

THE CORVUS CONCEPT
FORTRAN
REFERENCE MANUAL

 **CORVUS SYSTEMS**

DISCLAIMER OF ALL WARRANTIES & LIABILITIES

Corvus Systems, Inc. makes no warranties, either expressed or implied, with respect to this manual or with respect to the software described in this manual, its quality, performance, merchantability, or fitness for any particular purpose. Corvus Systems, Inc. software is sold or licensed "as is." The entire risk as to its quality or performance is with the buyer and not Corvus Systems, Inc., its distributor, or its retailer. The buyer assumes the entire cost of all necessary servicing, repair, or correction and any incidental or consequential damages. In no event will Corvus Systems, Inc. be liable for direct, indirect, incidental or consequential damages, even if Corvus Systems, Inc. has been advised of the possibility of such damages. Some states do not allow the exclusion or limitation of implied warranties or liabilities for incidental or consequential damages, so the above limitation may not apply to you.

Every effort has been made to insure that this manual accurately documents the operation and servicing of Corvus products. However, due to the ongoing modification and update of the software along with future products, Corvus Systems, Inc. cannot guarantee the accuracy of printed material after the date of publication, nor can Corvus Systems, Inc. accept responsibility for errors or omissions.

NOTICE

Corvus Systems, Inc. reserves the right to make changes in the product described in this manual at any time without notice. Revised manuals and update sheets will be published as needed and may be purchased by writing to:

Corvus Systems, Inc.
2029 O'Toole Avenue
San Jose, CA 95131

Telephone: (408) 946-7700
TWX 910-338-0226

This manual is copyrighted and contains proprietary information. All rights reserved. This document may not, in whole or in part be copied, photocopied, reproduced, translated or reduced to any electronic medium or machine readable form without prior consent, in writing, from Corvus Systems, Inc.

Copyright© 1982 by Corvus Systems, Inc. All rights reserved.

Mirror® patent pending, The Corvus Concept,™ Transporter,™ Corvus OMNINET,™ Corvus Logicalc,™ Time Travel Editing,™ EdWord,™ Constellation,™ Corvus,™ Corvus Systems,™ Personal Workstation,™ Tap Box,™ Passive Tap Box,™ Active Junction Box,™ Omninet Unit™ are trademarks of Corvus Systems, Inc.

FCC WARNING

This equipment generates, uses, and can radiate radio frequency energy and if not installed and used in accordance with the instruction manual, may cause interference to radio communications. As temporarily permitted by regulation it has not been tested for compliance with the limits for Class A computing devices pursuant to Subpart J of Part 15 of FCC Rules, which are designed to provide reasonable protection against such interference. Operation of this equipment in a residential area is likely to cause interference in which case the user at his own expense will be required to take whatever measures may be required to correct the interference.

* CORVUS SYSTEMS

*

* *

* The Corvus Concept Fortran
 Reference Manual

PART NO. : 7100-02823

DOCUMENT NO. : CCC/32-66/1.1

RELEASE DATE : February, 1982

CORVUS CONCEPT (TM) is a trademark of Corvus Systems, Inc.

Table of Contents

Chapter 1 Introduction	1
1.1 Overview of the FORTRAN Language	1
1.2 Notation and Terminology Used in this Manual	5
1.3 Basic Elements of FORTRAN	6
1.3.1 FORTRAN Character Set	6
1.3.2 Collating Sequence and Graphics	7
1.3.3 Use of Spaces or Blanks or Tabs	7
Chapter 2 Lines, Statements, and Control Flow	9
2.1 Lines	9
2.1.1 Comment Lines	9
2.1.2 Initial Lines	10
2.1.3 Continuation Lines	10
2.1.4 Compiler Directive Lines	11
2.2 Statements	11
2.2.1 Statement Labels	12
2.2.2 Order of Statements and Lines	13
2.3 Execution Sequence and Control Transfer	15
Chapter 3 Data Types and Constants	18
3.1 Data Type Rules	18
3.2 Constants	19
3.3 Integer Data Type	19
3.4 Real Data Type	20
3.4.1 Infinite and Indeterminate Real Values	21
3.5 Double Precision Data Type	22
3.6 Complex Data Type	22
3.7 Character Data Type	22
3.8 Logical Data Type	24
Chapter 4 FORTRAN Names, Arrays, and Substrings	25
4.1 FORTRAN Names	25
4.1.1 Scope of FORTRAN Names	26

4.1.2 Undeclared FORTRAN Names	27
4.2 Array Declarations	28
4.2.1 Dimension Declarations	28
4.2.2 Kinds of Array Declarations	29
4.2.3 Actual Arrays and Dummy Arrays	29
4.2.4 Referencing Array Elements - Array Subscripts	30
4.2.5 Using Unsubscripted Array Names	31
4.3 Character Substrings	32
Chapter 5 Expressions	33
5.1 Arithmetic Expressions	33
5.1.1 Arithmetic Operators	33
5.1.2 Arithmetic Operands	34
5.1.3 Constant Expressions	35
5.1.4 Type Conversion Rules for Arithmetic Expressions	35
5.1.4.1 Rules for Add, Subtract, Multiply and Divide	36
5.1.4.2 Rules for Exponentiation Operator	37
5.1.5 Coercion Rules for Integers of Different Size	38
5.1.6 Integer Division	38
5.2 Character Expressions	38
5.2.1 Restrictions on the use of String Expressions	39
5.3 Relational Expressions	39
5.3.1 Arithmetic Relational Expressions	40
5.3.2 Character Relational Expressions	40
5.4 Logical Expressions	41
5.4.1 Precedence of Logical Operators	41
5.5 Precedence of Operators	43
5.6 Evaluation Rules and Restrictions for Expressions	43
5.6.1 Restrictions on Function References	43

Chapter 6 Specification Statements	45
6.1 Type Statements - Declaring Data Types	45
6.1.1 Arithmetic Type Statements	46
6.1.2 CHARACTER Type Statement	47
6.1.3 LOGICAL Type Statement	48
6.2 DIMENSION - Declare Data Dimension	48
6.2.1 Dimension Declarators	49
6.3 COMMON - Declare a COMMON Block	49
6.4 PARAMETER - Make a Symbolic Association	50
6.5 IMPLICIT - Establish Default Data Type	50
6.6 EXTERNAL - Declare External or Dummy Procedure	52
6.7 INTRINSIC - Declare Intrinsic Function	53
6.8 SAVE - Retain Definition Status	54
6.9 EQUIVALENCE - Share Storage Between Elements	55
6.9.1 Restrictions on EQUIVALENCE Statements	55
Chapter 7 Data Initialization	57
7.1 Initializing Character Variables	58
7.2 Initializing Non CHARACTER Variables to CHARACTER Values	59
7.3 Implied DO in DATA Statements	60
Chapter 8 Assignment Statements	63
8.1 Arithmetic Assignment	63
8.2 Logical Assignment	65
8.3 Statement Label Assignment	65
8.4 Character Assignment	66
Chapter 9 Control Statements	69
9.1 Block IF THEN ELSE Statement	70
9.1.1 Block IF Statement	71
9.1.2 ELSEIF Statement	72
9.1.3 ELSE Statement	73
9.1.4 ENDIF Statement	73
9.2 Logical IF Statement	73
9.3 Arithmetic IF Statement	74
9.4 DO Statement - Loop Control	75

9.4.1 DO Loop Initialization Sequence	76
9.4.2 DO Loop Incrementation Processing	76
9.4.3 Events Which Terminate a DO Loop	77
9.5 CONTINUE Statement - Null Statement	77
9.6 STOP Statement - Stop Program Execution	78
9.7 PAUSE Statement - Suspend Program Execution	78
9.8 Unconditional GO TO Statement	78
9.9 Computed GO TO Statement	79
9.10 Assigned GO TO Statement	80
Chapter 10 Input and Output	81
10.1 Overview of the Input-Output System	81
10.1.1 Records	81
10.1.2 Files	82
10.1.3 Properties of Files	83
10.1.3.1 File Name	83
10.1.3.2 File Position	83
10.1.3.3 Formatted and Unformatted Files	83
10.1.3.4 Sequential and Direct Access Files	84
10.1.4 Internal Files	84
10.1.4.1 Special Properties of Internal Files	85
10.1.5 Units	85
10.2 General Discussion of the Input Output System	86
10.2.1 Pre-Connected Files	86
10.2.2 Examples of Common Input Output Operations	87
10.2.3 Less Common File Operations	88
10.2.4 Limitations of FORTRAN Input Output System	89
10.2.4.1 Direct Files must be on Blocked Devices	89
10.2.4.2 No Character Compression in Direct Files	89
10.2.4.3 BACKSPACE Only Applies to Files on Blocked Devices	90
10.2.4.4 Length Limitations on Formatted Records	90
10.2.4.5 BACKSPACE may not be used on	

Unformatted Sequential Files	90
10.2.4.6 Side Effects of Functions Used in Input Output Statements	90
10.3 Elements of Input and Output Statements	90
10.3.1 The Unit Specifier 'u'	91
10.3.2 The Format Specifier 'f'	91
10.3.3 The Record Number 'rn'	92
10.3.4 The End of File Exit Specifier	92
10.3.5 The Error Exit Specifier	92
10.3.6 The Input Output Status specifier 'ios'	93
10.3.7 The Input-Output List 'iolist'	93
10.3.7.1 Input and Output Objects	93
10.3.7.2 Implied DO Lists	94
10.4 The Specific Input and Output Statements	95
10.4.1 OPEN Statement	95
10.4.2 CLOSE Statement	97
10.4.3 READ, WRITE and PRINT Statements	97
10.4.4 File Positioning Statements	99
10.4.4.1 BACKSPACE Statement - Backspace a File	100
10.4.4.2 ENDFILE Statement - Write an Endfile Record	100
10.4.4.3 REWIND Statement - Rewind a File	100
10.4.5 INQUIRE Statement - Obtain File Properties	101
10.4.6 Notes on the INQUIRE Statement	104
10.5 List Directed Input and Output	105
10.5.1 List Directed READ	105
10.5.1.1 List Directed Value Separators	105
10.5.1.2 List Directed Input Values	106
10.5.2 List Directed WRITE and PRINT	107
Chapter 11 Format Specifications	109
11.1 FORMAT Specifications and the FORMAT Statement	109
11.2 Interaction Between Format Specifications and I/O List	112
11.3 Edit Descriptors	113

11.3.1 Repeatable Edit Descriptors	113
11.3.1.1 Numeric Editing	113
11.3.1.2 I - Integer Editing	114
11.3.1.3 F - Real Editing	115
11.3.1.4 E and D - Real Editing	115
11.3.1.5 G - Real Editing	116
11.3.1.6 L - Logical Editing	117
11.3.1.7 A - Character Editing	117
11.3.2 Non Repeatable Edit Descriptors	118
11.3.2.1 'xxx' - Apostrophe Editing	118
11.3.2.2 H - Hollerith Editing	118
11.3.2.3 X and T - Positional Editing	119
11.3.2.4 Slash Editing - End of Transfer on Record	120
11.3.2.5 Backslash or Dollar Editing - Inhibit End of Record	120
11.3.2.6 Colon Editing - Conditional Termination	121
11.3.2.7 P - Scale Factor Editing	121
11.3.2.8 EN and BZ - Blank Interpretation	121
11.3.2.9 S, SS and SP - Sign Control Editing	122
Chapter 12 Program and Subprogram Structure	123
12.1 Main Program	123
12.2 Access To Command Line Arguments	124
12.3 Formal Arguments and Actual Arguments	125
12.3.1 Argument Association	126
12.4 Subroutines	127
12.4.1 SUBROUTINE Statement	128
12.4.2 CALL Statement	128
12.5 Functions	130
12.5.1 External Functions	131
12.5.2 Intrinsic Functions	132
12.5.3 Statement Functions	132
12.6 ENTRY Statement	133
12.6.1 Restrictions on the ENTRY Statement	134
12.7 RETURN Statement	135

12.8	Definition Status	136
12.9	BLOCK DATA Subprogram	136
12.10	The FORTRAN Intrinsic Functions	138
Chapter 13	FORTRAN Compile Time Options	139
13.1	\$INCLUDE - Include Source File	139
13.2	\$XREF - Generate Cross Reference	139
13.3	\$SEGMENT - Designate Segment Name	140
13.4	\$BIGCODE - Set Maximum Code Sizes	140
13.5	\$COL72 - Restrict Source Lines to 72 Columns	141
13.6	FORTRAN-66 Compatibility Options	141
13.6.1	\$F66DO - Implement FORTRAN-66 DO Loops	141
13.6.2	\$CHAREQU - Character and Numeric Data Equivalence	141
13.6.3	\$INT2 - Make Integers 16-Bits	141
Appendix A	Messages from the FORTRAN System	143
A.1	Compile-Time Error Messages	143
A.2	Run-Time Error Messages	149
Appendix B	The FORTRAN Intrinsic Functions	153
B.1	Notes on the Intrinsic Functions	156
B.2	Restrictions on Ranges of Arguments	157
B.3	Non Standard Intrinsic Functions and Subroutines	158
Appendix C	Data Representations	161
C.1	Storage Allocation	161
C.2	Data Representations	163
C.3	Argument Passing Mechanism	172
Appendix D	ASCII Character Set Table	173
Appendix E	FORTRAN on the UNIX System	175
E.1	Running FORTRAN-77 On the UNIX System	175
E.2	UNIX System Dependencies	182
E.3	Interfacing FORTRAN-77 to UNIX C Programs	183

Chapter 1

Introduction

FORTRAN is a computer programming language oriented towards numerical computations. FORTRAN-77 is the latest (at the time of this writing) offering of the ANSI standardization committee. This FORTRAN reference manual describes the language called FORTRAN-77, as implemented by Silicon Valley Software, Inc. (SVS). From now on, the word FORTRAN is used to mean this implementation of FORTRAN-77.

1.1 Overview of the FORTRAN Language

A FORTRAN program is (ultimately) composed of characters. Characters are grouped into lines. Lines are grouped into program units. Program units are grouped into programs.

A line is either a comment line, an initial line of a statement or a continuation line of a statement. Lines appear in columns 1 thru 120. For compatibility with older FORTRAN implementations, the SVS FORTRAN-77 compiler will ignore lines past column 72 if the user selects the \$COL72 compiler option (see chapter 13 - "FORTRAN Compile Time Options" for a description of the compiler options.)

Comment lines are blank lines, as are lines with the letter C (upper-case or lower-case) or the asterisk character "*" in column one. Comment lines can appear anywhere in a FORTRAN program, including between initial lines and continuation lines of a statement.

The initial line of a statement has a zero or a space character in column 6. A continuation line of a statement has any other character in column 6. A continuation line is also signalled by an ampersand character (&) appearing in column one of a source line.

Statements may have up to 19 continuation lines. The initial line of a statement may have a statement label in columns 1 thru 5. A statement label is one to five digits in length. At least one of the digits must be non-zero. A statement label serves to "tag" a statement so that it can be referenced by other statements.

Statements are broadly divided into the two groups of executable and non-executable. Executable statements perform program actions that assign values to variables, evaluate expressions, affect flow of execution and perform data transmission. Non-executable statements generally are those that specify the forms and attributes of program objects. Statements are discussed in more detail a few paragraphs further on.

Program objects include constants and variables. A constant is a string of digits or other characters defining a value that does not change. Variables occupy storage and have values that can be changed during program execution. Variables and constants can have both a name and a data type. The name serves to identify that object in a program. The type of a data object defines, among other things, the amount of storage it occupies, its range and precision, and in some cases, the operations that can be performed on it. FORTRAN names can have default data types derived from a naming convention or the default rules can be overridden by explicit specifications.

A variable can be a single object or it can be an aggregate. There are two forms of aggregate data objects, namely array variables and character variables. An array variable is a collection of data occupying consecutive storage units. Arrays can have up to seven dimensions. A character variable represents string data and is a sequence of characters, which can be accessed individually, or collectively, in the form of a substring.

A complete FORTRAN program is composed of a main program and any number of subprograms. Subprograms fall into the categories of SUBROUTINE subprograms which can be activated via the CALL statement to perform out-of-line groups of statements, FUNCTION subprograms which compute and return a value in the context of an expression, and BLOCK DATA subprograms which serve to initialize data declared in COMMON blocks. The main program and subprograms form what is called program units. In general, the terms "subprogram" and "program unit" can be used interchangeably. User-defined subroutines and functions are also called "procedures".

A variable may be given more than one name by a process of association. There are several ways to associate data. The COMMON statement provides a way to share data between separate

program units. The EQUIVALENCE statement associates variables in the same program unit. Variables may also be associated through the argument passing mechanism when subroutines or functions are referenced.

Names of variables have a scope which is dependent on the way that they are defined. In general, most names (except names of program units, common areas and certain other names) have a scope that is local to the program unit in which they are defined. A name defined by being called as an external function has a global scope by default. Names defined in a common area are local to the program unit in which they are declared. The name of the common area itself is global. Formal parameters to statement-functions have a scope which is local to the statement-function statement itself.

Specification statements are one of the two major groupings of statements. Specification statements serve to declare variables and symbolic constants. Specification statements include the type statement for defining the data type of a variable, the DIMENSION statement to define the size of array variables, the COMMON and EQUIVALENCE statements to provide association of variables, the PARAMETER statement to give a symbolic name to a constant and the EXTERNAL and INTRINSIC statements to define attributes of other program units.

The DATA statement provides a mechanism for static initialization of data. The DATA statement includes an implied DO loop construct to facilitate initializing array variables in a concise manner.

Expressions combine data objects and operators to create new values. FORTRAN supports arithmetic, character, logical and relational expressions. Mixed-mode expressions are permitted, with well defined rules for conversions between the operands and generation of the result.

The assignment statement assigns the value of an expression to a variable. There are three variations of assignment, namely arithmetic, character and logical. The ASSIGN statement serves to assign the value of a statement label to an integer variable.

Control statements are those that control the flow of execution in a program. Various kinds of IF statements select other statements for execution, depending on the result of evaluating a logical or arithmetic expression. The DO statement provides for repetition of a block of statements while a control variable is assigned a sequence of values. The CALL and RETURN statements provide for subroutine and function execution. Variations of the GO TO statement provide for transfer of control within a program unit.

Statement-function statements are characterized by a single-statement "template" defined in a program unit, with operations on dummy arguments. The statement function is referenced in a program unit just like a function, with actual arguments supplied. The arguments are combined according to the statement-function definition to yield a result that can be used in an expression.

FORTRAN provides a powerful input and output capability. A file can be external (connected to an external device) or internal (refers to a character variable). Files can be formatted or unformatted. Files can be accessed sequentially or randomly. Formatted files can be the subject of data conversion operations from internal storage representations to external character string representations and vice-versa.

Format conversion is performed via READ, WRITE or PRINT statements. There is a rich set of format specifications to control the form and layout of converted data. There is a list-directed input-output capability, where default formatting rules are applied to the conversion process.

SUBROUTINE and FUNCTION program units may have arguments which a calling routine passes to them for processing. At the time a subroutine or function is declared, its formal arguments are declared. At the time the subroutine or function is referenced, actual arguments are substituted for the formal arguments. A subroutine or function may have multiple ENTRY points. An ENTRY statement can cause execution of a subroutine or function to begin at a statement other than the first executable statement. Control is returned from a subroutine or function program unit either by encountering the END statement, or by executing a RETURN statement. FORTRAN provides for an alternate return specification for subroutines, such that a subroutine program unit can return to a different place in the caller than the statement following the CALL statement.

FORTRAN supplies a comprehensive set of intrinsic functions which perform data type conversion and provide an extensive collection of arithmetic and transcendental functions.

1.2 Notation and Terminology Used in this Manual

This Section defines the notation that is used in this manual to define FORTRAN language constructs.

Upper-case letters and special characters are written as shown in programs. Lower-case letters and words indicate objects for which there is a substitution in actual statements described in the text. Once a lower-case object is defined, it can be assumed to retain that meaning for the remainder of the construct being defined.

Example of upper-case and lower-case usage

The format specification which describes integer editing is denoted **Iw**, where **w** is a non-zero, unsigned integer constant. In an actual FORMAT statement, the editing specification might be written as **I5** or **I21**. The editing specification for real numbers is **Fw.d**, where **d** is an unsigned integer constant. An actual FORMAT statement might contain an edit specification like **F8.4** or **F14.0**. Note that the period character is a special character as defined above, and is taken literally.

Brackets "[" and "]" enclose optional items. For example, **A[w]** indicates that either of the forms **A** or **A12** are valid (as a means of specifying a character format).

The ellipsis notation "..." indicates that the optional item preceding the ellipsis may appear one or more times. For example, the computed GOTO statement is described by the form:

```
GOTO (s [, s] ...) [,] i
```

which indicates that the syntactic item ',s' may be repeated any number of times.

Spaces (blanks) normally have no significance in describing FORTRAN statements. The general rules for spaces, supplied later in this chapter, provide the interpretation of spaces in all contexts. Throughout this manual, "space" and "blank" are considered synonymous. In general, the word "space" is used.

1.3 Basic Elements of FORTRAN

This section covers the basic lexical and syntactic elements that go towards constructing a FORTRAN program.

1.3.1 FORTRAN Character Set

The FORTRAN character set consists of 26 upper case letters, 26 lower case letters, and thirteen special characters.

A letter is one of the 52 characters:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z

A digit is one of the ten characters:

0 1 2 3 4 5 6 7 8 9

An alphanumeric character is a letter or a digit.

The special characters consist of the following characters:

Character	Name of Character
	Blank or Space
=	Equals sign
+	Plus sign
-	Minus sign
*	Asterisk
/	Slash
\	Reverse Slash
(Left Parenthesis
)	Right Parenthesis
,	Comma
.	Decimal Point
\$	Currency Symbol
'	Apostrophe
:	Colon
&	Ampersand

1.3.2 Collating Sequence and Graphics

SVS FORTRAN uses the ASCII character set. The collating sequence in ASCII is:

- . Space (blank) collates lowest, followed by:
- . Digits "0" thru "9", followed by:
- . Upper case letters "A" thru "Z", followed by:
- . Lower case letters "a" thru "z".

The special characters appear in between digits and upper-case letters and before and after lower-case letters. There is an ASCII character set chart in the appendices.

Within each of the ordered sets, digits, upper case letters, and lower case letters, the characters in those sequences are contiguous - there are no "holes" in those sequences.

1.3.3 Use of Spaces or Blanks or Tabs

The space (also called blank) and the tab characters have no meaning in a program unit, with the exceptions listed below. Otherwise, spaces or tabs can be used freely to improve the layout and readability of a program. Spaces or tabs are significant in the following cases:

- . Within string constants, and within Hollerith fields.
- . On compiler directive lines, discussed in the chapter on "Using the FORTRAN Compiler".
- . In column 6, when a space or tab distinguishes an initial line from a continuation line.
- . Counting in the total number of characters per line and per statement.

Chapter 2

Lines, Statements, and Control Flow

This chapter consists of three sections. The first section describes the notions of lines in a FORTRAN program. The second section covers the rules for FORTRAN statements. The final section in this chapter covers the concept of execution sequence or control flow - the order in which FORTRAN statements are executed.

2.1 Lines

A line in a program unit is a sequence of characters in columns 1 thru 120 (1 thru 72 if the \$COL72 compiler option is selected, as described in chapter 13). All characters in a line must be selected from the character set described in chapter 1. Comment lines (described below), character constants and Hollerith fields can contain any printable ASCII character.

The character positions on a line are called columns and are numbered consecutively from 1 thru 120 (1 through 72), left to right on a line.

The FORTRAN compiler ignores characters which appear to the right of column 120 (column 72 if the \$COL72 compiler option is selected) on a line, thus the user may use these columns for any purpose (such as sequence information).

2.1.1 Comment Lines

Comment lines can appear anywhere in a program unit, including before the first statement or after the last statement of a program unit. Comment lines may appear between an initial line and its first continuation line, or between two continuation lines.

Examples of Comment Lines

```

C   This is a comment line.
*   This is also a comment line.
C   The line following this one is all blank .....

C   ..... and is therefore considered to be a comment.
*
*   Comment lines are for documentary purposes and
*   have no effect on compilation or execution.

```

2.1.2 Initial Lines

An initial line generally indicates the start of a statement line. An initial line is any non-comment line containing the space or tab character or the digit 0 in column 6. Columns 1 thru 5 of the initial line of a statement may contain a statement label.

Examples of Initial Lines

```

C   Here are initial lines without statement labels.
C
    GO TO 999
0GO TO 999

C
C   Here are initial lines with statement labels.
C
379 GO TO 999
4850GO TO 999

```

2.1.3 Continuation Lines

A continuation line is any non-comment line containing any character from the FORTRAN character set (other than a space character or the digit 0) in column 6, or having the ampersand character & in column one. A continuation line must not have a statement label. There may be up to 19 continuation lines in a statement.

Examples of Continuation Lines

```

C      These contrived statements each span two lines.
C
      GO TO
      $ 999
*
      843 GO TO
      + 999
*
      GO TO
&      999

```

2.1.4 Compiler Directive Lines

Compiler Directives are an SVS extension to FORTRAN-77. Compiler directives provide additional controls over the compiler's actions. A compiler directive line is a line with a dollar sign "\$" in column 1. A compiler directive line can appear anywhere that a comment line can appear, although certain directives must appear in certain restricted places in the program. Spaces are significant in compiler directive lines and serve to delimit keywords and filenames. The compiler directives are listed in the chapter on "Running the FORTRAN Compiler".

Examples of Compiler Directives

```

*
*      The following directive instructs the compiler to
*      include the body of the file 'rasp.text' into the
*      program source code.
*
$INCLUDE rasp.text

```

2.2 Statements

FORTRAN language statements are described in Chapters 6 thru 12. Statements are used to form program units.

Statements are written in columns 7 thru 72 of an initial line, and as many as 19 continuation lines.

An END statement is the exception to the above rules. An END statement must appear on an initial line on its own. No other statement in a program unit may have an initial line that looks like an END statement.

In general, statements must begin on new lines, that is, a statement may not begin on the same line as another statement. The exception to this rule is the logical IF statement.

Spaces before, within, and after statements have no effect, except within character constants and Hollerith constants, where they indicate blank characters.

Examples of Statements

```
C      An assignment statement
C
      A = 5.0
C
      A subroutine call statement
C
      CALL COLECT(PAY, PHONE)
C
      A logical IF statement
C
      IF (DAY .EQ. 'FRIDAY') RETURN
```

2.2.1 Statement Labels

Statement labels provide the means to "tag" a statement such that other statements may refer to it.

Any statement may be labelled, but only labels associated with executable statements and FORMAT statements can be referenced by other statements.

A statement label is one to five digits appearing anywhere in columns 1 thru 5 of the initial line of a statement. At least one of the digits in a statement label must be non-zero.

In any given program unit, statement labels must be unique - duplication of statement labels is an error.

Examples of Statement Labels

```
123 FORMAT('The result is', I5)
C
C   An example of a DO block.
C
      DO 110 ICON = 1, 100
        DESK(ICON) = 0.0
110 CONTINUE
```

2.2.2 Order of Statements and Lines

In any given program unit, the order in which statements appear must obey certain rules. These rules are detailed below.

A PROGRAM statement may appear only as the first statement of a main program. The first statement of a subprogram must be either a FUNCTION, SUBROUTINE, or a BLOCK DATA statement.

In a program unit, statements must appear in the following order.

1. FORMAT statements can appear anywhere.
2. All specification statements must precede all DATA statements, statement-function statements, and executable statements.
3. All statement-function statements must precede all executable statements.
4. DATA statements may appear anywhere after the specification statements.
5. ENTRY statements may appear anywhere except between a block IF statement and its corresponding END IF statement or between a DO statement and the terminal statement of its DO-loop. In other words, a subprogram must not be entered (via an ENTRY statement), in the middle of a block IF or DO block.

Within a program unit's specification statements, IMPLICIT statements must precede all specification statements other than PARAMETER statements. Any specification statement that defines the type of a symbolic name must precede a PARAMETER statement that defines the symbolic name of a constant.

PARAMETER statements that define symbolic constant names must precede all uses of such names.

The last (non-comment) line of a program unit must be an END statement.

The diagram below pictorially describes the manner in which statements and comment lines may be interspersed.

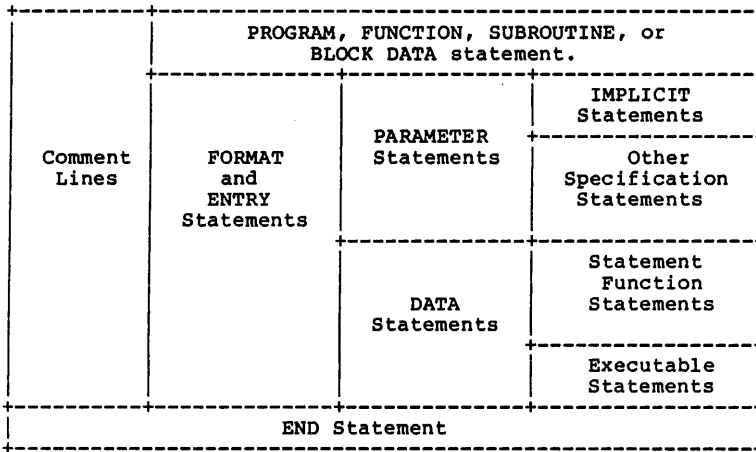


Figure 2-1
Ordering of Lines and Statements in a FORTRAN Program

In the diagram, vertical lines separate statement groups that may be mixed. For example, FORMAT statements can be mixed with statement-function statements and executable statements.

Horizontal lines separate statement groups that must not be mixed. For example, statement-function statements cannot be mixed with executable statements. Note that an END statement, in addition to being executable, must be the last statement in a program unit.

2.3 Execution Sequence and Control Transfer

Normal execution sequence means that executable statements are executed in the order in which they appear in the program unit. Program execution starts with the first executable statement in the main program. When an external procedure is referenced, execution proceeds with the first executable statement that follows the FUNCTION, SUBROUTINE or ENTRY statement in the subprogram.

A control transfer means that the normal execution sequence is altered. Statements that cause control transfer are:

1. unconditional GO TO statement, computed GO TO statement or assigned GO TO statement,
2. Arithmetic IF statement,
3. RETURN statement,
4. STOP statement,
5. input-output statement containing an error specifier or an end-of-file specifier,
6. CALL with an alternate return specifier,
7. logical IF statement containing any of the above forms as its subordinate statement,
8. Block IF and ELSE IF statements,
9. the last statement (if any) of an IF-block or ELSE IF-block,
10. DO statement,
11. the terminal statement of a DO-loop,
12. END statement.

Normal execution sequence is not affected by non-executable statements, comment lines or compiler directives appearing between executable statements in the source code.

Executing a function reference or a CALL statement is not considered a control transfer in the program that makes the reference, except when control is returned to a statement identified by an alternate return specifier in a CALL statement.

Executing a RETURN or an END statement in a referenced procedure, or a control transfer in a referenced procedure, is not considered a control transfer in the program unit that makes the reference.

Function and subroutine subprograms cannot be invoked recursively in SVS FORTRAN. Note, however, that FORTRAN subprograms can reference subprograms written in other languages (such as Pascal) that themselves can be recursive.

Chapter 3

Data Types and Constants

There are six data types in FORTRAN, namely: INTEGER, REAL (floating point), DOUBLE PRECISION (extended precision REAL), COMPLEX (elements of the complex number domain), CHARACTER (character string data) and LOGICAL (able to assume the values .TRUE. or .FALSE.). The various data types are discussed in the sections to follow. SVS FORTRAN extends ANSI FORTRAN in such a way that the user can specify the amount of computer storage which a particular data type consumes. This extension is covered in the appropriate sections below.

3.1 Data Type Rules

A symbolic name (associated with a constant, variable, array, external function or statement-function) can have its type specified in a type statement. The possible types are those listed in the paragraph above.

If no explicit type statement is supplied for a program element, the type is implied by the first letter of the name. A first letter of I, J, K, L, M or N implies type integer. Any other first letter implies type real. These default type rules can be overridden either by explicit type statements or by the IMPLICIT statement, which changes the default type-rules.

The data type of an array element is the same as that of the array. The data type of a function name specifies the type of the value which that function returns. Intrinsic functions have a type that is specified in the chart in the chapter on "Program Structure". Generic intrinsic functions do not have a default type. The type of a generic intrinsic function depends on the type of its argument(s). An external function reference is given a default type, based upon the first letter of its name, in the same way as variables and arrays.

3.2 Constants

A constant is a value that defines itself and does not change. A constant can be an arithmetic value, a logical value, or a character string value. The representation of a constant specifies both its value and its data type. A PARAMETER statement associates a symbolic name with a constant.

Arithmetic constants are INTEGER, REAL, DOUBLE PRECISION, and COMPLEX values.

For the purposes of definition in this manual, an unsigned constant is a constant without any leading sign. A signed constant is a constant with a leading plus or minus sign. An optionally-signed constant is a constant that can be either signed or unsigned. Integer, real, and double precision constants may be optionally-signed except where otherwise noted.

3.3 Integer Data Type

Integers, as represented in the finite word size of a computer, are only a subset of the infinite set of integers. An integer value as represented in FORTRAN is an exact representation of the corresponding integer.

An integer data value occupies two words (four bytes or 32 bits) of storage and can represent values in the range -2,147,483,648 thru +2,147,483,647. Integers can be designated as fitting in 16 bits if the \$INT2 compiler option is selected.

Note that integers are represented internally in two's complement notation and that it is a consequence of this notation that there is one more negative integer than there are positive integers.

SVS FORTRAN also provides a means to specify the amount of storage which integers occupy. This is an extension to ANSI FORTRAN. The extended forms of integer are:

INTEGER*1 occupies one byte (8 bits) and can assume values
 in the range -128 thru +127.

INTEGER*2 occupies one word (16 bits) and assumes values in the range -32768 thru +32767.

INTEGER*4 occupies two words (32 bits) and is the same as the standard integer type discussed above.

An integer constant consists of a sequence of decimal digits, preceded by an optional sign. Alternatively, an integer constant can be expressed in the hexadecimal radix by a dollar sign (\$) followed by a sequence of digits or letters in the range 'A' through 'F' or 'a' through 'f'. Note that hexadecimal numbers are considered unsigned. To obtain a signed hexadecimal constant, the user must explicitly code a 1 in the sign bit position.

Examples of INTEGER Constants

72	-32768	32767	0	+56
\$123	\$ffffffff	\$0a		\$3e8

3.4 Real Data Type

Real Data Types are intended to represent the set of real values which comprise the continuum. Because of a finite representation imposed by a finite word size in the computer, the real data type in FORTRAN can only represent a finite subset of the entire set of reals.

A basic real constant has an optional sign, an integer part, a decimal point, and a fractional part. Both the integer and the real part are sequences of digits. Either part can be omitted, but not both. Real constants are assumed to be decimal numbers.

Examples of Basic Real Constants

3.14159	+2.236	-1.4142
.7071	+5	-.618034
5.	+8.	-6.
0.0	0.	.0

A real exponent consists of the letter E followed by an optionally signed integer constant. A real exponent indicates a power of ten.

Examples of Real Exponents

E14	E+12	E-10	E0
-----	------	------	----

A real constant is any one of: a basic real constant, a basic real constant followed by an exponent part, or an integer constant followed by an exponent part.

SVS FORTRAN provides a means to specify the amount of data storage which a real data type is to occupy. The forms are:

REAL*4 occupies two words (32 bits) and is the same as a basic real datum as described above.

REAL*8 occupies four words (64 bits) and is the same as the DOUBLE PRECISION data type discussed below.

Examples of Real Constants

+7.52E-1	299793.5E3	20E-3
----------	------------	-------

A real constant containing an exponent part is the product of the constant preceding the E and the power of ten indicated by the integer following the E.

A real value occupies four bytes of storage. The range of real values is approximately $-3.4E38$ thru $+3.4E38$. The precision is about seven decimal places.

3.4.1 Infinite and Indeterminate Real Values

The representation of real values in FORTRAN allows for positive and negative "infinity", and for indeterminate values. This is primarily of interest when formatting such values for output. There is more on this subject in the chapter on input and output.

3.5 Double Precision Data Type

The Double-Precision data type is intended for applications where the range and precision of Single-Precision data is inadequate. Double-Precision extends the range to approximately $-10E308$ thru $+10E308$ and the precision to about 16 decimal digits.

A double precision exponent is the letter D followed by an optionally-signed integer constant. The forms of a double precision constant are either: a basic real constant followed by a double precision exponent, or an integer constant followed by a double precision exponent.

3.6 Complex Data Type

The Complex data type represents values from the complex number domain.

A complex number consists of an ordered pair of numbers, each of which is either an integer or a single-precision real number - the first representing the "real" (sic!) part of the number, and the second representing the "imaginary" part. A complex constant is written as two integers or single-precision real numbers enclosed in parentheses and separated by a comma.

Examples of Complex Constants

(1, 1) (0.707, -0.707) (-1.5E10, 2.6E-5)

3.7 Character Data Type

A character data element is a string of characters. The string can contain any of the printable ASCII characters. Spaces are significant in character strings. The length of a character string is the number of characters in the string.

The form of a character constant is a non-empty string of characters enclosed in apostrophe "'" signs. The apostrophes serve to delimit the string constant, but are not part of it. An apostrophe in the string is represented by two juxtaposed apostrophes.

The length of a character constant is the number of characters in the string, except that each pair of juxtaposed apostrophes in the string is counted as one character. The delimiting apostrophes are not part of the string and are not counted in the string length.

The maximum length of a character constant is 255 characters.

There is no provision for expressing empty (null) character strings.

Examples of Character String Constants

```
'x'           ' '           'The hunting of the Snark'
'The time is One O''Clock'      ''''
```

The last two examples illustrate the representation of embedded apostrophes.

FORTTRAN source lines extend up to column 72 on a line. Shorter lines are not space-filled to 72 columns but are left as typed. When a character constant extends across a line boundary, its value is as if the portion of the continuation line starting in column 7 abuts the last character on the preceding line. Thus the FORTRAN source statement:

```
200 example = 'First string part<cr>
$ Second string part'
```

(where <cr> is carriage-return) is equivalent to the statement:

```
200 example = 'First string part Second string part'
```

where the single space between the "t" at the end of the first line and the "S" at the start of the second line is the space in column 7 of the continuation line. Long character constants can be represented in this way.

3.8 Logical Data Type

A logical data element represents a Boolean quantity. It can only take on the values true or false.

The form of a logical constant is either: ".TRUE." (representing the truth value) or ".FALSE." (representing the false value).

SVS FORTRAN provides a means to specify logical data items which occupy less data storage than the standard logical type. These forms are:

LOGICAL*1	occupies one byte (8 bits).
LOGICAL*2	occupies one word (16 bits).
LOGICAL*4	occupies two words (32 bits) and is the same as the standard LOGICAL data type.

Chapter 4

FORTRAN Names, Arrays, and Substrings

This chapter introduces the rules for FORTRAN names - symbolic names which may be used to identify program objects. The second section describes the way that arrays are defined and referenced. The third section discusses the ideas of character variables and substrings of character variables.

4.1 FORTRAN Names

A FORTRAN name or identifier consists of one through six alphanumeric characters, and must start with a letter. A FORTRAN name can have embedded spaces in it - the spaces have no significance and are ignored. The FORTRAN compiler makes no distinction between upper-case letters and lower-case letters - the names **PASCAL**, **PaScAL**, **paScAL** and **pascal** are all equivalent as far as FORTRAN is concerned.

A name is used to denote a user-defined variable, a system-defined variable, array variable, subroutine or function. FORTRAN does not have any reserved words - the compiler recognizes keywords in context. For reasons of clarity and readability though, users are recommended to use names that are distinct from those of FORTRAN.

Examples of Valid FORTRAN Names

XPos	Eatup	FilSet	MAX0
RngKut	L5	Shell	Bubble

Examples of Invalid FORTRAN Names

2ndTime	Begins with digit
TooLarge	More than six characters
No_Good	Non-alphanumeric character

4.1.1 Scope of FORTRAN Names

The scope of a FORTRAN name is that region of a program over which the name is known or can be referenced. In general, the scope of a name is either local to a program unit or global to the entire FORTRAN program. There are certain exceptions which are described later.

A name with global scope can be used in more than one program unit (subroutine, function or the main program) and still refer to the same object. Names with global scope can only be used in a single, consistent manner within the same program. The names of all subroutine, function and block-data program units, the names of common areas, and the program name, have global scope. Therefore, there cannot be a subroutine program unit that has the same name as a function program unit or a common area. Similarly, two function program units cannot have the same name.

A name with local scope is only known within a single program unit. A name with local scope can therefore be used in other program units with the same or different meanings everywhere it is used. Within a specific local scope, a name must be used consistently and refer to the same object. The names of variables, arrays, constants, arguments and statement-functions all have local scope. A name with local scope can be used in the same compilation as the same name with global scope as long as the global name is not referenced within the program unit containing the local name. For example, there can be a function called PARTY, and a local variable called PARTY in another program unit, as long as the program unit containing the variable called PARTY does not try to reference the function called PARTY. The FORTRAN compiler detects all such scope errors and issues diagnostics concerning them.

Common block names are an exception to the scope rules. It is possible to refer to a globally scoped common block name in a program unit containing a locally scoped name identical to that of the common block. This situation is allowed because common block names always appear in slashes, such as /COLD/, and therefore the compiler can always distinguish such names.

Formal arguments to statement-functions are another exception to the scope rules. The scope of formal arguments of statement-functions is the body of the statement function itself. Any other use of those names in the statement-function is not allowed and neither is any other use of such names outside the statement-function. For example, if a formal argument to a statement-function has the same name as that of a function subprogram, that function subprogram may not be referenced from within the body of the statement-function. References to formal argument names of a statement-function from outside the body of the statement-function refer to objects which are different from the arguments of the statement-function.

Names used as implied-DO control variables in DATA statements and input-output statements have a scope which is local to the DATA statement or input-output statement.

4.1.2 Undeclared FORTRAN Names

When a user name that has not previously appeared in a program unit is referenced in an executable statement, FORTRAN decides how to classify that name from the context in which it appears. If the name appears to be a variable, FORTRAN creates a symbol-table entry for that name.

Its type is inferred from the first letter of the name. Variables starting with the letters I, J, K, L, M and N are considered to be of type integer; all others are considered to be of type real. If an undeclared name appears in the context of a function reference, the function's type is inferred from its name in the same manner as for variables. In both cases, these default type rules can be overridden by previous IMPLICIT statements (see the chapter on "Specification Statements"). Similarly, a name appearing in the context of a subroutine call has an entry created for it. If a symbol table entry exists for a subroutine or function name, its attributes are coordinated with those of the newly created entry. Inconsistencies such as a subroutine name used in the context of a function or vice versa give rise to error diagnostics.

In general, users are encouraged to declare all names used in each program unit, since it helps to assure that FORTRAN associates the proper definition with the name. Letting FORTRAN decide on the default can sometimes result in logical errors that are hard to find, usually at execution time when strange results or forms of behavior are exhibited.

4.2 Array Declarations

Arrays provide the means to deal with data aggregates where the elements of the aggregates are homogeneous. An array declaration specifies a symbolic name that identifies an array in a program unit. The declaration also serves to specify properties of the array, such as its dimension and, optionally, the type of its elements. In any given program unit only one array declaration is allowed for any given array - duplicate declarations are flagged as errors. The form of an array declaration is:

```
array name (dim [, dim] ... )
```

Each 'dim' above is a dimension declarator as defined below. The number of dimensions for an array is equal to the number of dimension declarations given when the array is declared.

4.2.1 Dimension Declarations

A dimension declarator serves to define the bounds of a specific dimension in an array. FORTRAN-77 provides for defining both the lower and the upper bound of a dimension. The form of a dimension declaration is:

```
[lower bound :] upper bound
```

The optional lower bound, and the upper bound are arithmetic expressions, called dimension bound expressions, in which all constants, symbolic constant names, and variables are of type integer. The upper bound of the last dimension declaration can be an asterisk (see 'assumed size arrays', later).

A dimension bound expression must not contain any function or array element references. Integer variables can appear in dimension bound expressions only in adjustable array declarations (see 'adjustable arrays', later). If a symbolic constant name or variable in a dimension bound expression is not of default implied integer type, it must be specified as integer via a type statement or an IMPLICIT statement before its use in a dimension bound expression.

Either dimension bound may have a positive, negative or zero value. The upper bound must not be less than the lower bound. If only the upper bound is specified, the lower bound has the value one (1). An upper bound of * is always greater than or

equal to the lower bound.

4.2.2 Kinds of Array Declarations

There are three basic forms of array declarations.

A constant array declaration is one in which all the dimension bound expressions are integer constant expressions.

An adjustable array declaration is one in which the dimension bounds contain integer variables. Adjustable arrays may be used as dummy arguments in subroutines and functions. Variables which define the bounds of adjustable arrays must either be formal arguments themselves, or they must be in common blocks.

An assumed size array declaration is one in which the upper bound of the last dimension is an asterisk character '*'. Assumed size arrays may also only be used as dummy arguments to subroutines and functions. Using assumed size arrays in procedures circumvents any range checking which the FORTRAN system can perform.

4.2.3 Actual Arrays and Dummy Arrays

An actual array declarator "actually" declares an array there and then. Each actual array declarator must be a constant array declarator as defined above. An actual array declarator can be used in the type statement, the DIMENSION statement and the COMMON statement, as defined in the chapter on "Specification Statements".

A dummy array declarator defines a dummy argument for a subroutine or function. A dummy array declarator can be any of the forms given above: constant, adjustable or assumed size. A dummy array declarator can only appear in subroutines and functions. A dummy array declarator may not appear in a COMMON statement.

Examples of Array Declarations

```

*
*   Constant array Declarations in type statements
*
C           a 100 element vector with bounds 1 - 100
  INTEGER VECTOR(100)
C           a 20 element matrix with 5 rows and 4 columns
  REAL MATRIX(5, 4)
C           a 256 element array with bounds 0 - 255
  CHARACTER*2 CHARS(0 : 255)
C           a 3 element array
  LOGICAL*2 BOOLS(-1 : +1)
C           a constant expression dimension
  REAL WOOD(2*4)
*
*   Adjustable Array Declaration in a DIMENSION statement
*
  DIMENSION SCREEN(1 : CHARS, 1 : LINES)
*
*   Assumed size array declaration in a type statement.
*
  REAL VARIAB(5, *)

```

4.2.4 Referencing Array Elements - Array Subscripts

An array subscript is the means to reference an element of the array. The form of an array subscript is:

```
(subexpr [, subexpr] ...)
```

Note that the term "subscript" includes the parentheses that enclose the subscript expression list.

A subscript expression is an integer expression. A subscript expression can contain array element references and function references. If a subscript expression contains a function reference, the function must not change the value of any other subscript expression in the same subscript.

In any given program unit, the value of each subscript expression should not be less than the lower bound for the dimension and should not be greater than the upper bound for the dimension. If the upper dimension bound is an asterisk, the subscript expression must not be greater than the size of the dummy array.

Examples of Arrays with Subscripts

```
SCREEN(2, 3)
```

```
VARIAB(N+1, MAX(3, 4))
```

4.2.5 Using Unsubscripted Array Names

Generally speaking, array names must be followed by subscripts. There are some exceptions where the array name alone can be used. An unsubscripted array name can be used in the following places:

- . a list of dummy arguments for a subroutine or function program unit,
- . a COMMON statement when declaring that the array resides in that common block,
- . a type statement when the type of the array is established,
- . an array declaration when the array dimensions are being established,
- . an EQUIVALENCE statement,
- . a DATA statement,
- . the list of actual arguments in a reference to an external procedure,
- . the list of an input-output statement if the array is not an assumed size dummy array,
- . a unit identifier for an internal file in an input-output statement if the array is not an assumed size dummy array,
- . the format identifier in an input-output statement if the array is not an assumed size dummy array,
- . a SAVE statement.

4.3 Character Substrings

A Character Substring is a contiguous portion of a character object. The type of a character substring is of type CHARACTER. A character substring can be identified by a symbolic name, and it can be referenced and assigned values by that name. The forms of a substring name are:

```
character variable([start] : [finish])
```

where 'start' and 'finish' are substring expressions. A character variable may be an element of a character array. 'start' specifies the leftmost character position of the substring. 'finish' specifies the rightmost character position of the substring. The values of 'start' and 'finish' must be such that:

$$1 \leq \text{start} \leq \text{finish} \leq \text{length}$$

where 'length' is the length of the character variable or character array element. If 'start' is omitted, the value one (1) is used. If 'finish' is omitted, the value 'length' is used. Both 'start' and 'finish' can be omitted. In such a case, a substring reference of the form `s(:)` is equivalent to `s`. The length of a character substring is 'finish' - 'start' + 1.

A substring expression is any integer expression which can contain array element references, and function references. The same restrictions (with regard to side effects) apply to substring expressions as apply to array subscripts.

Examples of Character Substrings

```
ROPEY(1:3)           THELOT(:)
ACHAR(5:5)          FOURCH(:4)
```

Chapter 5

Expressions

This chapter describes the rules for expressions. An expression is formed from operands, operators, and parentheses. FORTRAN has four classes of expressions:

- . Arithmetic expressions,
- . Character expressions,
- . Relational expressions,
- . Logical expressions.

5.1 Arithmetic Expressions

An arithmetic expression expresses a numeric computation and generates a numeric value.

5.1.1 Arithmetic Operators

The arithmetic operators are as follows:

Operator	Meaning
**	Exponentiation
/	Division
*	Multiplication
-	Subtraction or Negation
+	Addition or Identity

The **, /, and * operators are binary operators. The + and - operators can be unary or binary operators.

The ** operator has the highest precedence, then the * and / operators, and lastly the + and - operators. Parentheses may be used freely to change the order of evaluation.

5.1.2 Arithmetic Operands

An arithmetic operand consists of a primary, a factor, a term or an arithmetic expression. These various kinds of operands are discussed below.

The primary operands are:

- . Unsigned arithmetic constant,
- . Symbolic name of an arithmetic constant,
- . Arithmetic variable reference,
- . Arithmetic array element reference,
- . Arithmetic function reference,
- . Arithmetic expression enclosed in parentheses.

The factor operands are:

- . Primary,
- . Primary ** factor.

A factor is formed from a sequence of one or more primaries separated by an exponentiation operator. The second form means that an expression such as:

$$2**3**4$$

is to be interpreted as:

$$2**(3**4)$$

A term operand is:

- . Factor,
- . Term / factor,
- . Term * factor.

A term is formed from one or more factors separated by the multiply or divide operator. Factors are combined left to right.

An arithmetic expression consists of:

- . Term,
- . +term or -term,
- . Arithmetic expression + term,
- . Arithmetic expression - term.

An arithmetic expression consists of a series of terms separated by plus or minus operators. The first term in an expression can be preceded by a plus or minus sign. Terms are combined left to right. Note that the rules for expressions mean that two consecutive operators form an incorrect expression. Thus $A^{**}B$ is wrong, whereas $A^{**}(-B)$ is correct.

5.1.3 Constant Expressions

Constant expressions are used in many language constructs throughout FORTRAN, especially in specification statements. There are two forms of constant expressions, namely arithmetic constant expressions and integer constant expressions. These are discussed below.

An arithmetic constant expression is an expression in which each primary is an arithmetic constant, the symbolic name of an arithmetic constant or a constant expression enclosed in parentheses. Exponentiation is only allowed if the exponent is of type integer.

Examples of Arithmetic Constant Expressions

$5.0*2$	$2^{**}31-1$	$2*(4.5, 9.8)$
$-16/4$	$3.141592/2$	$5^{**}(3+2)$

An integer constant expression is an arithmetic constant expression in which each constant is of type integer.

Examples of Integer Constant Expressions

$3*5$	-10	$4+5*(9-2)$
-------	-------	-------------

5.1.4 Type Conversion Rules for Arithmetic Expressions

The data type of an expression is ultimately derived from the data types of its operands according to the rules stated below. When operands of mixed data types appear in an expression, FORTRAN performs implicit type conversion on the operands according to well-defined rules in order to generate the result.

When the plus "+" operator or the minus "-" operator operate upon a single operand (they are used as unary operators), the data type of the result is the same as the data type of the operand.

When an arithmetic operator applies to a pair of operands, the type of the results is as shown in the tables below. The letter I stands for an operand or result of type Integer, the letter R for Real, the letter D for Double-precision and the letter C for Complex. The rules are given in the form of assignments. The result type is indicated by the letter to the left of the equals sign and the derivation of that result is given by the expression to the right of the equals sign. The function names REAL, DBLE and CMLPX are as defined in the table of intrinsic functions in the appendix on "Intrinsic Functions".

5.1.4.1 Rules for Add, Subtract, Multiply and Divide

The two tables below define the types and interpretations for the +, -, * and / operators. For example, to obtain the rule for I1*C2 where 'I1' is an integer and 'C2' is a complex, look in the second part of the table; find the 'I1' entry under 'X1' and the 'C2' entry across from 'X2'; the rule is then:

```
'C = CMLPX(REAL(I1), 0.0) + C2'
```

which is interpreted as:

```
'the result is of type complex; the first operand is obtained by converting the integer to a real, then converting that to a complex with the imaginary part 0.0; the two complex numbers are then added'.
```

The rules for subtraction, multiplication and division are obtained by replacing the "+" signs with the desired operator.

X2 X1	I2	R2
I1	I = I1 + I2	R = REAL(I1) + R2
R1	R = R1 + REAL(I2)	R = R1 + R2
D1	D = D1 + DBLE(I2)	D = D1 + DBLE(R2)
C1	C = C1+CMPLX(REAL(I2), 0.0)	C = C1+CMPLX(R2, 0.0)

X2 X1	D2	C2
I1	D = DBLE(I1) + D2	C = CMPLX(REAL(I1), 0.0)+C2
R1	D = DBLE(R1) + D2	C = CMPLX(R1, 0.0)+C2
D1	D = D1 + D2	Not Allowed
C1	Not Allowed	C = C1+C2

5.1.4.2 Rules for Exponentiation Operator

The tables below define the types and interpretations for expressions of the form X1**X2.

X2 X1	I2	R2
I1	I = I1 ** I2	R = REAL(I1) ** R2
R1	R = R1 ** I2	R = R1 ** R2
D1	D = D1 ** I2	D = D1 ** DBLE(R2)
C1	C = C1**I2	C = C1**CMPLX(R2, 0.0)

X2 X1	D2	C2
I1	D = DBLE(I1) ** D2	C = CMPLX(REAL(I1), 0.0)**C2
R1	D = DBLE(R1) ** D2	C = CMPLX(R1, 0.0)**C2
D1	D = D1 ** D2	Not Allowed
C1	Not Allowed	C = C1**C2

Four of the entries in the above table specify what happens when a complex argument is raised to a complex power. In these cases, the value of the expression is the principal value, determined by the formula:


```
X1**X2 = EXP(X2 * LOG(X2))
```

where EXP and LOG are the exponential and natural logarithm intrinsics described in the chapter on "Program Structure".

Except for values raised to an integer power, in mixed mode expressions, the operand which differs from the type of the result is converted to the type of the result according to the rules given in the tables above. The operator then operates on a pair of operands of the same type. When a primary is raised to an integer power, the integer does not need to be converted.

5.1.5 Coercion Rules for Integers of Different Size

In expressions involving INTEGER*1, INTEGER*2 and INTEGER*4 (INTEGER), the smaller sized operand is always "promoted" to the size of the larger operand, and the arithmetic operation is performed in the larger sized field. In all cases, elements of type INTEGER*1 are always promoted to INTEGER*2. In any case, assigning the result of an expression to a variable of smaller size produces an undefined result if the value stored exceeds the range of values allowed for that specific variable. Note also that many FORTRAN statements and functions specifically require arguments of type INTEGER. In such cases, neither arguments of size INTEGER*1 nor INTEGER*2 may be used.

5.1.6 Integer Division

If an integer operand is divided by another integer operand, the result is not the strict mathematical quotient. Instead, the quotient is obtained by truncating towards zero. Thus $1/2$ is 0 and $(-8)/3$ is -2.

5.2 Character Expressions

A character expression operates on character strings and generates character values. The simplest form of character expressions are:

- . character constant,
- . character variable,
- . character array element reference,
- . character substring reference,
- . character function reference,
- . character expression enclosed in parentheses.

There is only one character operator - the "//" sign, meaning concatenation.

The result of a character concatenation operations such as:

X1 // X2

is a value which is 'X1' concatenated on the right with 'X2'. The length of the result is the sum of the lengths of the individual operands.

5.2.1 Restrictions on the use of String Expressions

Formal arguments to procedures can be character strings whose length is specified as (*). This designates the string as an assumed size character string whose length is determined at the time an actual string argument is associated with that formal argument. A character string expression involving concatenation of such a string argument may not be passed as an actual argument to any procedure, nor may it appear in the format specification of an input-output statement, nor may it appear as an item in the 'iolist' of an input-output statement.

Example of String Concatenation

'Left Side' // 'Right Side'

5.3 Relational Expressions

Relational expressions compare arithmetic expression values or character expression values. Relational expressions yield logical values. The relational operators are:

Operator	Meaning
.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to
.GT.	Greater than
.GE.	Greater than or equal to

5.3.1 Arithmetic Relational Expressions

An arithmetic relational expression expresses a relationship between arithmetic operands. The form of an arithmetic relational expression is:

$E1 \text{ relop } E2$

where 'E1' and 'E2' are arithmetic operands, and 'relop' is one of the operators selected from the table above.

Only the .EQ. (equality) and .NE. (inequality) operators are allowed for operands of complex type.

If the operands are of different types, the relational expression is treated as if it were in the form:

$(E1) - (E2) \text{ relop } 0$

where 0 (zero) is the same type as the expression.

Comparison of a double precision value and a complex value is not allowed.

5.3.2 Character Relational Expressions

A character relational expression is of the form:

$E1 \text{ relop } E2$

where 'E1' and 'E2' are character expressions and 'relop' is one of the relational operators selected from the table above. The ordering of character expressions is as defined in the ASCII character set table in the appendices. The .EQ. (equality) and .NE. (inequality) operators do not use the ordering. If the operands in a character relational expression are of different lengths, the shorter operand is considered to be padded on the

right with spaces until the operands are of the same length.

5.4 Logical Expressions

A logical expression operates on values of type logical and generates a result of type logical. The simplest forms of logical expressions are:

- . logical constant,
- . logical variable reference,
- . logical array element reference,
- . logical function reference,
- . relational expression.

Other logical expressions are built up from these simple forms by using parentheses and the logical operators as follows:

Operator	Meaning
.NOT.	Logical Negation
.AND.	Logical Conjunction
.OR.	Inclusive Disjunction
.EQV.	Logical Equivalence
.NEQV.	Logical Nonequivalence

5.4.1 Precedence of Logical Operators

The precedence of the logical operators is shown on the next page:

Operator	Precedence
.NOT.	Highest
.AND.	
.OR.	
.EQV. or .NEQV.	Lowest

The .AND. and .OR. operators are binary operators and must appear between their operands. The .NOT. operator is a unary operator and appears before its operand. Operators of equal precedence associate left to right.

A .AND. B .AND. C
 is equivalent to:
 (A .AND. B) .AND. C
 .NOT. A .OR. B .AND. C
 is equivalent to:
 (.NOT. A) .OR. (B .AND. C)

Two .NOT. operators must not appear adjacent to each other. The expression:

A .AND. .NOT. B

is an example of an allowable expression with two adjacent operators.

5.5 Precedence of Operators

When arithmetic, relational and logical operators appear in the same expression, their relative precedence is:

Operator	Precedence
Arithmetic	Highest
Relational	Intermediate
Logical	Lowest

5.6 Evaluation Rules and Restrictions for Expressions

Any variable, array element or function referenced in an expression must be defined at the time it is referenced. Integer variables must be defined with an arithmetic value rather than a statement label set by an ASSIGN statement. If a character string or substring is referenced in an expression, all the referenced characters should be defined at the time of the reference.

It is an error to divide by zero. It is also an error to raise a zero value to a zero or negative power. It is also an error to raise a negative value to a real or double precision power.

5.6.1 Restrictions on Function References

In any given statement, it is an error if a function reference within that statement changes any other object in the statement.

If a function reference causes an actual argument to the function to become defined, it is an error to reference that object anywhere else in the statement containing the function reference.

Chapter 6

Specification Statements

This chapter describes SVS FORTRAN specification statements. Specification statements are non-executable. They are used to define properties of user-defined variables, arrays and functions. There are nine types of specification statements:

- . Type statements,
- . DIMENSION statements,
- . COMMON statements,
- . PARAMETER statements,
- . IMPLICIT statements,
- . EXTERNAL statements,
- . INTRINSIC statements,
- . SAVE statements,
- . EQUIVALENCE statements.

Specification statements must precede all executable statements in a subprogram unit. If any IMPLICIT statements appear in the subprogram, they must precede all other specification statements. Other than that, specification statements can appear in any order within their own group.

6.1 Type Statements - Declaring Data Types

Type statements specify the data type of user-defined names. A type statement either confirms or overrides the default type rules for names. Type statements can also convey dimension information when declaring arrays. A user-defined name for a variable, array, formal argument, external function or statement-function can appear in a type statement. Such an appearance defines the type of that name for the entire program unit that contains the type statement. In any given program unit, a user-defined name may only appear once in a type statement.

A type statement can confirm the type of an intrinsic function, but it is not required to do so. A main program name or a subroutine name must not appear in a type statement. A type statement can define the dimensions of an array, or the dimensions can be declared in a DIMENSION statement (see below), independently of the type statement.

6.1.1 Arithmetic Type Statements

Arithmetic type statements are used to declare arithmetic data objects. The form of an arithmetic type statement is:

```
type var [, var] ...
```

'type' is one of INTEGER, INTEGER*1, INTEGER*2, INTEGER*4, REAL, REAL*4, REAL*8, DOUBLE PRECISION or COMPLEX.

INTEGER and INTEGER*4 are the same. REAL and REAL*4 are the same. DOUBLE PRECISION and REAL*8 are the same.

'var' is a variable name, array name, formal argument name, function name or array declarator. See the definition of array declarators in the chapter on "FORTRAN Names, Arrays and Substrings".

Examples of Arithmetic Type Statements

```
C declare some integer variables.
C
C INTEGER CLOCK, HANDS(2), TIME(24)
C
C declare some real and double precision variables.
C
C REAL RADIO, K101, VARBLS(10, 10, 5)
C DOUBLE PRECISION TWOS(50), TWICE, SECOND
C
C declare some complex data items
C
C COMPLEX FUNKS, ROCK, BACH(48)
```

6.1.2 CHARACTER Type Statement

The character type statement is used to declare CHARACTER data objects. The form of a CHARACTER type statement is:

```
CHARACTER [*nnn [,]] var [*nnn] [, var [*nnn]] ...
```

'var' is a variable name, array name, formal argument name or an array declarator. For a definition of an array declarator, see the chapter on "FORTRAN Names, Arrays and Substrings".

'nnn' is the length, in characters, of a character variable or character array element. The length must be an unsigned integer in the range 1 to 255 or a constant expression enclosed in parentheses, whose value lies in the range 1 to 255. The length can also be specified as (*), when the name is being defined either as a formal argument which is an assumed size character string, or for the purpose of establishing a type for later use in a PARAMETER statement.

The length 'nnn', following the type name CHARACTER, is the default length for any name in the list that does not have its length specified explicitly. In the absence of a length specification, the default length is one (1). A length immediately following a variable or array element overrides the default length for that item only. For an array, the length specifies the length of each element of that array.

A formal argument defined as CHARACTER*(*) cannot be used as an actual argument to a procedure if it is concatenated in a character string expression, whereas a symbolic name of a constant can be used in such a place.

Examples of CHARACTER Type Statements

```
CHARACTER FLIP*10, FLOP*20
CHARACTER WILD(15)*20
CHARACTER*80 LINE(24)
CHARACTER*(10*20) LSTR
CHARACTER* (*) VARBLE
```

6.1.3 LOGICAL Type Statement

The logical type statement is used to declare logical data objects. The form of a LOGICAL type statement is:

```
type var [, var] ...
```

'type' is one of LOGICAL, LOGICAL*1, LOGICAL*2 or LOGICAL*4.

LOGICAL is the same as LOGICAL*4.

'var' is a variable name, array name, formal argument name, function name, or an array declarator. For a definition of an array declarator, see the chapter on "FORTRAN Names, Arrays and Substrings".

Examples of LOGICAL Type Declarations

```
LOGICAL      SONG
LOGICAL*2    BLACK, WHITE
LOGICAL      YES(10), NO(10)
```

6.2 DIMENSION - Declare Data Dimension

A DIMENSION statement specifies the number of dimensions of a user-defined array. The form of a DIMENSION statement is:

```
DIMENSION var(dim) [, var(dim)] ...
```

where each one of the 'var(dim)' pairs is an array declarator of the form:

```
name(d [,d] ...)
```

'name' is the user-defined name of the array,

'd' is a dimension declarator.

6.2.1 Dimension Declarators

The number of dimensions in the array is the number of dimension declarators in the array declarator. The maximum number of dimensions is seven. The rules for array and dimension declarators are defined in the chapter on "FORTRAN Names, Arrays and Substrings".

Examples of DIMENSION Statements

```
DIMENSION forth(10, 5:15, 0:99)
```

```
DIMENSION axis(6)
```

6.3 COMMON - Declare a COMMON Block

Common blocks provide a means to share variables between multiple independently-compiled program units. Common blocks and their contents are defined via the COMMON statement. The form of the COMMON statement is:

```
COMMON [/cname]/ nlist [[,] /cname]/ nlist...
```

'cname' is a common block name. If any 'cname' is omitted, the blank common block is implied.

'nlist' is a list of variable names, array names and array declarators, all separated by commas. Formal argument names and function names must not appear in a COMMON statement.

In each COMMON statement, all variables and arrays appearing in each 'nlist', following a common block name, are declared to be in that common block. If the 'cname' is omitted, all elements appearing in the 'nlist' are specified to be in the blank common block.

Any common block name can appear more than once in COMMON statements in the same subprogram unit. All elements in all 'nlists' for the same common block are allocated storage, sequentially in that common block, in the order of their declaration.

All elements in a single common area must be all of type character or none of type character.

The size of a common block is equal to the number of bytes of storage needed to hold all elements in that common block. If the same named common block is referenced in several subprogram units, the size must be the same in all those units.

Examples of COMMON Statements

```
COMMON /horde/  TOKEN(100), SYMBOL(100)
```

6.4 PARAMETER - Make a Symbolic Association

A PARAMETER statement associates a symbolic name with a constant value. That constant is thereafter associated with that symbolic name, such that using the name is synonymous with a use of the constant. The form of a PARAMETER statement is:

```
PARAMETER (name=expr [,name=expr] ...)
```

'name' is the symbolic name to be defined,

'expr' is an expression that is to be associated with the name. The expression noted in the definition above must be a constant expression.

Examples of PARAMETER Statements

```
PARAMETER (TODAY = 'FRIDAY')
C
PARAMETER (BASE = 1, LIMIT = 100)
```

6.5 IMPLICIT - Establish Default Data Type

FORTRAN normally assigns a default type to a variable depending on the first letter of that variable. The IMPLICIT statement overrides the default type rules and establishes a new default type for variables. The form of the IMPLICIT statement is:

IMPLICIT type (letter-list) [, type (letter-list)] ...

- 'type' is one of the data types: INTEGER, INTEGER*1, INTEGER*2, INTEGER*4, REAL, REAL*4, REAL*8, LOGICAL, LOGICAL*1, LOGICAL*2, LOGICAL*4, DOUBLE PRECISION, COMPLEX or CHARACTER[*nnn]
- 'letter-list' is a list of single letters or ranges of letters. A range of letters is indicated by the first and last letters in the range, separated by a minus sign. If a range is specified, the letters must be in alphabetical order.
- 'nnn' is only applicable to a character data type, and is the size of the character type that is to be associated with that letter or letters. 'nnn' must be an unsigned integer in the range 1 thru 255. If 'nnn' is not specified, a value of one (1) is assumed.

An IMPLICIT statement defines the type and size for all user-defined names that begin with the letter or letters appearing in the specification. An IMPLICIT statement only applies to the program unit in which it appears. IMPLICIT statements do not change the type of any intrinsic functions.

Implicit types can be overridden or confirmed for any specific user-defined name if that name appears in a subsequent type statement. An explicit type in a FUNCTION statement also takes precedence over an IMPLICIT statement. If the type in question is a character type, the length is also overridden by any later type specification.

A program unit can have more than one IMPLICIT statement, but all IMPLICIT statements must precede all other specification statements.

Examples of IMPLICIT Statements

```
*   declare all names beginning with A as integer.
*
*   IMPLICIT INTEGER (A)
*
*   declare all names starting with the letters
*   Q, X, Y or Z to be complex.
*
*   IMPLICIT COMPLEX (Q, X-Z)
*
*   declare all names starting with C as CHARACTER.
*
*   IMPLICIT CHARACTER*255 (C)
```

6.6 EXTERNAL - Declare External or Dummy Procedure

An EXTERNAL statement specifies that a user-defined name is the name of an external procedure or of a dummy procedure. It also allows such a name to be used as an actual argument to a subroutine or function reference. The form of an EXTERNAL statement is:

```
EXTERNAL proc-name [, proc-name] ...
```

where each 'proc-name' is the name of an external procedure, dummy procedure or block data subprogram. A name appearing in an EXTERNAL statement declares that name to be an external procedure.

Statement-function names must not appear in an EXTERNAL statement.

If an intrinsic function name appears in an EXTERNAL statement, that name becomes the name of an external procedure and the corresponding intrinsic function can no longer be called from that program unit.

A user-defined name can only appear once in an EXTERNAL statement.

6.7 INTRINSIC - Declare Intrinsic Function

An INTRINSIC statement declares that a name is an intrinsic function. It also allows a specific intrinsic function name to be used as an actual argument to a subroutine or function reference. The form of an INTRINSIC statement is:

```
INTRINSIC name [, name] ...
```

where 'name' is an intrinsic function name.

Each name may appear only once in an INTRINSIC statement. If a name appears in an INTRINSIC statement, it may not appear in an EXTERNAL statement.

All names used in an INTRINSIC statement must be system-defined intrinsic functions. For a list of intrinsic functions, see the appendix on "Intrinsic Functions".

If a specific name of an intrinsic function is used as an actual argument in a program unit, that name must be declared in an INTRINSIC statement in that program unit.

If a generic function name appears in an INTRINSIC statement, that function still retains its generic properties.

In a given program unit, a name must not appear in more than one INTRINSIC statement.

Certain intrinsic functions may not be used as actual arguments. These are:

- . The type-conversion functions: INT, IFIX, IDINT, FLOAT, SNGL, REAL, DBLE, CMLPX, ICHAR and CHAR.
- . The lexical relationship functions: LGE, LGT, LLE and LLT.
- . The functions for choosing largest or smallest values: MAX, MAX0, AMAX1, DMAX1, AMAX0, MAX1, MIN, MIN0, AMIN1, DMIN1, AMIN0 and MIN1.

6.8 SAVE - Retain Definition Status

A SAVE statement is used to retain the definition of a program object after returning from the procedure which defines that program object. Within a subroutine or function subprogram, a program object specified in a SAVE statement remains defined after exit from the subroutine or function. The form of a SAVE statement is:

```
SAVE [thing [, thing] ... ]
```

where 'thing' is a common block name enclosed in slashes, a variable name or an array name. Any given name may only appear once in a SAVE statement. The names of dummy arguments, procedures and objects appearing in common blocks must not appear in a SAVE statement.

If a SAVE statement appears without an associated list of program objects, it is the same as if all objects in that program unit which could appear in the SAVE statement actually had appeared in the SAVE statement.

Specifying a common block name in a SAVE statement is the same as saving all the elements in that common block. A common block mentioned in a SAVE statement must be mentioned in a SAVE statement in every subprogram in which that common block appears. A SAVE statement has no effect in the main program, and is optional.

Examples of SAVE Statements

```
C
C   Save everything in the subprogram with
C
C   SAVE
C
C   Save some variables
C
C   SAVE dimes, nickels, pennies
C
C   Save all of common blocks
C
C   SAVE /Stamps/, /Lettrs/
```

6.9 EQUIVALENCE - Share Storage Between Elements

An EQUIVALENCE statement specifies that two or more variables or arrays are to share the same storage. If the shared variables are of different types, the EQUIVALENCE statement does not cause any kind of automatic type conversion. The form of an EQUIVALENCE statement is:

```
EQUIVALENCE (nlist) [, (nlist) ] ...
```

where 'nlist' is a list of at least two variable names, array names, array element names or character substring names. Argument names must not appear in EQUIVALENCE statements. Subscripts must be integer constant expressions and must be within the bounds of the array that they reference.

An EQUIVALENCE statement specifies that the storage sequences of the elements that appear in the list 'nlist' have the same first storage location. Two or more variables are said to be associated if they refer to the same actual storage. Thus an EQUIVALENCE statement causes its list of variables to become associated. If an array name appears in an EQUIVALENCE list, it refers to the first element of the array.

6.9.1 Restrictions on EQUIVALENCE Statements

An EQUIVALENCE statement must not specify that the same storage location is to appear more than once. For example:

```
REAL R, S(10)
EQUIVALENCE (R, S(1)), (R, S(5))
```

is in error because it forces the variable "R" to appear in two distinct memory locations, namely at S(1) and S(5).

An EQUIVALENCE statement must not specify that consecutive array elements be stored out of sequential order. For example:

```
REAL R(10), S(10)
EQUIVALENCE (R(1), S(1)), (R(5), S(6))
```

is in error because, having defined R(1) and S(1) to be associated, the statement then attempts to define R(5) and S(6) to be associated, and this means that the array "R" has somehow

been "stretched".

Names of dummy arguments must not appear in an EQUIVALENCE statement. Also, if a variable name is also a function name, that name must not appear in an EQUIVALENCE statement.

When EQUIVALENCE statements and COMMON statements are used together, there are further restrictions. An EQUIVALENCE statement must not try to associate storage elements in different common blocks. An EQUIVALENCE statement can extend a common block by adding storage elements following the common block, but not preceding the common block. For example:

```
COMMON /MASSES/ R(10)
REAL S(10)
EQUIVALENCE (R(1), S(10))
```

is in error because it tries to extend the common block by adding storage before the start of the block. That is, when R(1) and S(10) are associated, it means that S(1) would be nine locations before the defined start of the block.

Chapter 7

Data Initialization

The DATA statement is used to (statically) initialize data variables. The DATA statement is non-executable in the sense that the compiler does not generate any code for it.

If a DATA statement is present within a subprogram, it may appear anywhere after the specification statements (if there are any). The form of a DATA statement is:

```
DATA nlist /clist/ [[,] nlist /clist/] ...
```

'nlist' is a list of variables, arrays, array element names, substring names and implied-DO lists.

'clist' is a list of constants, or constants preceded by an integer-constant repeat-factor and an asterisk. Examples of repeated data items are:

```
5*3.14159      3*'Help'      100*0
```

There must be the same number of values in each 'clist' as there are variables or array elements in the corresponding 'nlist'. The appearance of an array in an 'nlist' is equivalent to a list of all the elements in that array in order of storage sequence. Array elements and substrings may be indexed by integer constant expressions (but see the implied-DO loop below).

The type of each element in a 'clist' must be the same as the type of the corresponding variable or array element in the accompanying 'nlist'. If necessary, the 'clist' constant is converted to the type of the 'nlist' object according to the rules for arithmetic conversion given in the table in the chapter on "Assignment Statements."

A DATA statement can be used to initialize any variable, array element or substring that is not one of the following:

- . a dummy argument,

- . an object in blank common or any object which is associated with an object in blank common,
- . a variable in a function subprogram whose name is also the same name as that of the function or one of its alternate entry-point names.

Objects may only be initialized once in any given program unit.

DATA statements in BLOCK DATA subprograms may only initialize objects in named COMMON areas.

Examples of DATA Statements

```

*
*   Declare some variables
*
*       REAL FIRST, SECOND
*       INTEGER NIG, NOG
*       COMPLEX WEIRD(10)
*       DOUBLE PRECISION VECT(5)
*
*   initialize some reals
*
*   DATA FIRST, SECOND /1.0, 2.0/
*
*   initialize some integers
*   DATA NIG /10/, NOG /20/
*
*   initialize two elements of the complex array
*
*   DATA WEIRD(2), WEIRD(5) /2 * (0.0, 0.0)/
*
*   initialize all the double precision array
*
*   DATA VECT /0.0, 0.0, 0.0, 0.0, 0.0/

```

7.1 Initializing Character Variables

If an 'nlist' item is of type character, the corresponding 'clist' item must be a character constant expression.

If the 'clist' item is shorter than the length of the 'nlist' item, the initial characters occupy the leftmost positions of the character data item and the remaining character positions are filled with space. If the 'clist' item is longer than its

corresponding 'nlist' variable, only the characters needed to initialize the 'nlist' item are used and the remaining characters are ignored.

Examples of Character Initialization

```
DATA STRING /'Old Rope'/  
DATA SVECT /6 * 'Attached'/
```

The second example assumes that SVECT is a 6 element character array.

7.2 Initializing Non CHARACTER Variables to CHARACTER Values

The FORTRAN-77 standard explicitly forbids initializing variables of any non-CHARACTER data type with CHARACTER values. Almost all FORTRAN-66 compilers do allow such initializations (they had to since there was no explicit CHARACTER data type in FORTRAN-66). SVS FORTRAN-77 therefore allows such initialization, provided the \$SHAREQU option has been selected. For a description of the \$SHAREQU option, see chapter 13 - "FORTRAN Compile Time Options".

When the \$SHAREQU compiler option is selected, non-CHARACTER variables may be initialized with CHARACTER constants. Each character constant in the 'clist' initializes precisely one variable in the 'nlist'. If the length of the CHARACTER constant is longer than the number of bytes which the target variable occupies, the CHARACTER constant is truncated on the right to the same size as the target variable. If the CHARACTER constant is shorter in length than the number of bytes which the target variable occupies, the CHARACTER constant is padded on the right with trailing spaces.

In all cases, one ASCII character is stored in each byte of the target variable. Thus a variable of type REAL*4 would receive exactly four bytes, and a variable of type LOGICAL*1 would receive exactly one byte.

Example of Initializing non-CHARACTER Variables

```

*      Select the option
$CHAREQU

*      Declare some variables
*
      LOGICAL*1 ARG
      DOUBLE PRECISION LARJ
*
*      Initialize the variables
*
      DATA ARG, LARJ /'SING', 'SONG'/

```

In the example above, the variable ARG would receive the single character value 'S' (because of truncation), while the variable LARJ would receive the value 'SONG ' (because of space padding).

7.3 Implied DO in DATA Statements

As an added convenience, the DATA statement can incorporate a form of DO loop for initializing arrays (for example) in a regular and concise way. This is known as an "implied-DO loop" and has the same form as a DO statement (see the chapter on "Control Statements"). The control variables of an implied-DO loop are declared implicitly and only for the duration of the DO loop. The following example should clarify the use of implied-DO loop initialization. In this case, the form of an implied-DO list in the DATA statement is:

```
(dlist, i = first, last [, inc])
```

'dlist' is a list of array element names and implied-DO lists.

'i' is the name of an integer variable, the implied-DO-variable.

'first', 'last' and 'inc' are each integer constant expressions. The expressions can contain implied-DO-variables of other implied-DO lists whose range includes this implied-DO list.

The range of an implied-DO list is the list 'dlist'. An iteration count and the values of the implied-DO-variable are established from 'first', 'last' and 'inc' just as for a DO loop, but the iteration count must be positive.

When an implied-DO list appears in a DATA statement, the items in 'dlist' are specified once for each iteration of the implied-DO list with the appropriate substitution of values for any occurrence of the implied-DO-variable 'i'.

The implied-DO-variable can have the same name as a variable in the subprogram unit containing the DATA statement - there is no conflict of such names.

Examples of Implied DO Loop Initialization

```

C   Declare some large arrays.
C
      INTEGER PRIMES(1000)
      INTEGER UPRTRI(20, 20)
      REAL MATRIX(25, 80)

C   Now initialize with DATA statements
C   containing an implied-DO loop.
C
      DATA (PRIMES(I), I = 1, 1000) /1000*1/
C
      DATA ((MATRIX(J, K), J = 1, 80), K = 1, 25) /2000*1.0/
C
C   The last DATA statement initializes the upper triangle
C   of the array called UPRTRI.
C
      DATA ((UPRTRI(I, J), J=1, 21-I), I= 1, 20) /210 * 0/

```


Chapter 8

Assignment Statements

An assignment statement computes a value which is then assigned to a program object. Because FORTRAN does not require that variables be declared ahead of time, it is possible that assignment actually causes that object to become allocated. There are four distinct types of assignment statements:

- . Arithmetic,
- . Logical,
- . Statement Label assignment (the ASSIGN) statement,
- . Character assignment.

8.1 Arithmetic Assignment

Arithmetic assignment evaluates an arithmetic expression, and assigns the result to a variable. The form is:

variable = expression

'variable' is a variable or an array element name, of type INTEGER, INTEGER*1, INTEGER*2, INTEGER*4, REAL, REAL*4, REAL*8, DOUBLE PRECISION or COMPLEX.

'expression' is an expression compatible with one of those types.

If the type of 'variable' and the type of 'expression' are not compatible, the value of 'expression' is automatically converted to the type of 'variable' according to the following table.

Type of variable or array element	Value Assigned
Integer	INT(expression)
Real	REAL(expression)
Double Precision	DBLE(expression)
Complex	CMPLX(expression)

Table 8-1
Type Conversion for Arithmetic Assignment Statements

The functions in the "Value Assigned" column in the above table are generic intrinsic functions described in the table of intrinsic functions in the appendices.

If an integer expression is of type INTEGER*1, INTEGER*2 or INTEGER*4, the result is automatically converted to the correct type. In general, an operand is "promoted" to the larger size when evaluating expressions. Operands of type INTEGER*1 are always promoted to INTEGER*2.

In the assignment statement, converting a longer INTEGER value to a shorter one is done by truncation. The FORTRAN-77 compiler does not issue any warning about it, and the running program issues no error messages because of truncation.

Examples of Arithmetic Assignment

```

INTEGER whole
REAL party
DOUBLE PRECISION huge
COMPLEX house

*
DATA whole /10/, party /5.5/
*
whole = 50 * party
party = 3.1415926388 * 5.7
huge = 547.987654e243
house = (1.0, -4.0)

```

8.2 Logical Assignment

Logical assignment assigns the value of an expression to a logical variable. The value of the expression must therefore evaluate to either of the values `.TRUE.` or `.FALSE.` The form of a logical assignment statement is:

```
logical variable = logical expression
```

If a logical variable is one of the types `LOGICAL*1`, `LOGICAL*2` and `LOGICAL*4`, a logical expression is automatically converted to the correct type.

Examples of Logical Assignment Statements

```
LOGICAL TELLME, NONO  
TELLME = .TRUE.  
NONO = .FALSE.
```

8.3 Statement Label Assignment

Statement label assignment is used to assign the value of a format label or a statement label to an integer variable. The form of statement label assignment is:

```
ASSIGN statement-label TO integer-variable  
'statement-label'  
    is a format label or a statement label.  
'integer-variable'  
    is an integer variable. The integer variable  
    must not be of type INTEGER*1 or INTEGER*2.
```

Executing an ASSIGN statement sets the integer variable to the value of the label. The label can be either a format label or a statement label. The label must appear in the same program unit as the ASSIGN statement. When used in an assigned GO TO statement, a variable must currently have the value of a statement label. When used as a format specifier in an input-output statement, a variable must have the value of a format statement label. The ASSIGN statement is the only way to assign the value of a label to a variable.

Examples of Statement Label Assignment

```
INTEGER SLAB, FORLAB
ASSIGN 666 TO SLAB
ASSIGN 905 TO FORLAB
```

8.4 Character Assignment

Character assignment evaluates a character expression and assigns the result to a character variable, character array-element or a character substring. The form of a character assignment statement is:

character-variable = character-expression

None of the character positions being defined in the left-hand side of the assignment may appear on the right-hand side of the assignment. If they do, the results are undefined.

The left-hand side and the right-hand side of the assignment may be of different lengths. If the left-hand side is longer, the effect is to extend the right-hand side value to the right with spaces until it is the same length as the left-hand side. If the left-hand side is shorter, the effect is to take a substring starting at position 1 of the right-hand side, short enough to assign to the left-hand side.

Only as much of the right-hand side need be defined as is necessary to define the left-hand side. For example, consider the following program fragment:

```
CHARACTER fred(4), bill*8
fred = bill
```

The assignment of `bill` to `fred` above requires that the substring `bill(1:4)` be defined, since that is enough to define `fred`. It is not required that the rest of `bill` - `bill(5:8)` - be defined.

If the left-hand side of the assignment is a substring reference, the right-hand side is assigned only to the substring. The definition status of the character positions not defined on the left-hand side does not change.

Chapter 9

Control Statements

Control statements are used to direct the sequence of execution of a FORTRAN program. Control statements include constructs to execute statements selectively depending on the outcome of a logical expression (the block IF and logical IF statements), perform blocks of statements repetitively (the DO statement), to select one of a number of statements to execute depending on the value of an integer expression (the computed GO TO statement, and to terminate or suspend program execution (the STOP and PAUSE statements). This chapter covers the control statements of FORTRAN, in this order:

Block IF THEN ELSE	a "structured coding" construct which was newly introduced to FORTRAN with FORTRAN-77,
Logical IF	which executes or does not execute a subordinate statement depending on the truth or falsity of a logical expression,
Arithmetic IF	which executes a three-way branch depending on the value of an arithmetic expression,
DO	which is FORTRAN's principal means of loop control,
CONTINUE	which acts as a "null" statement,
STOP	to stop program execution,
PAUSE	to suspend program execution,
Unconditional GO TO	which unconditionally transfers control to another part of the program unit,
Computed GO TO	which selects a statement label to execute, depending on the value of an expression,

Assigned GO TO which uses the value of an integer variable as a statement label.

9.1 Block IF THEN ELSE Statement

The block IF THEN ELSE statement group (described in the subsections below) represent "structured coding" constructs that control program execution flow without the need for indiscriminate jumping around via GO TO statements. As an overview of the subsections to follow, the three code skeletons below illustrate the basic ideas of the IF THEN ELSE statement groups.

Skeleton 1 - a simple block IF which skips a group of statements if the expression is false:

```
IF (I .LT. 10) THEN
    ..... some statements that are executed
    ..... only if I < 10
ENDIF
```

Skeleton 2 - block IF with a series of ELSEIF statements.

```
IF (J .GT. 1000) THEN
    ..... some statements executed only
    ..... if J > 1000
ELSEIF (J .GT. 100) THEN
    ..... some statements executed only
    ..... if J > 100 and ≤ 1000
ELSEIF (J .GT. 10) THEN
    ..... some statements executed only
    ..... if J > 10 and ≤ 100
ELSE
    ..... some statements executed only if
    ..... none of the above conditions were true
ENDIF
```

Skeleton 3 - shows that the constructs can be nested and that an ELSE statement can follow an IF block without intervening ELSEIF statements. The indentation is here to enhance readability - FORTRAN does not require it.

```

IF (I .LT. 100) THEN
  .. some statements executed only
  .. if I < 100
  IF (J .LT. 10) THEN
    .. some statements executed only
    .. if I < 100 and J < 10
  ENDIF
  .. some more statements executed only if
  .. I < 100
ELSEIF (I .LT. 1000) THEN
  .. some statements executed only
  .. if I > 100 and I < 1000
  IF (J .LT. 10) THEN
    .. some statements executed only
    .. if I > 100 and I < 10000 and J < 10
  ENDIF
  .. some more statements executed only if
  .. I > 100 and I < 1000
ENDIF
ENDIF

```

To provide a detailed understanding of the block IF and its associated statements, the concept of the IF-level is introduced. For any statement, its IF-level is:

$$n1 - n2$$

where 'n1' is the number of block IF statements (including the current statement) from the beginning of the current program unit and 'n2' is the number of ENDIF statements (not including the current statement) from the beginning of the current program unit. The IF-level of every statement must be greater than or equal to zero, and the IF-level of every block (IF, ELSEIF, ELSE and ENDIF) must be greater than zero. The IF-level of every END statement must be zero. The IF-level is used to define the nesting rules for the block IF and its associated statements and to define the extent of IF blocks, ELSEIF blocks, and ELSE blocks.

9.1.1 Block IF Statement

The general form of the Block IF statement is:

```
IF (logical expression) THEN
```

Execution of the block IF statement involves evaluating the 'logical expression'. If the value is true and there is at least one statement in the IF block, the next statement executed is the first statement of the IF block. After the last statement in the IF block is executed, the next statement to be executed is the

next ENDIF statement at the same IF-level as this IF statement.

If the value is true and there are no executable statements in the IF block, the next statement to be executed is the next ENDIF statement at the same IF-level as this IF statement.

If the value of the 'logical expression' is false, the next statement to be executed is the next ELSEIF, ELSE or ENDIF statement at the same IF-level as this IF statement.

Note that transfer of control into an IF block from outside the IF block is not allowed.

The block IF statement, in fact, looks no different from the logical IF statement described later; however, the presence of the THEN keyword as the next statement indicates that a structured block IF statement group follows.

9.1.2 ELSEIF Statement

The form of the ELSEIF statement is:

```
ELSEIF (logical expression) THEN
```

The ELSEIF block associated with an ELSEIF statement consists of the (possibly zero) executable statements up to, but not including, the next ELSEIF, ELSE or ENDIF statement that has the same IF-level as this ELSEIF statement.

Execution of an ELSEIF statement starts by evaluating the 'logical expression'. If the value is true and there is at least one statement in the ELSEIF block, the next statement executed is the first statement of the ELSEIF block. After the last statement in the ELSEIF block is executed, the next statement to be executed is the next ENDIF statement at the same IF-level as this ELSEIF statement.

If the value of 'logical expression' is true and there are no executable statements in the ELSEIF block, the next statement to be executed is the next ENDIF statement at the same IF-level as this ELSEIF statement.

If the 'logical expression' evaluates to false, the next statement to be executed is the next ELSEIF, ELSE or ENDIF statement that has the same IF-level as this ELSEIF statement.

Note that transfer of control into an ELSEIF block from outside of that ELSEIF block is not allowed.

9.1.3 ELSE Statement

The form of an ELSE statement is:

ELSE

The ELSE block associated with an ELSE statement consists of the (possibly zero) statements that follow the ELSE statement, up to, but not including, the next ENDIF statement that has the same IF-level as this ELSE statement. The "matching" ENDIF statement must appear before any intervening ELSE or ELSEIF statements at the same IF-level. In other words, there may only be one ELSE statement in a block IF statement. There is no effect in executing an ELSE statement.

Note that transfer of control into an ELSE block from outside that ELSE block is not allowed.

9.1.4 ENDIF Statement

The ENDIF statement marks the end of a block IF group. The form of the ENDIF statement is:

ENDIF

There is no effect in executing an ENDIF statement. There must be a "matching" ENDIF statement for every block IF statement in a program unit in order to specify which statements are in a particular block IF statement.

9.2 Logical IF Statement

The logical IF statement evaluates an expression for logical TRUE or FALSE, then either executes or does not execute a following statement based on the truth or falsity, respectively, of the expression. The form of the logical IF statement is:

IF (logical expression) statement

where 'logical expression' is a logical expression, and 'statement' is any executable statement except a DO statement, Block IF, ELSEIF, ELSE, ENDIF statement, an END statement, or another logical IF statement.

The 'logical expression' is evaluated, and if the value of the expression is true, the 'statement' is executed. If the 'logical expression' evaluates to false, the 'statement' is not executed and the execution sequence proceeds as if a CONTINUE statement had been encountered.

Note that functions in the 'logical expression' can affect objects in the 'statement'.

Example of Logical IF Statement

```
IF (Token (1) .NE. 32) RETURN
```

9.3 Arithmetic IF Statement

The arithmetic IF statement performs a GO TO to one of three statement labels depending on the value of an expression being negative, zero, or positive. The form of the arithmetic IF statement is:

```
IF (expression ) s1, s2, s3
```

The 'expression' in the description must be an INTEGER, INTEGER*1, INTEGER*2, INTEGER*4, REAL, REAL*4, REAL*8 or DOUBLE PRECISION expression.

's1', 's2' and 's3' are the statement labels of executable statements that appear in the same program unit as does the arithmetic IF statement itself. The same statement label can appear more than once among the three labels.

The effect of the arithmetic IF statement is to evaluate 'expression' and select a label based upon the value.

The statement labelled by 's1' is executed if the value of the expression is less than zero.

The statement labelled by 's2' is executed if the value of the expression is zero.

The statement labelled by 's3' is executed if the value of the expression is greater than zero.

The next statement executed is the statement labelled by the selected label. None of the labels may appear within the range of a DO loop or inside an IF, ELSEIF or ELSE block, unless the

arithmetic IF statement itself is also in the same range or block.

Example of the Arithmetic IF Statement

```

IF (I - 100) 20, 30, 40
C
  20 PRINT *, 'I is less than 100'
     STOP
C
  30 PRINT *, 'I is exactly 100'
     STOP
C
  40 PRINT *, 'I is more than 100'
     STOP

```

9.4 DO Statement - Loop Control

A DO statement block is used for "loop control" purposes such as applying some operation to a each element of an aggregate. The form of a DO statement is:

```
DO s [,] i=e1, e2 [, e3]
```

's' is the statement label of an executable statement. The label must follow this DO statement and be contained in the same program unit.

'i' is an integer, real or double precision variable, called the DO-variable.

'e1', 'e2' and 'e3' are each an integer, real or double precision expression.

The statement labelled by 's' is called the "terminal statement" of the DO statement. The terminal statement must not be an unconditional GO TO, assigned GO TO, arithmetic IF, block IF, ELSEIF, ELSE, ENDIF, RETURN, STOP, END or another DO statement. If the terminal statement is a logical IF, it may contain any executable statement except those not permitted inside a logical IF statement.

A DO statement is said to have a "range", beginning with the statement which follows the DO statement and ending with (and

including) the terminal statement. If a DO statement appears in the range of another DO loop, its range must be entirely contained within the range of the enclosing DO loop. If a DO statement appears within an IF block, ELSEIF block or ELSE block, the range of the associated DO loop must be entirely contained in the particular block. If a block IF statement appears within the range of a DO loop, its associated ENDF block must also appear within the range of the DO loop. More than one DO loop can have the same terminal statement.

The DO control variable, 'i', must not be set by any statement within the range of the DO loop associated with that control variable.

Transfer of control into the range of a DO loop from outside the range of a DO loop is not allowed.

9.4.1 DO Loop Initialization Sequence

Execution of a DO statement causes the following events to happen in order:

- . the expressions 'e1', 'e2' and 'e3' are evaluated, with conversion to the type of the DO variable if necessary, according to the rules specified for type conversion in the chapter on "Assignment Statements". If 'e3' is omitted, a default value of 1 is used. If 'e3' is present, it must not evaluate to zero,
- . the DO variable 'i' is set to the value of 'e1',
- . The iteration count for the DO loop is computed as:

$$\text{MAX}(\text{INT}((e2 - e1 + e3) / e3), 0)$$

which may be zero,

- . The iteration count is tested; if it exceeds zero, the statements in the range of the DO loop are executed.

9.4.2 DO Loop Incrementation Processing

After the terminal statement of the DO loop is executed, the following events occur in order:

- . The value of the DO control variable 'i' is incremented by the value of 'e3' which was computed when the DO statement was executed,

- . The iteration count is decremented by one (1),
- . The iteration count is tested; if it exceeds zero, the statements in the range of the DO loop are executed again.

The value of the DO control variable is well-defined, whether the DO loop is exited as a result of the iteration count being zero, or as a result of an explicit transfer of control out of the DO loop.

Example of the final value of a DO control variable:

```

C      This program fragment prints the number
C      1 thru 11 on the console

      DO 200 I = 1, 10
200    WRITE(*, '(I5)') I
      WRITE(*, '(I5)') I

```

9.4.3 Events Which Terminate a DO Loop

A DO loop is exited under the following conditions:

- . when its iteration count is zero, as described above under "DO Loop Execution Sequence",
- . a RETURN statement is executed within the range of the DO loop,
- . control is transferred to a statement in the same program unit but outside the range of the DO loop,
- . a subroutine called from within the range of the DO loop returns via an alternate return specifier to a statement which is outside the range of the DO loop,
- . the program terminates for any reason.

9.5 CONTINUE Statement - Null Statement

The CONTINUE statement is a "null" or "no operation" statement. It can appear just as any other statement in a program. CONTINUE has no effect on program execution. CONTINUE is almost always used as the final statement in a DO block, when the DO block would otherwise end in a statement that is disallowed in that context. The form of a CONTINUE statement is:

CONTINUE

9.6 STOP Statement - Stop Program Execution

The form of a STOP statement is:

```
STOP [n]
```

where 'n' is either a character constant or a string of not more than five digits. The effect of executing a STOP statement is to cause the program to terminate. If the argument 'n' is present on the STOP statement, it is displayed on the console upon termination.

9.7 PAUSE Statement - Suspend Program Execution

The form of a PAUSE statement is:

```
PAUSE [n]
```

where 'n' is either a character constant or a string of not more than five digits. The effect of executing a PAUSE statement is to cause the program to be suspended until there is an indication from the console that the program should proceed. If the argument 'n' is present on the PAUSE statement, it is displayed on the console as part of a prompt, requesting input from the console. If the indication is received from the console, program execution resumes as if a CONTINUE statement had been executed.

9.8 Unconditional GO TO Statement

The unconditional GO TO causes an unconditional transfer of control to a specific labelled statement. The form of the unconditional GO TO statement is:

```
GO TO statement label
```

The 'statement label' which is the target of the GO TO statement must be defined within the same program unit as the GO TO statement. The 'label' must be the statement label of an executable statement. Transfer of control into a DO, IF, ELSEIF or ELSE block from outside such a block is not allowed.

9.9 Computed GO TO Statement

A computed GO TO statement acts as a means of transferring control to one out of a set of labelled statements, depending on the value of an expression. The form of the computed GO TO statement is:

```
GO TO (s [,s] ...) [,] i
```

where 'i' is an integer expression and each 's' is a statement label of an executable statement in the same program unit as the computed GO TO statement. The same statement label may appear more than once in the list of statement labels.

The effect of the computed GO TO statement is to evaluate the integer expression indicated by 'i' to a value, say n . Control is then transferred to the n 'th statement label in the list, counting from 1. If the value of n is less than 1, or if the value of n is greater than the number of statement labels in the list, the computed GO TO statement has no effect, and program execution proceeds as if a CONTINUE statement had been executed.

The same restrictions on transfer of control apply to the computed GO TO statement as those that apply to the unconditional GO TO statement.

Example of Computed GO TO Statement

```
WHERE = 3
GO TO (100, 200, 300, 400, 500), WHERE
..... execution continues here when 5 < WHERE < 1
C
100 .... statements executed if WHERE = 1
C
200 .... statements executed if WHERE = 2
C
300 .... statements executed if WHERE = 3
C
400 .... statements executed if WHERE = 4
C
500 .... statements executed if WHERE = 5
```

9.10 Assigned GO TO Statement

The assigned GO TO statement uses the value of an integer variable as a statement label which is to be the target of a GO TO statement. The effect is as if an unconditional GO TO had been made to that statement label. The form of the assigned GO TO is:

```
GO TO i [[,] (s[, s] ...)]
```

where 'i' is the name of an integer variable and each 's' is a statement label of an executable statement in the same program unit as the assigned GO TO statement. The same statement label may appear more than once in the list of statement labels in an assigned GO TO statement.

At the time the assigned GO TO is executed, the integer variable 'i' must be defined with the value of a statement label of an executable statement which appears in the same program unit as the assigned GO TO statement. That variable must have been defined with an ASSIGN statement. The integer variable 'i' must be of type INTEGER or INTEGER*4. It must not be of type INTEGER*1 or INTEGER*2.

If the optional, parenthesized list of statement-labels is present, the value of the integer variable 'i' must be that of one of the statement-labels in the list. The same restrictions apply to the assigned GO TO statement as those that apply to the unconditional GO TO statement.

Example of Assigned GO TO Statement

```

ASSIGN 645 TO SYSTEM
GO TO SYSTEM, (360, 370, 635, 645, 1108)
.....
645      ..... statements starting here will be executed
          because 'SYSTEM' has the value 645.
```

Chapter 10

Input and Output

This chapter and chapter 11 describe the FORTRAN input-output system. This chapter covers the basic concepts of input and output in FORTRAN. Chapter 11 describes the FORMAT statement. Topics covered in this chapter and chapter 11 are:

- . an overview of the input-output system. Covers the basics of the FORTRAN file system. Defines the ideas of records, units and various forms of file access,
- . a general coverage of the input-output system,
- . input-output statements are covered, with the exception of the FORMAT statement, which is covered in chapter 11.

10.1 Overview of the Input-Output System

This section introduces the basic terms and concepts of the FORTRAN input-output system. Most tasks related to input-output can be done without a full understanding of this section, so the reader can skip to the next section on first reading and use this section for subsequent reference purposes.

10.1.1 Records

A record is the building block of the FORTRAN input-output system. A record is a sequence of characters or a sequence of values. There are three forms of records:

- . Formatted records,
- . Unformatted records,
- . Endfile records.

A formatted record is a sequence of characters terminated by the character value corresponding to the "end-of-line" key on a terminal (character value 13, or 10, or both, depending on the particular operating system). Formatted records are interpreted consistently on input the same way that the underlying operating system and any text editor interprets characters. A formatted record therefore corresponds to the notion of a "line" from a device. Formatted files are normally transportable between different language processors and other text-processing applications.

An unformatted record is a sequence of values. The system does not alter or interpret such records in any way; neither is there any representation for an end-of-record as is the case with a formatted record. Unformatted files are generally not transportable between different language processors or computers because of differences in the external representations of data.

An endfile record has no physical existence in a file, but the underlying input-output system supplies an indication of one, as if there had been some actual record after the last record in a file.

10.1.2 Files

A FORTRAN file is a sequence of records. FORTRAN files are either External or Internal.

An external FORTRAN file is a file on a physical peripheral device or it is an actual peripheral device.

An internal FORTRAN file is a character variable which is to serve as the source or destination of some input-output action.

From here on, FORTRAN files and the files known to the underlying operating system and any text processors are simply called "files". The correct meaning is determined by the context. The OPEN statement provides the association between the two notions of files, and in most cases, there is no ambiguity after the file is opened - the two notions of a file being the same.

10.1.3 Properties of Files

A file which is being operated upon by a FORTRAN program has a number of properties which are described in the paragraphs below. File properties which are discussed below encompass:

- . Name of the file,
- . Position of the file,
- . Record format,
- . Access method.

10.1.3.1 File Name

A file can have a name. If the name is present, it is a character string identical to that by which it is known to the operating system. A file can have a name such as `/source/fourier.text`.

10.1.3.2 File Position

A file has the property of position which is usually established by the preceding input-output operation. There is the notion of the beginning-of-file, endfile, the current record, the preceding record, and the next record in the file. It is possible to be between records, in which case the next record is the successor to the previous record, and there is no current record. The position of the file after a sequential write is the endfile but not beyond the endfile record. Executing an ENDFILE statement positions the file beyond the endfile record. A READ statement executed at end-of-file (but not beyond the endfile record) positions the file beyond the endfile record. The user can trap reading of an endfile record via the END= option in a READ statement.

10.1.3.3 Formatted and Unformatted Files

An external file is opened as either formatted or unformatted. Internal files are always formatted. Formatted files consist entirely of formatted records. Unformatted files consist entirely of unformatted records. Formatted files have the structural properties of being a sequence of lines with end-of-line indicators (usually carriage-return).

10.1.3.4 Sequential and Direct Access Files

An external file is opened as either sequential access or direct access. Sequential files contain records in an order determined by the order in which they were written. Sequential files must not be read or written using the REC= option which specifies a position for direct-access input-output. The system attempts to extend sequential access files if a record is written beyond the old endfile, if there is enough room to do this on the external device.

Direct-access files can be written in any order (random access). Records in a direct-access file are numbered sequentially with the first records having the number one (1). All records in a direct-access file have the same length, specified when the file is opened. Each record in the file is uniquely referenced by its record number, specified at the time the record is written. Records can be written out of order, with holes in the sequence if desired. For example, records 9, 5 and 11 could be written in that order without writing the intermediate records. Once written, a record cannot be deleted, but a record can be rewritten with a new value. It is an error to read a record that has not yet been written, but the system can only detect this if the attempted read is to a record beyond the highest numbered record in the file. Direct-access files must reside on block-structured storage devices, such that a position in the file is meaningful. The system attempts to extend a direct-access file if a write is made to a position beyond the current highest numbered record in the file, but the success of this depends on the amount of space on the storage device.

10.1.4 Internal Files

Internal files provide a means for using the formatting capabilities of the input-output system to convert values to and from their external character representations, within FORTRAN's internal storage structures. That means that reading from a character variable converts the character values into numeric, logical or character variables. Writing to a character variable converts values into their external character representation.

10.1.4.1 Special Properties of Internal Files

An internal file is a character variable, character array element, character array or character substring.

A record of an internal file is a character variable, character array element or character substring.

If an internal file is a character variable, character array element or character substring, such a file has exactly one element, whose length is that of the character variable, character array element or character substring. If an internal file is a character array, each element of that array is a record of the file, with each record being the same length.

If less than an entire record is written by a WRITE statement, the remainder of the record is filled with spaces.

The position of an internal file is always at the beginning of file prior to executing any input-output statement. Only sequential, formatted input-output is allowed on an internal file. Only the READ and WRITE statements can reference an internal file. List-directed input-output is not allowed on an internal file.

10.1.5 Units

A unit is a means of specifying a file. A unit specified in an input-output statement is either of:

- . External unit specifier,
- . Internal unit specifier.

An external unit specifier is either a positive integer expression or the character * which stands for the Standard Input and Standard Output files for the running program. In most cases, external unit specifiers are bound to physical devices (or files on those devices) by name when the OPEN statement is executed. Once this binding of a value to a system file name has occurred, FORTRAN input-output statements refer to the unit number as a means of referring to the external object. Once opened, the external unit specifier is uniquely associated with the external device or file until an explicit CLOSE statement is executed or until the program terminates.

The only exception to the above rules is that unit specifier 0 is initially associated with the Standard Input and Standard

Output files for reading and writing and an explicit OPEN statement is not needed. The system interprets the character * as specifying unit 0.

An internal unit identifier is the name of a character variable, character array, character array element or character substring. Internal unit specifiers may only be used in the READ, WRITE and PRINT statements - they are not allowed in any of the auxiliary input-output statements such as OPEN, CLOSE, INQUIRE and so on.

10.2 General Discussion of the Input Output System

FORTRAN supplies a rich selection of possible file structures. At first there might seem to be a confusing array of choices. However, two sorts of files will probably suffice for most applications.

- The Standard Input and Standard Output files. These are referred to as unit 0 or by the character *. The Standard Input and Output files are sequential, formatted files. When reading from the Standard Input, the backspace and line-delete keys familiar to the user have their normal editing functions. The Standard Output file has the special property that it is possible to write partial lines to it (lines which are not terminated by a carriage-return) by using the "\" or "\$" edit descriptors. These edit-descriptors are described in the next chapter. Writing to any other unit does not have this property, even if that unit is explicitly bound to the Standard Output file by an OPEN statement.
- Explicitly-opened external, sequential, formatted files. These files are bound to a system file by name in an OPEN statement. These files can be processed by system editors and other text-processing utilities.

10.2.1 Pre-Connected Files

The FORTRAN run-time system preconnects unit 0 to the Standard Input and Standard Output files, as mentioned above. On those operating systems which support a standard error file, the FORTRAN run-time system also preconnects unit 1 to that stream. On those systems which do not support a standard error file, the FORTRAN run-time system preconnects unit 1 to the same place as the Standard Output.

10.2.2 Examples of Common Input Output Operations

Here is an example program reading and writing the sorts of files discussed in the paragraph above. The specific input-output statements are discussed in detail in the next section.

```

C      Copy a file containing three columns of integers,
C      each column being 7 characters wide. The user
C      supplies the name of the input file. The output
C      file is called OUT.TEXT. The first and second
C      columns are swapped.
C
      PROGRAM SWITCH
      CHARACTER*23 FNAME
      INTEGER FIRST, SECOND, THIRD
C
C      Prompt to the Standard Output by writing to *.
C
      WRITE(*, 900)
900  FORMAT('Input file name - '\)
C
C      Read the file name from the Standard Input by
      reading *.
C
      READ(*, 910) FNAME
910  FORMAT(A)
C
C      Use unit 3 for input - any unit but 0 will do.
C
      OPEN(3, FILE=FNAME)
C
C      Use unit 4 for output - any but 0 and 3 are ok.
C
      OPEN(4, FILE='OUT.TEXT', STATUS='NEW')
C
C      Read and write until end-of-file.
C
100  READ(3, 920, END=200) FIRST, SECOND, THIRD
      WRITE(4, 920) SECOND, FIRST, THIRD
920  FORMAT(3I7)
      GO TO 100
C
200  WRITE(*, 910) 'Done Copying'
      END

```

10.2.3 Less Common File Operations

The less common file structures are suitable for certain classes of applications. In general, the application areas are these:

Direct-access files are suitable for random access applications such as maintaining a database.

Unformatted files are more efficient in input-output overhead and in file space requirements. Unformatted access can be used if data is to be written and read by FORTRAN on the same processor.

The combination of direct-access and unformatted files is ideal for a database management facility to be accessed exclusively through the FORTRAN system.

If data is to be transferred without any system interpretation (especially if all 256 character combinations are needed), unformatted input output is necessary, since .TEXT files are limited to the printable character set. A good example of unformatted input-output usage is to control a device with a single-byte binary interface. In this situation, formatted input-output would interpret certain characters (such as carriage-return) and would change their meanings in certain ways.

Internal files are not input-output in the conventional sense but they provide valuable character string operations and conversion via a standard mechanism.

Formatted direct-access files require special care. FORTRAN formatted files try to comply with the operating system's rules for .TEXT files. The FORTRAN input-output system is able to enforce these rules for sequential files, but cannot always enforce them for direct-access files. Direct-access files are not necessarily valid .TEXT files, since any unwritten records leave "holes" which do not follow .TEXT file rules. Direct-access files do, of course, follow FORTRAN'S input-output rules.

A file opened in FORTRAN is either "old" or "new". There is no notion of "open for reading" as distinct from "open for writing". Therefore "old" (existing) files may be opened and written on, with the effect of changing an existing file. Similarly the same file may be alternately read from and written to, providing that no attempts are made to read beyond end-of-file or to read unwritten records in direct-access files.

A write to a sequential file effectively deletes any records which existed beyond the fresh record. Normally, when a device (such as Standard Input or Standard Output) is opened as a file, it makes no difference whether it is "new" or "old". With disk files, opening "new" creates a new temporary file. If that file is closed with the "keep" option, or the program terminates before a CLOSE is performed on that file, a permanent file is created with the name given when the file was opened. If a file existed previously with the same name, the prior version is deleted. If a file is closed using the "delete" option, the newly created temporary file is deleted and any previously existing file of the same name remains intact. Opening a disk file as "old" generates a run-time error if the file does not exist; if it does exist, any writes to that file change its contents.

10.2.4 Limitations of FORTRAN Input Output System

This subsection discusses specific limitations that FORTRAN's input-output system imposes.

10.2.4.1 Direct Files must be on Blocked Devices

The operating system underlying FORTRAN has two kinds of devices, namely blocked and sequential.

Sequential files can be considered a stream of characters with no explicit positioning; the only operations are reading and writing. The Standard Input and Standard Output are examples of sequential devices.

Blocked devices, such as disks, have the additional capability to seek to a specific position. Blocked devices can be accessed either sequentially or randomly and therefore can support direct-access files. Since there is no notion of seeking to a position on an unblocked file, FORTRAN prohibits direct access to sequential devices.

10.2.4.2 No Character Compression in Direct Files

Direct-access formatted files must not contain any character compression information.

10.2.4.3 BACKSPACE Only Applies to Files on Blocked Devices

BACKSPACE can only be performed on files associated with blocked devices, it cannot be applied to sequential devices.

10.2.4.4 Length Limitations on Formatted Records

Formatted records must not be greater than 512 characters in length, including the terminating carriage-return character.

10.2.4.5 BACKSPACE may not be used on Unformatted Sequential Files

It is not possible to apply the BACKSPACE statement to an unformatted sequential file, because such a file contains no indications of its record boundaries. In principle it is possible to place end-of-record marks in such a file format, but this conflicts with the notion of an unformatted file as a "pure" sequence of values. Any structure imposed upon unformatted sequential files would interfere with their most common application, which is direct control of external instruments.

10.2.4.6 Side Effects of Functions Used in Input Output Statements

During execution of any input-output statement, expression evaluation can reference functions. Any function so referenced must not execute any other input-output statement.

10.3 Elements of Input and Output Statements

This section describes the elements that comprise input and output statements that FORTRAN supports. The next section covers the individual input-output statements in detail.

FORTRAN input-output statements require certain arguments and parameters which specify sources and destinations of data transfer, as well as other aspects of the operation.

10.3.1 The Unit Specifier 'u'

The unit specifier indicates the unit to be used in input-output operations. The format of a unit specifier is:

```
[UNIT=] u
```

where 'u' is an external or internal unit identifier.

The unit identifier, 'u', can take one of these forms in an input-output statement:

* refers to the Standard Input or Standard Output files.

positive integer expression

refers to an external file whose unit number is that of the expression. Unit 0 is the same as * - the Standard Input or Standard Output files.

name

the name of a character variable, character array, character array element or character substring.

In all input-output operations, if the UNIT= keyword is omitted, the unit specifier, 'u', must be the first item in the list of specifiers.

10.3.2 The Format Specifier 'f'

The format specifier is used to designate format lists when performing formatted input-output. The form of the format specifier is:

```
[FMT=] f
```

where 'f' is the format identifier, and can take one of these forms in an input-output statement:

statement label refers to the FORMAT statement labelled by that statement label.

integer variable name

refers to the FORMAT label assigned to that integer variable using the ASSIGN statement.

character expression

the current value of the character expression is

the format specifier.

asterisk character "**"

the asterisk is used to specify list-directed formatting.

If the `FMT=` keyword is omitted in a list of specifiers, the format specifier, 'f', must be the second item in the list and the unit specifier, 'u', must be the first item in the list, with the `UNIT=` keyword omitted.

See chapter 11 for a complete description of the format list and the elements it may contain.

10.3.3 The Record Number 'rn'

The record number is used in direct-access input-output only. It specifies the number of the record to be read or written. The format of the record number specifier is:

```
REC=rn
```

where 'rn' is the record number. The record number must be a positive integer.

10.3.4 The End of File Exit Specifier

The end-of-file exit specifier designates a statement label at which execution is to start when an end-of-file condition occurs while reading from a file. The format is:

```
END=s
```

where 's' is a statement label in the same program unit as the `READ` statement.

10.3.5 The Error Exit Specifier

The error exit specifier designates a statement label at which execution is to start when any errors occur during execution of an input-output statement. The format of the error exit specifier is:

```
ERR=s
```

where 's' is a statement label in the same program unit as the input-output statement.

10.3.6 The Input Output Status specifier 'ios'

The input-output status specifier specifies an integer variable into which an input-output status value is placed when the current input-output statement completes. The format of the specifier is:

```
IOSTAT=ios
```

where 'ios' is the name of an integer variable or integer array element. The integer variable specified by 'ios' must be an INTEGER (or INTEGER*4). When an input-output statement containing this specifier terminates, 'ios' becomes defined with the following values:

zero (0) indicates that the input-output operation completed normally. There were no errors and end-of-file was not encountered,

negative value indicates that an end-of-file was encountered during a READ statement,

positive value indicates that an error condition occurred during the input-output statement.

10.3.7 The Input-Output List 'iolist'

The input-output list - 'iolist' - specifies the objects whose values are transferred by READ and WRITE statements. An 'iolist' can be empty. If an 'iolist' is present, it is a list of elements separated by commas. Items in the list consist of:

- . Input and output objects,
- . Implied DOLists.

These two forms of 'iolist' are discussed in the two paragraphs below.

10.3.7.1 Input and Output Objects

An input or output object can be specified in the 'iolist' of a READ, WRITE or PRINT statement and is one of these forms:

- . Variable name,
- . Array element name,

- . Character substring name,
- . Array name. This form is a means of specifying all the elements of the array, in the order in which they are stored internally,
- . On output only, any other expression except a character expression which concatenates an assumed length character string.

Example of Input and Output

```
WRITE (0, 100) 'Results are: ', widget, blivet(j, 4)
100 FORMAT (A, I5, F10.5)
```

10.3.7.2 Implied DO Lists

Implied-DO lists can be specified as items in the 'iolist' of READ, WRITE and PRINT statements and are of the form:

```
(dlist, i = e1, e2 [, e3])
```

where the 'dlist' is an 'iolist' as defined previously..

In a READ statement, the DO variable, 'i', (or an associated object) must not appear as an input list item in the embedded 'dlist', but can be read in the same READ statement outside of the implied-DO list.

The embedded 'dlist' is effectively repeated for each iteration of 'i' with appropriate substitution of values for the DO control variable 'i'.

Example of an Implied DO List

```
WRITE (0, 150) (jinx(i), i = 1, 100)
150 FORMAT (10I7)
```

In the example above, the variable 'i' iterates through 100 elements of the array 'jinx'. The format specified in the FORMAT statement causes the results to be placed 10 per line on the output.

10.4 The Specific Input and Output Statements

The following input-output statements are supported in the FORTRAN system. The possible form of each statement is specified first, with an explanation of the meanings of the following forms. Certain items are specified as required if they must appear in the statement and are specified as optional if they need not appear in the statement. Optional items normally result in a default which is indicated if the item is omitted.

10.4.1 OPEN Statement

The format of the OPEN statement is:

OPEN(open list)

where 'open list' is a list of specifiers as described below. The argument list, 'open list' must contain one unit specifier and may contain one of each of the other specifiers listed.

[UNIT=]u	'u' is the unit specifier. It must not be an internal unit specifier. If the UNIT= keyword is omitted, the 'u' argument must be first in the argument list to OPEN.
IOSTAT=ios	'ios' is an input-output status specifier as defined above in "Elements of Input and Output Statements".
ERR=s	is an error exit specifier as defined above in "Elements of Input and Output Statements".
FILE=fname	The file name 'fname' is a character expression. If the 'fname' argument is omitted, OPEN opens a an anonymous file with a status of 'SCRATCH' (see below). This file is automatically deleted when CLOSE'd or upon program termination.
STATUS=sta	'sta' is the status of the file when opened. 'sta' is a character expression whose value must be one of 'OLD', 'NEW', 'SCRATCH' or 'UNKNOWN'. 'OLD' is the default for reading or writing existing files. 'NEW' may be used for writing

new files. If the 'OLD' or 'NEW' keywords are used, the FILE= argument must be supplied. If 'SCRATCH' is specified, the file is deleted when a CLOSE is performed on that file. If 'UNKNOWN' is specified, it is treated the same as if 'OLD' had been specified.

ACCESS=acc specifies the access mode for this file. 'acc' is a character expression whose value must be either 'SEQUENTIAL' or 'DIRECT'. 'SEQUENTIAL' is the default.

FORM=fm specifies whether the file is formatted or unformatted. 'fm' is a character expression whose value must be either 'FORMATTED' or 'UNFORMATTED'. 'FORMATTED' is the default.

RECL=rl The record length 'rl' is an integer expression. This argument to OPEN is for DIRECT-access files only and is required for that file type.

BLANK=blnk 'blnk' controls the default treatment of blanks (spaces) in formatted reads, which can be altered, in a particular formatted READ, by a BN or BZ edit-descriptor in a format specification. 'blnk' is a character expression whose value must be either 'NULL' or 'ZERO'. The default is 'NULL'. If 'NULL' is specified, all spaces are ignored in numeric input fields. If 'ZERO' is specified, all spaces other than leading spaces are treated as zeros.

BUFFERED='buffered'
The BUFFERED option selects buffered or unbuffered input or output on a unit in the system. If the option is 'BUFFERED', buffered input output is selected. If the option is 'UNBUFFERED', unbuffered input output is selected. Note that the operating system might override the option. Some operating systems show a substantial improvement in throughput if the 'BUFFERED' option is selected. All files are opened with the 'BUFFERED' option by default, except for the * unit and unit 1 (the standard error).

The OPEN statement binds a unit number with an external device, or file on an external device, by specifying its file name. If the file is to be a direct access file, the RECL=rl option specifies the length of records in that file. If the unit

specified in an OPEN statement is already open, it is closed before being bound to a file.

10.4.2 CLOSE Statement

The format of the CLOSE statement is:

```
CLOSE(close list)
```

where 'close list' is a list of specifiers. 'close list' must contain an external unit specifier and at most one of any of the other specifiers, which are as follows:

[UNIT=]u	'u' is the unit number of an external unit. If the UNIT= keyword is omitted, the unit specifier, 'u', must be the first specifier in the list.
IOSTAT=ios	'ios' is an input-output status specifier as defined above in "Elements of Input and Output Statements".
ERR=s	's' is an error exit specifier as defined above in "Elements of Input and Output Statements".
STATUS=sta	'sta' is an optional argument which applies only to files opened with STATUS='NEW'. 'sta' dictates the disposition of the file after it is CLOSE'd. 'sta' is a character expression whose value must be either 'KEEP' or 'DELETE'. In SVS FORTRAN-77, All files are closed with the KEEP option in effect.

CLOSE disconnects the specified unit and prevents input-output from being directed to that unit (unless the same unit number is re-opened, possibly bound to a different file or device). Files opened NEW are temporary files and are discarded if STATUS='DELETE' is specified. Normal termination of a FORTRAN program automatically closes all open files as if a CLOSE with STATUS='KEEP' had been specified. STATUS='KEEP' should not be specified for a file which was opened as 'SCRATCH'.

10.4.3 READ, WRITE and PRINT Statements

The READ statement transfers data into storage. The WRITE and PRINT statements transfer data from storage. All three statements have similar forms, as defined here:

```

READ (control list) [iolist]
READ f [, iolist]
WRITE (control list) [iolist]
PRINT f [, iolist]

```

where 'f' and 'iolist' are a format identifier and an input-output list as previously described in "Elements of Input and Output Statements". Note that the PRINT statement has no connection with "printing" on the system printer device (even if such a device exists).

The 'control list' is a list whose elements may be any of the following:

[UNIT=] u	is a unit specifier. If the optional UNIT= keyword is omitted from the specifier, the unit specifier, 'u', must be the first item in the list.
[FMT=] f	is a format specifier. If the optional FMT= keyword is omitted from this specifier, the format specifier, 'f', must be the second item in the list and the first item must then be the unit specifier, 'u', without the UNIT= keyword.
REC=rn	is a record number specifier. If this specifier is included in the list, the statement is a direct-access data transfer statement, otherwise the statement is a sequential access data transfer.
IOSTAT=ios	'ios' is an input-output status specifier as defined above in "Elements of Input and Output Statements".
ERR=s	's' is an error exit specifier as defined above in "Elements of Input and Output Statements".
END=s	is an end-of-file exit specifier as defined above in "Elements of Input and Output Statements". This specifier is only applicable to the READ statement.

If the 'control list' contains a format specifier, the statement is a formatted input-output statement, otherwise it is an unformatted input-output statement.

If the format identifier is an asterisk character, "*", the statement is a list-directed input-output statement. In this case, the record specifier must not appear in the 'control list'.

List-directed input-output must not be done on an internal file. If the unit specifier designates an internal file, the 'control list' must not contain a record specifier.

The end-of-file specifier must not appear in the 'control list' for a WRITE or PRINT statement.

The 'control list' must not contain both a record specifier and an end-of-file specifier.

10.4.4 File Positioning Statements

FORTRAN supplies three statements which position files explicitly:

BACKSPACE	to backspace a file by one record.
ENDFILE	to write an endfile record on a file.
REWIND	to position or re-position a file at its first record.

Each of the file positioning statements has two different forms:

BACKSPACE	u
BACKSPACE	(alist)
ENDFILE	u
ENDFILE	(alist)
REWIND	u
REWIND	(alist)

In each case, the 'u' is a unit number. The 'alist' is a parenthesized list of specifiers of the form:

[UNIT=j]u	'u' is a unit specifier for the unit. This argument is required. If the UNIT= keyword is omitted, the 'u' argument must be first in the list.
IOSTAT=ios	'ios' is an integer variable which is assigned the status of the specific input-output

statement. 'ios' is defined previously in "Elements of Input and Output Statements".

ERR=s 's' is a statement label to which control is passed if there are any errors in the input-output statement. The statement label 's' must appear in the program unit that contains the input-output statement. 's' is defined previously in "Elements of Input and Output Statements".

10.4.4.1 BACKSPACE Statement - Backspace a File

BACKSPACE positions the file connected to the specified unit, before the preceding record. If there is no preceding record, the file position is not changed. If the preceding record is the endfile record, the file is positioned before the endfile record.

A non-connected unit may be BACKSPACE'd without any error. In other words a BACKSPACE command issued against a non-connected unit has no effect.

BACKSPACE can only be issued on units which are sequential-formatted. BACKSPACE must not be applied to a sequential-unformatted file.

10.4.4.2 ENDFILE Statement - Write an Endfile Record

ENDFILE "writes" an endfile record as the next record on the file connected to the specified unit 'u'. The file is then positioned after the endfile record, so that further sequential data transfers are prohibited until either a BACKSPACE or a REWIND statement is executed.

10.4.4.3 REWIND Statement - Rewind a File

The REWIND statement positions the file associated with the unit 'u' to its initial point. The unit must be sequential.

A non-connected unit may be rewound without any error. In other words a REWIND command issued against a non-connected unit has no effect.

10.4.5 INQUIRE Statement - Obtain File Properties

The INQUIRE statement is used to obtain information about the properties of a particular named file or about the connection to a particular unit. There are two forms of INQUIRE, namely, inquire by file and inquire by unit.

The INQUIRE statement can be executed before, while or after a file is connected to a unit. Any values assigned as a result of INQUIRE are values which are current at the time the INQUIRE is executed. The two forms of the INQUIRE statement are:

INQUIRE(iflist) or INQUIRE(iulist)

where 'iflist', used for the inquire by file form, is a list of specifiers containing exactly one file specifier and 'iulist', used for the inquire by unit form, is a list of specifiers containing exactly one unit specifier. Both forms may then contain other specifiers as described below. The format of the file specifier is:

FILE=fin

where 'fin' is a character expression which, when trailing spaces are removed, is the name of the file which is the subject of the enquiry. The named file does not need to exist or to be connected to a unit.

The form of the unit specifier is as described previously in "Elements of Input Output Statements". The unit specified does not have to exist, nor be connected to a file. If the unit is connected to a file, the enquiry is being made about the connection and the file connected to it.

The remainder of the list in the INQUIRE statement is the inquiry specifiers. This is a list of one or more inquiry specifiers as described below. There may be only one of each inquiry specifier. Furthermore, each variable name may only appear once in the list of specifiers, in other words, the same variable name must not be given to more than one specifier.

The inquiry specifiers are summarized here and described in more detail below.


```

+-----+
| IOSTAT = ios
| ERR = s
| EXIST = ex
| OPENED = od
| NUMBER = num
| NAMED = nmd
| NAME = fn
| ACCESS = acc
| SEQUENTIAL = seq
| DIRECT = dir
| FORM = fm
| FORMATTED = fmt
| UNFORMATTED = unf
| RECL = rcl
| NEXTREC = nr
| BLANK = blk
+-----+

```

In every case where an integer variable is specified, it must be an INTEGER*4 (which is the same as INTEGER). In all cases where a logical variable is specified, it must be a LOGICAL*4 (which is the same as LOGICAL). The specifiers given here are as defined previously in "Elements of Input and Output Statements". The meaning of each of the inquiry specifiers is as follows:

IOSTAT=ios is an input-output status specifier. The variable 'ios' is set to zero (0) by the INQUIRE statement.

ERR=s 's' is a statement label to which control is passed if an error occurs. The INQUIRE statement never causes any error conditions.

EXIST=ex 'ex' is a logical variable. If a file with the specified name exists (in the FORTRAN milieu) 'ex' is set to .TRUE., else 'ex' is set to .FALSE. For an inquire by unit, 'ex' is set to true if and only if the unit actually exists. 'ex' always gets defined by the INQUIRE statement.

OPENED=od 'od' is a logical variable. If the specified file is opened (connected to a unit), 'od' is set to .TRUE., else 'od' is set to .FALSE. 'od' always gets defined by the INQUIRE statement.

NUMBER=num 'num' is an integer variable. 'num' is set to the the external unit number for the unit currently connected to the file. If the file is

not connected to a unit, 'num' is undefined.

- NAMED=nm**d If the file has a name, the logical variable 'nmd' is set to .TRUE., else 'nmd' is set to .FALSE.
- NAME=fn** 'fn' is a character variable that is set to the name of the file if the file has a name. If the file does not have a name, 'nmd' is undefined.
- ACCESS=acc** 'acc' is a character variable that is assigned the string 'SEQUENTIAL' if the file is connected for sequential access, and is assigned the value 'DIRECT' if the file is connected for direct access. If the file is not connected to a unit, the value of 'acc' is undefined.
- SEQUENTIAL=seq** 'seq' is a character variable which is assigned the value 'YES' if this file can be connected for sequential access. 'seq' receives the value 'NO' if the file cannot be connected for sequential access. If FORTRAN cannot determine what access methods are allowed for the file, 'seq' receives the value 'UNKNOWN'.
- DIRECT=dir** 'dir' is a character variable which is assigned the value 'YES' if this file can be connected for direct access. 'dir' receives the value 'NO' if the file cannot be connected for direct access. If FORTRAN cannot determine what access methods are allowed for the file, 'dir' receives the value 'UNKNOWN'.
- FORM=fm** 'fm' is a character variable which is assigned the value 'FORMATTED' if the file is connected for formatted input-output, and is assigned the value 'UNFORMATTED' if the file is connected for unformatted input-output. If the file is not connected, 'fm' is undefined.
- FORMATTED=fm**t 'fmt' is a character variable which is assigned the value 'YES' if this file can be connected for formatted input-output. 'fmt' receives the value 'NO' if the file cannot be connected for formatted input-output. If FORTRAN cannot determine if the file can be connected for formatted input-output, 'fmt' is assigned the value 'UNKNOWN'.
- UNFORMATTED=unf** 'unf' is a character variable which is assigned the value 'YES' if this file can be connected

for unformatted input-output. 'unf' receives the value 'NO' if the file cannot be connected for unformatted input-output. If FORTRAN cannot determine if the file can be connected for unformatted input-output, 'unf' is assigned the value 'UNKNOWN'.

- RECL=rcl** 'rcl' is an integer variable which is assigned the record length of the file connected for direct access. If the connection is not for direct-access or the file is not connected at all, 'rcl' is undefined.
- NEXTREC=nr** 'nr' is an integer variable which is assigned the number of the next record to be read or written on a direct-access file. If the file is connected but no data transfer has been done, 'nr' is assigned the value 1. If the file is not a direct-access file or if the position cannot be determined (possibly because of a previous error), 'nr' is undefined.
- BLANK=blnk** 'blnk' is a character variable which is assigned the value 'NULL' if the BN edit-descriptor is the default for blank control and is assigned the value 'ZERO' if the BZ edit-descriptor is the default for blank control (see the OPEN statement and the chapter on "Format Specifications"). If the file is not connected or if the file is connected but not for formatted input-output, 'blnk' is undefined.

10.4.6 Notes on the INQUIRE Statement

The INQUIRE statement never returns an error condition. The specifiers 'ex' (exists) and 'od' (opened) always get defined. When a specifier is said to be "undefined" in the descriptions above, it means that certain of the specifiers are meaningless in certain contexts. For example, if the access method for a unit is SEQUENTIAL, the 'rcl' (record length) and 'nr' (next record) specifiers have no meaning, and are thus said to be undefined (in the context that a program should not be examining those specifiers in such a case).

10.5 List Directed Input and Output

List-directed input-output serves to process formatted records without the use of a FORMAT statement. Values in the records are in a "free-form". List-directed input-output is convenient for those cases where the precise layout of the data is not important.

10.5.1 List Directed READ

The elements in the list-directed READ statement are the same as those for the READ statement previously described, with the exception that an asterisk character "*" is supplied as the format specifier.

Data is read into storage locations as specified in the 'iolist'. Input data consists of a number of values and value separators. The next paragraph describes value separators and the following paragraph describes values.

10.5.1.1 List Directed Value Separators

Value separators are used to delimit values in a list-directed input list. A value separator is one of the following:

- . a comma, with optional spaces on either side.
- . a slash, with optional spaces on each side.
- . one or more contiguous spaces between constants or after the last constant in the list.
- . end-of-record, which appears as a space between two constants.

A comma is used to separate values. Two commas in a row indicate a null value.

A slash has the effect that the remaining items in the input list are discarded and null values substituted.

A value separator adjacent to an end-of-record is not considered to be a null value.

10.5.1.2 List Directed Input Values

A value is a constant, a null (empty) value or one of the forms:

`r*c`

`r*`

where 'r' is a repeat factor which must be an unsigned non-zero integer constant. The form of 'r*c' represents 'r' successive appearances of the constant 'c'. The form 'r*' is the same as 'r' successive null values. Neither of these forms may contain embedded spaces except where 'c' is a character constant.

Individual values in list-directed input are in the forms described below. Values may not have embedded spaces, except character constants, where a space stands for itself. The acceptable forms of input values are:

Null A null value is indicated by the presence of two contiguous value separators, no characters or spaces preceding the first value separator in an input list, or by an 'r*' form. A null value cannot be used as the real or imaginary part of a complex constant but can be used as an entire complex constant. List items which are null have no effect on the definition status of variables in the corresponding 'iolist'. Variables in the 'iolist' which are already defined stay defined and variables which are not defined stay undefined.

Integers Must be the same form as for integer constants.

Real Numbers Any valid format for a FORTRAN real number. Furthermore the decimal point can be omitted from a real number, in which case the number is assumed not to have a fractional part.

Complex Numbers A complex value is represented by an ordered pair of real numbers in the format described above for real numbers. The pair of reals are separated from each other by a comma, and enclosed in parentheses. There can be spaces surrounding the numbers. An end of record can appear before or after the comma in a complex number.

Character string values

A string of characters enclosed in apostrophes. Embedded spaces are significant and are part of the constant. An apostrophe is represented by two juxtaposed apostrophes in the string. Character constants can span record boundaries. A record boundary in a character constant does not generate any gratuitous characters in the value. Character constants can be continued over as many records as needed. Note that unterminated character constants can lead to disastrous results.

If the character constant as read is longer than the length of the corresponding 'iolist' item, it is truncated to fit. If the character constant is shorter than the corresponding 'iolist' item, it is placed left justified in the variable and the remaining positions in the variable are filled with spaces.

Logical values Logical values are represented by a T (.TRUE.) or an F (.FALSE.). An optional period character "." may appear before the T or F character. The T or F can be followed by optional characters, but these characters must not be value separators - slashes or commas.

10.5.2 List Directed WRITE and PRINT

The list-directed WRITE and PRINT statements are similar to the formatted WRITE and PRINT statements described previously, with the exception that the FORMAT specifier is a *.

Data is transferred from the variables specified by the 'iolist' to the specified unit. In general, values are written on the output device in a manner consistent with list-directed input, but there are exceptions, the most notable being that character string data written out by a list-directed WRITE or PRINT statement cannot be re-read by a list directed input statement. FORTRAN starts new records when necessary. Values in the output are separated by three spaces, except that character values are not preceded or followed by any spaces. FORTRAN never generates slashes or null values on list-directed output. The forms of the different data types are as follows:

Integer values are generated as for an Iw edit-descriptor. 'w' is the minimum number of characters required to print the integer value.

Real and Double Precision

are generated with the effect of either an **F** edit-descriptor or an **E** edit-descriptor, depending on the magnitude of the numbers involved. The specific edit-descriptors used are as described in the table below.

Range of Number	Edit-Descriptor Used	
	Real	Double Precision
$1.0 \leq \text{val} < 10.0$	0PF9.6	0PF17.14
$\text{val} < 1.0$ or $\text{val} \geq 10.0$	1PE13.6E2	1PE22.14E3

Complex numbers are generated as a pair of real numbers enclosed in parentheses with a comma separating the real and imaginary parts.

Character string values

are generated as strings of characters. However, character constants are not surrounded by apostrophes, meaning that they cannot be read back in using list-directed input.

Logical values are generated as **T** for **.TRUE.** and **F** for **.FALSE.**

Chapter 11

Format Specifications

This chapter describes formatted input-output and the **FORMAT** statements available from **FORTRAN**. The reader is assumed to be familiar with the **FORTRAN** file system, units, records, access methods and input-output statements as described in the previous chapter.

A format specification is used in conjunction with formatted input-output to supply information which directs the editing or conversion between the internal representations of the machine and the representations of character strings in a file or character data item.

11.1 FORMAT Specifications and the FORMAT Statement

If a **READ**, **WRITE** or **PRINT** statement specifies a format, it is considered to be a formatted input-output statement as opposed to an unformatted input-output statement. To reiterate what was described in the previous chapter, a format can be specified by any of the following forms:

- . giving the label of a **FORMAT** statement,
- . the name of an integer variable containing the label of a **FORMAT** statement,
- . a character variable, character array element or character expression which has the correct form of a format specifier,
- . an asterisk character, **"**"**, indicating list-directed input-output.

The following examples illustrate valid ways to specify a format:


```

*           Example the first .....
*           reference to a FORMAT statement
*
  WRITE(*, 990) IGOR, JOHANN, KLUTZ
990 FORMAT(2I5, I3)
*
*           Example the second .....
*           assigned FORMAT label to integer
*
  ASSIGN 880 TO MORFAT
880 FORMAT(2I5, I3)
  WRITE(*, MORFAT) IVAN, JANUS, KLONG
*
*           Example the third .....
*           using a character variable
*
  CHARACTER*9 FORCH
  FORCH = '(2I5, I3)'
  WRITE(*, FORCH) IVOR, JACKO, KELP
*
*           Example the fourth .....
*           using a character expression
*
  CHARACTER*7 CHAREX
  DATA CHAREX /'2I5, I3'/
  WRITE(*, '(' // CHAREX // ')') IRENE, JANET, KLARG
*
*           Example the fifth .....
*           List-directed write
*
  WRITE(*, *) INEZ, JACKIE, KRON

```

The format specification itself must begin with a left parenthesis "(" and must end with a right parenthesis ")". Characters after a matching right parenthesis are ignored.

A FORMAT statement must have a label. Like all non-executable statements, a FORMAT statement may not be the target of a branching statement.

Between the "(" and ")" characters, the format specifications appear. The format specifications are a list of items, separated by commas. Each of the items in the format list is one of:

- [r] ed repeatable edit descriptors.
- ned non-repeatable edit descriptors.
- [r] fs a nested format specification. At most ten levels of nested parentheses are allowed within the outermost level.

where 'r' is an optional, non-zero, unsigned integer constant called a repeat specification. The comma separating two list items may be omitted if the resulting format is unambiguous, such as after a P edit descriptor or before or after a / edit descriptor.

The repeatable edit descriptors (described in more detail below) are:

Iw and Iw.m	for Integer editing
Fw.d	for Real editing
Ew.d and Ew.dEe	for Real editing
Dw.d	for Real editing
Gw.d and Gw.dEe	for Real editing
Lw	for Logical editing
A and Aw	for Character editing

The w and e are unsigned non-zero integer constants. The d and m are unsigned integer constants.

The non-repeatable edit descriptors (also described in detail below) are:

'xxxxxxx'	apostrophe editing
nHxxxxxxxxxxxx	Hollerith editing
Tc, TLc and TRC	Tabbing to column
nX	inserting spaces
/	starting a new record
\ or \$	inhibits starting new record
:	conditionally terminates format list
S, SS and SP	optional Sign control
kP	Scale factor editing
BN and BZ	Blank control

x is one of the characters from the character set that FORTRAN supports. n and c are unsigned, non zero integer constants. k is an optionally signed integer constant.

Note that the FORTRAN compiler performs as much checking of FORMAT statements as is possible at compilation time. There can however, be incorrect FORMAT statements which the compiler cannot detect, and such incorrect statements will not manifest themselves until the program is actually run.

11.2 Interaction Between Format Specifications and I/O List

Before going into the full details of how the various edit descriptors control the editing, it is necessary to describe how the format specification interacts with the input-output list ('iolist') in a given READ, WRITE or PRINT statement.

If an 'iolist' contains any items, at least one repeatable edit-descriptor must appear in the format specification. In particular, the empty edit specification "(" may only be used if there are no items in the 'iolist', in which case the only action the input-output statement performs is the implicit record skipping action associated with formatted input-output. Each item in the 'iolist' is associated in turn, with a repeatable edit-descriptor during the execution of the input-output statement. In contrast, the remaining non-repeatable format-control items interact directly with the record and are not associated with items in the 'iolist'.

Items in a format specification are interpreted from left to right. Repeatable edit-descriptors act as if they were present 'r' times - if 'r' is omitted it is treated as a repeat factor of 1. Similarly, a nested format specification is treated as if its list of items appears 'r' times.

Format specifications are interpreted at execution time. The term "format controller" is used here to describe the entity that interprets the list. The formatted input-output process proceeds as follows:

The format controller scans the format items in the order noted above. When a repeatable edit-descriptor is found, either:

- . a corresponding item appears in the 'iolist'. In this case the item and the edit-descriptor are associated and input or output of that item proceeds under format control of the edit-descriptor, or:
- . the format controller terminates the input-output process.

If the format controller encounters a colon edit-descriptor, ":" in the format list and there are no further items in the 'iolist', the format controller terminates input-output.

If the format controller encounters the matching final ")" of the format specification and there are no further items in the

'iolist', the format controller terminates input-output. If there are further items in the 'iolist', the file is positioned at the beginning of the next record and the format controller continues by re-scanning the format, starting at the beginning of the format specification terminated by the last preceding right parenthesis. If there is no such preceding right parenthesis, the format controller re-scans the format from the beginning. Within the portion of the re-scanned format, there must be at least one repeatable edit-descriptor. Should the re-scan of the format specification begin with a repeated, nested, format specification, the repeat factor indicates the number of times to repeat that nested format specification. The re-scan does not change the previously set scale-factor, BN or BZ blank control or S, SP or SS sign control.

When the format controller terminates, the remaining characters of an input record are skipped or an end-of-record is written on output, except as noted under the \ or \$ edit-descriptors.

11.3 Edit Descriptors

Edit-descriptors are used to specify a field of a record. A detailed description of the various edit-descriptors follows. The repeatable edit-descriptors appear first, followed by the non-repeatable edit-descriptors.

11.3.1 Repeatable Edit Descriptors

Repeatable edit-descriptors are associated with items from an 'iolist'. Repeatable edit-descriptors are concerned with the editing of numeric, logical, and character data items. These are described in the paragraphs to follow.

11.3.1.1 Numeric Editing

The I, E, F and G edit-descriptors are used for formatting integer, real and complex data. The following general rules apply to all of those edit-descriptors:

- On input, leading spaces are not significant. Other spaces are interpreted differently depending on the BN or BZ flag in effect, but all blank fields are always treated as the value zero (0). Plus signs are optional.

- . On input, with **E** and **F** editing, an explicit decimal point appearing in the input field overrides position of the decimal point as specified in the edit-descriptor.
- . On output, generated characters are right justified in the field with leading spaces if required.
- . On output, if the number of characters produced exceeds the field width, or the exponent exceeds its specified width, the entire field is filled with asterisks, but also see the next list item, concerning infinite and indeterminate values.
- . On output, a value of plus infinity fills the field with plus signs, a value of minus infinity fills the field with minus signs, and an indeterminate value fills the field with question marks.
- . Editing of complex numbers is controlled by two **E**, **F** or **G** edit-descriptors in succession, each of which controls the editing of "half" of the complex value. The two edit-descriptors for a given complex number do not have to be the same.

11.3.1.2 I - Integer Editing

The edit-descriptor **Iw** must be associated with an 'iolist' item which is of type **INTEGER** (which is the same as **INTEGER*4**), **INTEGER*1** or **INTEGER*2**. The field width is 'w' characters long. On input, an optional sign may appear in the field.

The **Iw.m** edit-descriptor specifies the field width 'w' as above, but the 'm' part specifies a minimum field width for the integer value.

On input, the 'm' specification has no effect.

On output, if the converted integer is shorter than 'm' characters, leading zeros are placed in the field. If 'm' is zero and the integer value to be formatted is also zero, the output field consists of 'w' spaces, regardless of any sign control in effect (see later). 'm' must not be greater than 'w'.

11.3.1.3 F - Real Editing

The edit-descriptor `Fw.d` must be associated with an 'iolist' item which is of type real, double-precision or one half of a complex number. The width of the field is 'w' characters. The fractional part is 'd' digits long.

The input field begins with an optional sign, followed by a string of digits optionally containing a decimal point. If the decimal point is present, it overrides the 'd' specified in the edit-descriptor. If the decimal point is not present, the rightmost 'd' digits of the string are interpreted as following the decimal point - leading spaces are converted to zeros if necessary. The number may be followed by an optional exponent which is either of the forms:

- . plus or minus followed by an integer, or
- . E or D, followed by zero or more spaces, followed by an optional sign, followed by an integer. E and D are treated identically.

The output field occupies 'w' digits, 'd' of which follow the decimal point. The output value is controlled both by the 'iolist' item and by the current scale-factor (see the paragraph later, on "Scale Factor Editing", in the discussion on non-repeatable edit-descriptors). The output value is rounded, not truncated.

11.3.1.4 E and D - Real Editing

The E or D edit-descriptors control formatting of real elements. These edit-descriptors must be associated with an 'iolist' item which is of type real, double-precision or one half of a complex number.

An E or D edit-descriptor takes one of the forms `Ew.d`, `Dw.d` or `Ew.dEe`. The forms `Ew.d` and `Dw.d` have identical editing effects. In each case, the field width is 'w' characters.

The 'e' has no effect on input. The input field for an E edit-descriptor is identical to that described by an F edit-descriptor which has the same 'w' and 'd' fields.

The form of the output field depends on the scale-factor (set by the P edit-descriptor) in effect. For a scale-factor of 0, the output field contains a minus sign (if needed), followed by a decimal point, followed by a string of digits, followed by an

exponent field for an exponent 'exp' of one of the following forms:

Ew.d -99 <= 'exp' <= 99
 E, followed by plus or minus, followed by the two digit exponent.

Ew.d -999 <= 'exp' <= 999
 plus or minus, followed by three digit exponent.

Ew.dEe -10^e-1 <= 'exp' <= 10^e-1
 E, followed by plus or minus, followed by 'e' digits of exponent with possible leading zeros.

The form **Ew.d** must not be used if the absolute value of the exponent to be printed exceeds 999.

The scale-factor controls decimal normalization of the printed 'E' field. If the scale-factor 'k' is in the range $-d < k \leq 0$, the output field contains exactly 'd'-'k' leading zeros after the decimal point and 'd'+k significant digits after that. If $0 < k < d+2$, the output field contains exactly 'k' significant digits before the decimal point and 'd'-'k'-1 places after the decimal point. Other values of 'k' are errors.

11.3.1.5 G - Real Editing

The G edit-descriptor is similar to the E and F edit-descriptors except that the G edit-descriptor provides for an "adaptable" output format depending on the magnitude of the number being converted - it gives the user a flexible choice of output formats without the need to pre-check on the size of the numbers ahead of time.

The G edit-descriptor must be associated with an 'iolist' item which is of type real, double-precision or one half of a complex number.

The G edit-descriptor takes one of the forms **Gw.d** or **Gw.dEe**. In either case, the final output field width is 'w' characters. The number of digits after the decimal point is 'd' digits unless a scale-factor greater than 1 is in effect.

On input, G editing acts the same as F editing (see above).

On output, the format of the converted number is dependent on its magnitude. If N is the number to be converted, the table below describes the action of the G edit-descriptor:

Magnitude of N	Equivalent Conversion
N < 0.1 or N > 10**d	Gw.d is the same as kPEw.d Gw.dEe is the same as kPEw.dEe
0.1 < N < 1	F(w-n).d, n('b')
1 < N < 10	F(w-n).(d-1), n('b')
⋮	⋮
10**(d-2) < N < 10**(d-1)	F(w-n).1, n('b')
10**(d-1) < N < 10**d	F(w-n).0, n('b')

where 'b' stands for a blank (space) in the above table, and 'n' is 4 for Gw.d editing and 'e'+2 for Gw.dEe editing.

11.3.1.6 L - Logical Editing

The edit-descriptor for a logical item is Lw, indicating that the field width is 'w' characters. The 'iolist' element associated with an L edit-descriptor must be of type LOGICAL (which is the same as LOGICAL*4), LOGICAL*1 or LOGICAL*2.

On input, the field consists of optional spaces, followed by an optional decimal point, followed by a T (for .TRUE.) or F (for .FALSE.). Any further characters in the field are ignored but accepted on input, so that the strings .TRUE. and .FALSE. are valid inputs.

On output, 'w'-1 spaces are followed by either the character T or F as appropriate.

11.3.1.7 A - Character Editing

The forms of the edit-descriptor for character items are A or Aw. The straight A format acquires an implied field width, 'w', from the number of characters in the 'iolist' item it is associated with. The 'iolist' item must be of type character to be associated with an A or Aw edit-descriptor.

On input, if 'w' equals or exceeds the number of characters in the 'iolist' element, the rightmost characters of the input field are used as the input characters, otherwise the input characters are left-justified in the input 'iolist' item, and trailing spaces are added.

On output, if 'w' exceeds the characters produced by the 'iolist' item, leading spaces are provided, otherwise the leftmost 'w' characters of the 'iolist' item are output.

It is also possible to read and write non-character items with the A edit-descriptor. When this is done, the variable is treated as if it were a CHARACTER variable whose length is the number of bytes that that variable occupies.

11.3.2 Non Repeatable Edit Descriptors

Non-repeatable edit-descriptors are format list items which are not associated with any 'iolist' items.

11.3.2.1 'xxx' - Apostrophe Editing

Apostrophe editing has the form of a character constant. It causes characters to be written from the enclosed characters, including spaces, of the edit descriptor itself. Within the string, two ' signs in a row are interpreted as one apostrophe. Apostrophe edit descriptors cannot be used on input.

11.3.2.2 H - Hollerith Editing

The nH edit-descriptor (called Hollerith editing) transmits the 'n' characters (including spaces) following the descriptor to the output. Hollerith editing cannot be used for input.

Examples of Apostrophe and Hollerith Editing

```

C      Each of the following WRITE statements writes
C      the characters
C      ABC'DEF
C      to the output.
C
C      WRITE(*, 970)
970  FORMAT('ABC''DEF')
C
C      WRITE(*, (('ABC''DEF'))
C
C      WRITE(*, (7HABC''DEF))
C
C      WRITE(*, 960)
960  FORMAT(7HABC'DEF)

```

11.3.2.3 X and T - Positional Editing

The **X** and **T** edit-descriptors described below have the effect of positioning the format controller within a record. They do not by themselves transmit any characters to or from a record.

When a formatted record is read on input, it is treated as if it were of infinite length, with as many trailing spaces as needed to satisfy input requests. Positioning using the **X** and **T** edit-descriptors determines the position of the next character to be read from the record. These edit-descriptors may therefore be used to skip portions of the input record or to re-read the same positions in the record more than once.

On output, it is as if the input-output system initially creates a record which is potentially of infinite length and filled with spaces. As formatted output transmits characters to the record, the final length of the record is determined by the rightmost position to which a character is transmitted. Changing the position with the **X** or **T** edit-descriptors does not directly affect the length of the record, but does affect the position at which the next character is transmitted to the output record. Using the **X** or **T** edit-descriptors, positions in the record may never have any characters transmitted to them (they are skipped), which means that those positions retain their original blank values, providing, of course, that characters are transmitted after the skipped positions so that those character positions are eventually included in the output record. It is also possible to overwrite positions of formatted output records using the **X** and **T** edit-descriptors by positioning to a place where data was previously written. In this case, only the last value written to a given character position becomes part of the final formatted record.

nX edit-descriptor advances the record position by 'n' spaces. If 'n' is omitted, the value 1 is used.

The **TC**, **TLC** and **TRC** edit-descriptors position the record to a specified column. The **TC** edit-descriptor positions the record to absolute column position 'c'. The **TLC** edit-descriptor moves the column position to 'c' characters to the Left (backwards), relative to the current position. The **TRC** edit-descriptor moves the column position 'c' characters to the Right (forwards) relative to the current position.

On input, the **T** edit-descriptors have the effect of skipping or allowing re-reading of portions of the input record. If the **TLC** edit-descriptor moves the character position to a place where input fields were previously transmitted, those items can be

re-transmitted.

On output, if a character is transmitted to a position where another character has already been transmitted, the earlier transmission is replaced.

11.3.2.4 Slash Editing - End of Transfer on Record

The slash character "/" indicates end of transfer on the current record.

On input, the file is positioned to the beginning of the next record.

On output, an end-of-record is written and the file is positioned to write at the start of the next record.

11.3.2.5 Backslash or Dollar Editing - Inhibit End of Record

The backslash "\ " and dollar "\$" edit-descriptors only apply when writing to the * device (the Standard Output).

Normally, when the format controller terminates a format list, data transmission to the current record ceases and the file is positioned so that a new WRITE starts a new record. If, while scanning the format list, the format controller finds a backslash character "\" or a dollar character "\$", the automatic end-of-record action is inhibited. This means that subsequent input-output statements can continue reading from or writing to the same record. The most common use for this mechanism is to prompt to the Standard Input and read a response on the same line. For example:

```
WRITE(*, '(\A)') 'Please type your weight -> '
READ(*, '(BN, I6)') LIGHT
```

The backslash or dollar edit-descriptor does not inhibit the automatic end-of-record generated when reading from the * unit. Input from the std-input must always be terminated by a carriage-return. This permits proper functioning of the backspace and line-delete keys.

11.3.2.6 Colon Editing - Conditional Termination

The `:` character appearing in a format list has the effect of terminating processing of the format list if there are no more items in the 'iolist'. If there are more items in the 'iolist' when the colon is encountered, the colon is ignored. There may be more than one colon in a format list.

The colon edit-descriptor is useful for terminating extraneous textual data that might otherwise be printed after all appropriate numeric items have been transferred. It is also useful for preventing further / edit-descriptors on input.

11.3.2.7 P - Scale Factor Editing

The `kP` edit-descriptor sets the scale-factor for subsequent `E`, `F` and `G` edit-descriptors until another `kP` edit-descriptor is encountered. At the start of each input-output statement, the scale-factor is zero (0). The scale-factor affects format editing as follows:

- . On input with `E`, `F` or `G` editing, providing that an explicit exponent does not appear, the externally represented number is equal to the internally represented number multiplied by 10^{**k} .
- . On input with `E`, `F` and `G` editing, the scale-factor has no effect if there is an explicit exponent in the input field.
- . On output with `E` editing, the real part of the quantity is multiplied by 10^{**k} and the exponent is reduced by 'k', effectively altering the column position of the decimal point, but not the actual output value.
- . On output with `F` and `G` editing, the externally represented number is equal to the internally represented number multiplied by 10^{**k} .

11.3.2.8 BN and BZ - Blank Interpretation

The `BN` and `BZ` edit-descriptors specify the interpretation of blanks (spaces) in numeric input fields.

The initial setting of this edit-descriptor is dependent on the `BLANK=` parameter to the `OPEN` statement when this file was opened. See the `OPEN` statement for a description of the `BLANK=` parameter.

If **BZ** editing is in effect, leading blanks are ignored and embedded blanks are treated as zeros. This edit-descriptor stays in effect until a **BN** edit-descriptor is encountered in the format list.

If **BN** editing is in effect, blanks in subsequent input fields are ignored until a **BZ** edit-descriptor is processed. The effect of ignoring blanks is to treat all the non-blank characters in the input field as if they were right justified in the field, with the number of leading blanks equal to the number of ignored blanks. In the example below, the **READ** statement treats the characters shown between the vertical bars as the value 123, where the term **<cr>** indicates a carriage-return:

```
READ(*, 100) I
100 FORMAT(BN, I6)
```

```
| 123      <cr> |
| 123    456<cr> |
| 123<cr> |
| 123<cr> |
```

Using the **BN** edit-descriptor in conjunction with the infinite blank padding at the end of formatted records makes interactive input very convenient.

11.3.2.9 S, SS and SP - Sign Control Editing

An output field generated by **I**, **D**, **E**, **F** or **G** editing includes an optional sign immediately preceding the digits of the value. The sign always appears if the number is negative, but if the number is positive, **FORTRAN** omits the plus sign.

At the start of a format list, **FORTRAN** opts to omit plus signs. An **S**, **SS** or **SP** edit-descriptor controls the option. Any option chosen remains in effect until another one is found in the format list.

An **SP** format code specifies that "optional" plus signs are always printed. An **SS** edit-descriptor specifies that they are to be suppressed always. An **S** option restores **FORTRAN**'s option to omit plus signs.

On input, these format codes have no effect and are ignored.

Chapter 12

Program and Subprogram Structure

A complete executable FORTRAN program consists of the following elements:

- . a main program,
- . any number of SUBROUTINE subprograms,
- . any number of FUNCTION subprograms,
- . any number of BLOCK DATA subprograms.

A subprogram is a program unit which begins with a SUBROUTINE, FUNCTION or BLOCK DATA statement. A subprogram is defined separately and can be compiled independently of the main program. Subroutine and function subprograms are procedures which can share values and results through argument lists, common blocks or both. A BLOCK DATA subprogram acts as a container for initializing common blocks.

A procedure can be a subroutine, external function, intrinsic function or statement-function.

12.1 Main Program

A main program is any program unit that does not start with a SUBROUTINE, FUNCTION or BLOCK DATA statement. A main program may start with a PROGRAM statement. Execution of a FORTRAN program starts with the first executable statement in a main program. Consequently there must be precisely one main program in every executable FORTRAN program. The form of a PROGRAM statement is:

```
PROGRAM progname
```

where 'progname' is a user-defined name of the main program.

The name 'programe' is a global name. Therefore it must not be the same as the name of another external procedure or the name of a common block. The name 'programe' is also local to the main program and must not be the same as any other local name in the main program. The PROGRAM statement may only appear as the first statement of a main program.

Example of a PROGRAM Statement

```
PROGRAM BESSEL
```

12.2 Access To Command Line Arguments

The SVS FORTRAN-77 system provides for access to the command line which called up the running program.

The FORTRAN-77 run-time library contains two routines which enable a FORTRAN-77 program to access its command line arguments.

IARGC (Argument Count) is an INTEGER FUNCTION which returns the number of arguments actually typed on the command line.

GETARG is a subroutine which returns a specified argument. The definition of GETARG is:

```
SUBROUTINE GETARG(ARGNUM, ARGCH)
  INTEGER ARGNUM
  CHARACTER *(*) ARGCH
```

ARGNUM is the number of the argument which is to be accessed from the command line. Arguments are numbered from 1 (not from 0 as on some operating systems). Indexing from 1 is done for compatibility with the Pascal numbering, and with FORTRAN-77's default lower array bounds. The value passed to ARGNUM must be in the range 1 through IARGC. If it is not, the results are undefined.

The first argument (argument 1) is the name of the program.

The receiving variable is treated just as it is in a character assignment statement in FORTRAN-77. If the source character string is shorter than the target variable, it is padded with spaces on the right. If the source character string is longer than the receiving variable, it is truncated.

Here is a short example of using the argument access facility to echo the command line to the standard output.

```
PROGRAM ECHO
INTEGER ARGC
CHARACTER *100 ARG
INTEGER I

DO 200 I = 1, IARGC()
    CALL GETARG(I, ARG)
200 WRITE(*, *) ARG
END
```

12.3 Formal Arguments and Actual Arguments

This section covers the relationship between formal arguments and actual arguments in function and subroutine subprograms. Throughout the discussion, the terms "formal argument" and "dummy argument" are synonymous.

A formal argument is the name (local to the subprogram) by which the argument is known during the execution of the subprogram.

The actual argument is the actual value (variable, expression, array and so on) passed to the subprogram in question at the time a caller references the subprogram.

There are a number of ways to pass values into and out of subprograms. One way is via common blocks. Another way is to use the argument mechanism of subroutines and functions. It is this second way that this section covers.

Arguments are used to pass values into and out of subprograms. The number of actual arguments must be the same as the number of formal arguments. The types of the corresponding formal and actual arguments must also agree.

12.3.1 Argument Association

On entry to a subroutine or function, the actual arguments become associated with the formal arguments. The association remains effective until execution of the subprogram terminates. Thus assigning a value to a formal argument while executing a subprogram can change the value of the corresponding actual argument. If an actual argument is a constant, function reference, or an expression other than a simple variable, assigning a value to the formal argument should not be done, and can lead to strange and hard to diagnose side effects..

A formal argument that is a variable can be associated with an actual argument that is a variable, an array element or an expression. The length attributes of integer and logical arguments must match exactly, that is, an actual argument of type `INTEGER*2` must only be associated with a formal argument of `INTEGER*2`.

Actual arguments which are integer expressions can only be associated with formal arguments of type `INTEGER` or `INTEGER*4`. Similarly, actual arguments which are logical expressions can only be associated with formal arguments of type `LOGICAL` or `LOGICAL*4`. An "expression" in this context is any construct which is not a variable, array or array element.

If an actual argument is an expression (anything not a variable, array, or array element), it is evaluated immediately prior to the association of actual and formal arguments. If an actual argument is an array element, its subscript expression is evaluated just prior to the association and remains constant during execution of the subprogram, even if the subscript expression contains variables that are re-defined while the subprogram executes.

A formal argument that is an array can be associated with an actual argument that is an array or an array element. The number and size of dimensions in the formal argument can differ from those of the actual argument, but any reference to the formal argument must be within the limits of the actual array's storage sequence. While the FORTRAN system cannot detect such an out-of-bounds reference, the results are generally unpredictable and undesirable.

A formal argument which is an asterisk character, `**`, may only appear in the argument list of a subroutine or in an `ENTRY` statement in a subroutine. The actual argument is an alternate return specifier in the subroutine `CALL` statement (see below).

Formal arguments which are arrays or character strings may have adjustable dimensions. This enables writing more general subprograms which can accept objects of varying size. A formal argument which is an array may have its dimensions specified by variables passed as actual arguments. This is an adjustable array. An array formal argument may also have the upper bound of its last dimension specified as an asterisk character, "*", which declares it to be an assumed size array. In this case, the value of that dimension is not passed as an actual argument, but is determined by the number of elements in the array. If an array is dimensioned as *, it is the programmer's responsibility to ensure that the calling program unit has provided an array big enough to contain all the elements stored into it in the subprogram.

Character strings may have their length specified as (*). This declares the string to be of varying size. The length of the string is not passed explicitly as an argument, but is determined by the system from the length of the actual argument.

A formal argument which is of type character must not be greater than the length of the actual argument. If the length of the actual argument is greater than that of the formal argument, the actual argument is truncated on the right.

If a formal argument is of type character whose length is specified as (*), a character expression involving concatenation of that argument must not be used as an actual argument to any other procedure, format specification or input-output list in an input-output statement.

12.4 Subroutines

A subroutine is a program unit that is called from other program units via the CALL statement. When invoked, the subroutine performs the actions that its executable statements define, and then returns control to the program unit that called it. A subroutine does not directly return a value, although values can be passed to the caller via the subroutine's arguments or via common variables.

12.4.1 SUBROUTINE Statement

A subroutine starts with a SUBROUTINE statement and ends with the first END statement that follows. A subroutine can contain any kind of statement except a PROGRAM statement, a FUNCTION statement or a BLOCK DATA statement. The form of a SUBROUTINE statement is:

```
SUBROUTINE subname [ ( farg [, farg] ... ) ]
```

'subname' is the user-defined name of the subroutine.

'farg' is a formal argument specification. A formal argument can be the user-defined name of a variable, array, dummy procedure, or it can be an alternate-return specifier designated by the asterisk character "*".

The subroutine name 'subname' is a global name. It is also local to the subroutine it names. The list of argument names defines the number (and with any subsequent IMPLICIT, type or DIMENSION statements) the type of arguments to that subroutine. Argument names must not appear in COMMON, DATA, EQUIVALENCE or INTRINSIC statements.

If a subroutine does not have any formal arguments, an empty argument list indicated by a pair of parentheses may follow the name, as shown in the examples below.

Examples of SUBROUTINE Statements

```
SUBROUTINE NOARGS
SUBROUTINE ZILCH()
SUBROUTINE ONEARG (RILEY)
SUBROUTINE ALTRET(LIMIT, *)
```

12.4.2 CALL Statement

A subroutine is executed when a CALL statement in another subprogram references that subroutines by name. The form of a CALL statement is:

```
CALL subname [ ( [arg [, arg] ... ] ) ]
```

'subname' is the name of the subroutine to call.

'arg' is an actual argument. The actual arguments are described below.

The actual arguments in the CALL statement must agree in type and number with the corresponding formal arguments specified in the SUBROUTINE statement of the referenced subroutine. Actual arguments may be one of the following:

- . An expression,
- . An array name,
- . An intrinsic function name,
- . An external procedure name,
- . A dummy procedure name,
- . An alternate-return specifier of the form *s, where 's' is the statement-label of an executable statement in the same program unit as the CALL statement.

If there are no arguments in the SUBROUTINE statement, a CALL statement that references that subroutine must not have any actual arguments. A pair of parentheses following the subroutine name is optional in the CALL statement. A formal argument can be used as an actual argument in another subroutine call.

Execution of a CALL statement proceeds as follows:

1. All actual arguments that are expressions are evaluated.
2. All actual arguments are then associated with their corresponding formal arguments.
3. The body of the specified subroutine is executed.
4. Control is returned to the subroutine's caller when a RETURN or and END statement is executed in the subprogram, either at the statement following the CALL statement, or at the alternate return specifier designated in a RETURN statement.

A subroutine specified in any program unit can be called from any other subprogram within the same executable program. Recursive subroutine calls are not allowed in FORTRAN. That is, a subroutine cannot call itself directly, nor may a subroutine

called by the current subroutine subsequently call the current subroutine.

12.5 Functions

A function is referenced in the context of an expression and returns a value that is used in the evaluation of that expression. There are three kinds of functions, namely: external functions, intrinsic functions and statement functions. The subsections to follow describe the three kinds of functions.

A function reference can appear in an expression. Referencing a function in the context of an expression causes that function to be executed. The resulting value that the function returns is used as an operand in the expression that references the function. The form of a function reference is:

```
funcname ( [arg [, arg]... ] )
```

'funcname' is the name of an external, intrinsic, or statement function.

'arg' is an actual argument to the function. The forms of the actual arguments are described below.

The number of actual arguments must be the same as the number of formal arguments. Except for generic intrinsic functions, the types of the actual arguments must agree with the types of the corresponding formal arguments. An actual argument can be any one of:

- . An expression,
- . An array name,
- . An external procedure name,
- . An intrinsic function name,
- . A dummy procedure name.

12.5.1 External Functions

An external function is specified by a function subprogram. It starts with a FUNCTION statement and ends with an END statement. A function can contain any kind of statement except a SUBROUTINE statement, PROGRAM statement or BLOCK DATA statement. The form of a FUNCTION statement is:

```
[type] FUNCTION function_name ( [farg [, farg]...] )
```

'type' defines the return type of the function. 'type' is one of INTEGER, INTEGER*1, INTEGER*2, INTEGER*4, REAL, REAL*4, REAL*8, DOUBLE PRECISION, CHARACTER[*len], LOGICAL, LOGICAL*1, LOGICAL*2, LOGICAL*4 or COMPLEX.

'function_name' is the user-defined name of the function.

'farg' is a user-defined name of the formal argument.

The function name 'function_name' is a global name. It must not be the same as the name of any other PROGRAM, SUBROUTINE, FUNCTION or BLOCK DATA subprogram. The function name is also local to the function it names.

If the 'type' specification is omitted from the function declaration, the function's type is determined by default and any subsequent IMPLICIT statements that would determine the type of an ordinary variable. If the 'type' specifier is present, the function name must not appear in any subsequent type statements.

The list of argument names determines the number (and with any subsequent IMPLICIT, type or DIMENSION statements) the type of the arguments to the function. Neither the argument names nor the function name may appear in any COMMON, DATA, EQUIVALENCE or INTRINSIC statements.

The function name must appear as a variable in the subprogram that defines the function. Each execution of the function must assign a value to that variable. The final value of this variable upon execution of a RETURN or END statement, defines the value of the function. After this variable is defined, its value can be referenced in an expression just like any other variable. In addition to the value returned, an external function can return values via assignment to one or more of its formal arguments or through variables in common areas.

A function which is declared as type CHARACTER*(*) derives its length from the specification (declaration) of the function in

the calling program unit.

12.5.2 Intrinsic Functions

Intrinsic functions are those functions that the FORTRAN compiler pre-defines. Intrinsic functions are available for use in a FORTRAN program. The table in the appendix on "Intrinsic Functions" gives the name, definition, number of arguments, and type of the intrinsic functions available in SVS FORTRAN-77. Those intrinsic functions which accept several types of arguments must have all those arguments of the same type in any given reference.

An intrinsic function can appear in an INTRINSIC statement, but only those intrinsic functions listed in the table in the appendix on "Intrinsic Functions" may do so. An intrinsic function may also appear in a type statement, but only if the type is the same as the standard type of that intrinsic function.

Certain intrinsic functions limit the range of their arguments in certain ways determined by the definition of the function being computed. For example, the logarithm of a negative number is mathematically undefined, and is therefore not allowed.

12.5.3 Statement Functions

A statement-function is a function that is defined by a single statement. It is similar in form to an assignment statement. A statement-function statement must appear in a subprogram after any specification statements and before any executable statements. A statement-function statement is not executable - rather the body of the statement-function statement serves to define the meaning of the statement-function. A statement-function is executed (in the body of the subprogram in which it is defined) by referencing it just like a function. The form of a statement-function statement is:

```
function_name ( [arg [, arg]... ] ) = expression
```

'function_name' is the name of the statement-function being defined.

'arg' is the user-defined name of the formal argument(s), if any.

'expression' is an expression that defines how the formal arguments are to be combined to generate a function result when the function is

referenced.

The type of the 'expression' must be assignment compatible with the type of the statement-function name. The list of formal argument names serves to define the number and types of arguments to the statement-function. The scope of the formal arguments is the statement-function. Therefore the formal argument names may be used as other user-defined names in the rest of the program unit that contains the statement-function statement. The name of the statement-function is local to the containing program unit, and therefore must not be used for any other purpose, other than as the name of a common block, or as the name of a formal argument to another statement-function statement. The type of all such other uses must be the same. If a formal argument to a statement-function statement is the same as a local name in the program unit, a reference to that name within the statement-function always refers to the formal argument, never to the other usage.

Within the 'expression', references to variables, formal arguments of the containing subprogram, other functions, array elements and constants, are all allowed. Statement-function references, however, must refer to statement-functions defined prior to the statement-function in which they appear. Statement-functions must not be called recursively, either directly or indirectly.

A statement-function can only be referenced in the subprogram in which it is defined. A statement-function name must not appear in any specification statement other than a type statement or a COMMON statement. If a statement-function name appears in a type statement, that name must not be defined as an array name. If a statement-function name appears in a COMMON statement, that name can only be the name of the common area.

12.6 ENTRY Statement

A subroutine or function subprogram has a primary entry-point which is established via the SUBROUTINE or FUNCTION statement which declares that program unit. A subroutine call or a function reference normally activates that subprogram at its primary entry-point, and the first statement which is executed is normally the first executable statement in the subprogram.

It is possible, however, to define alternate entry-points in a subroutine or function subprogram. These alternate entry-points are the start of sequences of statements which are different from

the sequence executed by entering the subprogram at its primary entry-point. In addition, such alternate entry-points can have formal argument lists which differ in number and type from those found in the primary entry-point, and from those of other ENTRY statements in the same subprogram. The format of the ENTRY statement is:

```
ENTRY entname [ ( farg [, farg] ... ) ]
```

'entname' is the user-defined name of the entry-point for the subroutine or function subprogram.

'farg' is a formal argument specification. A formal argument can be the user-defined name of a variable, array, dummy procedure, or, if the subprogram is a subroutine subprogram, it can be an alternate-return specifier designated by the asterisk character "*".

The entry-point name 'entname' is a global name. It is also local to the subprogram in which it appears. The list of argument names defines the number (and with any IMPLICIT, type or DIMENSION statements) the type of arguments to that subroutine. Argument names must not appear in COMMON, DATA, EQUIVALENCE or INTRINSIC statements.

If the entry-point name, 'entname' is in a function subprogram, the name can appear in a type statement.

An ENTRY statement must not appear within the body of an IF block or a DO block.

As with SUBROUTINE and FUNCTION statements, if there are no arguments to the ENTRY statement, an empty argument list can be supplied.

When a subprogram is referenced or called via an alternate entry-point, the actual arguments must agree in number, order and type with the formal arguments (except for subroutine names and alternate return specifiers which do not have a type).

12.6.1 Restrictions on the ENTRY Statement

An entry name must not appear as a dummy argument in a FUNCTION, SUBROUTINE or another ENTRY statement, and must not appear in an EXTERNAL statement.

In a function subprogram, the only place the entry-point name may be used prior to the ENTRY statement is in a type statement.

If a function subprogram is of type character, all entry-points to that function must also be of type character. If the length of the character function is specified as (*), all entry-points to that function must also have a length of (*), otherwise all entry-points must have the same length specification.

An argument in an ENTRY statement cannot appear prior to that ENTRY statement unless it:

- . is either a type statement,
- . is an argument in the SUBROUTINE or FUNCTION statement which begins the procedure containing the ENTRY statement,
- . appears in a prior ENTRY statement in the same procedure.

12.7 RETURN Statement

A RETURN statement returns control from a subprogram to the program unit which called it. A RETURN statement can only appear in a function or subroutine subprogram. The form of a RETURN in a function subprogram is:

```
RETURN
```

The form of a RETURN statement in a subroutine subprogram is:

```
RETURN [e]
```

Where the optional e is an integer expression.

Execution of a RETURN statement terminates the execution of the containing function or subroutine subprogram. If the RETURN statement is in a function subprogram, the value of the function is the current value of the variable with the same name as the function. If the function variable has not been assigned to prior to executing a RETURN or an END statement, the function value is undefined.

The RETURN statement is optional in a subprogram. Executing an END statement is equivalent to executing a RETURN statement.

If e is supplied on the RETURN statement, it indicates an alternate return from the subroutine. If e lies between 1 and 'n', where 'n' is the number of asterisks in the SUBROUTINE or ENTRY statement, the value of e selects the e'th asterisk from the formal argument list. Control then returns to the caller at the label specified by the e'th alternate return specifier.

If e is omitted, or if e lies outside the range 1 to 'n', the effect is to execute a normal return. Control then returns to the caller at the statement after the CALL statement that invoked the current subroutine.

12.8 Definition Status

When a RETURN statement or an END statement is executed in a subprogram, all objects within the subprogram become undefined, with the following exceptions:

- . Objects specified by SAVE statements.
- . Objects in blank common.
- . Anything in a named common block that appears in the current subprogram and also appears in at least one other subprogram that directly or indirectly references the current subprogram.
- . Initially-defined objects that have neither been re-defined nor become undefined.

If a named common block appears in the main program, anything in that common block does not become undefined.

12.9 BLOCK DATA Subprogram

A BLOCK DATA subprogram is a non-executable subprogram which is used to initialize the values of variables and array elements in named common areas. There may be more than one block data subprogram in a FORTRAN program, but if there is more than one block data subprogram, only one of them can be un-named. The format of a BLOCK DATA statement is:

```
BLOCK DATA [blockname]
```

where 'blockname' is the optional name of the block data subprogram.

The BLOCK DATA statement must appear as the first statement of the block data subprogram. The name, 'blockname', if present, must not be the same as any the name of any any external procedure, main program, common area or other block data

subprogram. The name, 'blockname' must not be the same as any local name in the subprogram.

A block data subprogram can contain type statements, IMPLICIT, PARAMETER, DIMENSION, COMMON, SAVE, EQUIVALENCE or DATA statements. A block data subprogram ends with an END statement.

More than one named common block can be initialized in the same block data subprogram. All the variables in a given named common block must be specified, even if they are not all initialized.

A given named common block may only be specified in one block data subprogram in the same executable program.

Examples of BLOCK DATA Subprogram

```

BLOCK DATA Whammo
*
*   Declare a common block with variables
*
COMMON /DRINKS/ Beer, Wine, Scotch
REAL Beer
COMPLEX Wine
DOUBLE PRECISION Scotch
*
*   Declare another common block with variables
*
COMMON /FOODS/ Burger, Dogs, Fries
LOGICAL Burger
REAL Dogs
COMPLEX Fries
*
*   Now initialize some of the variables.
*
DATA Beer /3.2/, Wine /(11.5, 1.5)/
DATA Burger /.TRUE./, Fries /(1.1, 2.8)/
*
END

```

In the example above, note that not all the variables were initialized.

12.10 The FORTRAN Intrinsic Functions

Intrinsic functions are those system supplied ("built-in") functions which are otherwise difficult to express in FORTRAN. An intrinsic function is supplied by FORTRAN. An intrinsic function returns a single value and is referenced in the same way as a user-defined function.

If a variable, array or statement-function is defined with the same name as that of an intrinsic function, the name is local to the program unit in which it is declared and the intrinsic function of that name is no longer available to that program unit.

If a function subprogram is defined which has the same name as that of an intrinsic function, use of that name references the intrinsic function, unless the name is declared as the name of an external function via the EXTERNAL statement.

Certain intrinsic functions are generic. In general, if a generic name exists, a generic name can be used in place of a specific name and permits greater flexibility than a specific name. Except for the type conversion functions, the type of the argument to a generic function determines the type of the result. For example, the generic function LOG computes the natural logarithm of its argument, which may be real, double precision or complex. The type of the result is the same as the type of its argument. The specific functions ALOG, DLOG and CLOG also compute the natural logarithm. ALOG computes the log of a real argument and returns a real result. Likewise, DLOG and CLOG accept double precision and complex arguments and return double precision and complex results, respectively.

Only the specific name can be used as an actual argument when an intrinsic function name is passed to a user-defined procedure or function.

The table in the appendix, "FORTRAN Intrinsic Functions", shows the intrinsic functions, their generic and specific names, their number of arguments and their argument types and result types.

Chapter 13

FORTRAN Compile Time Options

Compiler Directives are a SVS extension to ANSI FORTRAN. SVS FORTRAN compiler directives provide additional controls over the compiler's actions.

A compiler directive line is a line with a dollar sign \$ in column one. A compiler directive line can appear anywhere that a comment line can appear. Spaces are significant in compiler directive lines, where they delimit keywords and filenames.

Some of the compiler directives listed below are to make FORTRAN-77 cater to FORTRAN-66 features.

13.1 \$INCLUDE - Include Source File

\$INCLUDE filename

the file specified by "filename" is textually included in the program source, as if the actual contents of the included file had been written there.

Included files may be nested to a maximum depth of five.

13.2 \$XREF - Generate Cross Reference

\$XREF

generate a cross-reference listing at the end of each compiled subprogram.

13.3 \$SEGMENT - Designate Segment Name

`$SEGMENT [identifier]`

the generated object-code of subsequent procedures is placed into the segment named by 'identifier'. If the `$SEGMENT` directive appears without any 'identifier' field, the generated object-code is placed in a segment whose name is ' ' (eight spaces).

13.4 \$BIGCODE - Set Maximum Code Sizes

`$BIGCODE`

indicates that a procedure can be 32K bytes long. If `$BIGCODE` is not specified, the maximum length of a procedure is 16K bytes.

The maximum size of a single linked segment (a collection of procedures) is 32K bytes. This is a restriction on f77's internal code-generation scheme.

In addition, the maximum size of any subroutine or function is 16K bytes, unless the `$BIGCODE` option has been selected, in which case the limit is 32K bytes. But, the specific details of these restrictions are:

- . In any procedure, there may not be more than 16K bytes (32K bytes if `$BIGCODE`) of the `.text` plus `.data` segments. The `.text` is the executable code, and the `.data` is the initialized data areas.
- . Uninitialized data goes in the `.bss` area, but, if any of that data is initialized via a `DATA` statement, all that data ends up in the `.data` area. This is because of FORTRAN's rules that data declared in a procedure must be contiguous in storage.

13.5 \$COL72 - Restrict Source Lines to 72 Columns

\$COL72

indicates that source lines are to end in column 72. If this option is not specified, source lines can be up to 120 characters long. But, the ANSI FORTRAN-77 standard restriction of a maximum of 1360 characters per statement still applies. This corresponds to 20 lines of 66 columns.

13.6 FORTRAN-66 Compatibility Options

The FORTRAN compiler accepts options which change features of the language in a manner compatible with FORTRAN-66. These options are listed in the following paragraphs.

13.6.1 \$F66DO - Implement FORTRAN-66 DO Loops

If the \$F66DO option is used, DO loops always execute at least once.

13.6.2 \$CHAREQU - Character and Numeric Data Equivalence

The \$CHAREQU option means that CHARACTER and numeric data can now be assigned to the same COMMON areas. Using this option, CHARACTER and numeric data can also be EQUIVALENCE'd.

In addition, the CHAREQU option indicates that non-CHARACTER variables can be initialized with CHARACTER data constants via the DATA statement. See chapter 7 - "Data Initialization" for details.

13.6.3 \$INT2 - Make Integers 16-Bits

If the \$INT2 option is used, the INTEGER data type is INTEGER*2 by default, although all the length specifications are still available if explicitly used in specification statements.

If the \$INT2 option is used, LOGICAL variables default to LOGICAL*1. Just as for INTEGER, all the length specifications for LOGICAL are still available if explicitly used in specification statements.

It should be noted that these last two features conflict with the "storage unit" standards of FORTRAN-77, but are useful nevertheless.

Note: Although the \$INT2 option changes the default size of INTEGER and LOGICAL variables, the FORTRAN system still expects to see 4-byte variables in those contexts where an INTEGER*4 (or a LOGICAL*4) is required. For example, the assigned GO TO statement still expects a 4-byte variable as the subject of the ASSIGN statement and any GO TO statement which references that variable.

Appendix A

Messages from the FORTRAN System

A.1 Compile-Time Error Messages

- 0 Unknown error
- 1 Fatal error reading source block
- 2 Non-numeric characters in label field
- 3 Too many continuation lines
- 4 Fatal end-of-file encountered
- 5 Labeled continuation line
- 6 Missing field on \$ compiler directive line
- 8 Unrecognizable \$ compiler directive
- 9 Input source file not a valid text file format
- 10 Maximum depth of include file nesting exceeded

- 11 Integer constant overflow
- 12 Error in real constant
- 13 Too many digits in constant
- 14 Identifier too long
- 15 Character constant extends to end of line
- 16 Character constant is zero length
- 17 Illegal character in input
- 18 Integer constant expected
- 19 Label expected
- 20 Error in label

- 21 Type name expected (INTEGER[*n], REAL[*n], DOUBLE PRECISION,
COMPLEX, LOGICAL[*n], or CHARACTER[*n])
- 22 INTEGER constant expected
- 23 Extra characters at end of statement
- 24 '(' expected
- 25 Letter IMPLICIT'ed more than once
- 26 ')' expected
- 27 Letter expected
- 28 Identifier expected
- 29 Dimension(s) required in DIMENSION statement
- 30 Array dimensioned more than once

- 31 Maximum of 7 dimensions in an array
- 32 Incompatible arguments to EQUIVALENCE
- 33 Variable appears more than once in a type specification statement
- 34 This identifier has already been declared
- 35 This intrinsic function cannot be passed as an argument
- 36 Identifier must be a variable
- 37 Identifier must be a variable or the current FUNCTION name
- 38 '/' expected
- 39 Named COMMON block already saved
- 40 Variable already appears in a COMMON block

- 41 Variables in different COMMON blocks cannot be EQUIVALENCE'd
- 42 Number of subscripts in EQUIVALENCE statement does not agree with variable declaration
- 43 EQUIVALENCE subscript out of range
- 44 Two distinct cells EQUIVALENCE'd to the same location in a COMMON block
- 45 EQUIVALENCE statement extends a COMMON block in a negative direction
- 46 EQUIVALENCE statement forces a variable to two distinct locations, not in a COMMON block
- 47 Statement number expected
- 48 Mixed CHARACTER and numeric items not allowed in same COMMON block
- 49 CHARACTER items cannot be EQUIVALENCE'd to non-CHARACTER items
- 50 Illegal symbols in an expression

- 51 Cannot use SUBROUTINE name in an expression
- 52 Type of argument must be INTEGER or REAL
- 53 Type of argument must be INTEGER, REAL or CHARACTER
- 54 Types of comparisons must be compatible
- 55 Type of expression must be LOGICAL
- 56 Too many subscripts
- 57 Too few subscripts
- 58 Variable expected
- 59 '=' expected
- 60 Size of EQUIVALENCE'd CHARACTER items must be the same

- 61 Illegal assignment - types do not match
- 62 Can only call SUBROUTINES
- 63 Dummy arguments cannot appear in COMMON statements
- 64 Dummy arguments cannot appear in EQUIVALENCE statements
- 65 Assumed-size array declarations can only be used for dummy arrays
- 66 Adjustable-size array declarations can only be used for dummy arrays
- 67 Assumed-size array dimension specifier, "*", must be the upper bound of the last dimension
- 68 Adjustable bound must be either a dummy argument or be in COMMON prior to appearance

- 69 Adjustable bound must be simple integer expression containing only constants, COMMON variables, or PARAMETER constant names
- 70 Cannot have more than one main program
- 71 The size of a named COMMON block must be the same in all subprograms
- 72 Dummy arguments cannot appear in DATA statements
- 73 Variables in blank COMMON cannot appear in DATA statements
- 74 Names of SUBROUTINES, FUNCTIONS, INTRINSIC FUNCTIONS and such cannot appear in DATA statements
- 75 Subscripts out of range in DATA statement
- 76 Repeat count must be integer value greater than zero
- 77 Constant expected
- 78 Type conflict in DATA statement
- 79 Number of variables does not match the number of values in DATA statement list
- 80 Statement cannot have a label
- 81 No such INTRINSIC function
- 82 Type declaration for INTRINSIC function does not match actual type of INTRINSIC function
- 83 Letter expected
- 84 Type of FUNCTION does not agree with previous usage
- 85 This subprogram has already appeared in this compilation
- 86
- 87 Error in type of argument to INTRINSIC function
- 88 SUBROUTINE/FUNCTION previously used as a FUNCTION/SUBROUTINE
- 89 Unrecognizable statement
- 90 Expression not allowed
- 91 Missing END statement
- 92
- 93 Fewer actual arguments than formal arguments in a FUNCTION or SUBROUTINE reference
- 94 More actual arguments than formal arguments in a FUNCTION or SUBROUTINE reference
- 95 Type of actual argument does not agree with formal argument
- 96 The following procedures were called but not defined
- 97
- 98 Size of type CHARACTER item must be between 1 and 255
- 99 INTEGER*4 variable required
- 100 Statement out of order
- 101 Unrecognizable statement
- 102 Illegal jump into block
- 103 Label already used for FORMAT
- 104 Label already defined
- 105 Jump to FORMAT label
- 106 DO statement forbidden in this context
- 107 DO label must follow a DO statement
- 108 ENDIF forbidden in this context

- 109 No matching IF for this ENDIF
- 110 Improperly nested DO block in IF block

- 111 ELSEIF forbidden in this context
- 112 No matching IF for ELSEIF
- 113 Improperly nested DO or ELSE block
- 114 '(' expected
- 115 ')' expected
- 116 THEN expected
- 117 Logical expression expected
- 118 ELSE statement forbidden in this context
- 119 No matching IF for ELSE
- 120 Unconditional GOTO forbidden in this context

- 121 Assigned GOTO forbidden in this context
- 122 Block IF statement forbidden in this context
- 123 Logical IF statement forbidden in this context
- 124 Arithmetic IF statement forbidden in this context
- 125 ',' expected
- 126 Expression of wrong type
- 127 RETURN forbidden in this context
- 128 STOP forbidden in this context
- 129 END forbidden in this context

- 131 Label referenced but not defined
- 132 DO or IF block not terminated
- 133 FORMAT statement not permitted in this context
- 134 FORMAT label already referenced
- 135 FORMAT must be labelled
- 136 Identifier expected
- 137 Integer variable expected
- 138 'TO' expected
- 139 Integer expression expected
- 140 Assigned GOTO but no ASSIGN statements

- 141 Unrecognizable character constant as option
- 142 Character constant expected as option
- 143 Integer expression expected for unit designation
- 144 STATUS option expected after ',' in CLOSE statement
- 145 Character expression as filename in OPEN
- 146 FILE= option must be present in OPEN statement
- 147 RECL= option specified twice in OPEN statement
- 148 Integer expression expected for RECL= option in OPEN statement
- 149 Unrecognizable option in OPEN statement
- 150 Direct access files must specify RECL= in OPEN statement

- 151 Adjustable arrays not allowed as input-output list elements
- 152 End of statement encountered in implied DO, expressions beginning with '(' not allowed as input-output list elements
- 153 Variable required as control for implied DO
- 154 Expressions not allowed as reading input-output list elements

- 155 REC= option appears twice in statement
- 156 REC= options expects integer expression
- 157 END= option only allowed in READ statement
- 158 END= option appears twice in statement
- 159 Unrecognizable input-output unit
- 160 Unrecognizable format in input-output statement

- 161 Options expected after ',' in input-output statement
- 162 Unrecognizable input-output list element
- 163 Label used as format but not defined in FORMAT statement
- 164 Integer variable used as assigned format but no ASSIGN statement
- 165 Label of an executable statement used as format
- 166 Integer variable expected for assigned format
- 167 Label defined more than once as format
- 169 FUNCTION references require "()"
- 170 INTEGER expression expected for array dimension bound

- 171 Lower dimension bound must be less than or equal to upper dimension bound
- 172 DATA statement cannot initialize arrays of unknown size

- 200 Variable name of named COMMON name expected
- 201 This variable already SAVE'd
- 202 Cannot SAVE dummy arguments
- 203 Cannot SAVE COMMON variables
- 204 INTEGER and LOGICAL *1, *2, or *4 only
- 205 No *n allowed for DOUBLE PRECISION
- 206 Only REAL*4 or REAL*8 allowed
- 207 No *n allowed for COMPLEX
- 208 Size expression only allowed for CHARACTER
- 209 INTEGER constant expression expected
- 210 INTEGER constant or INTEGER constant expression expected

- 211 CHARACTER substring expression out of range
- 212 CHARACTER substring expression must be of type INTEGER
- 213 Error in CHARACTER substring expression
- 214 CHARACTER expression expected
- 215 LOGICAL expression expected
- 216 CHARACTER*(*) only allowed for dummy arguments
- 217 Undeclared PARAMETER constant
- 218 Constant expression not allowed
- 219 Arithmetic operators only apply to numeric values
- 220 Malformed COMPLEX constant

- 221 Maximum of seven levels of implied-DO allowed
- 222 Error in DATA statement variable list
- 223 Error in implied DO list in data statement
- 224 Variables in named COMMON can only appear in a DATA statement which is in a BLOCK DATA subprogram
- 225 Integer subscript expected
- 226 Subscript error

- 227 This identifier is already in use as an implied-DO control variable
- 228 Integer constant expression or implied DO control variable expected
- 229 Integer expression required
- 230 Division by zero

- 231 Error in COMPLEX primary
- 232 Numeric expression or CHARACTER expression expected
- 233 COMPLEX can only compare for equality
- 234 COMPLEX is not compatible with DOUBLE PRECISION
- 235 Constant expression expected
- 236 ENTRY statements must appear in SUBROUTINE or FUNCTION subprograms
- 237 ENTRY statements cannot be within a block IF or a DO statement range
- 238 Concatenation only applies to CHARACTER values
- 239 ':' expected
- 240 Substring operations only apply to CHARACTER variables or CHARACTER array elements

- 241 Error in implied DO expression in a DATA statement
- 242 Implied DO iteration count is zero in a DATA statement
- 243 Error in formal argument list
- 244 Alternate return is not allowed in a FUNCTION subprogram
- 245 Substring error in EQUIVALENCE statement
- 246 EQUIVALENCE statement must not require *2, *4, or *8 variables to be allocated on odd byte addresses
- 247 EQUIVALENCE statement must not require a COMMON block to be allocated on odd byte addresses
- 248 CHARACTER arguments cannot contain concatenation of values that are of size *(*)
- 249 Numeric expression expected
- 250 SUBROUTINE or FUNCTION name has already been used as a COMMON name

- 251 Recursive calls are not allowed
- 252 Statement-FUNCTIONS require variable or value arguments
- 253 Alternate ENTRY in CHARACTER FUNCTION must be of type CHARACTER and must be the same size as the FUNCTION
- 254 This INTRINSIC FUNCTION cannot be passed as an argument
- 255 Executable statements cannot appear in BLOCK DATA subprograms
- 256 An argument to an ENTRY statement has already appeared as a local variable

- 270 Assigned GO TO variable must be INTEGER or INTEGER*4
- 271 INTEGER, REAL, or DOUBLE PRECISION variable expected
- 272 INTEGER, REAL, or DOUBLE PRECISION expression expected
- 273 Unrecognizable element in option list
- 274 Option appears more than once in an option list
- 275 Incorrect type for variable

276 Variable must be *4 in size
277 CHARACTER variable or CHARACTER array element required
278 CHARACTER expression expected
279 Cannot have FILE and UNIT specifier in same INQUIRE statement
280 Must have a FILE or UNIT specifier in INQUIRE statement

281 Must have UNIT specifier
282 PRINT statement requires no option list - use WRITE
283 WRITE statement must have an option list
284 READ statement must not have both REC= and END= options
285 Must not specify REC= option with * format specifier
286 Internal Input Output must be formatted
287 Cannot use REC= specifier with internal input-output
288 Malformed implied DO loop
289 Implied DO loop must have simple variable for loop control
290 Wrong number of arguments to intrinsic function

291 Unit set more than once in input-output statement
292 No unit specified in input-output statement
293 Error in FORMAT statement
294 Hexadecimal constant expected
295 Too many characters in statement
296 Can't find \$INCLUDE file
297 Sub arrays cannot exceed 32766 bytes in size

400 Code file write error
401 Error in rereading code file
402 Error in reopening text file
403 Procedure too large (code buffer too small)
407 Not enough room for intermediate code file
408 Error in writing code file
409 Error in reading intermediate code file

A.2 Run-Time Error Messages

These messages are issued by the input-output run time system, and represent the possible values of 'iostat' in an 'iolist'.

-1 End of file found on a READ with no END= option.
600 FORMAT statement missing final ')'
601 Sign not expected in input
602 Sign not followed by digit in input
603 Digit expected in input
604 Missing N or Z after B in format
605 Unexpected character in format
606 Zero repetition factor in format not allowed
607 Integer expected for w field in format

608 Positive integer required for w field in format
609 '.' expected in format
610 Integer expected for d field in format

611 Integer expected for e field in format
612 Positive integer required for e field in format
613 Positive integer required for w field in A format
614 Hollerith field in format must not appear for reading
615 Hollerith field in format requires repetition factor
616 X field in format requires repetition factor
617 P field in format requires repetition factor
618 Integer appears before '+' or '-' in format
619 Integer expected after '+' or '-' in format
620 P format expected after signed repetition factor in format

621 Maximum nesting level (10 levels) for formats exceeded
622 ')' has repetition factor in format
623 Integer followed by ',' illegal in format
624 '.' is illegal format control character
625 Character constant must not appear in format for reading
626 Character constant in format must not be repeated
627 '/' in format must not be repeated
628 '\', '\$', ':', 'S', 'SP' and 'SS' in format must not be repeated
629 BN or BZ format control must not be repeated
630 Attempt to perform input-output on unknown unit number

631 Formatted or list-directed input-output attempted on file opened as unformatted
632 Format fails to begin with '('
633 I format expected for integer read
634 F, D, G or E format expected for real read
635 Two '.' characters in formatted real read
636 Digit expected in formatted real read
637 L format expected for logical read
639 T or F expected in logical read
640 A format expected for character read

641 I format expected for integer write
642 w field in F format not greater than d field + 1
643 Scale factor out of range of d field in E format
644 E, D, G or F format expected for real write
645 L format expected for logical write
646 A format expected for character write
647 Attempt to do unformatted input-output to a file opened as formatted
648 Unable to write blocked output - possibly no room on output device
649 Unable to read blocked input
650 Error in formatted text file - no carriage-return in last

512 bytes

- 651 Integer overflow on input
- 652 T, TL or TR in format must not be repeated
- 653 Positive integer expected for c field in T, TL or TR format
- 654 Attempt to open direct-access unit on unblocked device
- 655 Attempt to do external input-output on a unit beyond end-of-file record
- 656 Attempt to position a unit for direct-access on a non-positive record number
- 657 Attempt to do direct-access on a unit opened as sequential
- 658 Attempt to position direct-access unit on an unblocked device
- 659 Attempt to position direct-access unit beyond end-of-file for reading
- 660 Attempt to backspace unit connected to unblocked device or unformatted file

- 661 Attempt to backspace sequential, unformatted unit
- 662 Argument to ASIN or ACOS out of bounds - $ABS(x) > 1.0$
- 663 Argument of SIN or COS too large - $ABS(x) > 10$
- 664 Attempt to do unformatted input-output to internal unit
- 665 Attempt to put more than one record into an internal unit
- 666 Attempt to write more characters to an internal unit than its length
- 667 EOF called on unknown unit
- 668 Direct-access formatted input files must not use DLE blank compression
- 669 Error in opening file
- 670 Error in closing file

- 671 Can't specify KEEP in close if file opened SCRATCH
- 672 Unrecognizable option specified as character value in input-output statement
- 673 File name required unless status is SCRATCH
- 674 Must not name file if status is SCRATCH
- 675 Record length not allowed for sequential files
- 676 Record length must be positive
- 677 Record length must be specified for direct-access files
- 678 BLANK option only for formatted files
- 679 Rewind only allowed on sequential files
- 680 Endfile only allowed on sequential files

- 681 Backspace only allowed on sequential files
- 682 Formatted records must be less than or equal to 512 characters
- 683 More characters written to internal file record than record length
- 684 Incorrect number of characters read in formatted record of direct-access file
- 685 Attempt to write too many characters into formatted record of

direct-access file

686 No repeatable edit descriptor found and format exhausted
687 Digit expected in input field exponent
688 Too many digits in input real number
689 Numeric field expected in input
690 Unexpected character encountered in list-directed input

691 Repeat factor in list-directed input must be positive
692 ',' between reals for complex expected in list-directed input
693 ')' expected to terminate complex in list-directed input
694 Attempt to do list-directed input-output to direct-access file
697 Integer variable not currently assigned a FORMAT label
698 End-of-file encountered on a read with no END= option
699 Integer variable not assigned a label used in assigned
GOTO statement

701 Integer input item expected for list-directed input
702 Numeric input item expected for list-directed input
703 Logical input item expected for list-directed input
704 Complex input item expected for list-directed input
705 Character input item expected for list-directed input
706 Incorrect number of bytes read or written to direct-access
unformatted file
707 Substring index range error

1000+ Compiler debug error messages - should never appear in
correct programs

Appendix B
The FORTRAN Intrinsic Functions

Intrinsic Function	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Conversion to Integer	1	INT	INT IFIX IDINT	Real Real Double	Integer Integer Integer
Conversion to Real	1	REAL	REAL FLOAT SNGL	Integer Integer Double	Real Real Real
Conversion to Double	1	DBLE	-- -- -- --	Integer Real Double Complex	Double Double Double Double
Conversion to Complex	1 or 2	CMPLX	-- -- -- --	Integer Real Double Complex	Complex Complex Complex Complex
Conversion to Integer	1		ICHAR	Character	Integer
Conversion to Character	1		CHAR	Integer	Character
Truncation	1	AINT	AINT DINT	Real Double	Real Double
Nearest Whole	1	ANINT	ANINT DNINT	Real Double	Real Double

Intrinsic Function	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Nearest Integer	1	NINT	NINT IDNINT	Real Double	Integer Integer
Absolute Value	1	ABS	IABS ABS DABS CABS	Integer Real Double Complex	Integer Real Double Real
Remaindering	2	MOD	MOD AMOD DMOD	Integer Real Double	Integer Real Double
Transfer of sign	2	SIGN	ISIGN SIGN DSIGN	Integer Real Double	Integer Real Double
Positive Difference	2	DIM	IDIM DIM DDIM	Integer Real Double	Integer Real Double
Double Precision Product	2		DPROD	Real	Double
Choosing Largest Value	2 or more	MAX	MAX0 AMAX1 DMAX1	Integer Real Double	Integer Real Double
			AMAX0 MAX1	Integer Real	Real Integer
Choosing Smallest Value	2 or more	MIN	MIN0 AMIN1 DMIN1	Integer Real Double	Integer Real Double
			AMIN0 MIN1	Integer Real	Real Integer
Length	1		LEN	Character	Integer
Index of Substring	2		INDEX	Character	Integer

Intrinsic Function	Number of Arguments	Generic Name	Specific Name	Argument Type	Function Type
Imaginary Part of Complex Argument	1		AIMAG	Complex	Real
Complex Conjugate	1		CONJG	Complex	Complex
Square Root	1	SQRT	SQRT DSQRT CSQRT	Real Double Complex	Real Double Complex
Exponential	1	EXP	EXP DEXP CEXP	Real Double Complex	Real Double Complex
Natural Logarithm	1	LOG	ALOG DLOG CLOG	Real Double Complex	Real Double Complex
Common Logarithm	1	LOG10	ALOG10 DLOG10	Real Double	Real Double
Sine	1	SIN	SIN DSIN CSIN	Real Double Complex	Real Double Complex
Cosine	1	COS	COS DCOS CCOS	Real Double Complex	Real Double Complex
Tangent	1	TAN	TAN DTAN	Real Double	Real Double
Arcsine	1	ASIN	ASIN DASIN	Real Double	Real Double
Arccosine	1	ACOS	ACOS DACOS	Real Double	Real Double

Intrinsic Function	Number of Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Arctangent	1	ATAN	ATAN DATAN	Real Double	Real Double
	2	ATAN2	ATAN2 DATAN2	Real Double	Real Double
Hyperbolic Sine	1	SINH	SINH DSINH	Real Double	Real Double
Hyperbolic Cosine	1	COSH	COSH DCOSH	Real Double	Real Double
Hyperbolic Tangent	1	TANH	TANH DTANH	Real Double	Real Double
Lexically Greater or Equal	2		LGE	Character	Logical
Lexically Greater Than	2		LGT	Character	Logical
Lexically Less Than or Equal	2		LLE	Character	Logical
Lexically Less Than	2		LLT	Character	Logical

B.1 Notes on the Intrinsic Functions

The INT function truncates real or double precision arguments towards zero. If the argument to INT is a complex number, the function is applied to the real part of the complex argument. IFIX is the same as INT for real arguments.

If the REAL or DBLE functions are applied to a complex argument, the result is the real part of the argument.

CMPLX can have one or two arguments. If there is one argument, it can be of type integer, real, double precision or complex. If the argument is of type integer, real or double precision, the result is a complex value whose real part is that of the argument, and whose imaginary part is zero.

If CMPLX has two arguments, they must both be of the same type. The arguments can be of type integer, real or double precision. The result is a complex value whose real part is the first argument and whose imaginary part is the second argument.

ICHAR converts from character to integer. The first character in the collating sequence is position 0 and the last character in the sequence is 'n'-1, where 'n' is the number of characters in the character set.

In the trigonometric functions, all angles are in radians.

Functions of complex arguments yield a result which is the principal value of the function.

The INDEX function returns the index where its second argument starts in its first argument. If the first argument does not contain the second argument, or if the second argument is longer than the first argument, the INDEX function returns a value of zero.

All arguments in an intrinsic function reference must be of the same type.

B.2 Restrictions on Ranges of Arguments

When intrinsic functions are referenced by their specific names, the restrictions on ranges of arguments and results are as follows:

- | | |
|----------------------|--|
| Remaindering | MOD, AMOD and DMOD are undefined when their second argument is zero. |
| Transference of Sign | If the first argument to ISIGN, SIGN or DSIGN is zero, the result is zero. |
| Square Root | SQRT and DSQRT require an argument which is not less than zero. CSQRT returns a value which is the principal value and is greater than or equal to zero. If the real part of the result is |

zero, the imaginary part is greater than or equal to zero.

Logarithms ALOG, DLOG, ALOG10 and DLOG10 require an argument greater than zero. The argument to CLOG must not be (0.0, 0.0). If the real part of the argument is less than zero and the imaginary part is zero, the imaginary part of the result is 'pi', otherwise the imaginary part of the result lies in the range:

$$-\pi < \text{imaginary part} \leq \pi$$

Arcsine and Arccosine

ASIN, DASIN, ACOS and DCOS require that the absolute value of their argument be not greater than one. The result of arcsine lies in the range

$$-\pi/2 \leq \text{result} \leq \pi/2$$

and the result of arccosine lies in the range

$$0 \leq \text{result} \leq \pi$$

B.3 Non Standard Intrinsic Functions and Subroutines

The functions and subroutines described here are non-standard SVS FORTRAN-77 extensions to the FORTRAN-77 language.

POKE - Store Into Arbitrary Memory Location

The POKE subroutine stores a byte into an arbitrary memory location. The interface definition is:

```
SUBROUTINE POKE(IADDR, IVAL)
  INTEGER IADDR, IVAL
```

The POKE subroutine sets the memory location addressed by IADDR to the byte value of the variable IVAL.

IPEEK - Read From Arbitrary Memory Location

The IPEEK function gets a byte from an arbitrary memory location. The interface definition is:

```
INTEGER*4 FUNCTION IPEEK(IADDR)
INTEGER IADDR
```

IADDR is the address of a memory location. The IPEEK function returns the signed value of the byte stored at that location.

VERS - Print Date and Version

The VERS subroutine prints the date and version of the SVS FORTRAN-77 run-time system. The interface definition is:

```
SUBROUTINE VERS
```

There are no arguments to the VERS subroutine.

Appendix C

Data Representations

This appendix describes the ways that SVS FORTRAN represents data in storage and the mechanisms for passing arguments to subroutines and functions. This appendix is intended as a guide to those programmers who wish to write modules in languages other than FORTRAN and have those modules interface to FORTRAN.

C.1 Storage Allocation

This section describes the way in which storage is allocated to variables of various types.

In general, any word value (a value which occupies 16 bits) is always aligned on a word boundary. Anything larger than a word is also aligned on a word boundary. Values that can fit into a single byte are aligned on a byte boundary.

INTEGER, REAL and LOGICAL data types all occupy the same amount of storage, namely 32 bits (four bytes or two words). DOUBLE PRECISION occupies 64 bits (eight bytes or four words). COMPLEX is represented as a pair of single precision real data values and so occupies 64 bits (eight bytes or four words). There are provisions for indicating that integer and logical data types occupy less storage.

INTEGER*1 occupies 8 bits (one byte), aligned on a byte boundary.

INTEGER*2 occupies 16 bits (two bytes or one word), aligned on a word boundary.

INTEGER and INTEGER*4 occupy 32 bits (four bytes or two words), aligned on a word boundary.

REAL and REAL*4 occupy 32 bits (four bytes or two words), aligned on a word boundary. A REAL element has a sign bit, an 8-bit exponent and a 23-bit mantissa. SVS FORTRAN REAL elements conform to the IEEE standard for reals as defined in the March 1981 Computer magazine. The layout of a REAL element is shown below.

DOUBLE PRECISION and REAL*8 elements occupy 64 bits (eight bytes or four words), aligned on a word boundary. A DOUBLE PRECISION element has a sign bit, an 11-bit exponent and a 52-bit mantissa. SVS FORTRAN DOUBLE PRECISION elements conform to the IEEE standard for double precision data as defined in the March 1981 Computer magazine. The layout of a DOUBLE PRECISION element is shown below.

COMPLEX elements are represented by two REAL elements. The first element represents the real part of the number, the second represents the imaginary part.

LOGICAL*1 occupies one byte (8 bits) of storage, aligned on a byte boundary. A value of 0 represents the value **.FALSE.** . A value of 1 represents the value **.TRUE.** . Any other value is an "undefined" logical value.

LOGICAL*2 occupies two bytes (16 bits) of storage, aligned on a word boundary. A value of 0 represents the value **.FALSE.** . A value of 1 represents the value **.TRUE.** . Any other value is an "undefined" logical value.

LOGICAL and LOGICAL*4 occupies four bytes (32 bits) of storage, aligned on a word boundary. A value of 0 represents the value **.FALSE.** . A value of 1 represents the value **.TRUE.** . Any other value is an "undefined" logical value.

Representation of Extreme Numbers

zero (signed) is represented by an exponent of zero, and a mantissa of zero.

denormalized numbers

are a product of "gradual underflow". They are non-zero numbers with an exponent of zero. The form of a denormalized number is:

$$2^{\text{exponent-bias}+1} * 0.f$$

where 'f' is the bits in the mantissa.

signed infinity (that is, affine infinity) is represented by the largest value that the exponent can assume (all ones), and a zero mantissa.

Not-a-Number (NaN)

is represented by the largest value that the exponent can assume (all ones), and a non-zero mantissa. The sign is usually ignored.

Normalized REAL and DOUBLE PRECISION numbers are said to contain a "hidden" bit, providing for one more bit of precision than would normally be the case.

Hexadecimal Representation of Selected Numbers

Value	REAL	PRECISION
+0	00000000	0000000000000000
-0	80000000	8000000000000000
+1.0	3F800000	3FF0000000000000
-1.0	BF800000	BFF0000000000000
+2.0	40000000	4000000000000000
+3.0	40400000	4008000000000000
+Infinity	7F800000	7FF0000000000000
-Infinity	FF800000	FFF0000000000000
NaN	7F8xxxxx	7FFxxxxxxxxxxxxxxx

Deviations from the Proposed IEEE Standard

Deviations from the proposed IEEE standard in this implementation are as follows:

- . affine mapping for infinities,
- . normalizing mode for denormalized numbers,
- . rounds approximately to nearest - 7 or more guard bits are computed, but the "sticky" bit is not,
- . exception flags are not implemented,
- . conversion between binary and decimal is not implemented.

Arithmetic Operations on Extreme Values

This subsection describes the results derived from applying the basic arithmetic operations on combinations of extreme values and ordinary values.

No traps or any other exception actions are taken.

All inputs are assumed to be positive. Overflow, underflow, and cancellation are assumed not to happen.

In all the tables below, the abbreviations have the following meanings:

Abbreviation	Meaning
Den	Denormalized Number
Num	Normalized Number
Inf	Infinity (positive or negative)
NaN	Not a Number
Uno	Unordered

Addition and Subtraction					
Left Operand	0	Right Operand		Inf	NaN
		Den	Num		
0	0	Den	Num	Inf	NaN
Den	Den	Den	Num	Inf	NaN
Num	Num	Num	Num	Inf	NaN
Inf	Inf	Inf	Inf	Note 1	NaN
Nan	NaN	NaN	NaN	NaN	NaN

Note 1: $\text{Inf} + \text{Inf} = \text{Inf}$; $\text{Inf} - \text{Inf} = \text{NaN}$

Multiplication					
Left Operand	0	Right Operand		Inf	NaN
		Den	Num		
0	0	0	0	NaN	NaN
Den	0	0	Num	Inf	NaN
Num	0	Num	Num	Inf	NaN
Inf	NaN	Inf	Inf	Inf	NaN
Nan	NaN	NaN	NaN	NaN	NaN

Division					
Left Operand	0	Right Operand Den	Num	Inf	NaN
0	NaN	0	0	0	NaN
Den	Inf	Num	Num	0	NaN
Num	Inf	Num	Num	0	NaN
Inf	Inf	Inf	Inf	NaN	NaN
Nan	NaN	NaN	NaN	NaN	NaN

Comparison					
Left Operand	0	Right Operand Den	Num	Inf	NaN
0	=	<	<	<	Uno
Den	>		<	<	Uno
Num	>	>		<	Uno
Inf	>	>	>		Uno
Nan	Uno	Uno	Uno	Uno	Uno

Notes:

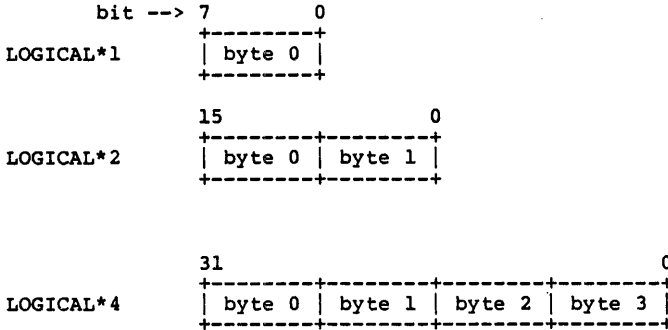
NaN compared with NaN is Unordered, and also results in inequality.

+0 compares equal to -0.

Max					
Left Operand	0	Right Operand Den	Num	Inf	NaN
0	0	Den	Num	Inf	NaN
Den	Den	Den	Num	Inf	NaN
Num	Num	Num	Num	Inf	NaN
Inf	Inf	Inf	Inf	Inf	NaN
Nan	NaN	NaN	NaN	NaN	NaN

Min					
Left Operand	0	Right Operand Den	Num	Inf	NaN
0	0	0	0	0	NaN
Den	0	Den	Den	Den	NaN
Num	0	Den	Num	Num	NaN
Inf	0	Den	Num	Inf	NaN
Nan	NaN	NaN	NaN	NaN	NaN

Representation of Logicals



Storage of Arrays

Arrays are stored with their elements in a specific storage order. The elements are actually stored in a linear sequence of storage elements.

FORTRAN arrays are stored in column major order, such that the first subscript in a multi-dimensional array varies fastest. The position of an arbitrary element in an array is:

$$1 + \text{SUM}((S_i - L_i) * \text{PRODUCT}(U_j - L_j))$$

where 'S_i' is the value of the subscript expression specified for dimension 'k', 'U_k' is the upper bound for dimension 'k' and 'L_k' is the lower bound of dimension 'k'. The subscript 'j' in the product above varies between 1 and 'i'-1 for any given dimension.

C.3 Argument Passing Mechanism

This section describes the way in which arguments are passed in SVS FORTRAN.

All arguments to FORTRAN subroutines and functions are passed by reference. For every argument except a CHARACTER object, a 32-bit pointer to the object is pushed onto the stack.

When CHARACTER objects are passed in FORTRAN-77, a 32-bit pointer to the CHARACTER object is pushed onto the stack, followed by a 16-bit length of the CHARACTER object.

Pointers to actual arguments are pushed onto the stack in the order in which they are declared in a subroutine or function declaration.

Actual arguments which are expressions are evaluated before the subroutine or function call. The result of the expression is assigned to a temporary storage area and a pointer to the temporary is pushed onto the stack.

If the callee is a FUNCTION, room for the function result is allocated on the stack before any arguments are pushed. Functions which return CHARACTER data types return a pointer and length pair on the stack.

A DOUBLE PRECISION FUNCTION in FORTRAN-77 returns its result with the following pair of instructions:

```
movl    d0,sp@
movl    dl,sp@(4)
```

Upon return from a procedure, all arguments are discarded from the stack. The only thing that should be on the stack upon return, is a function result.

Appendix D
ASCII Character Set Table

hex	char	hex	char	hex	char	hex	char
00	NUL	20	SP	40	@	60	`
01	SOH	21	!	41	A	61	a
02	STX	22	"	42	B	62	b
03	ETX	23	#	43	C	63	c
04	EOT	24	\$	44	D	64	d
05	ENQ	25	%	45	E	65	e
06	ACK	26	&	46	F	66	f
07	BEL	27	'	47	G	67	g
08	BS	28	(48	H	68	h
09	HT	29)	49	I	69	i
0A	LF	2A	*	4A	J	6A	j
0B	VT	2B	+	4B	K	6B	k
0C	FF	2C	,	4C	L	6C	l
0D	CR	2D	-	4D	M	6D	m
0E	SO	2E	.	4E	N	6E	n
0F	SI	2F	/	4F	O	6F	o
10	DLE	30	0	50	P	70	p
11	DC1	31	1	51	Q	71	q
12	DC2	32	2	52	R	72	r
13	DC3	33	3	53	S	73	s
14	DC4	34	4	54	T	74	t
15	NAK	35	5	55	U	75	u
16	SYN	36	6	56	V	76	v
17	ETB	37	7	57	W	77	w
18	CAN	38	8	58	X	78	x
19	EM	39	9	59	Y	79	y
1A	SUB	3A	:	5A	Z	7A	z
1B	ESC	3B	;	5B	[7B	{
1C	FS	3C	<	5C	\	7C	
1D	GS	3D	=	5D]	7D	}
1E	RS	3E	>	5E	^	7E	~
1F	US	3F	?	5F	_	7F	DEL

Appendix E

FORTRAN on the UNIX System

This appendix describes the specific details of using the SVS FORTRAN-77 compiler on the UNIX Operating System. Later sections cover specific dependencies and features of SVS FORTRAN-77 on the UNIX system.

E.1 Running FORTRAN-77 On the UNIX System

The FORTRAN-77 compiler, `f77`, compiles programs written in the FORTRAN-77 language, and optionally links (or binds) the result with intermediate files generated by `f77` or other language processors.

The description below covers the UNIX system interface to the compiler. At the end of this section can be found a description on how to run the individual phases separately.

NAME

`f77` - FORTRAN-77 Compiler

SYNOPSIS

`f77` [-o ofile] [-i] [-c] [-u] [-v] file ...

DESCRIPTION

`f77`, the FORTRAN-77 compiler, accepts a list of FORTRAN-77 source files and various intermediate texts contained in the list of files specified by file. The resulting executable object-module is placed in the file a.out, unless the `-o` option was specified (see below).

1. UNIX is a trademark of Bell Laboratories

To understand the use of `f77`, the reader should be aware of the steps through which the compiler goes in order to turn a FORTRAN-77 source program into an executable object-file. `f77` goes through several separate phases (or passes) when compiling a program.

`f77` source files must have a `.for` suffix. `f77` generates three intermediate files (`.i`, `.obj`, and `.o`) on the way to generating the final executable binary file. The first phase of the compiler generates intermediate files with the same names as the source files, but with a `.i` suffix. These intermediate files are destined for processing by the code generator.

Code generation is the second phase of the process. The code generator produces files of the same name as the source files, but with a `.obj` suffix. The `*.obj` files are the input to the next phase, called `ulinker`.

The `ulinker` phase of the process performs two functions: it links the `.obj` files, resolving external references to the FORTRAN-77 run-time library; then it converts the `.obj` file format into a UNIX-style object file, with a `.o` suffix. This means that one file in the list of files specified on the `f77` command line must contain a main program. This `.o` file can then be processed by `ld` - the UNIX system loader utility.

The final stage of the process is when the `ld` utility generates the final executable object code file.

When using `f77`, any combination of FORTRAN-77 source files (each with a `.for` suffix) can be combined with FORTRAN-77 (or SVS Pascal) intermediate files (each with a `.i` suffix), FORTRAN-77 or FORTRAN object-code files (each with a `.obj` suffix), and UNIX system object-code files (having `.o` suffixes). When the compilation completes successfully, the result of the combination of all those files is placed in the file `a.out`, or in the file specified by the `-o` option.

The `-o` option, if used, specifies that the file `ofile` (runnable file) whose name follows the `-o` option is the file to receive the final executable code. If the `-o` option is not specified, the resultant executable code is placed in the file `a.out`.

If the `-i` option is used, the FORTRAN-77 intermediate code (the result of running `/lib/fortran`) is placed in a file of the same name as the source file, but with a suffix of `.i` appended. The compilation then terminates.

If the `-c` option is used, the FORTRAN-77 unlinked object-code (the result of running `/lib/code`) is placed in a file of the same name as the source file, but with a suffix of `.obj` appended. The

compilation then terminates.

If the `-u` option is used, the linked object-code (the result of running `/lib/ulinker`) is placed in a file of the same name as the source file, but with a suffix of `.o` appended. The compilation then terminates at that point.

The `-v` (for **verbose**) option tells `f77` to display a running progress report as it goes through each of the compilation and linking phases. If the `-v` option is specified, the compiler also generates a listing file, of the same name as the source file, but with a suffix of `.lst` appended.

If only one `file` argument is supplied on the command line, then all the intermediate files (`*.i`, `*.obj`, `*.o`) are removed at the end of the compilation. If multiple `file` arguments are specified on the command line, any existing intermediate files are not removed.

EXAMPLES

```
f77 prog1.for
```

compiles prog1.for and places the resulting executable file in a.out.

```
f77 -o framnis prog2.for prog3.obj
```

compiles the FORTRAN-77 program called prog2.for and links the result with the object-file prog3.obj. The resulting executable binary is placed in the output file called framnis.

FILES

<code>*.for</code>	FORTRAN-77 source,
<code>*.i</code>	Intermediate code,
<code>*.obj</code>	Compiled but unlinked <code>f77</code> object-code,
<code>*.o</code>	Compiled but unlinked UNIX object-code,
<code>/lib/ftnlib.obj</code>	FORTRAN-77 run-time library,
<code>/lib/paslib.obj</code>	Pascal run-time library,
<code>/lib/fortran</code>	FORTRAN-77 compiler,
<code>/lib/code</code>	Code generator,
<code>/lib/ulinker</code>	SVS to UNIX object-file converter,
<code>/lib/ftncterr</code>	FORTRAN-77 compile-time error message file.
<code>/lib/ftnrterr</code>	FORTRAN-77 run-time error message file.
<code>/lib/wraplib</code>	Interfaces for calling C routines.

Note that all the files appearing in `/lib` above can alternatively reside in `/usr/lib`.

Running The Compiler Phases Separately

The separate parts of the FORTRAN-77 Language system can be run independently if required. The separate phases are:

- The FORTRAN-77 compiler (front end) is in /lib/fortran (or /usr/lib/fortran). The compiler turns FORTRAN-77 source text into an intermediate form suitable for processing by the code generator.
- The Code Generator is in /lib/code (or /usr/lib/code). It converts the intermediate file into a form which will be processed by ulinker.
- ulinker is in /lib/ulinker (or /usr/lib/ulinker). It links the output of the code generator, and converts the resulting linked file into UNIX compatible .o file format, ready for processing by the UNIX link-editor (ld) utility.

Examples shown in the following subsections assume that there is a program called 'hello.for', which is to be compiled, code-generated, linked, and link-edited, ready for execution.

Running the FORTRAN-77 Compiler Independently

The FORTRAN-77 compiler (front end) is run with a command line like this:

```
/lib/fortran [+q] [-p] sfile -llfile -iifile -eefile
or,
/usr/lib/fortran [+q] [-p] sfile -llfile -iifile -eefile
```

where 'sfile' is the name of a file containing the FORTRAN-77 source program. The source is expected to be on a file with a .for suffix.

Other files (such as listing or error files) are introduced by the options described below. Note that there must not be any spaces between an option letter and its corresponding filename.

```
-llfile      'lfile' is the optional name of the file to
              receive the compiler generated listing. No
```

suffix is appended to the list filename. If the `-l` option is omitted, no listing is generated.

`-iifile` 'ifile' is the optional name of the intermediate file destined for processing by the code generator. If the 'ifile' is not specified on the command line, the intermediate output is placed on a file with the same name as the source file (minus the `.for` suffix) with a `.i` suffix appended.

`-eefile` 'efile' is the optional name of a file to receive the compiler error summary. No suffix is appended to the error summary filename. An error summary file is not generated unless it is requested by the `-e` option.

The `+q` (for quiet)

option makes the compiler print less. The `-q` option increases the amount of data displayed. The default is `+q` (quiet).

In the `f77` UNIX command, the `-q` option is called up via the `-v` (for verbose) option on the command line.

The `+p` (for prompt)

option makes the compiler display each error on the standard output, and prompt for a user response from the standard input. The user has the option to continue or abort the compilation at that point. If the `-p` option is used, the compiler automatically continues processing and searching for more errors after an error is found. The normal default for this option is `-p` (continue automatically).

Example of Running the FORTRAN-77 Compiler

```
fortran hello.for
```

In this example, the FORTRAN-77 compiler compiles the source in the file 'hello.for'. The intermediate text is placed in the 'hello.i' file.

Running the Code Generator

The code generator is the second phase of a FORTRAN-77 compilation. The code generator processes the intermediate file from the compiler and generates a .obj file ready for processing by the ulinker utility. The code generator is run simply by typing the command:

```
/lib/code file  
or,  
/usr/lib/code file
```

where 'file' is the name of the intermediate file to process. The intermediate file must have a .i suffix. The output of the code generator is a file ready for processing by ulinker. The output filename is the same as the input filename, minus the .i suffix, and with a .obj suffix appended.

Example of Running the Code Generator

```
/lib/code hello.i
```

In this example, the code generator is generating code for the file 'hello.i'. The result of the code generation appears on a file called 'hello.obj'.

Running the Ulinker Utility

The ulinker utility links the output of the code generator, resolves external references to run-time routines, and converts the output of the code generator into a form that can be processed by the UNIX link-editor. The ulinker utility is run like this:

```
/lib/ulinker [-l] ofile objfile...  
or,  
/usr/lib/ulinker [-l] ofile objfile...
```

`-l` if supplied, suppresses link map information. If the `-l` option is omitted, the first filename on the ulinker command line must be the name of a file to receive the listing.

`'ofile'` is the name of the file to receive the final `.o` file.

`'objfile'` is a list of `.obj` files (and libraries) to process.

The output of ulinker is a file ready for processing by the UNIX link-editor.

The `-l` option, if typed on the command line, tells ulinker to display progress information on the standard output.

Example of Running ulinker

```
/lib/ulinker -l hello.o hello.obj /lib/ftnlib.obj /lib/paslib.obj
```

This example links the `'hello.obj'` file generated by the code generator, resolves external references to the FORTRAN-77 and Pascal run-time libraries, then converts the code format to a form acceptable to `ld`, the UNIX link-editor. The result appears in the `'hello.o'` file shown on the command line.

The FORTRAN-77 run-time routines make calls upon certain C library routines. The interface between these is supplied by a `.o` file called `'wraplib.o'`, which resides either in the `/lib` or `/usr/lib` directories. `'wraplib.o'` must be included in the link-edit process, after the user's `.o` files, and before the C library. A description of the interface between FORTRAN-77 and C routines can be found later in this appendix.

The final stage in the process of obtaining an executable binary file is to run the UNIX link-editor:

```
ld /lib/crt0.o hello.o /lib/wraplib.o -lc
```

This example links the `'hello.o'` file with the run-time routines from `'wraplib.o'`, and the C run-time library indicated by the `-lc` option on the `ld` command line. The `'crt0.o'` file must come first in the link sequence, so as to get the run-time startup routines in the correct place. The resulting executable binary appears on the file `'a.out'`.

E.2 UNIX System Dependencies

This section covers specific features of the SVS FORTRAN-77 compiler on the UNIX operating system.

Action on Closing Files

Contrary to what is said in chapter 10 - "FORTRAN Input and Output" about the actions taken on closing a file, on the UNIX system, all files are closed with the 'lock' option, regardless of what option was actually specified in the CLOSE statement.

English Error Messages

The FORTRAN-77 system now can display English style error messages by accessing the error message files which reside in the file /lib/ftncterrs (or in /usr/lib/ftncterrs) (fortran compile time errors), and in /lib/ftnrterr (or /usr/lib/ftnrterr) (fortran run time errors).

E.3 Interfacing FORTRAN-77 to UNIX C Programs

This part of the appendix is intended to give the user some clues in how to link programs written in SVS FORTRAN-77 with external routines written in C on the UNIX system.

Because the interfaces and calling conventions differ between FORTRAN-77 and C, FORTRAN-77 routines are linked with C routines via a small assembler language routine called a "wrapper". The purpose of the "wrapper" is to make the interface between FORTRAN-77 and C compatible with each other.

The user must be aware of the following relationships between FORTRAN-77 and C:

- . All C parameters expect four bytes on the stack for each argument, regardless of what they are.
- . C parameters are pushed onto the stack in the reverse order from those of FORTRAN-77. So the FORTRAN-77 call:

```
func(a, b, c)
```

is seen by the C routine as:

```
func(c, b, a)
```

- . The C language returns function results in register D0. FORTRAN-77 returns function results on the stack. The interface routine must be aware of this and act accordingly.
- . The representation of REAL and DOUBLE PRECISION data in FORTRAN-77 is incompatible with the float and double data types of C, so the user should avoid passing these data types.

Examples

The four sets of declarations below are representative samples of some C routines and the FORTRAN-77 declarations to provide the interface to those routines. Following the declarations, are the "wrappers" to interface those C routines with the FORTRAN-77 programs.

FORTRAN-77 Declaration:

```
INTEGER add(a, b)
```

C Function:

```
add(b, a)          /* add two numbers */
{
    int *a, *b;
    return (*a + *b);
}
```

C Function:

```
hello()           /* print the word 'HELLO' */
{
    printf ("HELLO!\n");
}
```

C Function:

```
swap(b, a)        /* swap two numbers */
{
    int *a, *b;
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

Note that in the declarations above we have one FUNCTION and two SUBROUTINES. One of the SUBROUTINES has no parameters. The other SUBROUTINE and the FUNCTION have parameters. Note that the FORTRAN program only needs to declare the FUNCTIONS, since it will simply CALL the SUBROUTINES, and these don't have to be declared. In the discussions to follow, these differences will illustrate the different pieces of assembler coding necessary for

the interface.

A fairly simple FORTRAN program to call those routines might look something like this:

```

C      program testC
      integer a, b, r
      integer add
C
      a = 10
      b = 13
      call hello()
      r = add (a, b)
      call swap (a, b)
      stop
C
      end

```

Connecting the Pieces

Unless the C program segments are already in the C library (that is, they are standard C functions), they should be compiled and left as .o files (using the cc -c compiler call).

An assembly language interface routine must be provided for each C routine to be called. This interface routine is called a "wrapper".

The external reference passed from the FORTRAN-77 program will be in upper case. The wrapper must call the corresponding lower-case routine (the C routine), and then clean up the stack on return. For every parameter that the FORTRAN-77 program passes, four bytes must be removed from the stack.

The wrapper must also ensure that the function return value (if there is one) must be pushed onto the stack from register D0.

Examples of Wrapper Routines

The assembly language fragments below are the wrappers to interface the three FORTRAN-77 and C routines shown in the examples above. There is a common theme throughout each

routine.

- Each wrapper declares as global (the `.globl` declaration) both the upper case name (which FORTRAN-77 generates) and the corresponding lower case name (which C generates). Note that the C routines have an underline character in front of them, but not the FORTRAN routine names, because the underline character cannot be used in FORTRAN names.
- In all cases, FORTRAN-77 places its return address on the stack. The wrapper pops that stack entry into register A3, which is guaranteed not to be clobbered by any C routines which are called (user written assembler codes must also honor this convention to avoid disaster).
- Upon return from the C routine, the wrapper pops four bytes for every parameter that was passed from the FORTRAN-77 program.
- Lastly, if the calling FORTRAN-77 routine is a function which expects a result, the wrapper must push the C function's return value onto the stack from register D0.

First there is the wrapper for the 'add' function:

```

      .globl ADD,_add      | declare ADD global
ADD:  movl  sp@+,a3      | save FORTRAN-77 return
      address
      jsr   _add         | call the C _add routine
      addl  8,sp         | remove 8 bytes from stack
      movl  d0,sp@      | put return value on stack
      jmp   a3@         | return to FORTRAN-77.

```

The 'add' function passes two parameters, so after the C routine returns it pops eight bytes from the stack. The caller is a FORTRAN-77 function, so the return value is pushed onto the stack.

Then there is the wrapper for the 'hello' procedure, which does not have any parameters:

```

      .globl HELLO,_hello | declare HELLO global
HELLO: movl  sp@+,a3      | save FORTRAN-77 return
      address
      jsr   _hello      | call the C _hello routine
      jmp   a3@         | nothing to remove/return,
                        | so just jump to FORTRAN-77.

```

This routine has no parameters, and the caller expects no return value, so there is nothing to do except return to the caller.

Lastly there is the wrapper for the 'swap' procedure which has two parameters:

<pre> .globl SWAP,_swap SWAP: movl sp@+,a3 address jsr swap addl 8,sp jmp a3@ </pre>	<pre> declare SWAP global save FORTRAN-77 return call the C routine remove 8 bytes from stack nothing to return, so just jump to FORTRAN-77 </pre>
--	--

The wrapper should be assembled into a .o module using the assembler. Now, all the pieces must be presented to the loader (which may be done by calling the compiler).

In addition to any "wrapper" routines that the user writes, there is also a file in /lib/wraplib (or /usr/lib/wraplib) which contains some predefined interfaces to the C run-time library.

For example, if the pieces of the C program are in the file called 'zork.o', and the wrapper is in 'wrp.o', and the FORTRAN-77 program is in 'prog.o', the pieces can be linked by typing:

```
f77 prog.o wrp.o zork.o
```

or (more complicated):

```
cc prog.o wrp.o zork.o /lib/wraplib.o
```

or (even more complicated):

```
ld /lib/crt0.o prog.o wrp.o zork.o /lib/wraplib.o -lc
```

The file /lib/wraplib (or /usr/lib/wraplib) already contains wrappers for the C language routines:

```

_CLOSE
_CREAT
_LSEEK
_OPEN
_READ
_SBRK
_WRITE

```

Note that these wrapper routines have underline characters in front of them, even though underline characters are not allowed in FORTRAN-77 names. The names in the library above are treated specially by the linker.