBTRACT Statement (Change)

Matching nonnumeric elementary data items in a SUBTRACT CORRESPONDING statement will be ignored, just as INDEX data items are ignored.

Symbolic-characters (Change)

A symbolic-character is a user-defined word that specifies a user-defined figurative constant.

Uniqueness of Reference (Change)

A user-defined word need not be unique or be capable of being made unique, unless referenced.

In COBOL74, two identical data names cannot appear in the DATA DIVISION as entries subordinate to a group item unless they are capable of being made unique through qualification. For example,

```
01  SOME-DATA.
     03  DATA-NAME1     PIC X(5).
     03  DATA-NAME1     PIC X(5).
```

In the example, the duplicated data-name, DATA-NAME1, is valid in COBOL85 only if there is no reference to it in the PROCEDURE DIVISION. COBOL74 treats the above duplication as a syntax error regardless of whether or not it is referenced.

USAGE Clause (Change)

PACKED-DECIMAL is a new feature of the USAGE clause.

USE BEFORE REPORTING Statement (Change)

The GLOBAL phrase specifies that the associated declarative procedures are invoked during the execution of any program contained within the program that includes the USE BEFORE REPORTING statement.

USE Statement (Change)

- A USE AFTER EXCEPTION/ERROR declarative statement specifying the name of a file takes precedence over a declarative statement that specifies the open mode of the file.
- The GLOBAL phrase specifies that the associated declarative procedures are invoked during the execution of any program contained within the program that includes the USE statement. This is mentioned earlier under changes that probably affect your programs.

User-defined Words and System-names (Change)

The same COBOL word can be used as a system-name and as a user-defined word within a source program; the context in which the COBOL word occurs determines what it is.

VALUE Clause (Change)

The VALUE clause can be specified in a data description entry that contains an OCCURS clause. The VALUE clause can be specified in a data description entry that is subordinate to an entry containing an OCCURS clause. In ANSI COBOL74 the VALUE clause is not permitted in a data description entry containing an OCCURS clause or in a data description entry subordinate to an entry containing an OCCURS clause.

WITH DATA Phrase of RECEIVE Statement (Change)

The WITH DATA phrase enables the programmer to specify procedures to be executed when the MCS makes data available during execution of a RECEIVE statement.

WITH NO ADVANCING Phrase of the DISPLAY Statement (Change)

The WITH NO ADVANCING phrase of the DISPLAY statement provides interaction with a hardware device that has vertical positioning.

Appendix G
COBOL Migration

Note: *Although this section is not highlighted, all of the information is relative to Unisys extensions to COBOL ANSI-85.*

This appendix is designed to aid in moving your COBOL68 and COBOL74 programs to COBOL85. In particular, it describes

- Options for the migration from COBOL68 and COBOL74 to COBOL85
- The COBOL Migration Tool (CMT)
- Error Messages
- Warning Messages

Refer to Appendix F, "Comparison of COBOL Versions," to understand more about why conversions are required.

Migration Methods

The suggested ways for you to convert programs to COBOL85 are as follows:

- To convert either COBOL 68 or COBOL 74 programs to COBOL85, use the COBOL Migration Tool (CMT), which automates much of the conversion process. The CMT greatly reduces the amount of work required in a conversion.
- To migrate from V Series COBOL, use the Evolution of V Series to A Series (EVA) product. For information, refer to the Evolution of V Series to A Series Transition Guide.
- For migration services, use the Unisys Portation Center. For information regarding complete portation services offered by Unisys, contact

  Unisys Portation Center
  MS 225
  25725 Jeronimo Road
  Mission Viejo, California 92691-2792

  Telephone: 949-380-5885
  Fax: 949-380-6550
  Internet: portation@unisys.com

COBOL Migration Tool (CMT)

The CMT is a screen-based utility that makes it easier for you to convert programs to COBOL85. The CMT is provided free of charge. Additionally, the CMT source files are provided free of charge on the release media to anyone who licenses either the COBOL74 or COBOL85 compiler. If you wish, you can update the CMT files to handle the specifics of your migration. The CMT is itself is a COBOL85 program. It is made up of the following files:

- SYMBOL/CMT/CONTROLLER
- SYMBOL/CMT/FILTER
- SYMBOL/CMT/CVDIR

While the CMT helps with your conversion, it is not designed to convert everything for you. Expect to find some code that you must convert manually.

The description of the tool on the following pages covers many of the conversions performed. However, some of the changes made the CMT are not documented because the CMT is built upon a preexisting tool (the BCT) for which no documentation is available.

When the CMT prompts you for a file to be converted, you can specify either a COBOL68 or COBOL74 program. The CMT converts the code that needs to be changed and outputs the file as a COBOL85 program. It saves the file after prompting you to choose a name for the file.

After the CMT makes its changes, it produces a report that identifies the areas of code in which migration is still required. You must then complete the conversion.

The CMT supplies you with online forms on which you specify the migration to be performed. Help is provided on each field for which you need to enter information.

CMT Migration Strategy

In general, you can convert all your COBOL68 and COBOL74 programs to COBOL85 with a single strategy. Use the following process to convert your COBOL68 or COBOL74 programs to COBOL85.

1. Run the program to be converted with standard data and obtain a listing of the output. Save this for checking the output of the modified program later.
2. Use the COBOL Migration Tool to upgrade the program to COBOL85. See "COBOL Migration Tool" described next for instructions on its use. The CMT will automatically make many required changes for you. There may be some changes which the CMT cannot automatically make. These will be indicated on the report created by the CMT.
3. Using the updated source created by the CMT, make any manual changes which were indicated by the CMT. Compile the updated source with the COBOL85 compiler, using $SET LIST to obtain a complete listing of the program.
4. Run the program with the standard data. Examine the output for data errors, comparing it with the output obtained from the run of the unmodified program (step 1), and fix any problems as necessary.

Conversion Module

To reduce the amount of work involved in converting many programs, you might want to create a COPY library module that contains general conversion statements. The following process describes the way you would develop such a module.

1. Create a source module that contains any conversion statements that are to be used in multiple conversions.
2. Place the source module into a COPY library.
3. COPY the library module into every program that you want to convert.

Verifying the COBOL Migration Tool is Available

The COBOL Migration Tool is installed using the Simple Install program. Verify the necessary files are on your system. The required files are

- HELPLIB/CMT/1CMT
- FORMLIB/CMT/1CMT
- OBJECT/CMT/CONTROLLER
- SYSTEM/CMT/FILTER
- SYSTEM/CMT/CVDIR

Running the COBOL Migration Tool

Perform the following steps to migrate a program to COBOL85 using the CMT.

1. Make sure the program being migrated is syntactically correct.
2. Generate test output before doing the migration so that you can check the output is correct after the migration.
3. From CANDE, enter and transmit the following command:

   ```
   RUN CMT/CONTROLLER
   ```

4. Fill in the forms as requested.
5. Obtain the CMT report, which is a printer backup file.
6. Read the CMT report and perform any remaining suggested migrations.
7. Check the test output to verify the program is outputting data correctly.

Getting Help

To get help on a particular field, position the cursor in the field and press SPCFY.

Understanding the COBOL Migration Tool Report

The COBOL Migration Tool report is a listing of all the messages generated during the migration. The messages types fall into these categories:

- Changes the COBOL Migration Tool made to the program.
- Warnings that you must make a change.

Changes Made by the CMT

The following paragraphs list the changes made by the CMT and the warning messages issued by the CMT when a change is required that the tool cannot make . The CMT identifies all the changes necessary to migrate a program to COBOL85. When possible, the tool also makes the required changes in the program. When the tool cannot make a change, it generates a message that indicates a change is required and the location of the required change in the program.

Language Elements

The following paragraphs describe the changes made to the language elements by the CMT.

Comment Line

In COBOL68 or COBOL74, a comment line was represented by characters such as plus (+), pound sign (#), or whatever character appeared before the IDENTIFICATION DIVISION header. COBOL85 allows only the asterisk (*) in column 7 to represent a comment line. The CMT replaces all such characters with an asterisk.

Compiler Control Options

COBOL85 does not allow several compiler option that are allowed in COBOL68. It also does not allow several compiler options that are allowed in COBOL68 and COBOL74.

The following table

- Lists the COBOL68 compiler options that are no longer available in COBOL85
- Shows the availability of the option in COBOL74
- Indicates the migration done by the CMT.

COBOL68 Compiler Option	In COBOL74?	Migration to COBOL85
ANALYZE	No	Delete option
ANSI74	No	Delete option
B2500	No	Delete option
CHECK	No	Delete option
CLEAR	Yes	Delete option
COMP	No	Delete option
GLOBAL	Yes	Change to COMMON
INTRINSIC	No	Delete option
LIB$	Yes	Delete option
LIBDOLLAR	Yes	Delete option
LIST$	Yes	Change to LISTDOLLAR
LISTDELETED	Yes	Delete option
OLDNOT	No	Delete option
SECGROUP	No	Delete option
STACK	No	Change to MAP

GLOBAL Clause

The term GLOBAL, which was used in the SELECT clause of the ENVIRONMENT DIVISION, in the data-name/FILLER definitions in the DATA DIVISION and in the USE PROCEDURE statement of the PROCEDURE DIVISION, was an extension to ANSI-68 and 74 COBOL. COBOL programs compiled at lexicographic level 3 or higher could use untyped procedures, files, and certain variables in the outer block of the host program by declaring them GLOBAL. In COBOL85, the term COMMON represents these features and the term GLOBAL is used to specify a global object in a nested source program environment. The CMT changes all references to GLOBAL to COMMON for COBOL85.

LOWER-BOUND and UPPER-BOUND

When LOWER-BOUND and UPPER-BOUND reference nonnumeric data items , the CMT changes them to LOW-VALUES and HIGH-VALUES respectively. However, when referencing a numeric item, the CMT changes LOWER-BOUND to the figurative constant ZERO and UPPER-BOUND to reference the SYMBOLIC CHARACTERS named UPPER-BOUND. The CMT inserts the SYMBOLIC CHARACTERS entry in the SPECIAL-NAMES of the converted program.

The following table shows the replacement of LOWER-BOUND and UPPER-BOUND by the CMT:

COBOL68	COBOL85
	SPECIAL-NAMES.
	SYMBOLIC CHARACTERS UPPER-BOUND
	UPPER-BOUNDS ARE 250.
01 GROUPDATA.	01 GROUPDATA.
03 DATAX PIC X(06).	03 DATAX PIC X(06).
03 DATA9 PIC 9(06).	03 DATA9 PIC 9(06).
MOVE LOWER-BOUND TO DATAX.	MOVE LOW-VALUES TO DATAX.
MOVE UPPER-BOUND TO DATAX.	MOVE HIGH-VALUES TO DATAX.
MOVE LOWER-BOUND TO DATA9.	MOVE ZERO TO DATA9.
MOVE UPPER-BOUND TO DATA9.	MOVE UPPER-BOUND TO DATA9.
IF DATAX = LOWER-BOUND . . .	IF DATAX = LOW-VALUES . . .
IF DATAX = UPPER-BOUND . . .	IF DATAX = HIGH-VALUES . . .
IF DATA9 = LOWER-BOUND . . .	IF DATA9 = ZERO . . .
IF DATA9 = UPPER-BOUNDS . . .	IF DATA9= UPPER-BOUNDS . . .

Reserved Words in COBOL68 and COBOL74

The CMT modifies a program that contains the COBOL74 and COBOL85 reserved words. They are turned into user-defined words by appending a character Q at the end of those reserved words. The reserved words are listed next in two groups.

The following COBOL74 reserved words were not reserved in COBOL68:

ALARM	BREAKOUT	COMMANDKEYS	COMMUNICATION
COPY-NUMBER	DEBUG-CONTENTS	DEBUG-ITEM	DEBUG-LINE
DEBUG-NAME	DEBUG-SUB-1	DEBUG-SUB-2	DEBUG-SUB-3
DEBUGGING	DICTIONARY	EGI	ESI
FIELD	FORM	FORM-KEY	INITIALIZE
KANJI	OFFSET	OPTIMIZE	PRINTING
PROCEDURES	READ-OK	READ-WRITE	REFERENCE
RELATIVE	RIBBON	SORT-MERGE	STACK
SUB-QUEUE-1	SUB-QUEUE-2	SUB-QUEUE-3	TAG-KEY
TAG-SEARCH	TIMER	TODAYS-NAME	WRITE-OK

More reserved words have been added to COBOL85 as follows:

ALPHABET	ALPHABETIC-LOWER	ALPHABETIC-UPPER
ALPHANUMERIC	ALPHANUMERIC-EDITED	ANY
BINARY	CLASS	COMMON
CONTENT	CONTINUE	CONVERTING
DAY-OF-WEEK	END-ABORT-TRANSACTION	END-ADD
END-ASSIGN	END-BEGIN-TRANSACTION	END-CALL
END-CANCEL	END-CLOSE	END-COMPUTE
END-CREATE	END-DELETE	END-DIVIDE
END-EVALUATE	END-FIND	END-FREE
END-GENERATE	END-IF	END-INSERT
END-LOCK	END-MODIFY	END-MULTIPLY
END-OPEN	END-PERFORM	END-READ
END-RECEIVE	END-RECREATE	END-REMOVE
END-RETURN	END-REWRITE	END-SAVE
END-SEARCH	END-SECURE	END-SET
END-START	END-STORE	END-STRING
END-STRUCT	END-TRANSACTION	END-UNSTRING
END-WRITE	EVALUATE	EXTERNAL

FALSE	GLOBAL	NATIONAL
NUMERIC-EDITED	ORDER	OTHER
PACKED-DECIMAL	PADDING	PURGE
REFERENCE	REPLACE	Standard-2
SW1	SW2	SW3
SW4	SW5	SW6
SW7	SW8	TEST
THEN	TRUE	

$SET BINDINFO for Binding Programs

For successful binding in COBOL85, you must set the BINDINFO compiler option in the subprogram and the host program. The CMT inserts the compiler option in

- A subprogram that has the LEVEL=n (n>2) compiler option.
- A host program that has the specification of USE EXTERNAL/USE AS GLOBAL clause in the DECLARATIVES section.

$SET LIBRARYPROG for Library Programs

The SHARING or TEMPORARY compiler option does not work in a COBOL85 program. COBOL85 uses the LIBRARYPROG compiler option to generate a COBOL68 or COBOL74 style library. The CMT inserts the LIBRARYPROG compiler option if the source program contains the SHARING or TEMPORARY compiler option.

Identification Division

The following paragraphs describe the changes made to the Identification Division by the CMT.

Abbreviation

The CMT changes an abbreviation as follows:

- ID changes to IDENTIFICATION

AUTHOR, INSTALLATION, DATE-WRITTEN, SECURITY Paragraphs

The AUTHOR, INSTALLATION, DATE-WRITTEN, and SECURITY paragraphs have been placed in the obsolete element category. COBOL85 still accepts these paragraphs, but the CMT comments them out for the sake of clarity by adding an asterisk (*) to column 7.

PROGRAM-ID Paragraph

The PROGRAM-ID entry was a comment entry in COBOL68 and thus might contain characters such as an asterisk (*) or a slash (/). COBOL85 does not allow those characters since a program-name is given in the PROGRAM-ID paragraph in COBOL85. The CMT modifies the PROGRAM-ID paragraph by

- Replacing the slash character with a hyphen (-).
- Suppressing characters, including that character itself, after any other disallowed character.

The original line remains a comment line and the original PROGRAM-ID entry is not lost.

User-Defined Paragraphs

User-defined Paragraphs are obsolete and the CMT comments them out.

Environment Division

The following paragraphs describe the changes made to the Environment Division by the CMT.

Abbreviation

The CMT changes an abbreviation as follows:

- I-O (SECTION) changes to INPUT-OUTPUT (SECTION).

ALPHABET-NAME Clause

The key word ALPHABET must precede alphabet-name-1 within the alphabet-name-1 clause of the SPECIAL-NAMES paragraph.

Because system-names and user-defined words could be the same word in COBOL85, the compiler might not be able to determine which use is intended. The introduction of the key word ALPHABET in the alphabet-name clause resolves this ambiguity.

```
SPECIAL-NAMES. WORD-1 IS WORD-2.
```

The CMT modifies any program that contains the SPECIAL-NAMES paragraph to use ALPHABET in front of alphabet-name-1. The preceding example would be modified as follows:

```
SPECIAL-NAMES. ALPHABET WORD-1 IS WORD-2.
```

APPLY, RERUN and MULTIPLE FILE Clause

The APPLY, RERUN and MULTIPLE FILE clauses of the I-O-CONTROL paragraph are not supported by COBOL85. The CMT deletes them.

AREAS and AREASIZE Attribute and INTERCHANGE Option

The CMT moves the AREAS and AREASIZE file attributes specification of the SELECT clause in the ENVIRONMENT DIVISION to the File Description in the DATA DIVISION. The INTERCHANGE option is removed.

Empty FILE-CONTROL Paragraph

The FILE-CONTROL paragraph with no SELECT entry was valid syntax in COBOL68 and COBOL74, but it is invalid in COBOL85. The CMT deletes the INPUT-OUT SECTION header and FILE-CONTROL paragraph if it includes no SELECT entry.

External Program Name in SPECIAL-NAMES Paragraph

An external program name for TASKING or BINDING is specified in SPECIAL-NAMES paragraph. Every node of the literal external program name was surrounded by quotation marks (") in COBOL68. In COBOL85, quotation marks are necessary only at the beginning and ending of the literal external program name. The CMT changes the COBOL68 format of literal external program names to the COBOL85 format.

Hardware Names

Hardware names in SELECT clause listed below are obsolete. The CMT changes them as follows:

- BACKUP TAPE, BACKUP DISK, BACKUP TAPE OR DISK, PRINTER BACKUP, PRINTER OR TAPE OR DISK and MESSAGE-PRINTER change to PRINTER
- PUNCH BACKUP, PUNCH OR TAPE OR DISK and CARD_PUNCH change to PUNCH
- CARD-READER and CARD-READERS change to READER
- DISKPACK and DISKPACKS change to REMOTE
- SPO, DISPLAY-UNIT and KEYBOARD change to REMOTE
- TAPES changes to TAPE

MEMORY SIZE Clause

The MEMORY SIZE clause of the OBJECT-COMPUTER paragraph has been placed in the obsolete element category. It is ignored in COBOL85 unless a SORT statement appears in the program. If a SORT statement is used in the program, the CMT moves the clause into SORT statement. If the program does not use a SORT statement, and has a MEMORY SIZE clause in the OBJECT-COMPUTER paragraph, the specification is commented out.

OBJECT-COMPUTER Paragraph

The SEGMENT-LIMIT and CODE SEGMENT-LIMIT clauses cause the COBOL85 compiler to emit a warning that the feature is not implemented. The STACK SIZE clause is ignored by the COBOL85 compiler without any warning. The CMT comments them out.

SAME AREA and SAME RECORD AREA Clauses

COBOL85 requires that all SAME AREA and SAME RECORD AREA clauses consist of only one sentence. A file may not appear in more than one SAME AREA clause. Also, a file may not appear in more than one SAME RECORD AREA clause. If one or more file-names of a SAME AREA clause appear in a SAME RECORD AREA clause, all of the file-names in that SAME AREA clause must appear in the SAME RECORD AREA clause. The CMT checks these rules.

SELECT Clauses

The following SELECT clauses are obsolete and are removed by the CMT.

- BY AREA
- BY CYLINDER
- FILE-LIMIT IS
- FILE-LIMITS ARE
- SINGLE

The CMT replaces the RESERVE NO clause with the RESERVE 1 clause. For the SAVE clause, the CMT issues a warning that it must be modified.

Data Division

The following paragraphs describe the changes made to the Data Division by the CMT.

Abbreviation

The CMT changes an abbreviation as follows:

- OC changes to OCCURS
- VA changes to VALUE
- PC changes to PIC
- CMP changes to BINARY EXTENDED

CONSTANT SECTION

The CMT replaces the CONSTANT SECTION header of the DATA DIVISION with the $SET OPT3 compiler option and inserts the $RESET OPT3 compiler option before the next SECTION or the PROCEDURE DIVISION. The items within the range of the OPT3 compiler option are regarded as constants by the COBOL85 compiler.

External File Name in File Description

Every node of the literal external file name in an FD entry was surrounded quotation marks (") in COBOL68. In COBOL85, the quotation marks are required only at the beginning and ending of the literal external file name. The CMT changes the COBOL68 format of literal external file name to the COBOL85 format. Additionally, the VALUE OF ID clause is replaced by the VALUE OF TITLE clause.

File Attribute Specification in File Description

COBOL68 allowed file attributes specified in an FD entry to contain hyphens. COBOL85, however, does not allow the hyphens. The CMT removes hyphens from such file attributes. Additionally, COBOL85 requires the VALUE OF clause to appear before the file attribute list. The CMT makes this change.

File Description Clauses

Some file description clauses are no longer supported. The CMT deletes the following clauses:

- DATA RECORDS
- LABEL RECORDS
- RECORDING MODE

Additionally, if the following clauses are expressed in words, the CMT changes them into characters:

- BLOCK CONTAINS
- FILE CONTAINS
- RECORD CONTAINS

Hexadecimal Literal Definition

The COBOL68 description of a hexadecimal literal was not the same as that of COBOL74 and COBOL85. COBOL68 permitted the following two ways to represent a hexadecimal literal and they are migrated as follows:

- Specifying a non-numeric literal as a COMP-2 group item. The literal was regarded as a hexadecimal literal and was left justified. The CMT changes USAGE COMP-2 to USAGE COMP and replaces the quotation mark (") of each non-numeric literal with the ampersand character (@).
- Delimiting both ends of the literal by the ampersand character, like COBOL74 or COBOL85. The feature was available only while the B2500 compiler option was set. The literal was aligned at the rightmost character position in the data item. The CMT deletes this B2500 compiler option.

HEX and WORDS File Support

A COBOL68 program could generate a HEX file whose INTMODE was HEX if the first 01 record entry under the file description was USAGE COMP-2. It also could generate a words file whose INTMODE was SINGLE if the first 01 record entry under the file description was USAGE COMP, COMP-1 or COMP-4. A COBOL74 program could generate a file whose INTMODE is HEX if the first 01 record entry under the file description is of USAGE COMP. In COBOL85, all group items are treated as EBCDIC arrays. To use the rule of COBOL68 or COBOL74, the following compiler options have to be declared before and after each file description.

- $SET COMPATIBILITY(RESET EBCDICFILES)
- $SET COMPATIBILITY(SET EBCDICFILES)

The CMT adds the appropriate compiler control option immediately before and after each file description if the first 01 record entry under it is of USAGE COMP, COMP-1, COMP-2 or COBOL-4.

COBOL68	COBOL85
	$SET COMPATIBILITY(RESET EBCDICFILES).
FD FILE-A BLOCK SIZE IS 30.	FD FILE-A BLOCK SIZE IS 30.
01 REC-A COMP-2.	01 REC-A COMP.
03 SUB-REC-A PIC 9(20).	03 SUB-REC-A PIC 9(20).
01 REC-AA.	01 REC-AA.
03 SUB-REC-AA PIC X(10).	03 SUB-REC-AA PIC X(10).
FD FILE-B MAXRECSIZE 80.	$SET COMPATIBILITY(SET EBCDICFILES).
01 REC-BPIC X(80).	FD FILE-B MAXRECSIZE 80.
	01 REC-BPIC X(80).

OCCURS Clause at 01 Level

The OCCURS clause on an 01 level data item is not supported by COBOL85. The CMT inserts an 01 level dummy group item above this item and bumps all necessary level numbers by 1. The CMT does nothing to 66, 77 or 88 level items.

COBOL68	COBOL85
	01 GEN-DATAITEM-nnn.
01 DATA-01 OCCURS 10.	02 DATA-01 OCCURS 10.
02 DATA-02-A.	03 DATA-02-A.
03 DATA-03-A .	04 DATA-03-A .
04 DATA-04.	05 DATA-04.
06 DATA-ITEM PIC 9(06).	06 DATA-ITEM PIC 9(06).
02 DATA-02-B REDEFINES DATA-02-A.	03 DATA-02-B REDEFINES DATA-02-A.
03 DATA-03-B PIC 9(05).	04 DATA-03-BPIC 9(05).

PICTURE Character J and K

The CMT changes the PICTURE characters J and K to the PICTURE character S. The SIGN IS LEADING specification and the LEADING SEPARATE CHARACTER specification is added as needed, respectively.

Qualifications in REDEFINES Clause

COBOL68 allowed qualifications in a REDEFINES clause, and COBOL85 does not allow this qualification. The CMT deletes qualifications encountered in a REDEFINES clause.

RANGE, RECORD AREA, SEGMENT and SIZE Clauses

Clauses in the DATA DIVISION such as RANGE, RECORD AREA, SEGMENT and SIZE are not implemented in COBOL74 or COBOL85. The CMT deletes them.

SIZE DEPENDING ON Clause

The CMT removes the SIZE clause when the DEPENDING ON phrase is not present. The SIZE DEPENDING ON <size > is also removed from WORKING-STORAGE items. All references to the associated data items in the PROCEDURE DIVISION are changed to reference modifications, i.e. <item> (1: <size>). For example, the CMT changes the phrase MOVE A TO B to MOVE A (1: <size>) TO B. When a SIZE DEPENDING ON clause is present in the FILE SECTION, the CMT replaces it with a RECORD CONTAINS clause.

A similar construction to the SIZE DEPENDING ON <size> clause is the PIC LX clause: PIC LX (n) DEPENDING ON <size>. The CMT changes LX (n) to X (n) and removes the DEPENDING ON <size> phrase. If the associated data item is in the FILE SECTION, the RECORD IS VARYING IN SIZE clause is inserted into the appropriate FD entry. Otherwise, all references to that data item in the PROCEDURE DIVISION are changed to reference modifications, i.e. <item> (1: <size>).

COBOL programs that use the SIZE...DEPENDING or PICTURE L...DEPENDING constructs create files with variable length records by using the BLOCKSTRUCTURE = EXTERNAL command. The SIZE...DEPENDING and PICTURE L...DEPENDING constructs are translated to use RECORD CONTAINS x TO y DEPENDING so that COBOL85 creates files with BLOCKSTRUCTURE = EXTERNAL.

USAGE BINARY Clause

In ANSI COBOL74, truncation of higher-order digits did not occur and, by default, SIZE ERROR conditions were limited to arithmetic faults such as INTEGER OVERFLOW conditions. In ClearPath and A Series COBOL74, it was necessary to specify the TRUNCATED phrase with the USAGE BINARY clause to use the contents of the PICTURE clause for higher-order digits truncation and SIZE ERROR determination. In COBOL85, the EXTENDED phrase is needed with the USAGE BINARY clause to achieve the same behavior as COBOL74 BINARY without the TRUNCATED phrase. Without it, behavior is the same as COBOL74 BINARY TRUNCATED. The CMT changes the syntax as shown in the following table, with one exception—01 level BINARY itmes in COBOL74 are kept unchanged because COBOL74 treated these items as BINARY TRUNCATED:

In COBOL74 . . .	Is equivalent to . . . in COBOL85
USAGE BINARY	USAGE BINARY EXTENDED
USAGE BINARY TRUNCATED	USAGE BINARY

USAGE COMP Clauses

The following USAGEs are not recognized by COBOL85. The CMT changes them as follows:

USAGE . . .	Is Changed To . . .
COMP	BINARY EXTENDED
COMP-1	BINARY EXTENDED
COMP-2	COMP
COMP-4	REAL
COMP-5	DOUBLE

The conversion of COMP and COMP-1 into BINARY EXTENDED and the conversion of COMP-5 to DOUBLE is required because these COBOL68 data types have a different meaning in COBOL85. BINARY EXTENDED is new for COBOL85.

When the compiler control option $SET B2500 is set in a COBOL68 program, USAGE COMP is not changed to USAGE BINARY EXTENDED.

USAGE CONTROL-POINT Clauses

The USAGE CP (CONTROL-POINT) is not recognized by COBOL85, and the CMT changes it to the USAGE TASK.

USAGE KANJI with PICTURE Character X

In COBOL85, the term NATIONAL replaces KANJI for data items representing national characters. The CMT changes the usage KANJI with the picture character X to the usage NATIONAL with the picture character N. The old specification is allowed as a synonym of the new representation in COBOL85 but will be deimplemented in a future release.

The characters NC, which were used to indicate the beginning of a national character string in COBOL74, are replaced by the character N in COBOL85. Using NC in COBOL85 programs results in a warning message for this release of COBOL85. The CMT replaces NC with N.

Procedure Division

The following paragraphs describe the changes made to the Procedure Division by the CMT.

Abbreviated Relational Conditions

The CMT modifies the conditional abbreviations as follows:

- 'IF X = 1 2 3 AND 4' becomes 'IF X = 1 AND 2 AND 3 AND 4'
- 'IF X = 1 2 3 OR 4' becomes 'IF X = 1 OR 2 OR 3 OR 4'

Additionally, the relational operator UNEQUAL is replaced by NOT EQUAL, and the relational operator EXCEEDS is replaced by GREATER THAN.

ANDs in MOVE Statements

COBOL68 allows AND between two operands in a MOVE statement. The COBOL85 compiler, however, issues a syntax error when AND is encountered in a MOVE statement. All ANDs encountered in a MOVE statement are changed to commas (,).

Audit Specification with BEGIN and END-TRANSACTION Statement

The BEGIN-TRANSACTION and END-TRANSACTION statements, when used in a COBOL68 program, might optionally omit the audit specification. In COBOL85, the specification of AUDIT and NO-AUDIT is mandatory. The CMT adds the AUDIT option to BEGIN-TRANSACTION statement and the NO-AUDIT option to END-TRANSACTION statement.

Example

In COBOL68, the syntax for default BEGIN and END-TRANSACTION statements were as follows:

```
BEGIN-TRANSACTION   <restart data set name>.

END-TRANSACTION     <restart data set name>.
```

In COBOL85, the equivalent syntax is as follows:

```
BEGIN-TRANSACTION   AUDIT   <restart data set name>.

END-TRANSACTION   NO-AUDIT <restart data set name>.
```

AWAIT Statement

The AWAIT statement is not implemented in COBOL74 or COBOL85. The CMT replaces occurrences of the AWAIT statement with the WAIT statement.

Blind MOVE Statement

In the blind MOVE statement, the sending field was DISPLAY or COMP-2 and the receiving field was COMP, COMP-1, or COMP-4 of any size within one word (48 bits). Data from the first six bytes of the sending field item was moved unchanged into the low-order six bytes of the receiving field item. The CMT changes the sending field to DISPLAY or COMP and the receiving field to BINARY EXTENDED. The CMT changes the blind MOVE into MOVE selected bits. In COBOL74 and COBOL85, however, the sending operand in MOVE selected bits has to be BINARY or REAL. The CMT creates a redefines item for the sending data name with BINARY EXTENDED usage and then moves this redefines item to the receiving data name instead. If the size of sending data name is less than 6, the CMT inserts a MOVE ZEROS TO <the receiving data name> statement before translating the blind MOVE.

CALL PROGRAM DUMP

CALL PROGRAM DUMP requested a program dump in COBOL68. The CMT changes this to the COBOL74 and 85 CALL SYSTEM DUMP syntax.

CALL SYSTEM WITH or ZIP Statement

In COBOL68, you could use the CALL SYSTEM WITH <data-name or file-name> statement or the ZIP <data-name or file-name> statement to pass a control message to the operating system. The CMT changes are as follows:

- CALL SYSTEM WITH or ZIP <data name> changes to CALL SYSTEM WFL USING <data-name>
- CALL SYSTEM WITH or ZIP <file name> changes to CALL SYSTEM WFL USING "BEGIN JOB; START <External File Name>; ENDJOB."

Class Condition

The ALPHABETIC test is true for the uppercase letters, the lowercase letters, and the space character. The ALPHABETIC-UPPER test is true for uppercase letters and the space character. The ALPHABETIC-LOWER test is true for lowercase letters and the space character.

Because lowercase letters have been added to the ALPHABETIC test, a COBOL68 or COBOL74 program that contains this test could behave differently when compiled in COBOL85. In COBOL68 and COBOL74, the ALPHABETIC class condition test is true only for the uppercase and space characters. The condition test in the following example returns a true value in COBOL85 but a false value in COBOL74:

```
77  ALPHA-DATAPIC X(10) VALUE "ABCDEfghij".
 .
 .
 .
    IF ALPHA-DATA IS ALPHABETIC . . .
```

The CMT replaces the reference to ALPHABETIC with ALPHABETIC-UPPER.

CLOSE HERE Statement

In COBOL68, the CLOSE HERE [NO REWIND] statement enabled you to write over the last portion of a tape file or to add to an existing tape file. The CMT changes the CLOSE HERE statement into a combination of CLOSE NO REWIND and OPEN OUTPUT NO REWIND statements.

CLOSE WITH LOCK

The CMT replaces the CLOSE WITH LOCK statement with the CLOSE WITH SAVE statement.

COMPILETIME Registers

COBOL85 does not support the COMPILETIME registers of COBOL68. The CMT replaces the COMPILETIME(5) and COMPILETIME(15) registers with a combination of MOVE statements and the COBOL85 intrinsic function WHEN-COMPILED using the dummy paragraph and variables. The CMT emits a warning message requiring a manual change for the COMPILETIME register when used with other integer numbers because COBOL85 contains no equivalent syntax.

COMPUTE with FROM or EQUALS Statement

The COMPUTE with FROM or EQUALS statement is not implemented in COBOL74 or COBOL85. The CMT replaces occurrences of the FROM or EQUALS clause of COMPUTE statement with the equal sign (=).

DIV and MOD Operators

The operators DIV (integer divide) and MOD (remainder divide) are not allowed in COBOL85. The CMT converts them to the appropriate intrinsic functions of COBOL85.

DIVIDE Statement with the MOD Option

The MOD option of the DIVIDE statement is not implemented in COBOL74 or COBOL85. The CMT replaces occurrences of the DIVIDE . . . MOD statement with the DIVIDE ... GIVING/REMAINDER option using a dummy variable.

DUMP Statement

The CMT deletes the DUMP statement. The COBOL85 Test and Debug System (TADS) should be used.

ELSE Phrase of Statements in PROCEDURE DIVISION Change

The ELSE phrase, which was associated with the phrases INVALID KEY, AT END, AT END-OF-PAGE, ON EXCEPTION and ON SIZE ERROR in COBOL68 in not supported by COBOL74 and COBOL85. Instead, COBOL85 supports the phrase NOT INVALID KEY, NOT AT END, NOT END-OF-PAGE, NOT ON EXCEPTION and NOT ON SIZE ERROR, respectively. The CMT changes are as follows:

COBOL68 Syntax	COBOL85 Syntax
ELSE phrase with ON SIZE ERROR phrase	NOT ON SIZE ERROR phrase used in COMPUTE, MULTIPLY, DIVIDE, ADD and SUBTRACT
ELSE phrase with ON EXCEPTION phrase	NOT ON EXCEPTION used in DMSII statements such as FIND, LOCK, OPEN, CREATE, STORE and so on.
ELSE phrase with AT END phrase	NOT AT END phrase used in READ and RETURN
ELSE phrase with END-OF-PAGE phrase	NOT END-OF-PAGE used in WRITE
ELSE phrase with INVALID KEY phrase	NOT INVALID KEY used in statements such as READ, WRITE, DELETE, REWRITE and START

ENTER Statement

The ENTER statement was the predecessor of the CALL statement for the calling of external subprograms. It is not allowed in COBOL74 or COBOL85 and has been placed in the obsolete category. The CMT replaces the ENTER statement with the CALL statement.

EXAMINE Statement

The EXAMINE statement is not implemented in COBOL74 or COBOL85. The CMT replaces occurrences of the EXAMINE statement with the INSPECT statement.

EXECUTE Statement

The EXECUTE statement is not implemented in COBOL74 or COBOL85. The CMT replaces occurrences of the EXECUTE statement with the RUN statement.

Group COMP/COMP-2 Item vs. Figurative Constant ZERO

When a figurative constant ZERO was moved to a COMP or COMP-2 group item, this item was initialized with binary zeros (LOW-VALUES) in COBOL68. In COBOL85, moving ZERO to a BINARY or COMP group item results in an alphanumeric value F0F0F0 . . . The CMT changes ZERO to LOW-VALUE in this case. The CMT applies the same changes for an IF statement in a similar situation.

Example

In COBOL68, the syntax for MOVE and IF statements for the COMP and COMP-2 group items was as follows:

```
MOVE  ZERO  TO  COMP-GROUP,  COMP-2-GROUP.

IF  ZERO  =  COMP-GROUP,  COMP-2-GROUP ...
```

In COBOL85, the equivalent syntax is as follows:

```
MOVE LOW-VALUE  TO  COMP-GROUP,  COMP-2-GROUP.

IF  LOW-VALUE  =  COMP-GROUP,  COMP-2-GROUP . . .
```

IF Statement for Task and File Attributes

The CMT changes the COBOL68 format of an IF statement for a task or file attribute to the COBOL85 format.

```
IF  <task/file-identifier>  (<attribute-name>)  becomes

IF  ATTRIBUTE  <attribute-name>  OF  <task/file-identifier>
```

Intrinsic Functions

Unlike COBOL68, the COBOL85 syntax for an intrinsic function consists of the word FUNCTION, the name of a specific predefined function, and one or more arguments. The CMT

- Adds the word FUNCTION before each function name.
- Changes the function name ARCTAN and LN of COBOL68 to ATAN and LOG, respectively.

MONITOR Statement

The CMT deletes the MONITOR statement. COBOL85 TADS should be used.

NOTE Statement

The NOTE statement is not implemented in COBOL74 or COBOL85. The CMT comments out the NOTE statement.

REMOTE File

The CMT changes the READ...INVALID clause for a REMOTE file to the READ...AT END clause.

SET and MOVE Statements for Task and File Attributes

The CMT replaces the SET statement for a task or file attribute with the CHANGE statement and changes the COBOL68 format of a MOVE statement for a task or file attribute to the COBOL85 format.

Example

In COBOL68, the syntax for SET and MOVE statements for the TASKVALUE task attribute was as follows:

```
SET  TASK-ID(TASKVALUE)  TO  1.

MOVE  TASK-ID(TASKVALUE)  TO TASK-ITEM.
```

In COBOL85, the equivalent syntax is as follows:

```
CHANGE  ATTRIBUTE  TASKVALUE  OF  TASK-ID  TO  1.

MOVE  ATTRIBUTE  TASKVALUE  OF  TASK-ID  TO  TASK-ITEM.
```

SORT Statement

In COBOL68 or COBOL74, when MEMORY SIZE and DISK SIZE clauses were specified in a SORT statement, they had to follow the INPUT PROCEDURE and OUTPUT PROCEDURE clauses. In COBOL85, the MEMORY SIZE and DISK SIZE clauses must precede the INPUT PROCEDURE and OUTPUT PROCEDURE clauses. The CMT moves the clauses to the location required by COBOL85.

In addition to this, the KEYS entry is changed to KEY by the CMT. Although either KEY or KEYS is valid in COBOL68 or COBOL74, COBOL85 allows only the KEY entry.

STOP Literal Statement

The literal variation of the STOP statement is now in the obsolete element category. In COBOL85, this format of the STOP statement continues to work as it did in COBOL74; however, the CMT replaces any literal variations of the STOP statement with a set of DISPLAY and ACCEPT statements:

```
PROCEDURE DIVISION.
      PARA-1.
STOP "Error in PARA-1".
```

In this example, the STOP statement suspends the run unit. You must reinitiate the run unit by typing ?OK on your terminal.

To modify this code for COBOL85, the CMT replaces the literal with a DISPLAY statement, and replaces the STOP statement with an ACCEPT statement, as follows:

```
PROCEDURE DIVISION.
      PARA-1.
DISPLAY "Error in PARA-1".
ACCEPT keyboard-option.
```

In this example, the DISPLAY statement will display the message on the ODT, and the ACCEPT message will reinitiate the run unit.

TIME Registers

COBOL85 does not support the TIME registers of COBOL68. The CMT changes the registers as follows:

- TIME(0) and TIME(10) registers change to the ACCEPT . . . FROM DAY statement.
- TIME(1) and TIME(11) registers change to the ACCEPT . . . FROM TIMER statement. The CMT inserts the extra COMPUTE statement for the migration of TIME(1) to change the value from increments of 2.4 microseconds to sixtieths of a second.
- TIME(2) and TIME(12) change to the MOVE statement with the dummy variable GEN-ACCUMPROCTIME. The variable gets the value from the task value ACCUMPROCTIME. The CMT inserts the extra COMPUTE statement for the migration of TIME(2) to change the value from increments of 2.4 microseconds to sixtieths of a second.
- TIME(3) and TIME(13) change to the MOVE statement with the dummy variable GEN-ACCUMIOTIME. The variable gets the value from the task value ACCUMIOTIME. The CMT inserts the extra COMPUTE statement for the migration of TIME(3) to change the value from increments of 2.4 microseconds to sixtieths of a second.
- TIME(4) and TIME(14) are marked as an error by the CMT. You must modify the program manually.

- TIME(5) and TIME(15) change to ACCEPT . . . FROM TODAYS-DATE statement

The DATA (n) functions are similarly changed by the CMT.

TRUE, FALSE

In COBOL68, the clause VALUE TRUE/FALSE can be used in arithmetic or conditional expressions. The semantic clause depends on the data name associated with it in the expression. In COBOL85, the use of TRUE, FALSE can be specified only for Boolean variables. The word VALUE is not required except when used in association with MYSELF, MYJOB, or TASK identifiers. The CMT deletes the word VALUE if TRUE/FALSE is specified for a Boolean variable and translates the clause to 1 or 0, respectively, if the clause is used with a numeric operand. No translation is performed when the clause is associated with MYSELF, MYJOB, or TASK identifiers.

TODAYS-DATE

The TODAYS-DATE was synonymous with the TIME(15). The CMT changes the TODAYS-DATE as well as the TIME(15).

WRITE Statement

In COBOL68, if the ADVANCING phrase was not used, automatic advancing was provided to cause single spacing after writing (that is, BEFORE ADVANCING 1 LINES). In COBOL74 or COBOL85, automatic advancing is provided as if AFTER ADVANCING 1 LINE were specified if the ADVANCING phrase is not used. The CMT inserts the BEFORE ADVANCING 1 LINE to the WRITE statement, if the statement is to a printer file and has no ADVANCING phrase.

Warnings Issued by the CMT

A number of COBOL68 and COBOL74 constructs cannot be translated to COBOL85 for the following reasons:

- It deeply depends on the algorithm.
- To be done correctly, the program itself must be changed.
- Some functions are obsolete and will be deleted from the next version of standard COBOL although this implementation of COBOL85 supports them.
- They should not be used because they do not provide portability.

Language Element

The following paragraphs describe the warnings issued for language elements by the CMT.

Direct I/O

Direct I/O is not implemented in COBOL74 or COBOL85. The CMT changes the SELECT clause that assigns a file to DIRECT. The CMT issues a warning message to the RECORD AREA clause for WORKING-STORAGE or any of the PROCEDURE DIVISION statements such as DEALLOCATE.

Data Division

The following paragraphs describe the warnings issued in the Data Division by the CMT.

COMP-2 Group Item Alignment

In COBOL68, group items were aligned according to their USAGE. For example, a COMP-2 group item could be aligned on a DIGIT boundary. In COBOL74 or COBOL85, group items must be treated as USAGE IS DISPLAY items, and as such must both begin and end on a byte boundary. To meet this change, any existing files with the old alignment must be converted. The CMT emits a warning message and recommends file conversion.

PICTURE DEPENDING ON Clause with PICTURE Character L

This feature was used to denote a variable length elementary item in COBOL68. Neither COBOL74 nor COBOL85 allows the syntax. The CMT emits a warning message to modify the program.

USAGE ASCII Clause

A data item with this clause is assumed to contain 8-bit-coded ASCII characters. Neither COBOL74 nor COBOL85 support the USAGE ASCII clause. The CMT issues a warning message for the clause because users must modify the program.

USAGE INDEX FILE Clause

The USAGE INDEX FILE clause is not supported by COBOL74 or COBOL85. In COBOL68, this clause is permitted only for DIRECT files. The CMT issues a warning message for the clause because users must modify the program.

Procedure Division

The following paragraphs describe the warnings issued in the Procedure Division by the CMT.

ALTER Statement

The ALTER statement has been placed in the obsolete element category. The use of the ALTER statement results in a program that could be difficult to understand and maintain because it changes the procedure referred to in a GO TO statement. The CMT issues a warning message recommending the use of the EVALUATE statement instead of the ALTER statement.

COPY...REPLACING Statement

The CMT does not update a copy library if the associated copy statement contains a REPLACING clause. If the translated source contains a warning about copy REPLACING, do the following. Make a temporary copy of the program, delete the REPLACING clauses and then translate it. This will generate translated copy libraries. Remember to discard the temporary source and its translation, because it will not include the necessary copy REPLACING clause.

Explicit Terminators

Explicit scope terminators are inserted on nested statements that consist of I-O or DMS statements with conditional phrases. However, there might be cases where the CMT is unable to determine the correct locations of explicit scope terminators. When explicit scope terminators are inserted in nested statements with nesting levels of 2 and above, the CMT issues a warning message recommending the manual verification of such statements.

HEX to EBCDIC Translation

A COMP-2 group item declared in a COBOL68 program was regarded as a hexadecimal array. If the group item was moved to a DISPLAY item, COBOL68 performed a HEX to EBCDIC translation. The CMT issues a warning message for each MOVE statement of this type.

Installation Intrinsic

Neither COBOL74 nor COBOL85 supports installation intrinsic functions. When the CMT finds the compiler control option INTRINSICS, the CMT deletes the option and recommends replacing it with COBOL85 intrinsic functions or a run-time library.

LOCK with COMP or with COMP-1 Statement

COBOL85 neither supports the LOCK with COMP or with COMP-1 statement nor provides an equivalent construct. The program must be modified. The CMT emits a warning message.

OPEN with REEL-NUMBER (Format 2) Statement

COBOL85 neither supports the OPEN with REEL-NUMBER (Format 2) statement nor provides an equivalent construct. The CMT emits a warning message.

OPEN with REVERSED Statement

The REVERSED phrase has been placed in the obsolete element category. Although COBOL85 continues to support the REVERSED phrase of the OPEN statement as defined by Standard COBOL, the CMT emits a warning to modify the program.

PL/I ISAM

The indexed sequential access method (ISAM) facility supports indexed files in COBOL68. In COBOL85, KEYEDIOII supports indexed files. The CMT emits a warning to modify the program.

You must convert ISAM data files to KEYEDIOII data files by following this process.

- Write a COBOL68 program to create a flat file from the ISAM file.
- Write a COBOL85 program that reads records from the flat file and writes the records into a COBOL85 indexed file that has the same key declarations as the COBOL68 ISAM file.

During the reading and writing process, the COBOL85 program converts the flat file into a KEYEDIOII data file with the same access keys that existed in the COBOL68 ISAM file.

SEEK with KEY CONDITION Clause Statement

COBOL85 neither supports the SEEK with KEY CONDITION clause statement nor provides an equivalent construct. The CMT issues a warning message for this.

SYNC LEFT/RIGHT

SYNCHONIZED effects are different between COBOL68 and COBOL85. CMT issues a warning asking the user for a manual change. For more information, refer to Apendix F, "Comparison of COBOL Versions."

USE AFTER RECORD SIZE ERROR Statement

COBOL85 neither supports the USE AFTER RECORD SIZE ERROR statement nor provides an equivalent construct. The CMT emits a warning message to modify the program.

USE Procedure For Tape Files

COBOL68 and COBOL74 support USE procedures that allow manipulation of tape label information. COBOL85 does not support this feature. The CMT emits a warning message to modify the program.

Error Messages

The following list arranges the error messages in numerical order based on the number that is displayed immediately preceding the error message. Each error message is accompanied by a brief explanation of its meaning. In general, an error message means the construct causes a syntax error for the translated program at compile time, the construct has no equivalent construct in COBOL85, or the construct is too complicated to be translated correctly by the translator. In most cases, an error construct requires a manual change in order for the translated program to compile and run successfully.

1810 : SAVE IS FLAGGED IN SELECT STATEMENT

SAVE in a SELECT statement is not supported in COBOL85.

1824 : NUMBER OF TAPES MUST BE >= 3, MANUAL CHANGE REQD

In COBOL85, the number of tapes required for a SORT file must be 3 or greater, whereas in COBOL68, this number can be less than 3. This error message is issued if the number of tapes is less than 3.

1829 : STRING LENGTH EXCEEDS THE MAX LGTH ALLOWED IN C85

If the length of the string being considered is greater than 160, this error is issued because the COBOL85 compiler will issue a syntax error on this string.

1842 : FILE-LIMITS CANNOT BE HANDLED

If the construct FILE-LIMIT(S) is specified in a SELECT statement, the associated file can be translated into a file with RANDOM ACCESS mode if the file is opened for INPUT. If the file is opened for output or I-O, however, this error message will be issued.

1862 : 'CHANNEL' IS FLAGGED FOR MANUAL CHANGE

If the number specified for the CHANNEL phrase is greater than 11, then the CMT issues this error message because the number must be within the range 01 to 11.

1864 : NO EQUIVALENT FOR READER-SORTER IN COBOL85

The hardware device READER-SORTER is not supported in COBOL85.

1867 : FD NOT FOUND FOR THIS SELECT STATEMENT

The CMT cannot find an FD entry for the file under consideration.

1872 : LIMIT NEEDS MANUAL CHG - COPY/REPLACING PRESENT

If the program contains a COPY...REPLACING statement, the CMT does not translate any FILE-LIMIT phrase encountered in the program.

1878 : RESERVE <DATA-NAME> AREA MUST BE MANUALLY CHANGED

If the token that follows the keyword RESERVE is a data name, instead of an integer, the CMT issues this error message because the feature is not supported in COBOL85.

1902 : LOWER-BOUND(S) IN THIS EXPRESSION NEEDS MANUAL CHG

Since there are several ways to translate the figurative constant LOWER-BOUND(S) depending on the data name associated with it, the CMT does not translate this constant if it sees the constant is specified within an expression.

1903 : UPPER-BOUND(S) IN THIS EXPRESSION NEEDS MANUAL CHG

Since there are several ways to translate the figurative constant UPPER-BOUND(S) depending on the data name associated with it, the CMT does not translate this constant if it sees the constant is specified within an expression.

2038 : ATTRIBUTE CAN'T BE USED WITH ABBREVIATED CONDITION

The COBOL85 compiler issues a syntax error if a file or task attribute is used with an abbreviated condition in an IF statement. Example: IF ATTRIBUTE FILEKIND OF FILE1 = VALUE DISK OR TAPE generates an error message.

2821 : USAGE ASCII INVALID IN COBOL85

USAGE ASCII is not supported in COBOL85.

2827 : SIZE...DEPENDING ON REQUIRES MANUAL CHANGE

If a group item contains an OCCURS clause and an elementary item declared under the group item contains a SIZE...DEPENDING ON clause, this error message is issued if one of the following conditions is met:

- The OCCURS clause of the group item is specified with the phrase FROM <integer1> TO <integer2>.
- There are additional elementary items declared under the group item with the elementary item that contains the SIZE...DEPENDING ON clause.

If one of the previous conditions is met, the CMT might not produce correct translations in the PROCEDURE DIVISION for all items declared under that group item.

2831 : 'DEPENDING' IN PICTURE CLAUSE INVALID IN COBOL85

If the phrase DEPENDING ON is specified without a SIZE clause or an LX picture string, the CMT does not recognize the construct.

2835 : INVALID USAGE IS FLAGGED FOR MANUAL CHANGE

Usage specifications such as ASCII, DISPLAY-1, INDEX FILE are not supported in COBOL85.

2844 : USAGE IS INVALID IN THE RECORD CONTAINS CLAUSE

CHARACTERS is the only keyword allowed in the RECORDS CONTAINS clause in COBOL85.

2850 : EMBEDDED SPACE IN QUOTED 'VALUE' WITH NUMERIC PIC

Spaces or non-numeric characters exist in the VALUE clause of a numeric data item.

2884 : 'READER96' IS FLAGGED AS HAVING NO EQUIVALENT

The hardware device READER96 is not supported in COBOL85.

2886 : 'TAPECASSETTE' IS FLAGGED AS HAVING NO EQUIVALENT

The hardware device TAPECASSETTE is not supported in COBOL85.

2887 : MAY BE DMSII TPS FUNCTION – NOT SUPPORTED IN C85

DMS TPS functions such as TRFORMAT, TRSUBFORMAT, and TRDATASIZE are not supported in COBOL85. CMT issues this error message when it encounters these reserved words in the PROCEDURE DIVISION.

2888 : DMSII TPS CONSTRUCT NOT SUPPORTED IN COBOL85

DMS TPS constructs such as TRANSACTION and TB are not supported in COBOL85. CMT issues this error message when it encounters these reserved words in the DATA DIVISION.

2917 : 'RECORD AREA' FLAGGED -- NO DIRECT I/O IN COBOL85

A direct array in the input program is not supported in COBOL85.

2918 : DEALLOCATE AND DIRECT I/O NOT AVAILABLE IN C85

The DEALLOCATE verb specified for direct arrays is not supported in COBOL85.

3826 : VALUE LENGTH > PIC LENGTH FLAGGED

The length of a declared PICTURE clause is smaller than that of the associated VALUE clause.

3840 : VARIABLE USED FOR FILE TITLE, NEED MANUAL CHANGE

CMT cannot translate a ZIP or CALL SYSTEM with a file name if the FD entry associated with the file includes an IDENTIFICATION attribute specified with a variable instead of a literal string.

3842 : USASI OR NON-STANDARD FLAGGED, LABEL RECORDS

The keywords USASI and NON-STANDARD in a LABEL RECORDS clause are not supported in COBOL85.

3846 : VALUE IS DATE-COMPILED IS FLAGGED

The clause VALUE IS DATE-COMPILED is not recognized in COBOL85.

5019 : BAD USE OF LOWER-BOUND - FLAGGED FOR MANUAL CHANGE

The CMT cannot recognize the use of LOWER-BOUND in the syntax.

5020: BAD USE OF UPPER-BOUND - FLAGGED FOR MANUAL CHANGE

The CMT cannot recognize the use of UPPER-BOUND in the syntax.

5029 : DATA ITEM NOT QUALIFIED, USED 1ST DEF

The CMT found a duplicate name in the input program and the duplicate name is not adequately qualified. The first found qualification is assumed. An adequate form of qualification would be A OF B OF ... X, where X is not a duplicate itself.

5068 : COBOL85 SWITCH VALUES RESTRICTED TO 0 OR 1

In COBOL85, a switch value can only be either OFF or ON, which are 0 or 1.

5205 : ALPHA LIT > 23 CHARS WITH NUMERIC PIC NOT ALLOWED

The CMT strips the surrounded quotes from an alpha literal if this literal is associated with a numeric data item. If the length of the literal is greater than 23, this error message is issued.

5621 : 'CALL' STATEMENT REQUIRES MANUAL VERIFICATION

The syntax used in the CALL statement cannot be translated to an equivalent construct in COBOL85.

5807 : STACKER SELECT INVALID IN COBOL85

The STACKER option in a WRITE statement is not supported in COBOL85.

5808 : WITH KEY CONVERSION OR LOCK FLAGGED IN SEEK

The SEEK statement with the options KEY CONVERSION or LOCK is not supported in COBOL85.

5809 : WITH LOCK OPTION IS FLAGGED IN A WRITE STATEMENT

The LOCK option in a WRITE statement is not supported in COBOL85.

5849 : MUST BE REFERENCED IN THE SPECIAL–NAMES PARAGRAPH

The CMT detects a non-numeric token that follows the keyword CHANNEL in a WRITE.ADVANCING statement. This error is issued to recommend that either the CHANNEL clause be defined in the SPECIAL-NAMES paragraph or an integer follow the CHANNEL keyword.

5890 : UNKNOWN ATTRIBUTE VALUE SETTING

This error is issued if the keyword BY, instead of TO, UP BY or DOWN BY, is encountered in a SET statement.

5893 : 'CHECKPOINT' IS NOT AVAILABLE IN COBOL85

The CHECKPOINT statement is not available in COBOL85. A manual translation must be made using the CALL verb as specified in Appendix D.

5894 : TOO MANY NESTED ATTRIBUTES, TRANSLATE MANUALLY

If the construct MYSELF (EXCEPTIONTASK) is encountered more than once in a task related statement, this error message is issued.

5905 : 'CHECKPOINT-STATUS' IS NOT AVAILABLE IN COBOL85

The syntax CHECKPOINT-STATUS is not supported in COBOL85. A manual translation must be made using the CALL verb as specified in Appendix D.

5907 : THIS 'TIME' FUNCTION IS NOT AVAILABLE IN COBOL85

If the numeric parameter of a TIME function is a number other than 0, 1, 2, 3, 5, 10, 11, 12, 13 or 15, the CMT issues this error message because it does not have an equivalent structure in COBOL85.

5911 : GROUP ITEM IN BLIND MOVE - REQUIRES MANUAL CHANGE.

BLIND MOVE can be translated into a BIT MOVE only if the source data item is an elementary item. If the source data item is a group item, then this error message is issued.

5913 : NON-01 ITEM W/ SIZE < 6 IN BLIND MOVE, CHECK REQD

In COBOL85, the source item in a bit MOVE statement must be BINARY, REAL, or DOUBLE, whereas in COBOL68, the source item in a BLIND MOVE must be DISPLAY or COMP-2. Therefore, CMT must create a REDEFINES item for the source item with usage BINARY EXTENDED. If the size of the source item is less than 6, then the REDEFINES BINARY EXTENDED item receives an error message when compiled with the COBOL85 compiler. For this reason, this error message is issued for the source item if its size is less than 6.

5933 : RESTART WITH DATANAME/FORMULA NEEDS MANUAL CHANGE

COBOL85 only supports RESTART IS <integer> in a SORT/MERGE statement. If a dataname or an expression is specified for the RESTART clause, this error message is issued.

5938 : 'OPEN REEL-NUMBER' NOT AVAILABLE IN COBOL85

The REEL-NUMBER option is not supported in an OPEN statement in COBOL85.

5942: I/O FROM READER-SORTER INVALID IN COBOL85

If the file specified in a READ statement is associated with a READER-SORTER device, this error message is issued because the READER-SORTER device is not supported in COBOL85.

5944 : DIRECT I/O CLAUSES IN READ AND WRITE INVALID

A direct I/O operation is detected in a READ or WRITE statement. This error message is issued for the statement because direct I/O is not supported in COBOL85.

5945 : DIRECT I/O 'KEY' CLAUSE INVALID

The KEY clause is used with direct I/O in a READ statement. This error message is issued for the statement because direct I/O is not supported in COBOL85.

5948 : COPY WITH REDEFINES CLAUSE REQUIRES MANUAL CHANGE

If a REDEFINES item is followed by a COPY statement, the CMT might produce an incorrect translation for the REDEFINES item. This error message is issued to require a manual change.

5957 : ZIP WITH UNKNOWN PARAMETER - REQUIRES MANUAL CHANGE

The parameter for the ZIP statement under consideration is not a data name or a file name.

6807 : PICTURE STRING EXCEEDS 23 DIGIT POSITIONS

If a data name is declared as a numeric item and its size specified in the picture string is greater than 23, the CMT issues this error message.

6838 : WAIT STATEMENT IS FLAGGED FOR MANUAL VERIFICATION

If the identifier used in a WAIT statement is a task (control point) variable, this error message is issued because WAIT with a task variable is not supported in COBOL85.

6845 : <IDENTIFIER> MUST BE CONTROL-POINT IDENTIFIER

This error message is issued when the variable found in the CALL under consideration is not a task variable.

6851 : COBOL68 ISAM REQUIRES MANUAL CHANGE TO KEYEDIOII

COBOL68 INDEXED FILE is not compatible with that of COBOL85. This error message is issued for an indexed file feature if it is detected in the input program.

9002 : NESTED COPY NOT SUPPORTED IN C85, CHANGE REQUIRED

If a COPY library contains other COPY statements, the CMT issues this error message because COBOL85 does not allow nested COPY statements.

9009 : UNEXPECTED TOKEN IN COPY STATEMENT

An unrecognized token is encountered in a COPY statement.

9010 : TABLE EXCEEDED - MORE THAN 20 REPLACE SYMBOLS

An internal table that saves REPLACING entries of COPY statements is exceeded.

9011 : DIRECT SWITCH FILE NOT SUPPORTED IN C85, CHG REQD

A direct switch file in a statement in the PROCEDURE DIVISION is not supported in COBOL85.

9801 : THRU/THROUGH IN COPY REPLACING, CHECK REQUIRED

If the keyword THRU or THROUGH is encountered in a COPY...REPLACING statement, the CMT issues this error message because it cannot process such a situation.

9804 : COPY LIBRARY MUST BE GENERATED WHERE L OCCURS

A character "L" appears at column 7 of a source line. This indicates that the source line comes from a COPY library in an unusual situation.

9820 : $FROM <INTEGER> NOT TRANSLATED

The CMT cannot translate $FROM <integer> structures.

9824 : OPTION SHOULD BE DELETED AND EXPRESSION CHANGED

The dollar option with a right hand side expression has no equivalent meaning in COBOL85.

9829 : DOLLAR OPTION HAS NO EQUIVALENCE -- FLAGGED

The dollar option has no equivalent meaning in COBOL85.

9843 : B7700 FLAGGED -- MANUAL CHANGE TO TARGET REQUIRED

A RESET to the dollar option B7700 is detected in the input program. The CMT cannot translate this specification.

9844 : DELETE OPTION -- DOLLAR OPTION ASSUMED RESET

The dollar option that has no meaning in COBOL85 is deleted and is assumed to be reset in the program.

9848 : OVERFLOW ON COMPILER CTRL RECORD, MANUAL CHG REQD

If the line containing translated dollar options is exceeded, the CMT issues this error message.

Warning Messages

The following list arranges the warning messages in numerical order based on the message number that is displayed immediately preceding the warning message. Each warning message is accompanied by a brief explanation of its meaning. In general, a warning message means the construct does not cause a syntax error for the translated program at compile time, but might cause a different result in comparison with the original statement.

1808 : LINAGE IN FD SHOULD BE DELETED FOR OPEN EXTEND

In COBOL85, files for which the LINAGE clause has been specified must not be opened in the extend mode. When a file was opened in the extend mode in COBOL68, LINAGE-COUNTER remained at 0. This warning message is issued at an OPEN EXTEND statement if the associated file contains a LINAGE clause in its FD entry.

1875 : SEARCH ALL STMT IS NOT COMPATIBLE WITH COBOL85

SEARCH ALL in COBOL68 or in COBOL74 is a linear search, while in COBOL85 it is a binary search.

1893 : HEX TO EBCDIC CONVERSION APPLIED FOR THIS MOVE

A HEX to EBCDIC conversion will be applied for this move in COBOL85 while there is no conversion for such a MOVE in COBOL68. An example of this kind of MOVE is when the sending item is a COMP group item and the receiving item is a DISPLAY group item.

1895 : ZERO COMPARED WITH VARIOUS ITEMS, CHECK REQUIRED

The CMT translates ZERO to either SPACE or LOWER-BOUNDS depending on the data type of the data name being compared. However, if ZERO is being compared with a variety of data items, then this warning is issued to require a manual check. For example,

if ZERO = A OR B OR C OR D, where A is DISPLAY usage, B is a group COMP-2 item, C and D are COMP usage.

1896 : ZERO COMPARED WITH COMPLEX EXP, MANUAL CHECK REQD

The CMT translates ZERO to either SPACE or LOWER-BOUNDS depending on the data type of the data name being compared. However, if the translator detects that ZERO is being compared with a complex expression then this warning is issued. For example,

```
IF ZERO = (A - 5 + B)
```

1897 : FILLING SPACE, NOT NULL, IF DESTINATION IS LONGER

When a literal surrounded by quotes is compared or moved to a COMP-2 elementary item, the CMT translates the quotes into at signs (@). It then issues this warning because the COBOL85 compiler appends spaces to the literal, instead of null as in COBOL68, if the length of the associated data name is longer than that of the literal.

1898 : ALPHA LIT COMPARED WITH VARIOUS ITEMS, CHANGE REQD

When a literal surrounded by quotes is compared with a COMP-2 elementary item, the CMT translates the quotes into at signs (@). However, if the literal is compared with a sequence of data items with a variety of data types, CMT cannot process the translation.

For example, IF "1234" = A OR B, where A is a COMP-2 elementary item and B is an alpha item.

1904 : MOVE LOWER-BOUND(S) TO DMS ITEMS, MANUAL CHG REQUIRED

The CMT is unable to make this translation because it has no knowledge of DMS data types.

1905 : MOVE UPPER-BOUND(S) TO DMS ITEMS, MANUAL CHG REQUIRED

The CMT is unable to make this translation because it has no knowledge of DMS data types.

1906 : COMPARE LOWER-BOUND(S) DMS ITEMS, MANUAL CHG REQUIRED

The CMT is unable to make this translation because it has no knowledge of DMS data types.

1907 : COMPARE UPPER-BOUND(S) DMS ITEMS, MANUAL CHG REQUIRED

The CMT is unable to make this translation because it has no knowledge of DMS data types.

2032 : GROUP/ELEMENTARY MOVE NOT COMPATIBLE WITH COBOL85

A MOVE of a group data name to an elementary data name, or the reverse, does not have the same result as that of the same MOVE in COBOL68.

2036 : NUM LIT COMPARED WITH VARIOUS ITEMS, CHANGE REQD

When a numeric literal is compared with a COMP-2 group item, the CMT translates it into a hex literal, which is surrounded by at signs (@). If the literal, however, is compared with a sequence of data items with a variety of data types, this warning message is issued.

2838 : COMP-2 GROUP ITEM CHANGED TO COMP, CHECK REQUIRED

In COBOL85, a COMP group item is treated as a DISPLAY group item whereas in COBOL68 a COMP-2 group item is treated as a hex item. This can cause different results in certain operations such as MOVE and IF statements.

3444 : FILLER WILL BE ADDED AFTER THIS LEVEL IN COBOL85

In COBOL85, if a group item is COMP and contains an OCCURS clause, and the total number of bytes within that group level is odd, the compiler adds a FILLER item to make the number of bytes even. This action does not happen in COBOL68 for a COMP-2 OCCURS group item. The CMT issues this warning message when it detects such a situation in the input program.

5018 : PACK NAME IN COPY CHANGED, MANUAL CHECK REQUIRED

If a pack name for COPY statements is specified from a user interface screen, the CMT issues this warning if a pack name is also specified in the COPY statement under consideration.

5030 : DATA ITEM QUALIFIER NOT FOUND, USED 1ST DEF

Upon encountering a qualification in a statement, the CMT looks for its parent in order to have an appropriate translation. If the parent of the qualified item is not found, the CMT assumes the first qualifier found in the DATA DIVISION and issues this warning message.

5804 : ALTER SHOULD BE REPLACED WITH EVALUATE

The ALTER statement is obsolete in COBOL85 and will be deleted from the next revision of Standard COBOL. Use the EVALUATE statement.

5826 : ELEMENTARY TO MULTIPLE GROUP MOVE, CHANGE REQUIRED

For a MOVE from an elementary item to a group item, results can be different between COBOL68 and COBOL85. For a single MOVE, the CMT generates a combination of multiple MOVE statements. However, for a MOVE to multiple destination items in which these destination items are group items, it is too complicated for the translator to handle the case.

5875 : SIGNED NUM MOVED TO COMP-2 GROUP, CHECK REQUIRED

When a signed numeric literal is moved to a COMP-2 group item, the result is different between COBOL68 and COBOL85.

5882 : SIGNED NUM MOVED TO ALPHANUMERIC, CHECK REQUIRED

When a signed numeric literal is moved to an alphanumeric item, the result is different between COBOL68 and COBOL85.

5999 : 'LOCK' STATEMENT REQUIRES MANUAL VERIFICATION

COBOL85 does not support use of a LOCK statement with a data name.

6861 : NON-DECLARATIVES PROC REF NOT SUPPORTED IN C85

When the DECLARATIVES SECTION is specified as an interrupt procedure, referencing from PERFORM statements inside the DECLARATIVES SECTION to sections or paragraphs declared outside the DECLARATIVES SECTION is not supported in COBOL85.

6899 : C74 INDEXED FILE MAY BE INCOMPATIBLE WITH COBOL85

COBOL74 supports KEYEDIO and KEYEDIOII while COBOL85 supports only KEYEDIOII. An indexed file declared in COBOL74 can be either KEYEDIO or KEYEDIOII. This warning is issued when the CMT detects an indexed file in a COBOL74 input program.

8011 : FILE SECTION MAY REQUIRE MANUAL CORRECTION

The CMT detects a file name duplication in the program and the message indicates a manual correction is required.

9006 : COPY..REPLACING:TRANSLATED COPY LIBRARY DISCARDED

The CMT does not update a copy library if the associated copy statement contains a REPLACING clause. To generate one, refer to COPY...REPLACING Statement in the section titled Warnings Issued by the CMT, Procedure Division.

9803 : COPY WITH SEQUENCE RANGE REQUIRES MANUAL CHECK

If only a portion of a copy library is desired for a program, the CMT does not update the copy file. Users should manually check the copy file for correct syntax after the translation.

9839 : LEVEL CHANGED TO 14, MANUAL VERIFICATION REQUIRED

A lex level greater than 14 is detected in the program. The CMT changes the lex level to 14 and issues this warning message.

9845 : OPTION DELETED -- ASSUMED RESET FOR TRANSLATION

The dollar option under consideration is not supported in COBOL85. It is deleted and is assumed to be reset.

9846 : COMPATIBILITY OPTION & EXP DELETED, CHECK REQUIRED

A system compatibility option with a right hand side expression is not supported in COBOL85. The option and the expression are deleted and a manual check is required for the program.

9847 : COMPATIBILITY OPTION DELETED, CHECK REQUIRED

A system compatibility option without a right hand side expression is not supported in COBOL85. The option is deleted and a manual check is required for the program.

Appendix H
Migrating V Series Intrinsics

Many of the V Series COBOL74 intrinsic routines are available as COBOL85 procedures that you can initiate by using Format 5 of the CALL statement. Located in the EVASUPPORT library, these procedures are intended primarily to ease the migration of V Series COBOL programs to ClearPath and A Series COBOL. While you can use some of these procedures in new ClearPath and A Series programs, it is recommended that you use the equivalent ClearPath and A Series code as indicated in Table H–1 and at the beginning of the discussion of each procedure.

If you use the V Series-to-A Series COBOL conversion filter to migrate your V Series COBOL programs to ClearPath and A Series COBOL85, the filter replaces the V Series calls with the syntax for the corresponding ClearPath and A Series procedure. If you do not use the filter, you must manually make these changes. For details about the automatic migration process provided by this filter, refer to the *EVA Application Program Transition Guide*.

Note: *Hereafter, the V Series-to-ClearPath and A Series COBOL conversion filter is referred to as the COBOL conversion filter.*

Summary of Procedures

Table H–1 summarizes the EVASUPPORT procedures for COBOL, which are described individually in this section. Note that the EVASUPPORT procedure name is similar to the V Series intrinsic name. One significant difference is the DISKFILEHEADER intrinsic name, which is VDISKFILEHEADER in the EVASUPPORT library.

Table H–1. EVASUPPORT Library Procedures

ClearPath and A Series EVASUPPORT Library Procedure	Description	ClearPath and A Series Equivalent
BINARYDECIMAL	Converts a binary data item to its decimal equivalent.	None
DATECOMPILED	Obtains the date and time of compilation for the calling program.	WHEN-COMPILED ANSI intrinsic function (Section 7)
DATENOW	Obtains the symbolic representation of the current date.	ACCEPT statement with either the DATE or TODAYS-DATE special register (Section 6) or the intrinsic function CURRENT-DATE (Section 7)
DECIMALBINARY	Converts a decimal data item to its binary equivalent.	None
EVA_TASKSTRING	Converts the data stored in the TASKSTRING attribute to three parameters and stores them in a global array where they can be queried by the GETPARAM and GETSWITCH procedures	None
GETMCP	Moves system log information generated by the EVA Logging Tools to the specified data item.	None
GETPARAM	Moves parameters that follow the V Series format to the specified data item.	TASKSTRING attribute (Refer to the Task Attributes Programming Reference Manual for details.)
GETSWITCH	Moves switches that follow the V Series format to the specified data item.	ClearPath and A Series task attributes SW1 through SW8 (Refer to the Task Attributes Programming Reference Manual for details.)

Table H–1. EVASUPPORT Library Procedures

ClearPath and A Series EVASUPPORT Library Procedure	Description	ClearPath and A Series Equivalent
INTERROGATE	Searches for a specified file and indicates whether it is resident.	RESIDENT file attribute (Refer to the File Attributes Programming Reference Manual for details.)
JOBINFO	Returns a structure that contains information about the job environment.	None
JOBINFO5	Returns a structure that contains information about the environment in which the job is running. MIX numbers in this structure are rendered in a five-digit format	None
MIX	Obtains a count of the programs in the mix.	None
MIX5	Obtains a count of the programs in the mix and returns the count as a five-digit number.	None
MIXID	Obtains a count of the programs in the mix that have a specific name. Replaces the V Series MIXID intrinsic.	None
MIXID5	Obtains a count of the programs in the mix that have a specific name. The five-digit result is placed in identifier-4 and returned.	None
MIXNUM	Obtains the mix number of the specified program.	MIXNUMBER task attribute (Refer to the Task Attributes Programming Reference Manual for details.)
MIXNUM5	Obtains the mix number of the specified program and returns it in a five-digit format. The result is placed in identifier-4.	MIXNUMBER task attribute
MIXTBL	Returns information about the programs in the mix.	None
MIXTBL5	Returns information about the programs in the mix into a table structure that you have allocated. Specific MIX number information is in a five-digit format within the structure.	None

Table H–1. EVASUPPORT Library Procedures

ClearPath and A Series EVASUPPORT Library Procedure	Description	ClearPath and A Series Equivalent
PROGINFO	Returns a structure that contains information about the job that is running.	ANSI intrinsic function WHEN-COMPILED (Section 7), and the ClearPath and A Series task attributes NAME, USERCODE, MIXNUMBER, and JOBNUMBER (Refer to the Task Attributes Programming Reference Manual for details.)
PROGINFO5	Returns information about the program that initiates the procedure. MIX information returned with it is formatted as five-digit values.	ANSI intrinsic function WHEN-COMPILED and the ClearPath and A Series task attributes NAME, USERCODE, MIXNUMBER, and JOBNUMBER.
SETSWITCH	Causes V Series switch settings passed by the program to be moved into a global array created by the EVA_TASKSTRING procedure.	ClearPath and A Series task attributes SW1 through SW8 (Refer to the Task Attributes Programming Reference Manual for details.)
SPOMESSAGE	Accepts and processes a V Series ODT command and returns a response to the calling program.	None. Use the CALL SYSTEM WFL statement to initiate tasks (Section 6).
TIMENOW	Obtains the symbolic representation of the current time.	ANSI intrinsic function CURRENT-DATE (Section 7) or the ACCEPT statement with the DATE, TODAYS-DATE, and TIME options (Section 6)
UNIQUENAME	Appends the mix number of the calling program to the name of a file as an additional node.	UNIQUETOKEN file attribute (Refer to the File Attributes Programming Reference Manual for details.)
VDISKFILEHEADER	Returns file information in the format of a V Series disk file header.	See Table H–10.
VREADTIMER	Obtains the symbolic representation of the current year, month, day, and time in the form YYYYMMDDFmmmmmmmmmmm	ANSI intrinsic function CURRENT-DATE (Section 7), or the ACCEPT statement with the TIMER option (Section 6)

Table H–1. EVASUPPORT Library Procedures

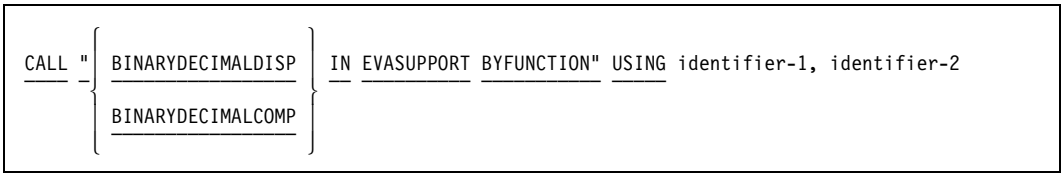
ClearPath and A Series EVASUPPORT Library Procedure	Description	ClearPath and A Series Equivalent
VTRANSLATE	Translates a string of digits or characters according to a specified equivalence table.	None
ZIP	Accepts and processes ClearPath and A Series ODT commands and the V Series commands EX, CH, and RM, but does not return a response to the calling program.	None. Use the CALL SYSTEM WFL statement to initiate tasks (Section 6).
ZIPSPO	Accepts and processes an ClearPath and A Series WFL command and the V Series EX, CH, and RM commands, and returns a response to the calling program only if the command is COMPILE or RUN and task initiation results.	CALL SYSTEM statement with the WFL option

BINARYDECIMAL Procedure

ClearPath and A Series Equivalent: None

The BINARYDECIMAL procedure converts a binary data item to its decimal equivalent. You can use this procedure in new ClearPath and A Series code.

Syntax



Explanation

BINARYDECIMALDISP

The keyword BINARYDECIMALDISP indicates that the binary item being converted is to be stored as a DISPLAY numeric data item.

BINARYDECIMALCOMP

The keyword BINARYDECIMALCOMP indicates that the binary item being converted is to be stored as an unsigned COMPUTATIONAL numeric data item.

identifier-1

This identifier references a binary data item to be converted to decimal format. This identifier must reference a COMPUTATIONAL numeric data item of 12 digits.

identifier-2

This identifier references the data item where the data converted to decimal format is stored.

- If the BINARYDECIMALDISP option is used, this identifier must reference an unsigned DISPLAY numeric data item of either 6 or 12 digits.
- If the BINARYDECIMALCOMP option is used, this identifier must reference an unsigned COMPUTATIONAL numeric data item of exactly 12 digits.

Details

The maximum value that can be converted is (2**39–1), which is hexadecimal 7FFFFFFFFF and decimal 549755813887. If the size of the converted data item is less than the size of identifier-2, the converted data is stored right-justified with leading zeros.

DATECOMPILED Procedure

ClearPath and A Series Equivalent: ANSI intrinsic function WHEN-COMPILED

The DATECOMPILED procedure obtains the date and time of compilation for the calling program in the form *hhmmMMDDYY*.

The letters . . .	Represent the . . .
hh	Hour
mm	Minutes
MM	Month
DD	Day
YY	Year

Syntax

```
CALL "{ DATECOMPILED     } IN EVASUPPORT BYFUNCTION" USING identifier-1
      { DATECOMPILEDCOMP }
```

Explanation

DATECOMPILED

The keyword DATECOMPILED specifies that the compilation date and time is to be stored as either an alphanumeric or a numeric DISPLAY data item.

DATECOMPILEDCOMP

The keyword DATECOMPILEDCOMP specifies that the compilation date and time is to be stored as a numeric COMPUTATIONAL data item.

identifier-1

This identifier references the data item where the compilation date and time is to be stored. The data is returned left-justified in identifier-1.

- If the DATECOMPILED option is used, identifier-1 must reference an alphanumeric data item of 10 characters declared as PIC X(10), or a numeric data item of 10 digits declared as PIC 9(10) or PIC 9(10) DISPLAY.
- If the DATECOMPILEDCOMP option is used, identifier-1 must reference a numeric COMPUTATIONAL data item of 10 digits declared as PIC 9(10) COMP.

If your program requires a numeric field with a length other than 10 digits, you must move the result to the appropriate field.

Details

This procedure builds a file title by using the NAME task attribute of the calling program. It then interrogates the CREATIONDATE and the CREATIONTIME file attributes of that file.

By default, the NAME task attribute is the same as the file title. To use this procedure, a program must not change the NAME task attribute. It must be the same as the code file title.

Because the procedure must interrogate file attributes, the code file must not have SECURITYUSE=SECURED, or be guarded. If either of these conditions exists, a run-time security violation can result. If the code file must be secured or guarded, use the ANSI intrinsic WHEN-COMPILED, described in Section 9.

DATENOW Procedure

ClearPath and A Series Equivalent: use the ACCEPT statement with either the DATE or TODAYS-DATE options, or the ANSI intrinsic function CURRENT-DATE

The DATENOW procedure obtains the symbolic representation of the current date in the form *MMM DD YYYY* or *MMM D YYYY*.

The letters . . .	Represent the . . .
MMM	Abbreviated name of the month
DD or D	Day
YYYY	Year

Syntax

```
CALL "DATENOW  IN  EVASUPPORT  BYFUNCTION" USING  identifier-1
```

Explanation

identifier-1

This identifier references an elementary alphanumeric data item of at least 12 characters, in which the result from this procedure is stored.

DECIMALBINARY Procedure

ClearPath and A Series Equivalent: None

The DECIMALBINARY procedure converts a decimal data item to its binary equivalent. You can use this procedure in new ClearPath and A Series code.

Syntax

```
CALL "{ DECIMALDISPBINARY  } IN EVASUPPORT  BYFUNCTION"
      { DECIMALCOMPBINARY  }

   USING identifier-1, identifier-2
```

Explanation

DECIMALDISPBINARY

The keyword DECIMALDISPBINARY indicates that the decimal item being converted is a DISPLAY numeric data item.

DECIMALCOMPBINARY

The keyword DECIMALCOMPBINARY indicates that the decimal item being converted is an unsigned COMPUTATIONAL numeric data item.

identifier-1

This identifier references the decimal data item to be converted to binary format.

- If the DECIMALDISPBINARY option is used, this identifier must reference an unsigned DISPLAY numeric data item whose length is either 6 digits, 12 digits, or any value greater than 12 digits.
- If the DECIMALCOMPBINARY option is used, this identifier must reference an unsigned COMPUTATIONAL numeric data item of exactly 12 digits.

identifier-2

This identifier references the data item where the converted data is to be stored. This identifier must reference an unsigned COMPUTATIONAL numeric data item of 12 digits.

Details

The maximum value that can be converted is (2**39–1), which is hexadecimal 7FFFFFFFFF and decimal 549755813887. If the size of the converted data is less than the size of identifier-2, the converted data is stored right-justified with leading zeros.

EVA_TASKSTRING Procedure

ClearPath and A Series Equivalent: None

The EVA_TASKSTRING procedure converts the data stored in the TASKSTRING attribute to three parameters that can be used by V Series programs that are migrated to ClearPath and A Series COBOL85. The three resulting parameters are stored in global arrays in the EVASUPPORT library for subsequent use by the GETSWITCH and GETPARAM procedures, which are described later in this section.

Note: *This procedure is intended for use only in V Series programs being migrated to ClearPath and A Series COBOL85.*

Syntax

```
CALL "EVA_TASKSTRING  IN  EVASUPPORT  BYFUNCTION"
```

Details

Each COBOL program that is filtered by the ClearPath and A Series-to-V Series COBOL conversion filter calls this procedure at the beginning of execution. When called, the procedure determines whether any data exists in the TASKSTRING attribute. When the call is completed, the global arrays for the program calling this procedure are updated with any parameters found in the TASKSTRING attribute. The program can subsequently access these parameters by calling the GETSWITCH and GETPARAM procedures.

The underscore character (_) in EVA_TASKSTRING is required.

Data in the TASKSTRING attribute is expected to be in the following format:

```
"value\parameter-1\parameter-2"
```

In this syntax,

value

Is the eight switches expressed in one numeric sequence in the following order: 01234567. If the value parameter is not used, 00000000 is stored in its position in the global array. Note that for ClearPath and A Series switches, position 0 (zero) is SW8.

parameter-1

Is the first 6-byte parameter. If parameter-1 is not used, spaces are stored in its position in the global array.

parameter-2

Is the second 6-byte parameter. If parameter-2 is not used, spaces are stored in its position in the global array.

If a preceding parameter is omitted, the backslash (\) is still required. The following are samples of possible syntaxes:

```
RUN PROG;TASKSTRING="01010101\ABCDEF\GHIJKL"

RUN PROG;TASKSTRING="01010101"

RUN PROG;TASKSTRING="\\GHIJKL"
```

GETMCP Procedure

ClearPath and A Series Equivalent: None

The GETMCP procedure reports information about the system logs being generated by the EVA Logging Tools. You can use this information to properly read the system logs.

Note: *This procedure is intended for use only in V Series programs being migrated to ClearPath and A Series COBOL85.*

Syntax

```
CALL "GETMCP IN EVASUPPORT BYFUNCTION" USING identifier-1
```

Explanation

identifier-1

This identifier references a group data item where the system log information is to be returned. For an example of the required structure of this data item, see "Details."

Details

The COBOL conversion filter changes calls to the V Series GETMCP intrinsic to the following CALL statement:

```
CALL "GETMCP IN EVASUPPORT BYFUNCTION" USING GETMCP-DATA.
```

If you do not use the COBOL conversion filter, you must manually make this change.

Note that a value of 0 (zero) is returned for the following items, because they are not supported in ClearPath and A Series COBOL:

- MLOG-NEXT-RECORD-POINTER
- RLOG-NEXT-FILE-NUMBER
- SLOG-NEXT-FILE-NUMBER
- MLOG-NEXT-FILE-NUMBER

If the EVA Logging Tools are not running, all pointer and number fields return a value of 0 (zero); however, the TIME-OF-RESPONSE and GETMCP-HOSTNAME fields still return correct values from the system.

GETPARAM Procedure

ClearPath and A Series Equivalent: ClearPath and A Series TASKSTRING attribute

The GETPARAM procedure returns the two TASKSTRING attribute parameters stored in a global array by the EVA_TASKSTRING procedure, described earlier in this section. For the GETPARAM procedure to return meaningful results, your program must have previously called the EVA_TASKSTRING procedure. The COBOL conversion filter automatically inserts a call to the EVA_TASKSTRING procedure during program migration. If you do not use the filter, you must manually add this call.

Note: *This procedure is intended for use only in V Series programs being migrated to ClearPath and A Series COBOL85.*

Syntax

```
CALL "GETPARAM IN EVASUPPORT BYFUNCTION" USING identifier-1
```

Explanation

identifier-1

This identifier references a group data item that contains two elementary alphanumeric data items of at least 6 characters each to which the parameter values are moved. Refer to "Details" for the structure of this data item.

Details

If the TASKSTRING attribute parameters are not present in the global array created by the EVA_TASKSTRING procedure, the data items defined for storing the result of the GETPARAM procedure are filled with spaces.

The COBOL conversion filter changes calls to the V Series GETPARAM intrinsic to the following CALL statement:

```
CALL "GETPARAM IN EVASUPPORT BYFUNCTION" USING PARAM.
```

If you do not use the COBOL conversion filter, you must manually make this change.

GETSWITCH Procedure

ClearPath and A Series Equivalent: ClearPath and A Series task attributes SW1 through SW8

The GETSWITCH procedure queries the value parameter of the TASKSTRING task attribute, which contains the values of the V Series switch settings, and returns the value in the data item referenced by identifier-1. If the value parameter is not used, 00000000 is returned.

For details about how the value parameter is returned in the TASKSTRING task attribute, refer to the discussion of the EVA_TASKSTRING procedure earlier in this section.

For the GETPARAM procedure to return meaningful results, your program must have previously called the EVA_TASKSTRING procedure. The COBOL conversion filter automatically inserts a call to the EVA_TASKSTRING procedure during program migration. If you do not use the filter, you must manually add this call.

Note: *This procedure is intended for use only in V Series programs being migrated to ClearPath and A Series COBOL85.*

Syntax

```
CALL "GETSWITCH IN EVASUPPORT BYFUNCTION" GIVING identifier-1
```

Explanation

identifier-1

This identifier references a computational numeric data item of 8 digits, described as PIC 9(8) COMP VALUE ZERO, to which the result from this procedure is returned.

Details

The COBOL conversion filter changes calls to the V Series GETSWITCH intrinsic to the following ClearPath and A Series CALL statement:

```
CALL "GETSWITCH IN EVASUPPORT BYFUNCTION" GIVING identifier-1
```

If you do not use the COBOL conversion filter, you must manually make this change.

INTERROGATE Procedure

ClearPath and A Series Equivalent: RESIDENT file attribute

The INTERROGATE procedure queries the RESIDENT attribute of the specified file and returns a flag that indicates whether the file exists. This procedure is intended for use only in V Series programs being migrated to ClearPath and A Series COBOL85.

Syntax

```
CALL "INTERROGATE IN EVASUPPORT BYFUNCTION" USING
    identifier-1, identifier-2, identifier-3
```

Explanation

identifier-1

This identifier is the name of the file to be interrogated. The file name can contain a maximum of 94 characters and must exist under your usercode.

identifier-2

This identifier is the name of the family on which the file resides. The family name can contain a maximum of 17 characters with no spaces. If the family name contains spaces, your default family is used.

identifier-3

This identifier references the data item where the result of the search is to be returned. The data item must be described as a 1-digit COMPUTATIONAL data item (PIC 9 COMP). If identifier-1 is longer than 1 digit, the result is returned in the left-most digit.

Details

This procedure tests the RESIDENT file attribute of the specified file under your usercode.

If the file is . . .	Then the RESIDENT attribute is . . .	And the value returned in identifier-1 is . . .
Present	TRUE	1
Not Present	FALSE	0

If you have archived or cataloged a file with the ClearPath and A Series Archive or Catalogue feature, the RESIDENT attribute is set to TRUE even though the files are not present. You must load the files to disk before they can be opened.

JOBINFO Procedure

ClearPath and A Series Equivalent: None

The JOBINFO procedure returns a structure that contains information about the environment in which the job is running.

Note: *This procedure is intended for use only in V Series programs being migrated to ClearPath and A Series COBOL85.*

Syntax

```
CALL "JOBINFO IN EVASUPPORT BYFUNCTION" USING identifier-1
```

Explanation

identifier-1

This identifier references an alphanumeric data item of at least 100 bytes where the returned structure is to be stored.

Details

The returned structure has the following format:

```
02  IDENTIFIER-1.
    03  RESPONSE-CODE                          PIC X(02).
    03  MCP-NAME                               PIC X(17).
    03  MCP-RELEASED-LEVEL                     PIC X(04).
    03  MCP-PATCH-LEVEL                        PIC X(06).
    03  MCP-VERSION-DATE                       PIC 9(06) COMP.
    03  SYSTEM-NUMBER-OF-CALLER                PIC 9(02) COMP.
    03  HOSTNAME                               PIC X(17).
    03  MIX-NUMBER-OF-CALLER                   PIC 9(04) COMP.
    03  BATCH-OR-TIMESHARING-FLAG              PIC 9(01) COMP.
    03  PACK-STATUS                            PIC 9(01) COMP.
    03  PRIMARY-BACKUP-FAMILY                  PIC X(06).
    03  SECONDARY-BACKUP-FAMILY                PIC X(06).
    03  CODE-PACK                              PIC X(06).
    03  RESERVED-2                             PIC X(28).
```

The fields in the result structure are filled as described in Table H–2.

Table H–2. Values in JOBINFO Result Structure

Field Identifier	Value Returned
RESPONSE-CODE	00 (zero) is returned for a normal response or 04 if the response area that you allocated is smaller than the response structure. The values 01 and 02 are not returned.
MCP-NAME	The value is the name of the current MCP. A leading asterisk (*) and the first node (SYSTEM/) are not displayed so that the version information is less likely to be truncated.
MCP-RELEASE-LEVEL	The values are the MARKDIGIT and LEVELNO fields.
MCP-PATCH-LEVEL	The value is the CYCLENO field translated to a displayable form.
MCP-VERSION-DATE	The value is the CREATIONDATE attribute of the MCP code file on disk converted to the mmddyy format or 999999 if the MCP code file is not found.
SYSTEM-NUMBER-OF-CALLER	0 (zero) is returned. ClearPath and A Series systems do not use system numbers.
HOSTNAME	The value is the HOSTNAME task attribute.
MIX-NUMBER-OF-CALLER	The value is the MIXNUMBER task attribute.
BATCH-OR-TIMESHARING-FLAG	0 (zero) is returned. This field is meaningless on both ClearPath and A Series systems and V Series systems running MCP/VS 2.0 or later.
PACK-STATUS	0 (zero) is returned.
PRIMARY-BACKUP-FAMILY	The value is the first six characters of the BACKUP family name of the system.
SECONDARY-BACKUP-FAMILY	The value is blank.
CODE-PACK	The value is blank.
RESERVED-2	Null (4-bit zeros) is returned.

JOBINFO5 Procedure

ClearPath and A Series Equivalent: None

The JOBINFO5 procedure returns a structure that contains information about the environment in which the job is running. It is designed to allow the return of five digit mix numbers.

Note: *This procedure is intended for use only in V Series programs being migrated to ClearPath and A Series COBOL85.*

Syntax

```
CALL "JOBINFO5 IN EVASUPPORT BYFUNCTION" USING identifier-1
```

Explanation

identifier-1

This identifier references an alphanumeric data item of at least 101 bytes where the returned structure is to be stored. The field MIX-NUMBER-OF-CALLER has been expanded to accommodate five-digit mix numbers.

Details

The returned structure has the following format:

```
02  IDENTIFIER-1.
    03  RESPONSE-CODE                          PIC X(02).
    03  MCP-NAME                               PIC X(17).
    03  MCP-RELEASED-LEVEL                     PIC X(04).
    03  MCP-PATCH-LEVEL                        PIC X(06).
    03  MCP-VERSION-DATE                       PIC 9(06) COMP.
    03  SYSTEM-NUMBER-OF-CALLER                PIC 9(02) COMP.
    03  HOSTNAME                               PIC X(17).
    03  MIX-NUMBER-OF-CALLER                   PIC 9(05) COMP.
    03  BATCH-OR-TIMESHARING-FLAG              PIC 9(01) COMP.
    03  PACK-STATUS                            PIC 9(01) COMP.
    03  FILLER                                 PIC 9(01) COMP.
    03  PRIMARY-BACKUP-FAMILY                  PIC X(06).
    03  SECONDARY-BACKUP-FAMILY                PIC X(06).
    03  CODE-PACK                              PIC X(06).
    03  RESERVED-2                             PIC X(29).
```

The fields in the result structure are filled as described in Table H–3.

Table H–3. Values in JOBINFO5 Result Structure

Field Identifier	Value Returned
RESPONSE-CODE	00 (zero) is returned for a normal response or 04 if the response area that you allocated is smaller than the response structure. The values 01 and 02 are not returned.
MCP-NAME	The value is the name of the current MCP. A leading asterisk (*) and the first node (SYSTEM/) are not displayed so that the version information is less likely to be truncated.
MCP-RELEASE-LEVEL	The values are the MARKDIGIT and LEVELNO fields.
MCP-PATCH-LEVEL	The value is the CYCLENO field translated to a displayable form.
MCP-VERSION-DATE	The value is the CREATIONDATE attribute of the MCP code file on disk converted to the mmddyy format or 999999 if the MCP code file is not found.
SYSTEM-NUMBER-OF-CALLER	0 (zero) is returned. ClearPath and A Series systems do not use system numbers.
HOSTNAME	The value is the HOSTNAME task attribute.
MIX-NUMBER-OF-CALLER	The value is the five-digit MIXNUMBER task attribute.
BATCH-OR-TIMESHARING-FLAG	0 (zero) is returned. This field is meaningless on both ClearPath and A Series systems and V Series systems running MCP/VS 2.0 or later.
PACK-STATUS	0 (zero) is returned.
FILLER	The value is a single-digit filler to accommodate the odd number of digits in the preceding MIX-NUMBER-OF-CALLER field.
PRIMARY-BACKUP-FAMILY	The value is the first six characters of the BACKUP family name of the system.
SECONDARY-BACKUP-FAMILY	The value is blank.
CODE-PACK	The value is blank.
RESERVED-2	Null (4-bit zeros) is returned.

JOBINFO5 enables the calling program to receive a five-digit mix number in the structure. JOBINFO only allows 4 digits and is a prerequisite to a conversion to the five-digit version.

Note: *Five-digit mix numbers are used by certain A Series systems that have been configured with the MoreTasks option when larger mix numbers are enabled.*

Transition Information

Where the JOBINFO5 procedure is used in an A Series application:

- A translation from the V Series form to the A-Series 4-digit form, JOBINFO, must have already taken place. At this point it is possible to convert the 4-digit form into JOBINFO5.
- Do not use the system number, the batch/timesharing flag, the pack status, the secondary backup or code pack names. These fields will not be available in the A Series intrinsic.
- Note that the returned A Series MCP name might be truncated. It can exceed 17 characters in length.
- Note that the returned A Series response code contains 04 if the response area provided by the caller is too small. Otherwise, this field contains 00.

MIX Procedure

ClearPath and A Series Equivalent: None

This procedure obtains a count of the programs in the mix.

Note: *This procedure is intended for use only in V Series programs being migrated to ClearPath and A Series COBOL85.*

Syntax

```
CALL "MIX IN EVASUPPORT BYFUNCTION" USING identifier-1
```

Explanation

identifier-1

This identifier references an elementary numeric COMPUTATIONAL data item of four digits in length where the result is to be returned. If your program requires the result to be of a different length or usage, you must move the result to another field.

MIX5 Procedure

ClearPath and A Series Equivalent: None

This procedure obtains a count of the programs in the mix. Since mix numbers can be up to five digits in length, the returned value is increased to accommodate five digits.

Note: *This procedure is intended for use only in V Series programs being migrated to ClearPath and A Series COBOL85.*

Syntax

```
CALL "MIX5 IN EVASUPPORT BYFUNCTION" USING identifier-1
```

Explanation

identifier-1

This identifier references an elementary numeric COMPUTATIONAL data item of five digits in length where the result (mix count) is to be returned. If your program requires the result to be of a different length or usage you must move the result to another field.

Details

For instance, when the USAGE of A-IDENTIFIER (for example, 77 A-IDENTIFIER PIC X(5) DISPLAY) is DISPLAY, convert the program to:

```
77  FLT-MIX-COMP                                    PIC 9(5) COMP.
.
.
.
CALL "MIX5 IN EVASUPPORT BYFUNCTION" USING FLT-MIX-COMP.
MOVE FLT-MIX-COMP TO A-IDENTIFIER.
```

MIXID Procedure

ClearPath and A Series Equivalent: None

This procedure obtains a count of the programs in the Mix that have a specific name.

Note: *This procedure is intended for use only in V Series programs being migrated to ClearPath and A Series COBOL85.*

Syntax

```
CALL "MIXID IN EVASUPPORT BYFUNCTION" USING
      identifier-1, identifier-2,
      identifier-3, identifier-4
```

Explanation

identifier-1

This identifier is the name of the program for which the count is to be performed. You can specify a program-name with a maximum of 94 characters. Embedded periods (.) and blanks are not valid in a program-name.

identifier-2

This identifier is the name of the disk family on which the object code file for the program resides. If you want to interrogate a program running under your default family, you must specify your default family as the family name. If the family name is spaces, it is assumed that the program for which you are performing the MIXID procedure appears in the mix without a family name.

identifier-3

This identifier is the usercode under which the object code file for the program resides. If you want to interrogate a program running under your own usercode, you must specify your usercode. If the usercode is spaces, it is assumed that the program for which you are performing the MIXID procedure appears in the mix without a usercode.

identifier-4

This identifier references a 4-digit COMPUTATIONAL numeric data item (PIC 9(4) COMP) where the result from this procedure is to be stored. If your program requires a field of a different length or usage, such as USAGE DISPLAY, you must move the result to the desired field.

MIXID5 Procedure

ClearPath and A Series Equivalent: None

This procedure obtains a count of the programs in the mix that have a specific name. Since the total number in the count can be up to five digits, the returned value of this routine is increased to five digits.

Note: *This procedure is intended for use only in V Series programs being migrated to ClearPath and A Series COBOL85.*

Syntax

```
CALL "MIXID5 IN EVASUPPORT BYFUNCTION" USING

      identifier-1, identifier-2,
      identifier-3, identifier-4.
```

Explanation

identifier-1

This identifier is the name of the program for which the count is to be performed. You can specify a program-name with a maximum of 94 characters. Embedded periods (.) and blanks are not valid in a program-name.

identifier-2

This identifier is the name of the disk family on which the object code file for the program resides. If you want to interrogate a program running under your default family, you must specify your default family as the family name. If the family name is spaces, it is assumed that the program for which you are performing the MIXID5 procedure appears in the mix without a family name.

identifier-3

This identifier is the usercode under which the object code file for the program resides. If you want to interrogate a program running under your own usercode, you must specify your usercode. If the usercode is spaces, it is assumed that the program for which you are performing the MIXID5 procedure appears in the mix without a usercode.

identifier-4

This identifier references a five-digit COMPUTATIONAL numeric data item (PIC 9(5) COMP) where the result from this procedure is to be stored. If your program requires a field of a different length or usage, such as USAGE DISPLAY, you must move the result to the desired field.

Details

Parameters 2 and 3 are supported because A Series program titles require not only a name, but a usercode and family name. Normally they would not be needed on a V Series call. These parameters are supplied in a user's program when they convert from the V Series form to the 4-digit, A Series form of the procedure call.

Along with the change to the procedure call, the following change must be made to the data structure in Working-Storage:

```
     77  FLT-MIXID-NAME                    PIC X(06).
     77  FLT-MIXID-FAMILY                  PIC X(17) VALUE SPACES.
     77  FLT-MIXID-USERCODE                PIC X(17) VALUE SPACES.
change the final line to
     77  FLT-MIXID-COMP                    PIC 9(05) COMP.
```

The routine uses only the contents of the parameter fields to construct the program title. When the FLT-MIXID-FAMILY or FLT-MIXID-USERCODE field is blank, the caller's family and usercode are not inserted.

With the added parameters, any A Series program can be specified. However, names with embedded blanks or periods are not supported. This routine also accepts a program name field up to 94 bytes in length. If a six-character name (of either the program being sought or its family name) is passed to the routine, but the A Series name is longer than six characters, the program will not be found.

Example of a Variable as a Parameter

Where M-COUNT is USAGE COMPUTATIONAL:

```
CALL "MIXID5 IN EVASUPPORT BYFUNCTION"
  USING P-NAME
  FLT-MIXID-FAMILY
  FLT-MIXID-USERCODE
  M-COUNT.
```

Where M-COUNT is USAGE DISPLAY:

```
CALL "MIXID5 IN EVASUPPORT BYFUNCTION"
  USING P-NAME
  FLT-MIXID-FAMILY
  FLT-MIXID-USERCODE
  FLT-MIXID-COMP.
MOVE FLT-MIXID-COMP TO M-COUNT.
```

Example of a Literal as a Parameter

Where M-COUNT is USAGE COMPUTATIONAL:

```
MOVE "MYPROG" TO FLT-MIXID-NAME.
CALL "MIXID5 IN EVASUPPORT BYFUNCTION"
  USING FLT-MIXID-NAME
  FLT-MIXID-FAMILY
  FLT-MIXID-USERCODE
  M-COUNT.
```

Where M-COUNT is USAGE DISPLAY:

```
MOVE "MYPROG" TO FLT-MIXID-NAME.
CALL "MIXID5 IN EVASUPPORT BYFUNCTION"
  USING FLT-MIXID-NAME
  FLT-MIXID-FAMILY
  FLT-MIXID-USERCODE
  FLT-MIXID-COMP.
MOVE FLT-MIXID-COMP TO M-COUNT.
```

MIXNUM Procedure

ClearPath and A Series Equivalent: MIXNUMBER task attribute

This procedure obtains the mix number of the specified program.

Note: *This procedure is intended for use only in V Series programs being migrated to ClearPath and A Series COBOL85.*

Syntax

```
CALL "MIXNUM IN EVASUPPORT BYFUNCTION" USING

     identifier-1, identifier-2
     identifier-3, identifier-4
```

Explanation

identifier-1

This identifier is the name of the program for which the mix number is to be returned. You can specify a program-name with a maximum of 94 characters. Embedded periods (.) and blanks are not valid in a program-name.

identifier-2

This identifier is the name of the disk family on which the object code file for the program resides. If you want to interrogate a program running under your default family, you must specify your default family as the family name. If the family name is spaces, it is assumed that the program for which you are performing the MIXNUM procedure appears in the mix without a family name.

identifier-3

This identifier is the usercode under which the object code file for the program resides. If you want to interrogate a program running under your own usercode, you must specify your usercode. If the usercode is spaces, it is assumed that the program for which you are performing the MIXNUM procedure appears in the mix without a usercode.

identifier-4

This identifier references a 4-digit COMPUTATIONAL numeric data item (PIC 9(4) COMP) where the result from this procedure is to be stored. If your program requires a field of a different length or usage, such as USAGE DISPLAY, you must move the result to the desired field.

MIXNUM5 Procedure

ClearPath and A Series Equivalent: MIXNUMBER task attribute

This procedure obtains the mix number of the specified program. It returns it in a five-digit format.

Note: *This procedure is intended for use only in V Series programs being migrated to ClearPath and A Series COBOL85.*

Syntax

```
CALL "MIXNUM5 IN EVASUPPORT BYFUNCTION" USING

     identifier-1, identifier-2,
     identifier-3, identifier-4.
```

Explanation

identifier-1

This identifier is the name of the program for which the mix number is to be returned. You can specify a program-name with a maximum of 94 characters. Embedded periods (.) and blanks are not valid in a program-name.

identifier-2

This identifier is the name of the disk family on which the object code file for the program resides. If you want to interrogate a program running under your default family, you must specify your default family as the family name. If the family name is spaces, it is assumed that the program for which you are performing the MIXNUM5 procedure appears in the mix without a family name.

identifier-3

This identifier is the usercode under which the object code file for the program resides. If you want to interrogate a program running under your own usercode, you must specify your usercode. If the usercode is spaces, it is assumed that the program for which you are performing the MIXNUM5 procedure appears in the mix without a usercode.

identifier-4

This identifier references a five-digit COMPUTATIONAL numeric data item (PIC 9(5) COMP) where the result from this procedure is to be stored. If your program requires a field of a different length or usage, such as USAGE DISPLAY, you must move the result to the desired field.

The conversion to use the MIXNUM5 function follows the conversion from the V Series to the 4-digit A Series function, MIXNUM. MIXNUM is established through the automated EVA workbench translation process. MIXNUM5 is edited manually later when the user moves to a MoreTasks environment.

Details

The following changes should be noted for the conversion process:

- After the following items in Working-Storage:

```
77  FLT-MIXNUM-NAME                    PIC X(6).
77  FLT-MIXNUM-FAMILY                  PIC X(17) VALUE SPACES.
77  FLT-MIXNUM-USERCODE                PIC X(17) VALUE SPACES.
```

  update the following line to

```
77  FLT-MIXNUM-COMP                    PIC 9(05) COMP.
```

- Convert the CALL statement as follows:

```
CALL "MIXNUM IN EVASUPPORT BYFUNCTION"...
```

  becomes

```
CALL "MIXNUM5 IN EVASUPPORT BYFUNCTION"...
```

The routine uses only the contents of the parameter fields to construct the program title. When the FLT-MIXNUM-FAMILY or FLT-MIXNUM-USERCODE field is blank, the caller's family and usercode are not inserted. Both FLT-MIXNUM-FAMILY or FLT-MIXNUM-USERCODE can be blank. A caller wishing to interrogate a program running under the caller's own FLT-MIXNUM-FAMILY or FLT-MIXNUM-USERCODE must supply the contents of those fields. With the added parameters, any A Series program can be specified. However, names with embedded blanks or periods are not supported. This routine also accepts a program name field up to 94 bytes in length.

Example of a Variable as a Parameter

Where M-COUNT is USAGE COMPUTATIONAL:

```
CALL "MIXNUM5 IN EVASUPPORT BYFUNCTION"
  USING P-NAME
  FLT-MIXNUM-FAMILY
  FLT-MIXNUM-USERCODE
  M-COUNT.
```

Where M-COUNT is USAGE DISPLAY:

```
CALL "MIXNUM5 IN EVASUPPORT BYFUNCTION"
  USING P-NAME
  FLT-MIXNUM-FAMILY
  FLT-MIXNUM-USERCODE
  FLT-MIXNUM-COMP.
MOVE FLT-MIXNUM-COMP TO M-COUNT.
```

Example of a Literal as a Parameter

Where M-COUNT is USAGE COMPUTATIONAL:

```
MOVE "MYPROG" TO FLT-MIXNUM-NAME.
CALL "MIXNUM5 IN EVASUPPORT BYFUNCTION"
  USING FLT-MIXNUM-NAME
  FLT-MIXNUM-FAMILY
  FLT-MIXNUM-USERCODE
  M-COUNT.
```

Where M-COUNT is USAGE DISPLAY:

```
MOVE "MYPROG" TO FLT-MIXNUM-NAME.
CALL "MIXNUM5 IN EVASUPPORT BYFUNCTION"
  USING FLT-MIXNUM-NAME
  FLT-MIXNUM-FAMILY
  FLT-MIXNUM-USERCODE
  FLT-MIXNUM-COMP.
MOVE FLT-MIXNUM-COMP TO M-COUNT.
```

MIXTBL Procedure

ClearPath and A Series Equivalent: None

This procedure returns information about the programs in the mix into a table structure that you have allocated.

Note: *This procedure is intended for use only in V Series programs being migrated to ClearPath and A Series COBOL85.*

Syntax

```
CALL "MIXTBL IN EVASUPPORT BYFUNCTION" USING identifier-1
```

Explanation

In this syntax, identifier-1 references the data item where the result is returned. The structure is a group data item of the following format:

```
02  ID1.
      03  HEADER.
           04  JOBS-IN-MIX           PIC 9(4)  COMP.
           04  MEM-AVAILABLE         PIC 9(10) COMP.
      03  ENTRY                 OCCURS nnn TIMES.
           04  PROGRAM-NAME          PIC X(6).
           04  MULTI-PROG-NAME       PIC X(6).
           04  MIX-NUMBER            PIC 9(4)  COMP.
           04  MEMORY-USED-BY-JOB    PIC 9(7)  COMP.
           04  PROCESSOR-PRIORITY    PIC 9     COMP.
           04  MEMORY-PRIORITY       PIC 9     COMP.
           04  SPECIAL-PROGRAM-CODE  PIC 9     COMP.
           04  PROGRAM-STATUS-CODE   PIC 99    COMP.
```

The letters nnn in the 03-level data entry description, ENTRY OCCURS nnn TIMES, represent the number of entries in the table that you want to allocate. Be sure to allocate a table of sufficient size for your needs. You might need a larger table for ClearPath and A Series entries than you needed for V Series entries.

If the response area is not large enough for all the entries in the ClearPath and A Series Mix, the MIXTBL routine returns as many complete entries as fit. If the response area is larger than the response, the last entry is followed by blanks.

The other elementary data items in this group item are described in Table H–4.

Table H–4. Table Structure for MIXTBL Procedure

The elementary item . . .	Stores . . .
JOBS-IN-MIX	The total number of mix entries.
MEM-AVAILABLE	The amount of available system memory in words.
PROGRAM-NAME	The first six characters of the last node of the task name.
MULTI-PROG-NAME	The first six characters, excluding parentheses, of the usercode under which the task was initiated.
MIX-NUMBER	The mix number of this job or task.
MEMORY-USED-BY-JOB	The number of total words of memory referenced by the segment and data descriptors in the D1 stack. (No exact ClearPath and A Series equivalent exists for this V Series field.)
PROCESSOR-PRIORITY	The priority of the task, divided by 10.
MEMORY-PRIORITY	Zero.
SPECIAL-PROGRAM-CODE	A value that indicates the type of program. The value is mapped from internal ClearPath and A Series mix attributes. The ClearPath and A Series mapping for V Series program codes is shown in Table H–5.
PROGRAM-STATUS-CODE	A value that indicates the status of the program. This value is usually 0 (zero). If the task is determined to be a compiler, this value is 01.

The SPECIAL-PROGRAM-CODE values for the MIXTBL procedure are described in Table H–5.

Table H–5. Values of the SPECIAL-PROGRAM-CODE Field for the MIXTBL Procedure

A value of . . .	Means that the program . . .
1	Is a generator.
2	Is DMPALL.
6	Has MCS status.
B	Is a DMS control program.
C	Is a job.
E	Is COPY.

MIXTBL5 Procedure

ClearPath and A Series Equivalent: None

This procedure returns information about the programs in the mix in a table structure that you have allocated. Specific mix number information has been expanded to return five-digit mix numbers.

Note: *This procedure is intended for use only in V Series programs being migrated to ClearPath and A Series COBOL85.*

Syntax

```
CALL "MIXTBL5 IN EVASUPPORT BYFUNCTION" USING identifier-1
```

Explanation

identifier-1

In this syntax, identifier-1 references the data item where the result is returned. The structure is a group of data items in the following format:

```
02  ID1.
      03  HEADER.
           04  JOBS-IN-MIX           PIC 9(5)  COMP.
           04  MEM-AVAILABLE         PIC 9(10) COMP.
      03  ENTRY                 OCCURS nnn TIMES.
           04  PROGRAM-NAME          PIC X(6).
           04  MULTI-PROG-NAME       PIC X(6).
           04  MIX-NUMBER            PIC 9(5)  COMP.
           04  MEMORY-USED-BY-JOB    PIC 9(7)  COMP.
           04  PROCESSOR-PRIORITY    PIC 9     COMP.
           04  MEMORY-PRIORITY       PIC 9     COMP.
           04  SPECIAL-PROGRAM-CODE  PIC 9     COMP.
           04  PROGRAM-STATUS-CODE   PIC 9(2)  COMP.
```

The letters nnn in the 03-level data entry description, ENTRY OCCURS nnn TIMES, represent the number of entries in the table that you want to allocate. Be sure to allocate a table of sufficient size for your needs. You might need a larger table for ClearPath and A Series entries than you needed for V Series entries.

If the response area is not large enough for all the entries in the ClearPath and A Series mix, the MIXTBL5 routine returns as many complete entries as will fit. If the response area is larger than the response, the last entry is followed by blanks.

The other elementary data items in this group item are described in Table H–6.

Table H–6. Table Structure for MIXTBL5 Procedure

The elementary item . . .	Stores . . .
JOBS-IN-MIX	The total number of mix entries in a five-digit format.
MEM-AVAILABLE	The amount of available system memory in words.
PROGRAM-NAME	The first six characters of the last node of the task name (truncated if longer than six characters).
MULTI-PROG-NAME	The first six characters, excluding parentheses, of the usercode under which the task was initiated.
MIX-NUMBER	The mix number of this job or task in a five-digit format. This is a two-digit or four-digit value on V Series.
MEMORY-USED-BY-JOB	The number of total words of memory referenced by the segment and data descriptors in the D1 stack (no exact ClearPath and A Series equivalent exists for this V Series field).
PROCESSOR-PRIORITY	The priority of the task, divided by 10. It is a two-digit number on A Series.
MEMORY-PRIORITY	Zero. Does not exist on A Series.
SPECIAL-PROGRAM-CODE	A value that indicates the type of program. The value is mapped from internal ClearPath and A Series mix attributes. The ClearPath and A Series mapping for V Series program codes is shown in Table H–7. The mapping does not have direct equivalence.
PROGRAM-STATUS-CODE	A value that indicates the status of the program. This value is usually 0 (zero). If the task is determined to be a compiler, this value is 01.

The values of the SPECIAL-PROGRAM-CODE field for the MIXTBL5 procedure are described in Table H–7.

Table H–7. Values of the SPECIAL-PROGRAM-CODE Field for the MIXTBL5 Procedure

A Series Type	V Series SPECIAL-PROGRAM-CODE	Which means that the program...
0 (Unknown)	0 (Undefined)	
1 (MCS)	6 (DCP or MCS)	Has MCS status.
2 (Library)	0	
3 (Database)	B (DMS control program)	Has a DMS control program.
4 (WFL job)	C (WFL handler)	Is a job.
5 (Compiler)	1 (Generator)	Is a generator.
6 (External)	0	
7 (Internal)	0	
8 (Segment dictionary)	0	
*SYSTEM/DUMPALL	2 (DMPALL)	Is DMPALL.
*LIBRARY/ MAINTENANCE	E (COPY)	Is COPY.

Details

Like MIXTBL, the MIXTBL5 function provides data in the same structure as in V Series. However, certain field values are different on the A Series. Some of these are:

- The PROGRAM-NAME form differs on A Series
  - It can begin with an asterisk (*).
  - It can begin with a parenthesized usercode.
  - If compiled under CANDE, the OBJECT/ follows the usercode.
  - It can consist of multiple levels.
- The MULTI-PROG-NAME field is used on V Series to distinguish between several running copies of the same code file. This function does not work on the A Series.
- The mix number is two or four digits on the V Series. Under this function it is five digits long.
- PRIORITY is a two-digit number on the A Series.
- MEMORY-PRIORITY does not exist on the A Series.
- SPECIAL-PROGRAM-CODE and PROGRAM-STATUS-CODE values do not map directly to A Series equivalents.

PROGINFO Procedure

ClearPath and A Series Equivalent:

ANSI intrinsic function WHEN-COMPILED and the ClearPath and A Series task attributes NAME, USERCODE, MIXNUMBER, and JOBNUMBER.

This procedure returns information about the program that initiates the procedure.

Note: *This procedure is intended for use only in V Series programs being migrated to ClearPath and A Series COBOL85.*

Syntax

```
CALL "PROGINFO IN EVASUPPORT BYFUNCTION" USING identifier-1
```

Explanation

identifier-1

This identifier must reference an alphanumeric data item of at least 26 bytes where the result from this procedure is returned.

Details

The result returned by this procedure has the following format:

```
02  ID1.
    03  PROGRAM-NAME                          PIC X(6).
    03  MULTI-PROG-NAME                       PIC X(6).
    03  MIX-NUMBER                            PIC 9(4) COMP.
    03  RLOG-NUMBER                           PIC 9(4) COMP.
    03  RJE-LINK                              PIC 9(4) COMP.
    03  PROGRAM-DATE-COMPILED                 PIC 9(6) COMP.
    03  PROGRAM-TIME-COMPILED                 PIC X(5).
```

Table H–8 shows the fields in the result structure and their values.

Table H–8. Values in PROGINFO Result Structure

The field identifier . . .	Returns the . . .
PROGRAM-NAME	First six characters of the low-order node of the task name.
MULTI-PROG-NAME	First six characters (excluding parentheses) of the USERCODE under which the task is running.
MIX-NUMBER	Mix number of the task in decimal.
RLOG-NUMBER	Mix number of the task. (Run log numbers do not exist on ClearPath and A Series systems.)
RJE-LINK	JOBNUMBER of the task. (The RJE link does not exist on ClearPath and A Series systems.)
PROGRAM-DATE-COMPILED	Date expressed as mmddyy converted from the CREATIONDATE attribute of the code file of the task.
PROGRAM-TIME-COMPILED	Time expressed as hh:mm extracted from the CREATIONTIME attribute of the code file of the task.

This procedure builds a file title by using the NAME task attribute of the calling program. It then interrogates the CREATIONDATE and the CREATIONTIME file attributes of that file.

By default, the NAME task attribute is the same as the file title. To use this procedure, a program must not change the NAME task attribute. It must be the same as the code file title.

Because the procedure must interrogate file attributes, the code file must not have SECURITYUSE=SECURED, or be guarded. If either of these conditions exists, a run-time security violation can result. If the code file must be secured or guarded, use the ANSI intrinsic WHEN-COMPILED, described in Section 9, and the ClearPath and A Series task attributes listed at the beginning of the discussion of this procedure.

PROGINFO5 Procedure

ClearPath and A Series Equivalent: ANSI intrinsic function WHEN-COMPILED and the ClearPath and A Series task attributes NAME, USERCODE, MIXNUMBER, and JOBNUMBER.

This procedure returns information about the program that initiates the procedure. The mix number returned with this information is five digits long.

Note: *This procedure is intended for use only in V Series programs being migrated to ClearPath and A Series COBOL85.*

Syntax

```
CALL "PROGINFO5 IN EVASUPPORT BYFUNCTION" USING identifier-1
```

Explanation

identifier-1

This identifier must reference an alphanumeric data item of at least 29 bytes where the result from this procedure is returned.

Details

The result returned by this procedure has the following format:

```
02  ID1.
    03  PROGRAM-NAME                          PIC X(6).
    03  MULTI-PROG-NAME                       PIC X(6).
    03  MIX-NUMBER                            PIC 9(5) COMP.
    03  RLOG-NUMBER                           PIC 9(5) COMP.
    03  RJE-LINK                              PIC 9(5) COMP.
    03  DATE-COMPILED                         PIC 9(6) COMP.
    03  TIME-COMPILED                         PIC X(5).
```

Table H–9 shows the fields in the result structure and their values.

Table H–9. Values in PROGINFO5 Result Structure

The field identifier . . .	Returns the . . .
PROGRAM-NAME	First six characters of the low-order node of the task name.
MULTI-PROG-NAME	First six characters (excluding parentheses) of the USERCODE under which the task is running.
MIX-NUMBER	The five-digit mix number of the task in base ten.
RLOG-NUMBER	The five-digit mix number of the task (run log numbers do not exist on ClearPath and A Series systems).
RJE-LINK	The five-digit JOBNUMBER of the task (the RJE link does not exist on ClearPath and A Series systems).
DATE-COMPILED	Date expressed as mmddyy converted from the CREATIONDATE attribute of the code file of the task.
TIME-COMPILED	Time expressed as hh:mm extracted from the CREATIONTIME attribute of the code file of the task.

This procedure builds a file title by using the NAME task attribute of the calling program. It then interrogates the CREATIONDATE and the CREATIONTIME file attributes of that file.

By default, the NAME task attribute is the same as the file title. To use this procedure, a program must not change the NAME task attribute. It must be the same as the code file title.

Because the procedure must interrogate file attributes, the code file must not have SECURITYUSE=SECURED or be guarded. If either of these conditions exists, a run-time security violation can result. If the code file must be secured or guarded, use the ANSI intrinsic WHEN-COMPILED, described in Section 9, and the ClearPath and A Series task attributes listed at the beginning of the discussion of this procedure.

Transition Information

Where PROGINFO5 is used, observe the following precautions:

- Avoid use of the run log number and the RJE link fields.
- Do not use the multi-program name field on A Series. Instead, retain the 6-character V Series program name.

A V Series program that obtains its own name from PROGINFO5 and then passes that name to MIXID or MIXNUM to determine whether copies of itself are running, will not run correctly if its A Series name is longer than six characters. The reason is that the response from the A Series version of PROGINFO5 is formatted identically to that of the V Series version. The sizes of the program name and multiprogram name fields permit only six characters of program names and usercode to be returned. There is no provision for family names.

To perform the equivalent inquiry on the A Series, the program should include the following code, instead of a call to PROGINFO5:

```
03  MY-NAME-AREA                    PIC X(nnn).
    .
    .
MOVE ATTRIBUTE NAME OF MYSELF TO MY-NAME-AREA.
<Response is in Title form: (usercode)name ON family.>
<Add code to extract usercode, name, and family name>
```

SETSWITCH Procedure

ClearPath and A Series Equivalent: SW1 through SW8 task attributes

The SETSWITCH procedure places the V Series switch settings passed by the program into the global array that contains the switch values. (The global array is established by the EVA_TASKSTRING procedure, described earlier in this section.) The setting established by the SETSWITCH procedure overrides any setting generated by the EVA_TASKSTRING procedure.

Note: *This procedure is intended for use only in V Series programs being migrated to ClearPath and A Series COBOL85.*

Syntax

```
CALL "SETSWITCH IN EVASUPPORT BYFUNCTION" USING identifier-1
```

Explanation

identifier-1

This identifier references an 8-digit COMPUTATIONAL numeric data item, described as PIC 9(8) COMP VALUE ZERO, from which the switch settings are to be retrieved.

Details

The COBOL conversion filter changes calls to the V Series SETSWITCH intrinsic to the following ClearPath and A Series CALL statement:

```
CALL "SETSWITCH IN EVASUPPORT BYFUNCTION" USING SWITCHES.
```

If you do not use the COBOL conversion filter, you must manually make this change.

SPOMESSAGE Procedure

ClearPath and A Series Equivalent: CALL SYSTEM WFL statement

The SPOMESSAGE procedure accepts and processes these types of input messages:

- ClearPath and A Series system commands
- ClearPath and A Series WFL statements
- V Series syntax for the EX, CH, and RM commands

Only the V Series commands inherit the usercode-related attributes of the calling task. To enable ClearPath and A Series WFL jobs to inherit the usercode-related attributes of the calling tasks, use the CALL SYSTEM WFL statement (see Section 6.)

The SPOMESSAGE procedure returns the result of the command to the specified data item.

This procedure is intended for use only in V Series programs being migrated to ClearPath and A Series COBOL85.

Syntax

```
CALL "SPOMESSAGE IN EVASUPPORT BYFUNCTION" USING identifier-1, identifier-2
```

Explanation

identifier-1

This identifier references a 200-character alphanumeric data item where the input ODT command submitted by the calling program is stored. If the input data item is less than 200 characters, the data item is padded on the right with spaces.

identifier-2

This identifier references an alphanumeric data item in which the response from identifier-1 is returned. The response is stored exactly as it would appear on the ODT (including the removal of extraneous blanks, if applicable). Each line ends with a carriage return character; the last line ends with carriage return and ETX characters. The size of the data item cannot exceed the maximum size for an 01–level item. Output responses larger than the size of the data item are truncated and are indicated by the last line ending with carriage return and NULL characters.

Details

The COBOL conversion filter changes calls to the V Series SPOMESSAGE intrinsic to the following ClearPath and A Series CALL statement:

```
CALL "SPOMESSAGE IN EVASUPPORT BYFUNCTION" USING identifier-1,
identifier-2.
```

If you do not use the COBOL conversion filter, you must manually make this change.

TIMENOW Procedure

ClearPath and A Series Equivalent: ANSI intrinsic function CURRENT-DATE, or the ACCEPT statement with the DATE, TODAYS-DATE, and TIME options

The TIMENOW procedure obtains the symbolic representation of the current time in the form *HH:MM XXXX*.

The letters . . .	Represent the . . .
HH	Hour
MM	Minutes
XXXX	Value A.M. or P.M.

Note: *This procedure is intended for use only in V Series programs being migrated to ClearPath and A Series COBOL85.*

Syntax

```
CALL "TIMENOW IN EVASUPPORT BYFUNCTION" USING identifier-1
```

identifier-1

This identifier references an elementary alphanumeric data item of at least 10 characters in which the result from this procedure is stored.

UNIQUENAME Procedure

ClearPath and A Series Equivalent: UNIQUETOKEN file attribute

This procedure appends the mix number of the calling program to the file name as an additional node. This practice helps to ensure that work files created by multiple, simultaneously running copies of the same program have unique names. To obtain similar functionality in new COBOL85 code, use the UNIQUETOKEN file attribute.

Note: *This procedure is intended for use only in V Series programs being migrated to ClearPath and A Series COBOL85.*

Syntax

```
CALL "UNIQUENAME IN EVASUPPORT BYFUNCTION" USING identifier-1
```

Explanation

identifier-1

This identifier references the data item that contains the name specified in both the SELECT statement and the file description (FD) entry of the file to which you want the mix number to be appended.

Details

This procedure must be called before the file is opened.

Each time it is called, this procedure appends a mix number node to the file name. It does not attempt to determine whether a mix number is already appended. In addition, this procedure does not create an attribute that remains with the file. If you change the name of the file without preserving the mix number node or without calling the UNIQUENAME procedure again, the mix number node does not appear with the file name.

The length of the mix number in the appended node is four or more characters. If the actual length of the mix number is less than four characters, leading zeros are added.

You must ensure that the file name with the mix number node appended does not exceed the maximum allowable length for file names or nodes. If the operating system encounters an invalid file name, the UNIQUENAME procedure keeps the original file name.

VDISKFILEHEADER Procedure

ClearPath and A Series Equivalent: See Table H–10

The VDISKFILEHEADER procedure returns file information in the format of a V Series disk file header. This procedure obtains its information from file attributes.

This procedure is intended for use only in V Series programs being migrated to ClearPath and A Series COBOL85.

Caution

Because ClearPath and A Series file attributes have a larger range of values than V Series file attributes, some of the returned values might be truncated. Thus, the use of this routine is discouraged.

Syntax

```
CALL "VDISKFILEHEADER IN EVASUPPORT BYFUNCTION" USING identifier-1, identifier-2,
      identifier-3
```

Explanation

identifier-1

This identifier specifies the name of the file whose disk file header information is to be accessed. The file must be under your usercode.

identifier-2

This identifier specifies the name of the family where the file resides. You must specify the family name if the family is not your current default family. Otherwise, this field can contain spaces or nulls.

identifier-3

This identifier references a 40-digit COMPUTATIONAL result field whose data items are redefined during the filtering process to match equivalent ClearPath and A Series file attributes as shown in Table H–10. If you do not use the COBOL conversion filter, you must manually make these changes. The result field is shown under "Details."

Details

Before filtering, the result field represented by identifier-3 has the following format:

```
02  DFH-RESULT                      PIC 9(40) COMP.
02  DFH-RESULT-FIELD REDEFINES DFH-RESULT.
          03 REC-SIZE-IN-DIGITS      PIC 9(5)   COMP.
          03 RECORDS-PER-BLK         PIC 9(3)   COMP.
          03 NUMBER-OF-AREAS         PIC 9(2)   COMP.
          03 END-OF-FILE-PTR         PIC 9(8)   COMP.
          03 NUM-OF-USERS-ON-1       PIC 9(2)   COMP.
          03 NUM-OF-USERS-ON-2       PIC 9(2)   COMP.
          03 NUM-OF-USERS-ON-3       PIC 9(2)   COMP.
          03 NUM-OF-USERS-ON-4       PIC 9(2)   COMP.
          03 RESERVED                PIC 9(5)   COMP.
          03 FILE-TYPE-1             PIC 9      COMP.
          03 FILE-TYPE-2             PIC 9      COMP.
          03 SECTORS-PER-AREA        PIC 9(7)   COMP.
```

Table H–10 maps these data items to their equivalent ClearPath and A Series file attributes.

Table H–10. ClearPath and A Series File Attributes for VDISKFILEHEADER Fields

V Series Data Item	ClearPath and A Series File Attribute	Response Picture	Details
REC-SIZE-IN-DIGITS	MAXRECSIZE	PIC 9(5) COMP	This attribute is expressed in 4-bit digits, truncated to 5 digits.
RECORDS-PER-BLK	BLOCKSIZE	PIC 9(3) COMP	The value for this field is determined by the BLOCKSIZE divided by the MAXRECSIZE.
NUMBER-OF-AREAS	AREAS	PIC 9(2) COMP	Values of 100 or greater are shown as 00.
END-OF-FILE-PTR	LASTRECORD	PIC 9(8) COMP	The value for this field is determined by adding 1 to the LASTRECORD value. (This calculation produces a 1-relative result.)
NUM-OF-USERS-ON-0	POPULATION	PIC 9(2) COMP	Returns the number of users of the file on the host, in the range 0 through 99.
NUM-OF-USERS-ON-1 NUM-OF-USERS-ON-2 NUM-OF-USERS-ON-3	None	PIC 9(2) COMP	A zero is returned in these fields.
RESERVED		PIC 9(5) COMP	A null value is returned in this field.
FILE-TYPE-1 FILE-TYPE-2		PIC 9 COMP	A null value is returned in these fields.
SECTORS-PER-AREA	AREASECTORS	PIC 9(7) COMP	Values greater than 9999999 are shown as 9999999.

VREADTIMER Procedure

ClearPath and A Series Equivalent: ACCEPT statement with the TIMER option for the microsecond timer and the CURRENT-DATE function for the date

This procedure obtains the symbolic representation of the current, year, month, day, and time in the form YYYYMMDDFmmmmmmmmmmm.

The letters . . .	Represent the . . .
YYYY	Year.
MM	Month.
DD	Day.
F	Filler.
mmmmmmmmmmm	Time in 2.4 microsecond intervals.

Note: *This procedure is intended for use only in V Series programs being migrated to ClearPath and A Series COBOL85.*

Syntax

```
CALL "VREADTIMER IN EVASUPPORT BYFUNCTION" USING identifier-1
```

Explanation

identifier-1

This identifier references the data item where the result is to be stored. The data item must be a 20-digit COMPUTATIONAL numeric data item (PIC 9(20) COMP). If the data item exceeds 20 digits, the result is returned in the leftmost 20 digits.

Details

The data item where the result is to be stored must be redefined to access the individual fields as follows:

```
02  VREADTIMER-RESULT                                PIC 9(20) COMP.
02  VREADT-FIELDS      REDEFINES VREADTIMER-RESULT.
        03  VRT-YEAR                                 PIC 9(04)  COMP.
        03  VRT-MONTH                                PIC 9(02)  COMP.
        03  VRT-DAY                                  PIC 9(02)  COMP.
        03  FILLER                                   PIC 9(01)  COMP.
        03  VRT-TIMER                                PIC 9(11)  COMP.
```

Note: *Because the ClearPath and A Series timer result is different from the V Series timer result, the VREADTIMER procedure must perform multiple operations to prepare its response. If your purpose in using the VREADTIMER procedure is to have the fastest access to the system timer, you could probably save time by using the ACCEPT FROM TIMER statement instead.*

VTRANSLATE Procedure

ClearPath and A Series Equivalent: None

The VTRANSLATE procedure translates a string of digits or characters from V Series format to ClearPath and A Series format and stores the translated string in the specified location. Each time a VTRANSLATE procedure is called, it copies the V Series translation table to the appropriate ClearPath and A Series format. The converted table is not saved between calls.

You can use this procedure in new ClearPath and A Series code.

This procedure has six formats, which are described in the following table:

Format	Use
Format 1	Translates an 8-bit DISPLAY source to an 8-bit DISPLAY destination.
Format 2	Translates an 8-bit DISPLAY source to a 4-bit COMP destination.
Format 3	Translates a 4-bit COMP source to a 4-bit COMP destination.
Format 4	Translates a 4-bit COMP source to an 8-bit DISPLAY destination.
Format 5	Translates a 4-bit signed numeric source to a 4-bit COMP destination. The leading digit in the source (the sign) is bypassed. Thus, the output field begins with the translation of the first character following the sign.
Format 6	Translates a 4-bit signed numeric source to an 8-bit DISPLAY destination. The leading digit in the source (the sign) is bypassed. Thus, the output field begins with the translation of the first character following the sign.

Format 1: Translate DISPLAY Source to DISPLAY Destination

This format of the VTRANSLATE procedure translates an 8-bit DISPLAY source to an 8-bit DISPLAY destination using a 64-word table.

Syntax

```
CALL "VTRANSLATEDISPDISP IN EVASUPPORT BYFUNCTION"
   USING identifier-1, identifier-2, identifier-3
```

Explanation

identifier-1

This identifier references the data item to be translated. It can contain a maximum of 10,000 digits or characters. If identifier-1 has an operational sign, the sign is ignored; translation begins with the first character following the sign.

identifier-2

This identifier specifies the name of a standard, 390-byte V Series translation table.

identifier-3

This identifier references the data item in which the translated data is to be stored. The data item must be of the same size or larger than the data item referenced by identifier-1 and must be one of the following types:

- Alphanumeric
- Numeric without editing PICTURE symbols
- Unsigned numeric COMPUTATIONAL

Format 2: Translate DISPLAY Source to COMP Destination

This format of the VTRANSLATE procedure translates an 8-bit DISPLAY source to a 4-bit COMP destination.

Syntax

```
CALL "VTRANSLATEDISPCOMP IN EVASUPPORT BYFUNCTION"
   USING identifier-1, identifier-2, identifier-3
```

For information about the identifiers in this syntax, refer to the explanation of Format 1.

Format 3: Translate COMP Source to COMP Destination

This format of the VTRANSLATE procedure translates a 4-bit COMP source to a 4-bit COMP destination.

Syntax

```
CALL "VTRANSLATECOMPCOMP IN EVASUPPORT BYFUNCTION"
   USING identifier-1, identifier-2, identifier-3
```

For information about the identifiers in this syntax, refer to the explanation of Format 1.

Format 4: Translate COMP Source to DISPLAY Destination

This format of the VTRANSLATE procedure translates a 4-bit COMP source to an 8-bit DISPLAY destination.

Syntax

```
CALL "VTRANSLATECOMPDISP IN EVASUPPORT BYFUNCTION"
   USING identifier-1, identifier-2, identifier-3
```

For information about the identifiers in this syntax, refer to the explanation of Format 1.

Format 5: Translate Signed Numeric Source to COMP Destination

This format of the VTRANSLATE procedure translates a 4-bit signed numeric source to a 4-bit COMP destination.

Syntax

```
CALL "VTRANSLATESNCOMP IN EVASUPPORT BYFUNCTION"
   USING  identifier-1, identifier-2, identifier-3
```

For information about the identifiers in this syntax, refer to the explanation of Format 1.

Format 6: Translate Signed Numeric Source to DISPLAY Destination

This format of the VTRANSLATE procedure translates a 4-bit signed numeric source to an 8-bit DISPLAY destination.

Syntax

```
CALL "VTRANSLATESNDISP IN EVASUPPORT BYFUNCTION"
   USING identifier-1, identifier-2, identifier-3
```

For information about the identifiers in this syntax, refer to the explanation of Format 1.

Example

The following example demonstrates the use of various formats of the VTRANSLATE routine.

```
IDENTIFICATION DIVISION.
  PROGRAM-ID.  TRANSALTEDEMO.
* Demonstrate the use of the VTRANSLATE routines.
ENVIRONMENT DIVISION.
  INPUT-OUTPUT SECTION.
    FILE-CONTROL.
     SELECT REM
         ASSIGN TO REMOTE.
DATA DIVISION.
  FILE SECTION.
  FD REM
     VALUE OF MAXRECSIZE IS 80.
     VALUE OF FILEUSE IS IO.
  01 REM-REC.
     05  P-REC-PT1         PIC X(35).
     05  P-REC-SEPARATOR   PIC X(35).
     05  P-REC-PT2         PIC X(40).

  WORKING-STORAGE SECTION.
    77  LOWER-CASE-TEXT      PIC X(15)  VALUE "WORDS & LETTERS".
    77  DEC-NUMBER           PIC 9(12)  VALUE 16500000.
    77  HEX-NUMBER           PIC 9(12)  VALUE ZERO  COMP.
```

```
*****************************************************************
* The following pattern is repeated 8 times to define the       *
* translated result for each of the 256 characters in the       *
* collating sequence:                                           *
*     ccccfccccfccccfccccfccccfccccfccccf fffffffff             *
* In this sequence, "c" represents a character, and "f"         *
* represents a filler byte.                                     *
*****************************************************************

  01  VTRN-LOWER-TO-UPPER            COMP.

*****************************************************************
* This table leaves all the characters in the collating         *
* sequence, except for the lowercase letters, unchanged.        *
* The table performs this function by defining the original     *
* character at that character's position in the table and       *
* inserting uppercase letters in the lowercase letter positions.*
*****************************************************************

        03  FILLER     PIC 9(20) VALUE @000102030004050607OO@
        03  FILLER     PIC 9(20) VALUE @08090A0B000C0D0E0F00@
        03  FILLER     PIC 9(20) VALUE @1011121300141516170O@
        03  FILLER     PIC 9(20) VALUE @18191A1B001C1D1E1F00@
        03  FILLER     PIC 9(20) VALUE ZEROS.
        03  FILLER     PIC 9(20) VALUE @2021222300242526270O@
        03  FILLER     PIC 9(20) VALUE @28292A2B002C2D2E2F00@
        03  FILLER     PIC 9(20) VALUE @3031323300343536370O@
        03  FILLER     PIC 9(20) VALUE @38393A3B003C3D3E3F00@
        03  FILLER     PIC 9(20) VALUE ZEROS.
        03  FILLER     PIC 9(20) VALUE @4041424300444546470O@
        03  FILLER     PIC 9(20) VALUE @48494A4B004C4D4E4F00@
        03  FILLER     PIC 9(20) VALUE @5051525300545556570O@
        03  FILLER     PIC 9(20) VALUE @58595A5B005C5D5E5F00@
        03  FILLER     PIC 9(20) VALUE ZEROS.
        03  FILLER     PIC 9(20) VALUE @6061626300646566670O@
        03  FILLER     PIC 9(20) VALUE @68696A6B006C6D6E6F00@
        03  FILLER     PIC 9(20) VALUE @7071727300747576770O@
        03  FILLER     PIC 9(20) VALUE @78797A7B007C7D7E7F00@
        03  FILLER     PIC 9(20) VALUE ZEROS.
```

```
*****************************************************************
* At this point, replace the lowercase letters (EBCDIC codes 81 *
* through 89, 91 through 99, and A2 through A9) with the        *
* uppercase letters (EBCDIC codes C1 through C9, D1 through D9, *
* E2 through E9).                                               *
*****************************************************************

        03  FILLER     PIC 9(20) VALUE @80C1C2C300C4C5C6C700@
        03  FILLER     PIC 9(20) VALUE @C8C98A8B008C8D8E8F00@
        03  FILLER     PIC 9(20) VALUE @90D1D2D300D4D5D6D700@
        03  FILLER     PIC 9(20) VALUE @D8D99A9B009C9D9E9F00@
        03  FILLER     PIC 9(20) VALUE ZEROS.
        03  FILLER     PIC 9(20) VALUE @A0A1E2E300E4E5E6E700@
        03  FILLER     PIC 9(20) VALUE @E8E9AAAB00ACADAEAF00@
* End of lowercase letters.
        03  FILLER     PIC 9(20) VALUE @B0B1B2B300B4B5B6B700@
        03  FILLER     PIC 9(20) VALUE @B8B9BABB00BCBDBEBF00@
        03  FILLER     PIC 9(20) VALUE ZEROS.
        03  FILLER     PIC 9(20) VALUE @C0C1C2C300C4C5C6C700@
        03  FILLER     PIC 9(20) VALUE @C8C9CACB00CCCDCECF00@
        03  FILLER     PIC 9(20) VALUE @D0D1D2D300D4D5D6D700@
        03  FILLER     PIC 9(20) VALUE @D8D9DADB00DCDDDEDF00@
        03  FILLER     PIC 9(20) VALUE ZEROS.
        03  FILLER     PIC 9(20) VALUE @E0E1E2E300E4E5E6E700@
        03  FILLER     PIC 9(20) VALUE @E8E9EAEB00ECEDEEEF00@
        03  FILLER     PIC 9(20) VALUE @F0F1F2F300F4F5F6F700@
        03  FILLER     PIC 9(20) VALUE @F8F9FAFB00FCFDFEFF00@
        03  FILLER     PIC 9(20) VALUE ZEROS.
  01  VTRN-HEX-TO-EBCDIC.
```

```
*****************************************************************
* At this point, the only input characters are the hexadecimal  *
* digits 0 through F, which are to be translated to the         *
* displayable characters 1 through 9, and A through F. The      *
* V Series treats these hex digits as though they contain the   *
* zone digit F. Thus, the translation table disregards all      *
* values preceding F0 in the collating sequence and defines F0  *
* through F9 and C1 through C6 as their equivalents.            *
*****************************************************************

        03  FILLER     PIC 9(20) VALUE @F0F1F2F300F4F5F6F700@
        03  FILLER     PIC 9(20) VALUE @F8F9C1C200C3C4C5C600@
*
PROCEDURE DIVISION.
      FIRST-PARA.
        OPEN I-O REM.
        MOVE SPACES TO REM-REC.
*
        MOVE LOWER-CASE-TEXT    TO P-REC-PT1.
*
        CALL "VTRANSLATEDISPDISP IN EVASUPPORT BYFUNCTION"
          USING LOWER-CASE-TEXT, VTRN-LOWER-TO-UPPER, P-REC-PT2.
        MOVE " --> "    TO P-REC-SEPARATOR.
        PERFORM DISPLAY-IT.
*
        MOVE DEC-NUMBER TO P-REC-PT1.
        CALL "DECIMALDISPBINARY IN EVASUPPORT BYFUNCTION"
          USING DEC-NUMBER, HEX-NUMBER.
        CALL "VTRANSLATECOMPDISP IN EVASUPPORT BYFUNCTION"
          USING HEX-NUMBER, VTRN-HEX-TO-EBCDIC, P-REC-PT2.
        MOVE " --> "    TO P-REC-SEPARATOR.
        PERFORM DISPLAY-IT.
*
        STOP RUN.
*
*
DISPLAY IT.
        WRITE REM-REC.
        MOVE SPACES TO REM-REC.
```

The execution of this program yields the following:

```
#RUNNING 3880
#?
Words & Letters                 --> WORDS & LETTERS
000016500000                    --> 000000FBC520
#ET=0.2 PT=0.1 IO=0.0
```

ZIP Procedure

ClearPath and A Series Equivalent: CALL SYSTEM WFL statement

The ZIP procedure accepts and processes the following types of input messages:

- ClearPath and A Series system commands
- ClearPath and A Series WFL statements
- V Series syntax for the EX, CH, and RM commands

Only the V Series commands inherit the usercode-related attributes of the calling task. To enable ClearPath and A Series WFL jobs to inherit the usercode-related attributes of the calling tasks, use the CALL SYSTEM WFL statement (see Section 6.)

The ZIP procedure does not return results to the calling program, as does the SPOMESSAGE procedure described earlier in this section.

Note: *This procedure is intended for use only in V Series programs being migrated to ClearPath and A Series COBOL85.*

Syntax

```
CALL "ZIP IN EVASUPPORT BYFUNCTION" USING identifier-1
```

Explanation

identifier-1

This identifier references a 200-character alphanumeric data item that contains the input ODT command. If the size of the input data is smaller than the data item, the data item is padded to the right with spaces.

Details

The COBOL conversion filter changes the call to the V Series ZIP intrinsic to the following ClearPath and A Series CALL statement:

```
CALL "ZIP IN EVASUPPORT BYFUNCTION" USING USERIN.
```

If you do not use the COBOL conversion filter, you must manually make this change.

ZIPSPO Procedure

ClearPath and A Series Equivalent: CALL SYSTEM WFL statement

The ZIPSPO procedure accepts and processes the following types of input messages:

- ClearPath and A Series system commands
- ClearPath and A Series WFL statements
- V Series syntax for the EX, CH, and RM commands

Only the V Series commands inherit the usercode-related attributes of the calling task. To enable ClearPath and A Series WFL jobs to inherit the usercode-related attributes of the calling tasks, use the CALL SYSTEM WFL statement (see Section 6.)

The ZIPSPO procedure returns information to the calling program only if the input command is COMPILE or RUN, and a task is started. In these cases, the procedure returns the task number of the CONTROLCARD procedure that processed the task to the calling program.

Note: *This procedure is intended for use only in V Series programs being migrated to ClearPath and A Series COBOL85.*

Syntax

```
CALL "ZIPSPO IN EVASUPPORT BYFUNCTION" USING identifier-1, identifier-2
```

Explanation

identifier-1

This identifier references a 200-character alphanumeric data item that stores the input message submitted by the calling program.

identifier-2

This identifier references a group data item that consists of two numeric elementary data items. For an example of the structure of this data item, refer to "Details."

Details

The COBOL conversion filter changes calls to the V Series ZIPSPO intrinsic to the following ClearPath and A Series CALL statement:

```
CALL "ZIPSPO IN EVASUPPORT BYFUNCTION" USING USERIN, USEROUT.
```

If you do not use the COBOL conversion filter, you must manually make this change.

Appendix I
Tips and Techniques

This appendix is intended to help you write better COBOL applications by taking advantage of COBOL85. This appendix consists of the following major topics:

- Improving Performance of COBOL85 Programs
- Using Key Features of COBOL85

Improving Performance of COBOL85 Programs

Which COBOL compiler performs better, COBOL74 or COBOL85? The answer, in general, is that the COBOL85 compiler tends to be more processor intensive than COBOL74 and takes more elapsed time to compile a program. At run time, however, the COBOL85 program is usually faster in executing COBOL verbs such as PERFORM, ADD, and MOVE. The I/O performance of a COBOL85 program is nearly identical to that of a COBOL74 program.

The following topics provide tips and techniques that help you improve the performance and efficiency of your COBOL85 programs.

Distinguishing CALL Statements

The COBOL85 compiler cannot distinguish between calls to a nested program and calls to an external program, such as a library entry point. This is because the COBOL85 call for nested procedures, CALL "<procedure name>", has similar syntax to the COBOL74 call for libraries, CALL "<library object title>". The compiler requires additional run-time code to distinguish between these calls.

You can improve CALL performance and the speed of your program by implementing the following changes for nested calls and library calls.

Nested Calls

To distinguish nested calls to the COBOL85 compiler, set the $CALLNESTED statement. When set, the compiler assumes that the specified nested call is internal.

Example

```
$SET CALLNESTED
   CALL "<nested program id>".
$RESET CALLNESTED
```

Library Calls

To distinguish library calls to the COBOL85 compiler, replace COBOL74 style library calls

```
CALL "<library object title>".
```

with the following COBOL85 style library call:

```
CALL "PROCEDUREDIVISION OF <library object title>".
```

Reading STREAM Files Faster

The COBOL85 compiler reads STREAM files, also known as PC files, one character at a time. STREAM files typically have a MAXRECSIZE file attribute value of 1 and consist of data that is organized into records delimited by CR-LF (carriage return/linefeed) characters.

You can speed up the processing of a STREAM file by

- Setting the ANYSIZEIO file attribute value to TRUE
- Increasing the size of the record that you use to read the file

  Note: *When increasing the record description of the file in your program, use a size that is a multiple of the number of characters that you expect each CR-LF character to delimit. Be sure to include the CR-LF characters in the count; the program must scan the data for each delimiting CR-LF character so that it can process the data as meaningful records.*

These actions enable the compiler to read large amounts of the file rather than individual characters.

Input Process

A common input process for STREAM files follows:

1. Read a chunk of data.
2. UNSTRING it into complete records delimited by CR-LF.
3. Continue the process of unstringing and using the data until the data is exhausted.
4. Loop back to the statement where the program reads the next chunk of data.
5. Repeat this process until EOF terminates the READ process.

For a detailed discussion of the ANYSIZEIO attribute, refer to the *File Attributes Reference Manual*. For additional information about STREAM files, refer to the *I/O Subsystem Programming Guide*.

Example

The following example is a portion of a COBOL85 program that shows how to read a STREAM file and then display each record at the user's terminal:

```
00100 IDENTIFICATION DIVISION.
00200 PROGRAM-ID.
00300     P1.
00400 ENVIRONMENT DIVISION.
00500 INPUT-OUTPUT SECTION.
00600 FILE-CONTROL.
00700      SELECT VF ASSIGN TO DISK.
00800      SELECT REM ASSIGN TO REMOTE.
```

```
00900 DATA DIVISION.
10000 FILE SECTION.
11000 FD VF
12000      VALUE OF
14000               FILENAME      IS "SOME/STREAM/FILE"
14100               DEPENDENTSPECS IS TRUE
17000               EXTMODE       IS ASCII
18000               INTMODE       IS EBCDIC
19000               ANYSIZEIO     IS TRUE.
23000*              The record establishes the length of the READ.
24000 01 VF-REC     PIC X(3000).
25000 FD REM.
26000 01 REM-REC    PIC X(80).
27000 WORKING-STORAGE SECTION.
28000 01 REC-ARRAY.
29000    03 REC-DATA  PIC X(3000).
31000 01 REM-ARRAY.
32000    03 REM-DATA PIC X(80).
33600 77 UNSTRING-START REAL.
33800 77 SAVED-START    REAL.
34500 77 DUMMY          REAL.
35000 77 UNSTRING-CNT   REAL.
36000 77 MSG-LEN        REAL.
37000 77 REC-CNT        REAL.
39000 PROCEDURE DIVISION.
40000  MAIN-PARA.
41000    PERFORM OPEN-FILES.
42000    PERFORM READ-AND-DISPLAY-VF THRU READ-AND-DISPLAY-VF-EXIT.
43000    DISPLAY "RECORDS READ:" REC-CNT.
44000    PERFORM CLOSE-FILES.
048000    STOP RUN.
049000
050000  OPEN-FILES.
052000    OPEN INPUT VF.
054000    OPEN I-O REM.
056000
058000  CLOSE-FILES.
060000    CLOSE REM.
062000    CLOSE VF.
064000
066000  READ-AND-DISPLAY-VF.
067000    READ VF
068000         AT END GO TO READ-AND-DISPLAY-VF-EXIT.
071000    MOVE ATTRIBUTE CURRENTRECORDLENGTH OF VF TO MSG-LEN.
074000    PERFORM REMOTE-DISPLAY THRU REMOTE-DISPLAY-EXIT.
075000    GO TO READ-AND-DISPLAY-VF.
076000
077000  READ-AND-DISPLAY-VF-EXIT.
078000    EXIT.
079000
080000  REMOTE-DISPLAY.
082000    MOVE VF-REC TO REC-ARRAY.
```

```
082050    MOVE 1 TO UNSTRING-START.
083000    PERFORM WRITE-RECS-TO-REMOTE THRU
084000            WRITE-RECS-TO-REMOTE-EXIT UNTIL MSG-LEN <= 0.
084700
084800  REMOTE-DISPLAY-EXIT.
084900    EXIT.
085000
086000  WRITE-RECS-TO-REMOTE.
087000    MOVE 0 TO UNSTRING-CNT.
088500    MOVE UNSTRING-START TO SAVED-START.
089000    MOVE SPACES TO REM-ARRAY.
090000    UNSTRING REC-ARRAY
091000             DELIMITED BY ALL @0D25@
092000             INTO REM-ARRAY
092400             COUNT IN UNSTRING-CNT
093000             WITH POINTER UNSTRING-START.
093050*   UNSTRING-CNT characters have been moved to REM-ARRAY.
093060*   UNSTRING-START is next position in source after UNSTRING,
093080*   including the delimiter(s).
095000    WRITE REM-REC FROM REM-ARRAY.
095100    ADD 1 TO REC-CNT.
095300    SUBTRACT SAVED-START FROM UNSTRING-START GIVING UNSTRING-CNT.
095400*   UNSTRING-CNT now includes the delimiting CRLF in the input
095500*   string.
096000    SUBTRACT UNSTRING-CNT FROM MSG-LEN.
099000
100000  WRITE-RECS-TO-REMOTE-EXIT.
101000    EXIT.
102000
```

Generating Temporary Arrays with the $LOCALTEMP Option

The $LOCALTEMP option specifies whether the compiler generates local or global temporary arrays. Each type of array provides you with performance advantages.

Local temporary arrays

Setting $LOCALTEMP to the default value TRUE causes temporary arrays to be allocated locally within a program. You must include this option in the source code before the Identification Division. Allocating temporary arrays locally

- Optimizes subprogram memory usage
- Reduces the use of lex level 2 stack cells

Setting $LOCALTEMP to TRUE for shared-by-all libraries prevents the data corruption that can result from multiple users accessing the same temporary data.

Note: *Avoid using local temporary arrays in subprograms that are entered frequently. Each time a program containing a local array is entered and the array is first used, the system performs a p-bit interrupt.*

Global temporary arrays

Resetting $LOCALTEMP to FALSE causes temporary arrays to be allocated globally at lex level 2. Allocating temporary arrays globally can speed up the performance of a program.

Using the $LOCALTEMPWARN option

The $LOCALTEMPWARN option helps you determine when to set the $LOCALTEMP option.

Setting $LOCALTEMPWARN to TRUE enables the compiler to emit the following warning when a statement causes the compiler to generate a local temporary array:

```
A LOCAL TEMPORARY ARRAY HAS BEEN GENERATED FOR THIS STATEMENT, WHICH MAY
CAUSE AN INITIAL PBIT TO OCCUR. TO AVOID A PERFORMANCE PROBLEM CAUSED BY THE
PBIT, RESET THE LOCALTEMP CCI OR MODIFY THE STATEMENT.
```

Diagnosing Performance with the $STATISTICS Option

The $STATISTICS option indicates where a program spends most of its run time, enabling you to identify the areas where performance can be improved.

You can specify $STATISTICS to obtain the following information about the program:

- The number of times each paragraph was entered
- The CPU time spent within each paragraph

You can specify the following forms of $STATISTICS for additional information:

- $STATISTICS(PBITS)

  Includes the number of initial pbits within each paragraph.
- $STATISTICS(TERSE)

  Omits the paragraphs that were not used.
- $STATISTICS(SYSTEM)

  Includes the procedures called in SLICESUPPORT and the MCP.

Example

The following display shows sample output from the $STATISTICS option.

```
Processor Time:  Times are shown in seconds     Date=10242002  Time=141324

Sequence Name                  Count           Routine Time
                                                 Total     Per Trip
    1900 START-P1                  1          0.010786    0.010786
    2400 NOT-USED-LABEL
    2600 FIRST-LABEL            1000          1.559390    0.001559
    2900 SECOND-LABEL            100          0.841349    0.008413
    4000 ALMOST-DONE               1          0.000096    0.000096

                                            Routine Time
                                              Total
Total for entire program:                    2.411621

                             Top 90 Percent Sorted by Processor Time

Sequence Name                  Count           Routine Time  Cumulative
                                                 Total   Percent    Percent
    2600 FIRST-LABEL            1000          1.559390  64.66 %    64.66 %
    2900 SECOND-LABEL            100          0.841349  34.89 %    99.55 %
```

Using Multiple Versions of COBOL85 on One Server

Multiple versions of COBOL85 can exist on one server, with or without the Test and Debugging System (TADS) component. This capability enables you to test software on one level while using a previous level for other software.

By default, the *SYSTEM/COBOL85 compiler uses the following library:

```
COMPILER LIBRARY SUPPORT (FUNCTIONNAME=SLICESUPPORT);
```

Also, by default, programs produced by the COBOL85 compiler use the following libraries:

```
LIBRARY SUPPORT (FUNCTIONNAME=SLICESUPPORT);
LIBRARY C85_LIB_SUPPORT (FUNCTIONNAME=COBOL85SUPPORT);
LIBRARY SLICETADS (FUNCTIONNAME=COBOL85TADS);
```

Multiple Versions of COBOL85 without TADS

In most cases, it is recommended to use the most current versions of SLICESUPPORT and COBOL85SUPPORT. For example,

```
SL SLICESUPPORT = *SYSTEM/SLICESUPPORT/491 ON DISK
SL COBOL85SUPPORT = *SYSTEM/COBOL85SUPPORT/491 ON DISK
```

To use the same versions of the support libraries as the COBOL85 compiler, perform the following instructions:

1. Define two new support libraries with the SL (Support Library) system command. For example,

   ```
   SL SLICESUPPORT451 = *SYSTEM/SLICESUPPORT/451 ON DISK
   SL COBOL85SUPPORT451 = *SYSTEM/COBOL85SUPPORT/451 ON DISK
   ```

2. When compiling programs with the COBOL85 compiler, add the following commands to the WFL compile statement:

   ```
   COMPILE <file> WITH COBOL85/451 ON DISK;
   COMPILER LIBRARY SUPPORT (FUNCTIONNAME=SLICESUPPORT451);
   LIBRARY SUPPORT (FUNCTIONNAME=SLICESUPPORT451);
   LIBRARY C85_LIB_SUPPORT (FUNCTIONNAME=COBOL85SUPPORT451);
   ```

   The last two library equations modify the run-time library attributes for the user program.

Multiple Versions of COBOL85 with TADS

Complete the instructions for handling multiple versions of COBOL85 without TADS. Then perform the following instructions to use multiple versions of the COBOL85 compiler with TADS. You must use the same version of COBOL85 and COBOL85 TADS.

1. When using COBOL85 TADS, you can usually use the most current version of the SLICESUPPORT library. To use the same version of this library as the compiler, you must modify the COBOL85TADS library with the correct SLICESUPPORT library:

```
WFL MODIFY *SYSTEM/COBOL85TADS/451 ON DISK;
LIBRARY SUPPORT (FUNCTIONNAME=SLICESUPPORT451);
```

2. Using the MP (Mark Program) system command, reassign COMPILER status to the COBOL85TADS library:

```
MP *SYSTEM/COBOL85TADS/451 ON DISK + COMPILER
```

3. Using the SL (Support Library) system command, define a new library for the modified COBOL85TADS library:

```
SL COBOL85TADS451 = *SYSTEM/COBOL85TADS/451 ON DISK : TRUSTED
```

4. Run the user program as follows:

```
RUN; TADS; LIBRARY SLICETADS (FUNCTIONNAME=COBOL85TADS451);
```

Improving Reliability of Non-numeric Information in COMPUTATIONAL Fields

The USAGE COMPUTATIONAL field is typically used in calculations and consists of strictly numeric information, containing only decimal digits of 0 through 9.

Although some ClearPath MCP servers treat COMPUTATIONAL fields as essentially hexadecimal strings that are numeric only when the context requires, the COBOL74 and COBOL85 compilers presume that COMPUTATIONAL fields are numeric. Non-numeric information in COMPUTATIONAL fields can produce unpredictable results.

You can improve the reliability of non-numeric information in COMPUTATIONAL fields as follows.

Retrieving the Numeric Value of a COMPUTATIONAL Item

To retrieve the numeric value of a COMPUTATIONAL item, you must convert the value from packed-decimal to binary integer form. As long as the numeric portion of the packed-decimal field contains only numeric digits, the resulting binary value is reliable.

If any digit position in the COMPUTATIONAL field contains hexadecimal digits A through F, the value retrieved from that data item is unpredictable and can vary depending on

- The release level of the compiler producing the object code file
- The server for which the program was compiled
- The server on which the program is run and, possibly, the microcode level of that server

Moving Numeric-edited Data Items

You can move a numeric-edited data item to either a numeric or a numeric-edited data item. If you move a numeric-edited data item to a numeric data item, de-editing occurs. De-editing is the logical removal of all editing characters from a numeric-edited data item to determine the unedited numeric value of that item.

Maintaining Precision in Programs

DOUBLE and REAL data items represent values that the COBOL85 compiler must approximate. To maintain precision in your programs, be aware that when DOUBLE and REAL values are assigned to DISPLAY, COMP, or BINARY data items, the intended values might be changed.

Example

In the following example, A is declared as REAL and B is declared as PIC 9V999. Because A has the approximate value of 1.1199999999953433871, the following statements yield the value of 1.119 for B:

```
MOVE 1.12 TO A.

COMPUTE B = A.
```

Producing Object Files for Multiple ClearPath MCP Servers

You can produce object files for multiple ClearPath MCP servers by specifying PRIMARY and SECONDARY target servers with the TARGET option. Specifying these targets causes the COBOL85 compiler to produce object files that run on all your ClearPath MCP servers. The resulting object code is optimized for the PRIMARY target, but does not contain any operators that are invalid on the SECONDARY targets.

Similarly, the syntax TARGET=ALL optimizes the code for the server that is compiling the program, but produces code that can run on any ClearPath MCP server.

For additional information about the TARGET option, refer to Section 15, "Compiler Operations." For details about the COMPILERTARGET system command, refer to the *System Commands Reference Manual*.

Using Key Features of COBOL85

The COBOL85 compiler provides many improved capabilities over the COBOL74 compiler. The following topics enable you to take advantage of some of the best features of COBOL85.

Nested Programs

The COBOL85 capability to handle nested programs enables you to place one complete COBOL85 program within another. In fact, nested programs can be contained within other nested programs.

Nested programs enable you to segment large programs into smaller logical units, making your applications

- Easier to develop and test
- More reliable
- Easier to maintain

To identify a nested program so that it can be called, each program must contain the following statements:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. <idname>.
END PROGRAM <idname>.
```

Calling Nested Programs from Outside Programs

To call a nested program from the outside program that contains it, specify the following command:

```
CALL "INSIDE-NP1"
```

Using Items Declared in Outside Programs in Nested Programs

To use an item that is declared in an outside program in a nested program, specify the item with the GLOBAL clause.

Example

The following example shows a nested program.

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. OUTSIDE-NP.
PROCEDURE DIVISION.
LAB0.
 CALL "INSIDE-NP1".
 CALL "INSIDE-NP2".
IDENTIFICATION DIVISION.
 PROGRAM-ID. INSIDE-NP1.
PROCEDURE DIVISION.
LAB1.
 DISPLAY "INSIDE-NP1".
 END PROGRAM INSIDE-NP1.
IDENTIFICATION DIVISION.
 PROGRAM-ID. INSIDE-NP2.
PROCEDURE DIVISION.
LAB2.
 DISPLAY "INSIDE-NP2".
 END PROGRAM INSIDE-NP2.
 END PROGRAM OUTSIDE-NP.
```

Intrinsic Functions

COBOL85 provides you with a wide variety of intrinsic functions. The following list includes some of the most commonly used functions:

- Date and Day

  CURRENT-DATE, DAY-OF-INTEGER
- Arithmetic

  LOG, MAX, MEAN, RANDOM
- Accounting

  ANNUITY, PRESENT-VALUE
- Trigonometric

  SIN, COS, TAN, ASIN, ACOS, ATAN
- Compile-time

  WHEN-COMPILED, LINENUMBER

LINENUMBER Function

The LINENUMBER function returns the sequence number of the source file record on which it appears, which is very useful for debugging COBOL85 programs.

When you use this function in an INCLUDE file or a COPY statement, the sequence number pertains to the line number in the included file.

Example

```
MOVE FUNCTION LINENUMBER TO LN.
```

Scope Terminators

Scope terminators are phrases that delimit the scope of a statement within a COBOL85 program. By using explicit scope terminators at the end of your statements, you can improve the clarity and reliability of your program. The presence of a scope terminator indicates that a statement contains no more phrases.

Example

The scope of a statement can be terminated either explicitly or implicitly. The following example uses explicit scope terminators to indicate where the phrase "NOT ON SIZE ERROR" applies:

```
DIVIDE I BY J GIVING I
   ON SIZE ERROR
      MULTIPLY I BY K GIVING I
      END-MULTIPLY
   NOT ON SIZE ERROR
      IF X GREATER THAN Y THEN
         DISPLAY Y
      END-IF
END-DIVIDE.
```

In-line Performs

In-line performs keep small logic in the location where it is executed in the program. You can enclose the statements to be performed in your program by using an END-PERFORM statement.

Example

```
PERFORM VARYING I FROM 1 BY 1 UNTIL (I > 5)
   DISPLAY AN-ENTRY(I)
   MOVE I TO LAST-ENTRY
END-PERFORM.
```

EVALUATE Option

The EVALUATE option causes the COBOL85 compiler to evaluate multiple conditions. You can use this option to test one or more subjects against corresponding multiple objects. This capability is similar to a CASE statement in other programming languages.

Example

```
EVALUATE Patient-Age
  WHEN 0            PERFORM Infant
  WHEN 1 THROUGH 2  PERFORM Toddler
  WHEN 3 THROUGH 4  PERFORM Preschooler
  WHEN 5 THROUGH 17 PERFORM Student
  WHEN OTHER        PERFORM Adult
END-EVALUATE.
```

$IF Option

The $IF option controls whether the COBOL85 compiler ignores all source language records except for compiler control records.

The COBOL85 compiler always processes compiler control options encountered in the source language input between the IF, ELSE IF, ELSE, and END compiler control options. Even when $IF evaluates to FALSE, the compiler processes all compiler control records encountered in the source language input in the normal fashion.

You can use the $IF option to conditionally include or omit certain source records in the compilation of your COBOL85 program. You can also use $IF to replace the $SET OMIT = NOT DEBUGGING and $POP OMIT options in your program.

Example

```
$IF DEBUGGING
     MOVE "DEBUGGING" TO VERSION.
$ELSE IF INTERNAL
     MOVE "INTERNAL 48.1" TO VERSION.
$ELSE
     MOVE "SSR 48.1" TO VERSION.
$END IF
```

You cannot use the $IF option to omit a DICTIONARY compiler control option. For example, the following options set DICTIONARY to the value PRODUCTION even if the NEEDTEST value is TRUE:

```
$IF NOT NEEDTEST
$DICTIONARY=PRODUCTION
$END IF
```

The OMIT compiler control option also does not affect compiler control records. For more information, refer to “Conditional Compilations Options” or to the OMIT option in Section 15, “Compiler Operations.”

$INCLUDE Option

The $INCLUDE option causes the COBOL85 compiler to temporarily accept input from a different file. The compiler uses the file as input until the file is completely read or a specified range within the file is read.

Example

```
$INCLUDE FILE1 ("RANGE1")
```

Within FILE1, you must include

```
$COPYBEGIN "RANGE1"
<text>
$COPYEND "RANGE1"
```

The $INCLUDE option allows hard-coded sequence ranges within the file to be specified. Specifying a symbolic name such as "RANGE 1" precludes the need to hard-code the range sequence into the file that is included in <text>.

INITIALCCI File

The INITIALCCI file is an optional input file that is used to specify initial settings for compiler options. The COBOL85 compiler searches for the INITIALCCI file using standard usercode and family name conventions. If the compiler cannot find the file, the compiler proceeds without it.

You can use the INITIALCCI file to implement system-wide global settings. You can also use the INITIALCCI file to implement customized individual settings. The COBOL85 user programs do not need to be modified.

The INITIALCCI file can be file-equated at compilation time by modifying the file attributes for the INITIALCCI file.

Example

```
COBOL85:     XREFFILES RESET BOUNDS
BATCH:       ERRORLIMIT=9 SET ERRORLIST
INTERACTIVE: ERRORLIMIT=5
             PAGESIZE=122
```

For more information, refer to the INITIALCCI file in Section 15, "Compiler Operations."

CONSTANT Entry

The CONSTANT entry defines a constant value that can be used in place of a literal value within a COBOL85 program. This entry replaces the use of the $OPT3 option for creating references to literal values by data names.

The CONSTANT entry enables you to improve the performance of your COBOL85 program by using constant values instead of variables. You must declare CONSTANT data items at the 01 level with the following syntax:

```
01 <id> CONSTANT AS <literal>
```

Example

```
01 MAX-SIZE CONSTANT AS 200.
01 AN-ARRAY.
   03 AN-ENTRY  PIC X910)
                OCCURS MAX-SIZE TIMES.
```

USE AS EPILOG Procedure

Specifying a declarative as an epilog procedure enables you to designate that a certain procedure executes before the program exits. This capability enables you to facilitate tasks that are performed prior to termination, such as clean up or the release of locked resources. The procedure that you designate executes each time the program exits, whether it exits normally or abnormally.

Example

To determine if your program terminated normally, include the test in the epilog procedure as follows:

```
IF ATTRIBUTE HISTORYCAUSE OF MYSELF = 0 THEN
    <program terminated normally>
ELSE
    <program terminated abnormally>
END-IF.
```

COBOL85 Dump Analysis

A COBOL85 program dump is basically like any other program dump. You use the same skills to find information in a COBOL85 dump that you would for a COBOL74 dump.

There are several items present in a COBOL85 program dump that do not appear in a COBOL74 dump, such as the SLICESUPPORT library template; however, these items are necessary for the execution of the COBOL85 code.

The key difference between COBOL74 and COBOL85 program dumps is that COBOL74 allocates a stack cell for each 01 item and COBOL85 pools 01 items together in one array. A compiler program listing with the $MAP option set shows the difference.

Example

Consider the following WORKING STORAGE items:

```
WORKING-STORAGE SECTION.
01 GRP-1.
   03 X1 PIC X(10) VALUE "HI THERE!!".
   03 N1 PIC 9(10) VALUE 1111111111.
   03 B1 PIC 9(11) BINARY VALUE 1.
01 GRP-2.
   03 X2 PIC X(10) VALUE "BYE NOW!!!".
   03 N2 PIC 9(10) VALUE 2222222222.
   03 B2 PIC 9(11) BINARY VALUE 2.
01 ELEM-1 PIC X(13) VALUE "01 ELEMENTARY".
```

The COBOL74 $MAP listing of the WORKING-STORAGE items shows that separate stack cells are generated for the 01 items:

```
GRP-1 = (02,003)
GRP-2 = (02,004)
ELEM-1 = (02,005)
```

The COBOL85 $MAP listing shows that one stack cell is allocated for all the data items and the offset into the array for each item is listed:

```
GRP-1 = (2,006)
offset = 21 Byte
    X1 = (2,006)
offset = 21 Byte
    N1 = (2,006)
offset = 31 Byte
    B1 = (2,006)
offset = 41 Byte
 GRP-2 = (2,006)
offset = 47 Byte
    X2 = (2,006)
offset = 47 Byte
    N2 = (2,006)
offset = 57 Byte
```

```
     B2 = (2,006)
offset = 67 Byte
ELEM-1 = (2,006)
offset = 73 Byte
```

You can use the address of the array and the offset of the field to find the value of the data item in the dump.

COBOL85 Library Interfaces

COBOL85 provides the capability for COBOL85 programs to call libraries. COBOL85 allows two types of library interfaces: implicit and explicit.

Implicit Library Interface

The implicit library interface is similar to the COBOL74 method of calling a library. The CALL statement for this interface contains information about whether the parameters are declared by content or by reference.

Explicit Library Interface

The explicit library interface provides most of the library capabilities that are available in other languages, such as ALGOL. This interface also removes the restrictions imposed by COBOL74 type libraries, providing more control over library parameters and enabling a library to have multiple entry points.

For more information, refer to the CALL statement in Section 6, "Procedure Division Statements A-H."

SHAREDBYALL Libraries

COBOL85 enables you to create multithreaded libraries as follows:

Library Type	Action
For a COBOL74 style library with the LIBRARYPROG option enabled . . .	Set the LIBRARYLOCK option and set the SHARING option to the value SHAREDBYALL. Set the LOCALTEMP option (LOCALTEMP is set by default).
For COBOL85 subprograms bound into a SHAREDBYALL library written in another language . . .	Set the LOCALTEMP option.
For a SHAREDBYALL COBOL85 library written with explicit library syntax . . .	Write locking code using USAGE LOCK variables for instances where global variables need protection. Set the LOCALTEMP option.

For more information, refer to the LOCALTEMP, LIBRARYPROG, LIBRARYLOCK, and SHARING options in Section 15, "Compiler Operations."

Index

A

B

C

D

E

F

G

H

I

J

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Special Characters

86001518-307