IBM®

NPL Technical Report.

NPL Technical Report.

IBM

This manual provides a detailed and comprehensive description of a new programming language, NPL. This new language is designed not only for applications programming in the traditional commercial and scientific fields, but also for programming in other applications areas. At the same time, the language is so designed that different levels of language facility can be selected for given classes of applications or for given levels of programmer experience.

The NPL Technical Report is only intended to describe the language and not to serve as a specification of the language for implementation by a particular compiler.

December 1964

PREFACE

This manual constitutes a description of NPL. It is a technical report of NPL, not a student text, nor a user's guide for a particular compiler implementation of the language. Further publications describing the language are planned for a later date.

In general, this manual assumes a relatively high level of programming knowledge and experience on the part of the reader. Specifically, it assumes a thorough knowledge of modern programming concepts and techniques and some knowledge of current high-level programming languages. Accordingly, the manual is not intended for general distribution.

The "Introduction" chapter provides the reader with a review of the design criteria of NPL and a discussion of the more significant features of the language; it also gives an indication of those parts of the manual that are of interest to particular classes of users. The language description comprises the succeeding twenty-seven chapters. Various kinds of reference information have been organized into eight appendices, the last of which is concerned with implementation of NPL for the IBM System/360.

This description of NPL is based largely on reports issued by the SHARE Advanced Language Development Committee which included GUIDE representation. IBM wishes to express its deep appreciation to that committee and to acknowledge the efforts of its members.

This publication was prepared for production using an IBM computer to update the text and to control the page and line format. Page impressions for photo-offset printing were obtained from an IBM 1403 Printer using a special print chain.

A modern data processing system should serve as a comprehensive tool for the solution of today's data processing problems. In addition, it should provide a broad base for meeting the future needs of the ever increasing spectrum of applications. Whereas, in the past, data processing equipment was often designed specifically for either scientific or for business requirements, today's equipment should lend itself equally well to both business and scientific applications and to newer areas, such as real time processing.

Similarly, a modern operating system should offer comprehensive programming support for the efficient solution of traditional computing problems in many fields and, at the same time, should provide an up-to-date environment for newer techniques, such as asynchronous program execution and shared data processing facilities.

An advanced programming language is a critical element of a modern data processing system. This idea provided the basic motivation for the design and development of NPL, a new programming language. NPL is designed to serve the needs of an unusually large group of programmers, including scientific, business, real time, and systems programmers. The language is so organized that each programmer, no matter how extensive his experience, finds facilities at his own level. The NPL programmer can write programs simply, without concern for arbitrary restrictions; he can devote his energy to the problem and its analysis, rather than to its programming.

This chapter consists of three sections. The first section describes the design criteria or philosophical bases of NPL. The second section discusses some of the significant features of the language, with special emphasis on those advanced features that are critically important for a new programming language. The third section discusses the organization of the remainder of this publication and attempts to assist the reader in his study of NPL.

DESIGN CRITERIA

In order to better understand NPL and to appreciate some of its characteristics, it is helpful to know the basic rules that governed its design. Following is an explanation of six such bases.

Freedom of expression: If a particular combination of symbols had a useful meaning, that meaning was embodied in the language. Invalidity was a last resort. This will help to insure uniformity between different NPL compilers.

Full access to machine and operating system facilities: The NPL programmer will rarely, if ever, need to resort to assembly language coding. No facility was discarded because it belonged more properly to assembly or control languages.

Modularity: NPL has been designed so that many of its features can be ignored by the programmer without fear of adverse effects. Thus, manuals can be constructed of subsets of the language for different applications and different levels of complexity. These need not mention the unused facilities. To accomplish this end, every attribute of a variable, every option, and every specification was given a default interpretation, and this was chosen to be the one most likely to be required by the programmer who does not know that alternatives exist.

Relative machine independence: Although NPL allows the programmer to take full advantage of the powerful facilities of System/360, it is essentially a machine-independent language. Parameters which would reflect the characteristics of a particular machine were not allowed to intrude into the language. Thus, for example, the programmer specifies the precision of an arithmetic variable in digits rather than by "single" or "double" precision; input/output is specified in a device-independent manner.

Catering to the novice: Although the general specification is there for power and growth, the frequently used special case is specifiable redundantly in an explicit way. This approach allows the compiler to maximize efficiency for the commonly-used case, and, again, permits the novice to learn only the notation which is most natural to him.

A programming language, not an algorithmic language: Programming languages are most often written on coding sheets, punched at key punches or terminals, and listed on printers. While the specification of a publication language is considered essential, the first and most important goal of syntax design has been to make the listings as readable, and to make the writing and punching as error-free, as possible. A free-field format has been chosen to help meet these goals.

NEW FEATURES

A large part of NPL is, of course, based on earlier programming languages. On the other hand, several concepts not manifested in previous higher-level languages are reflected in NPL. Also, certain ideas have been extended or modified so as to take on broader significance. The following paragraphs describe some of the salient features to be found in the new language.

PROGRAM STRUCTURE

In most high level programming languages the basic element that denotes a certain action to be executed is called a statement; the collection of all the statements required to achieve solution to the problem at hand is called a program. Generally, it has proved necessary or desirable to introduce into programs a structure which is more complex than that of a single statement. The motivations for such structuring are:

1. to delimit a procedure which may then be invoked from several different places with different arguments;
2. to delineate the scope of applicability or uniqueness of a name so that names may be non-unique within a program and yet well-defined locally;
3. to group a set of statements for control purposes so that they are treated syntactically as a single statement;
4. to specify the duration of allocation of storage for variables.

In NPL, four syntactically different methods are used to accomplish the four functions mentioned above.

Blocks

In specifying a block to delimit scope of names but not to be called out-of-line it is inefficient to be prepared to store register contents and return locations. Therefore, PROCEDURE ... END are used for the first and second purposes and BEGIN ... END for the second purpose. The procedure may be thought of as a block with the additional properties of argument handling and return mechanisms.

Groups

The grouping of a set of statements for control purposes requires a much simpler and more common structure than one in which the scope of names is delimited. Therefore, in NPL, DO ... END are used for the third purpose. The DO statement also takes on the naturally related function of loop control.

Storage Allocation

Unlike the first three purposes discussed above, storage allocation frequently requires dynamic rather than static structure. In fact, the allocation structure may not even be well nested. Therefore, in NPL, an artificial correspondence between scope of names and storage allocation is rejected in favor of a scheme which allows the programmer to specify for a variable the appropriate treatment in each of the two categories.

A variable may be EXTERNAL or INTERNAL depending on whether its name is or is not known to other blocks. Its allocation may be STATIC, AUTOMATIC, or CONTROLLED. An AUTOMATIC allocation takes place upon entry to a block, storage being freed or unallocated upon exit from the block. A STATIC allocation is one made once for the entire execution of the program.

If a variable is declared to be CONTROLLED, allocation (or freeing) of storage takes place when and only when an explicit ALLOCATE (or FREE) statement is encountered during program execution. To meet the need for flexibility, a built-in procedure ALLOCATION (arg) returns the value 1 (True) if and only if allocated storage exists for arg.

In keeping with the philosophy of simplicity of expression for straight-forward applications, the default allocation attribute for EXTERNAL data is STATIC while the default for INTERNAL is AUTOMATIC.

Storage allocation, whether automatic or controlled, causes previous storage allocated for the given variable to be stacked. Similarly, freeing results in the previous allocation being unstacked.

Procedures

NPL procedures may have multiple entry points. Thus, the initialization part of a procedure may be invoked the first time and the usual entry called thereafter. The parameters of the two entries need not agree.

Normal return of control from a procedure to the invoking procedure is specified by a RETURN statement, which may specify return with a function value.

TYPES OF DATA

In NPL, a variable is described in a DECLARE statement. In order to improve documentation and because of the number of attributes specifiable, all of the attributes of a variable are listed together rather than including the variable name in the list for each attribute. Common attributes may be factored, however, to reduce the amount of text.

For every category of attribute there is a default which is the one most frequently used by the novice programmer. The programmer may therefore choose to ignore the existence of a facility without concern for the related attributes and their meaning. As an example, the programmer would not normally need to declare a storage class attribute for an item since, by default, allocation is static for EXTERNAL variables and automatic for INTERNAL variables.

Data are basically of two types, string and numeric. String data may be either CHARACTER string or BIT string and may be declared to be of fixed or varying length. Numeric data may be of two radices, binary and decimal; two scales, fixed and float; and two modes, real and complex. The size of numeric data, in bits if binary or digits if decimal, is specifiable directly, as is the location of a binary or decimal point. Such data is stored internally in a standard encoded representation.

If the programmer wishes he may specify a nonstandard representation by means of a picture, thereby attaching numeric significance to a string. Picture specifications apply to input/output formatting as well. This facility allows the programmer to specify conveniently zero suppression , insertion of blanks and other special characters, and general editing.

Conversion

In keeping with the freedom of expression concept, mixed expressions are allowed in NPL. Thus, in

```
DECLARE F FIXED, G FLOAT, H CHARACTER(10);
H = F + G;
```

F will be converted to floating, the floating addition will be performed, and the result will be converted to a character string of length ten and assigned to H.

Aggregates

Aggregates of data are defined in NPL as arrays or structures. Structures are defined by a level number notation, and their elements are represented by qualified names.

```
DECLARE 1 RECORD1,
          2 NAME,
            3 LAST CHARACTER(14),
            3 FIRST CHARACTER(6),
          2 ADDRESS,
            3 STREET CHARACTER(20),
            3 CITY CHARACTER(12),
            3 STATE CHARACTER(8),
          2 AGE CHARACTER(2);
```

defines a structure containing name, address, and age information, the name and address portions being further structured. The qualified name,

```
RECORD1.ADDRESS.STATE
```

represents the fifth elementary item in the structure.

Array elements are represented in the conventional manner, by subscripting. For example, the statement

```
DECLARE Q(3,5);
```

defines Q to be a 3x5 array. Q(2,3) represents the element in the second row, third column.

The concept of cross sections of arrays is introduced in NPL as a logical extension of the subscripting notation. If Q is defined as above, Q (2,*) denotes the second row of the matrix while Q (*,1) refers to the first column, the * indicating that the corresponding subscript is to vary between its defined bounds. Q(*,*) is therefore equivalent to Q, meaning the entire array.

To many programmers, the word "variable" has always meant a single item which may assume one value at a time. Yet, in matrix algebra, an entire matrix may be treated as a variable. For example, one value of a matrix might be the identity matrix, another value the zero matrix, another value a matrix consisting of all 5's, etc. In this concept, if any one element of the matrix changes its value, the entire matrix has changed its value.

In NPL, arrays and structures are treated as variables in their own right. Arrays (or structures) may be used as operands of an expression. The expression is then an array (or structure) expression and returns an array (or structure) result.

Additionally, if several structures have elements with identical names, operations may be specified on these structures to be applied only to these corresponding elements. For example, if in addition to RECORD1 defined above, there is the further declaration

```
DECLARE 1 RECORD2,
          2 ADDRESS,
            3 STATE CHARACTER(8),
            3 CITY CHARACTER(12),
            3 STREET CHARACTER(20),
          2 OCCUPATION CHARACTER(10),
          2 NAME,
            3 FIRST CHARACTER(6),
            3 LAST CHARACTER(14);
```

then the assignment statement

```
RECORD2 = RECORD1, BY NAME;
```

would cause the values of the elements of NAME and ADDRESS in RECORD1 to be rearranged and assigned to the corresponding elements of RECORD2. AGE and OCCUPATION do not participate in the operation.

All operations performed on arrays are performed on an element by element basis. Therefore, all arrays in an array expression must be of identical bounds.

Built-in functions are provided in NPL to assist the compiler in producing efficient in-line code and/or in selecting the appropriate member of a family of functions available.

Several built-in functions help to provide a string handling capability. BIT and CHAR allow arithmetic data to be treated as a string. SUBSTR (string,m,n,) refers to the n bits (or characters) of string beginning with the m'th bit (or character). INDEX(a, b) finds the first occurrence of the string b in the string a. UNSPEC (item) is a bit string whose value is the internal representation of item.

When the elements of a structure are all character strings or all bit strings the structure may be treated as a string by use of the STRING built-in function. Unedited transfers of collections of data may be accomplished in this manner.

INPUT/OUTPUT

One of the most successful fulfillments of the design criteria for NPL is its broad input/output facility. Here, in a machine independent manner, the programmer may control input/output activity to whatever degree he requires, invoking normal transmission and conversion simply, and utilizing the full capability of the language, in a consistent manner, to meet more sophisticated needs.

The programmer who uses the standard input and output formats and media may cause data-directed transmission without resorting to format or file descriptions.

List-directed transmission assumes a one-to-one correspondence between data names and data elements and permits specification of data element delimiters other than the standard.

Format-directed input/output is accomplished in a conventional way by giving a list of data names and a corresponding format specification list. The format list may appear in the READ or WRITE statement or in a remote FORMAT statement.

The most general form of input/output specification is the CALL option which invokes a programmer's procedure as part of the input/output process. This option allows the full NPL language facilities to be used in describing the transmissions. The procedure assumes a record or logical grouping of data has been read (in the case of input) or is to be written (in the case of output). Whereas READ and WRITE statements generally involve an entire record, by means of GET and PUT statements within the invoked procedure one may secure or dispose of portions of a record. Thus the programmer might use a GET statement to transmit the value of a key and then by testing that value determine what list and/or format should apply in getting the next portion of the record, or whether to process the remainder of the record at all.

A wide variety of options may be specified in a READ or WRITE statement. For example, one may specify by means of the PRINT option that data transmitted in the corresponding READ statement also be written on the standard output file.

By means of the KEY option one may specify that a specific record is to be transmitted or that a programmer-supplied selection procedure is to be invoked to locate the next record to be transmitted. In this manner, a sequence may be imposed on what otherwise would be a random file, or the implied sequence of a sequential file may be overridden. Thus a user may define and refer to files chained in random storage or files which are indexed by a dictionary.

The SEARCH statement enables one to access a record from a random file based either on a logical key address or on the content of the record. The user may specify the beginning and end of the search and may specify the action to be taken on successful completion of the search.

The programmer may specify in an OPEN or CLOSE statement that a file be opened or closed and positioned at a specific point. If no explicit OPEN is given, the file is opened when the first READ or WRITE statement is encountered.

Useful report generation facilities are available to the programmer in the form of statements which enable him to describe a printed page (with heading, footing, sub-total lines, pagination, etc.). Other statements permit setting tab positions and margins on a line, restoring a page, and skipping to a specified line.

The full conversion facilities of input/output are available to the user for internal data transmission. Internal transmission is accomplished by specifying a string name instead of a file name in the READ or WRITE statement. Scattering and gathering of collections of data may be accomplished in this manner.

READ,WRITE,OPEN,CLOSE, and SEARCH may all be invoked asynchronously by means of the TASK option as discussed in the following section.

NONSEQUENTIAL FACILITIES

This section and the next describe facilities that are available for use in more advanced programs.

When writing programs for a multiprocessor data processing system, or for a single processor system with either overlapped input/output facility or real time processing requirements, it is necessary to be able to specify concurrent execution of portions of a program. This is a critically important and relatively new requirement that has received great attention in the design of NPL. Programmers normally describe concurrent execution as either asynchronous operations or as interrupts. NPL allows both, or any mix of the two, so that programs can be approached in the way which seems most natural and leads to most efficient code.

Asynchronous Operations

In languages that can describe only sequential algorithms, it is possible to confuse two different concepts:

1. The program, which is a collection of procedures loaded into storage as needed.
2. The execution of one, or many programs, or of part of a program, to perform some task upon some data.

In NPL, this confusion is not possible. The collection of procedures is called a program, and that which has a job to do, a task. Sequential languages describe a single task executing a single program.

Suppose P is an NPL program, consisting of procedures P1, P2, ..., PN. A task A wishes to execute P, starting, say, at P1. If there is a place in P1 when concurrent execution of the part of P beginning at P2 is possible, the programmer writes,

```
CALL P2 (arg1, arg2, ...), TASK (B);
```

This statement, identical to a sequential call except for the TASK option, causes the creation of a second task, called B, which will begin its execution at procedure P2 simultaneously with A's execution of the rest of its job.

It is possible to specify the relative priority of B with respect to A, as a second argument to the task option.

B can communicate with A by the explicit arguments listed at the call or through shared storage.

Often both P1 and P2 will invoke P3. In this case, P3 must be able to be executed simultaneously by two tasks. This requirement is called re-entrance and is a declarable attribute of NPL procedures. It imposes several restrictions on the object code. The code must not modify its own instructions (i.e., it must be read-only). It must also refer to all data areas indirectly through the task which is in control. The property is extremely useful in such programs as message processors and central control programs of a multi-terminal system.

Once task A has created B, it proceeds with its own execution. It may come to a point where no further execution is possible until B has been completed. The NPL programmer writes,

```
WAIT (B);
```

This causes execution of task A to be suspended until task B is completed. By writing the statement DELAY (n), the programmer will cause the current task to go into wait status for n milliseconds before resuming execution.

It may be that A would like to discover whether B is complete or not, but is not willing to give up control. The programmer then can use the built-in procedure COMPLETE (B), which has the value 1 (True) if, and only if, B is completed.

A task may be terminated either by returning up past the main procedure for that task, with a RETURN statement, or by the explicit EXIT statement. An entire family of tasks may be terminated by the execution of the STOP statement in any member of the family.

Interrupt Operations

Whereas asynchronous operations involve one task asking for the initiation of another task, and later verifying its completion, interrupt operations involve the establishment of what code should be executed when, later, some event occurs.

There is an executable statement in NPL which is powerful. It is written:

ON condition action;

For example,

```
ON OVERFLOW Y = YMAX;
```

enables an asynchronous interrupt of the task which executes it when the specified condition occurs (regardless of what procedure the task is executing) so that the specified action may be taken. This action consists of a group of any NPL statements optionally preceded by the word SNAP, indicating the writing of machine status information for later inspection. The statements may contain a GO TO out of the group, which implies that control will never return to the point of interrupt.

The conditions fall into three categories:

DEBUGGING AIDS: These are programmed interrupts which can check whether subscripts are out of range, can document every possible change of value of a set of variables, and trace every execution of a set of statements, e.g.,

```
ON SUBSCRIPT RANGE SNAP COUNT = COUNT + 1;
```

UNUSUAL CONDITIONS: The programmer may override the system action on most machine interrupts, such as overflow, underflow, end-of-file, transmission error, or many system interrupts such as conversion error, and fixed-point overflow. There are built-in functions in NPL to help detect the cause of error and correct it.

CONCURRENT EXECUTION: The enabled condition may be a programmer-defined name. He can simulate a machine interrupt by executing the statement SIGNAL, at which time control is interrupted exactly as if the machine interrupt had occurred.

ON statements have as scope the block in which they appear. They may be stacked (for each condition) in push-down fashion as blocks and procedures are invoked, and unstacked on returns. In the same block, conditions may be overridden, or unstacked (by the REVERT statement).

COMPILE-TIME FACILITIES

A programmer describes an algorithm for the solution of his problem. The description may be processed by several programs (such as preprocessor, compiler, loader) and finally executed in its machine language

version. A subset of the information present in the description is used by each of these programs. Previous languages have addressed themselves almost entirely to the last phase, the execution. What attention is given to the other phases consists almost entirely of statements about the nature of the data.

And yet, as compilers become more sophisticated and preprocessors more efficient, much processing is performed before execution time. Common subexpressions are found and evaluated only once; constant expressions are evaluated at compile time; statements which will never be executed are not compiled at all. But this task is difficult and limited without the active cooperation of the programmer. Allowing him to help not only leads to efficient code, but results in more natural problem-oriented languages which are compatible dialects of the base language, and provides for compile-time editing of large general-purpose programs for special applications.

The compile-time facilities in NPL can be categorized as follows:

Hints and Commands to the Compiler

The NPL programmer may include in his program information which will aid the compiler to compile more quickly or to produce more efficient code, documentation, and diagnostics.

He can impart special information relevant only to some compilers by the open-ended attribute OPTIONS (attribute 1, attribute 2, ...). He can describe some characteristics of another procedure which is to be invoked. These characteristics may include the exact nature of each argument, what data it will use (via USES attribute), what data it will change (via SETS attribute), whether it has side effects or will sometimes produce different results with the same set of arguments (via ABNORMAL attribute). He can specify, via ABNORMAL attribute, that a variable is subject to change from outside, in a multiprocessing environment or due to asynchronous interrupt. He can suggest, via SECONDARY attribute, that if high-speed storage is unavailable, this table or procedure should be stored in secondary addressable storage. He can declare the set of statement labels to which a GO TO can transfer (by a list appended to the LABEL attribute).

Compile-Time Statements

Most programming languages are written explicitly on one level only, as statements to the computer to perform certain operations on the data. As stated above, any higher level assertions that are present must be ferreted out by an intelligent compiler. NPL not only commands the computer to operate on the data but also commands the compiler to operate on the program. This operation on the program determines the statements to be constructed and compiled. Compile-time statements are ordinary NPL statements distinguished by being immediately preceded by a %. A set of compile-time statements operates on the program to determine which source statements will be compiled. The following kinds of statements are allowed:

DECLARATIONS: Variables and procedures may be declared in compile-time declarations. These are called macro variables and macro procedures (see below).

ASSIGNMENTS: Macro variables may be assigned new values during compilation by execution of compile-time assignment statements.

CONDITIONAL COMPILATION: The compile-time statement IF macro Boolean expression THEN group causes the group of statements after the THEN to be compiled only if the macro expression is True.

TRANSFER OF CONTROL: All compile-time statements may have labels. The compile-time statement,

```
GO TO label;
```

causes compilation to proceed starting from the compile-time label specified. For example, to generate a series of similar statements,

```
% DECLARE I FIXED INITIAL (0), LABEL CHARACTER (2);
%L:I=I+1;
%LABEL='L'||I;
LABEL: X(I)=Y(I)+I;
%IF I<4 THEN % GO TO L;
```

will compile the following NPL statements:

```
L1:X(1)=Y(1)+1;
L2:X(2)=Y(2)+2;
L3:X(3)=Y(3)+3;
```

Macro Variables and Procedures

In order to aid the compile-time facility described above, to facilitate program modification, and, most important, to allow for a reasonably efficient, easy-to-specify development of the language into many different problem-oriented dialects, a macro facility has been included in NPL, as follows:

MACRO VARIABLES: A variable may be declared in a compile-time declaration statement. It may be given an initial value there; it may have values assigned to it by compile-time assignment statements. Whenever a compile-time statement which contains it is executed, its current value is used. Whenever a base language statement is being compiled, all appearances of any macro variable in this statement result in the current value of the variable being substituted for each occurrence. For example,

```
%DECLARE X FIXED INITIAL (3),
Y CHARACTER (10) VARYING INITIAL ('JOE+2');
L: Q=Y+X;
```

Compilation of statement L will first produce, as the statement to be compiled:

```
L: Q=JOE+2+3;
```

Subsequent optimization could compile:

```
L: Q=JOE+5;
```

MACRO PROCEDURES: A procedure may be declared in a compile-time declaration statement. It may, or may not, require an argument list. Whenever a base language statement which contains a reference to this procedure is encountered by the compiler, the procedure is first invoked and its returned value substituted for its appearance. Note that since all data to the compiler are character strings, all macro procedures must have character string values.

Thus one can introduce special purpose statements and make them appear to be part of the NPL language.

SUGGESTIONS FOR USE OF THIS PUBLICATION

This publication is a language definition, intended to be used primarily as a reference manual. It has not been designed as a tutorial document. In this section an attempt is made to guide the user according to his programming experience and application requirements.

Basic components of the language of interest to every reader, no matter what his level of experience or area of interest, are found in Chapter 2 in its entirety; Chapter 3 in its entirety; Chapter 5 in its entirety; Chapter 6, "Function References" through "Array Expressions"; Chapter 7, "Statements" through "The Procedure Statement," and "Programs and Procedures" through "Procedure and Block Termination"; Chapter 9, "Subroutine Procedures" and "Functions"; Chapter 11, "Scalar Assignment"; Chapter 12 in its entirety; Chapter 13, "The GO TO Statement" through "The RETURN Statement"; Chapter 15, "Attribute Classes" through "Label Variable Attributes," "Scope Attributes," "The INITIAL Attribute," and "Symbol Table Attributes"; Chapter 16, "Name Declaration" through "The DECLARE Statement"; Chapter 20 in its entirety; Chapter 22, "List Directed Transmission" et seq.; Chapter 23, "The READ and WRITE Statements"; Chapter 25 in its entirety; and Appendices 7 and 8.

The commercial or business program will frequently involve facilities such as the definition and use of structures and the use of picture specifications for editing and numeric computation. The pertinent sections are Chapter 4, "Structures"; Chapter 6, "Structure Expressions" and "Structure Expressions BY NAME"; Chapter 11, "Simple Structure Assignment" and "BY NAME Structure Assignment"; Chapter 15, "The LIKE Attribute"; Chapter 16, "Structures"; and Appendix 6. These facilities may be less useful in sientific programs.

On the other hand, other NPL features are vital to the solution of purely scientific problems. These are concerned primarily with arrays and mathematical functions, discussed in Chapter 4, "Arrays"; Chapter 6, "Structure Expressions"; Chapter 11, "Array Assignment"; Chapter 15, "The Dimension Attribute"; and Appendix 1.

The more advanced programmer, whether scientifically or commercially oriented, will be interested in the subject of storage allocation and related subjects. His attention is drawn to Chapter 7, "The ENTRY Statement" and "The BEGIN Statement"; Chapter 8 in its entirety; Chapter 9, "The ENTRY Attribute" through "Abnormality of Procedures"; Chapter 10 in its entirety; Chapter 11, "Statement Label Assignment"; Chapter 14, "The ON Compound Statement"; Chapter 15 in its entirety; Chapter 22 in its entirety; Chapter 23 "The GET and PUT Statements"; and Appendix 5.

The remaining sections provide additional topics of interest, such as program modification and asynchronous operation.

CHAPTER 2: CHARACTER SET AND IDENTIFIERS

The NPL language uses a basic character set of 60 characters. All the elements of the language (names, constants, etc.) are constructed from these characters. However, data in an NPL program is not restricted to the basic character set, but can be any character or bit pattern (legal card-column punch combination) permitted by a particular implementation.

A specific use of some of the basic characters is in identifiers, which are names, statement labels, keywords, etc., that appear in an NPL program.

LANGUAGE CHARACTER SET

The NPL language is constructed from the following basic characters: alphabetic characters, digits, and special characters.

The alphabetic characters are the 26 characters of the alphabet, A through Z, and, in addition, three characters that are defined to be, and are treated as, alphabetic characters. These additional characters and the graphics by which they are represented in this manual are given in the following list.

Name	Graphic
Currency symbol	$
Commercial At sign	@
Number sign	#

Digits are either decimal or binary. Decimal digits are the digits 0 through 9. A bit (binary digit) is either a 0 or a 1.

The names of the special characters used in the language and the graphics by which they are represented in this manual are given in the following list.

Name	Graphic
Blank	
Equal or Assignment symbol	=
Plus	+
Minus	-
Asterisk or Multiply symbol	*
Slash or Divide symbol	/
Left Parenthesis	(
Right Parenthesis	)
Comma	,
Decimal Point or Period	.
Quote	'
Percent symbol	%
Semicolon	;
Colon	:
Not symbol	¬
And symbol	&
Or symbol	|
Greater Than symbol	>
Less Than symbol	<
Break Character	_
Question Mark	?

DATA CHARACTER SET

Characters permitted in data are defined for each particular implementation.

IDENTIFIERS

An identifier is a string of alphabetic characters, digits, and break characters with the initial character always alphabetic. Any number of break characters are allowed within an identifier; however, consecutive break characters are not permitted. Also, a break character cannot be the final character of an identifier.

Identifiers in the language are used for scalar variable names, array names, structure names, statement labels, entry names, file names, keywords, task identifiers, condition names, headings for external names, macro variable names, and macro function names.

Examples:

```
A                   #32_45
$L32                BCD320
X@_52               XR20A
RATE_OF_PAY         @531
```

Information that is operated on by an NPL object program during execution is called data. Each data item has a well determined type and representation as described in the following paragraphs.

DATA TYPES

The permitted data types are: arithmetic, character string, bit string, and label.

ARITHMETIC DATA

Arithmetic data is represented either as a numeric field or in coded form. A numeric field is a string of characters which is given a numeric interpretation using the PICTURE attribute. The picture may include editing characters, such as currency symbols or commas, which are ignored during arithmetic computation. The absence of the PICTURE attribute specifies that arithmetic data is of the coded form.

Arithmetic data has the characteristics radix, scale, mode, and precision. These characteristics are implicitly specified for numeric fields in the PICTURE attribute. They may be explicitly declared for arithmetic data of coded form.

Radix

The permitted radices are decimal and binary.

Scale

The permitted scales are fixed point and floating point. Fixed-point data consists of rational numbers for which the number of decimal or binary digits and the position of the decimal or binary point may be specified. Floating-point data consists of rational numbers considered in the form of a fraction and an exponent; the number of significant digits may be specified.

Mode

Arithmetic data may be operated on in two modes, real and complex.

Precision

The precision of fixed-point data involves two quantities

1. the total number of decimal or binary digits to be maintained (w).
2. the scale factor for the data (d); If d is omitted, it is assumed to be zero. The scale factor may be negative; its magnitude need not be less than (w). If r is the radix of the data, then the scale factor, d, effectively multiplies the w-digit integer data by r**-d. For example, decimal data of precision (5,2) will represent numbers less than 1,000 and at least 0.01 in magnitude.

The preceding values are specified as either (w,d) or (w).

The precision of floating-point data is the number of significant binary or decimal digits to be maintained. This value is specified as (w).

CHARACTER STRING DATA

Character string data consists of strings of characters. A character string may be of fixed or variable length.

BIT STRING DATA

Bit string data consists of strings of bits. A bit string may be of fixed or variable length. In the former case, the actual length may be specified; in the latter case, the maximum length may be specified.

STATEMENT LABEL DATA

Statement label data consists of statement labels (see "Labels" in Chapter 7).

CONSTANTS

A constant is a data item that cannot take on different values during the execution of a program. The types of constants permitted in NPL are described in the following paragraphs.

REAL ARITHMETIC CONSTANTS

Arithmetic constants are of radix binary or decimal. Both radices use a decimal representation.

Decimal Fixed-Point Constants

A decimal fixed-point constant is represented by one or more decimal digits with an optional decimal point.

Examples:

```
72.192
.308
255
```

Binary Fixed-Point Constants

A binary fixed-point constant is represented by a decimal fixed-point constant followed by the letter B.

Examples:

```
127B
3.24B
.001B
```

Sterling Fixed-Point Constants

Sterling quantities may be specified and will be interpreted as decimal fixed-point pence. A sterling quantity consists of the following concatenated fields:

a pounds field that is a decimal integer
a period
a shillings field that is a decimal integer less than 20
a period
a pence field that is one or more decimal digits with an optional decimal point. The integral part must be less than 12.
an L

Examples:

101.13.8L
1.10.0L
0.0.2.5L

Decimal Floating-Point Constants

A decimal floating-point constant is represented by one or more decimal digits with an optional decimal point, followed by the letter E, followed by an optionally signed decimal exponent.

Examples:

12.E23
317.5E-16
0.1E+03
0.42E+73

Binary Floating-Point Constants

A binary floating-point constant is represented by a decimal floating-point constant followed by the letter B.

Examples:

27E+3B
.123E-45B

THE PRECISION OF REAL ARITHMETIC CONSTANTS

Real decimal fixed-point constants have apparent precision (p,q) where q significant digits are specified after the decimal point, and (p-q) before the decimal point.

Real binary fixed-point constants have apparent precision (p*3.32,q*3.32) where p and q are as defined above. The ceiling of these products is used. (The ceiling of a number is the smallest integer greater than the number.)

Real decimal floating-point constants have apparent precision (p) where p significant digits are specified before the E.

Real binary floating-point constants have apparent precision (p*3.32) with p defined as above. The ceiling of the product is used.

Implementations may specify an assumed minimum precision for constants which are involved in expression evaluation, and apply the minimum precision to a constant if its apparent precision is less.

IMAGINARY ARITHMETIC CONSTANTS

An imaginary constant represents a complex value whose real part is zero, and whose imaginary part is the value specified.

An imaginary constant is represented by a real constant, other than a sterling constant, followed by the letter I. The language does not define complex constants with nonzero real parts but provides the facility to specify such data through expressions, e.g., 10.1+9.2I.

Examples:

```
27I
3.968E10I
```

BIT STRING CONSTANTS

A bit string constant is one or more binary digits enclosed in quotes followed by the letter B. The constant may optionally be preceded by a decimal integer constant in parentheses to specify replication.

Examples:

```
'01011'B
(10)'1'B
```

The latter is exactly equivalent to

```
'1111111111'B
```

CHARACTER STRING CONSTANTS

A character string constant is one or more characters enclosed in quotes. A quote mark used in a character string constant is represented by two immediately adjacent quote marks. The constant may optionally be preceded by a decimal integer constant in parentheses to specify replication.

Examples:

```
'$123.45'
'INBUSLAB'
'IT''S'
(3)'TOM'
```

The latter is exactly equivalent to

```
'TOMTOMTOM'
```

STATEMENT LABEL CONSTANTS

A statement label constant is an identifier which appears in the program as a statement label (see "Labels" in Chapter 7).

SCALAR VARIABLES

A scalar variable may take on values over one and only one data type, and, in the case of type arithmetic, only one radix, scale, mode and precision. If its range is not restricted, it may assume values over the entire set of data of that type.

A scalar variable is represented in the language by a name that is an identifier, a qualified name, a subscripted name, or a subscripted qualified name.

CHAPTER 4: DATA AGGREGATES

A data aggregate is a variable which may take on a set of data as a value. In NPL, this type of variable is either an array variable or a structure variable.

ARRAYS

An array is an ordered collection of data, all of which must be of the same type. Arithmetic data in an array must be of the same radix, scale, mode and precision, and, where applicable, the same picture. String data must have the same length (if fixed) or maximum length (if variable).

An array name is declared as an identifier with the dimension attribute (see "The Dimension Attribute" in Chapter 15). This specifies the number of dimensions of the array, and the upper and lower bounds of each dimension.

Reference to an array in the language is by an array name, which may be an identifier, a qualified name, a subscripted name, or a subscripted qualified name.

SUBSCRIPTED NAMES

General Form:

array name (subscript 1 , ... , subscript n)

An element of an array is referenced in the language by a subscripted name which is an array name followed by a list of subscripts. The subscripts are separated by commas and the list is enclosed in parentheses. A subscript is an expression which is evaluated and converted to integer before use (see Chapter 6). The number of subscripts must be equal to the number of dimensions of the array and the value of a specified subscript must fall within the bounds declared for that dimension of the array.

Examples:

```
A (3)
FIELD (B,C)
PRODUCT (SCOPE * UNIT + VALUE, PERIOD)
ALPHA (1,2,3,4)
```

CROSS SECTIONS OF ARRAYS

A cross section of an array is represented in the language by an array name, followed by a list of subscripts and asterisks, separated by commas, and enclosed in parentheses. The number of items in the list must be equal to the number of dimensions of the array. If the nth list position is occupied by an asterisk, the cross section of the array includes elements covered by varying the nth subscript between its bounds. The dimensionality of the cross section is equal to the number of asterisks in the subscript list. If all subscript positions are occupied by asterisks, this is equivalent to a reference to the entire array. Subsequently, this document will use the word "array" to include cross sections of arrays.

Examples:

```
A(*, J)
B(X, *, Y, *)
```

```
                              1  2  3
If MATRIX is the array        4  5  6
                              7  8  9
```

```
                                 2
MATRIX(*, 2) is the vector       5
                                 8
```

CONSTRUCTED ARRAYS

Scalars or arrays of the same dimensionality may be collected in a form that is considered to be an array by the following notation:

ARRAY $(A_1,A_2,....,A_n)$ All A are array expressions of dimensionality m with the same bounds. The function value is an array of dimensionality m+1 with bounds (1:n) for the first dimension, and the bounds of the A for the next m dimensions. The A will be converted to the highest type, radix, scale, mode, and precision of the arguments.

STRUCTURES

A structure is an hierarchical collection of scalars, arrays, and structures. These need not be of the same type and characteristics.

Structures may contain structures. The outermost structure is the major structure; contained structures are minor structures. A major structure is at level 1. Items contained in structures at level n are at levels greater than n.

Example:

```
1 A,2 B,2 C,3 D(2),3 E,2 F
```

has form

```
A ┌B
  │C ┌D(1)
  │  │D(2)
  │  └E
  └F
```

ARRAYS OF STRUCTURES

A structure may be given a dimension attribute; it is then an array of structures. All contained items are arrays as a result of the structure having a dimension attribute.

Examples:

```
1 CARDIN(3), 2 NAME, 2 WAGES, 3 NORMAL, 3 OVERTIME
```

The decimal integers before the identifiers specify the level. The name CARDIN represents a one-dimensional structure of bounds 1 to 3. Each element of the structure contains the variable NAME and the minor structure WAGES. WAGES contains the variables NORMAL and OVERTIME.

Because CARDIN is dimensioned, NAME, NORMAL and OVERTIME are arrays, and their elements are referred to by subscripted names.

The form of the data is illustrated as follows:

```
CARDIN (1)   ┌NAME (1)
             |WAGES (1)   ┌NORMAL (1)
             └            └OVERTIME (1)
CARDIN (2)   ┌NAME (2)
             |WAGES (2)   ┌NORMAL (2)
             └            └OVERTIME (2)
CARDIN (3)   ┌NAME (3)
             |WAGES (3)   ┌NORMAL (3)
             └            └OVERTIME (3)
```

```
1 X, 2 Y, 2 Z(2), 3 P(2:3,2), 3 Q, 2 R
```

X is an undimensioned major structure
Y is a scalar variable
Z is a dimensioned structure containing P and Q
P is an array
Q is an array
R is a scalar variable

The form of the data is

```
    ┌Y
    |          ┌P (1,2,1)
    |          |P (1,2,2)
    |Z (1)     |P (1,3,1)
    |          |P (1,3,2)
X   |          └Q (1)
    |          ┌P (2,2,1)
    |          |P (2,2,2)
    |Z (2)     |P (2,3,1)
    |          |P (2,3,2)
    |          └Q (2)
    └R
```

QUALIFIED NAMES

At any point within a program, an identifier usually has only one use specified by the programmer; however, an identifier may have more than one use if all uses represent elements of structures. The separate uses are then referred to by qualified names, which are a sequence of containing structure names in order of increasing level, followed by the ambiguous identifier. The items are separated by periods; blanks may be placed as desired around the periods. The sequence need not include all the containing structures, but it must include sufficient items to resolve any ambiguity.

The qualified name, once composed, is itself a name. Subsequently in this language document, when the terms scalar variable name, array name, or structure name are used, they should also be taken to include qualified names.

Examples:

A program may contain the structures

```
1 CARDIN, 2 PARTNO, 2 DESCRIPTION, 2 PRICE and
1 CARDOUT, 2 PARTNO, 2 DESCRIPTION, 2 PRICE
```

Elements are then referred to by qualified names such as

```
CARDIN.PARTNO
CARDOUT.PARTNO
CARDIN.PRICE
```

A program may contain the structure

```
1 MARRIAGE, 2 MAN, 3 NAME, 3 DATE, 2 WOMAN, 3 NAME, 3 DATE
```

Elements are then referred to by qualified names such as

```
MAN.NAME or MARRIAGE.MAN.NAME
WOMAN.NAME or MARRIAGE.WOMAN.NAME
```

If the same program also contained the structure

```
1 BIRTH, 2 WOMAN, 3 NAME, 3 DATE, 2 COMPLEXION
```

Elements must then be referred to by qualified names such as

```
MAN.NAME or MARRIAGE.MAN.NAME
MARRIAGE.WOMAN.NAME
BIRTH.NAME or BIRTH.WOMAN.NAME
COMPLEXION
```

and the minor structures by

```
MARRIAGE.WOMAN
BIRTH.WOMAN
```

SUBSCRIPTED QUALIFIED NAMES

General Form:

major structure name (subscript list) minor structure name (subscript list) ... array name (subscript list)

The elements of an array contained in a structure and requiring name qualification for identification are referred to by subscripted qualified names. A subscripted qualified name is a sequence of names and subscripted names separated by periods. The names represent the structures containing the array, followed by the array name. The structure names must be in order of increasing level. The subscript list following each name refers to the dimensions associated with that name (as specified in the structure description). If no dimensions are associated with a particular name in the list, the subscript list and its containing parentheses may be omitted.

As long as the order of the subscripts remains unchanged, subscripts may be moved to the right and attached to names at a deeper level.

Provided that the total number of subscripts is the same as the total dimensionality of the array and that no ambiguity of identification may occur, structure names may be omitted. Ambiguity of names cannot be resolved by subscripting.

Examples:

A is an array of structures with the following description

```
1 A(10,12), 2 B(5), 3 C(7), 3 D
```

The following subscripted qualified names refer to the same element of C:

(a) A(I,J) . B(K) . C(L)

(b) A(I) . B(J,K) . C(L)

(c) A(I) . B(J) . C(K,L)

(d) A.B(I,J,K) . C(L)

(e) A.B(I,J) . C(K,L)

(f) A.B(I) . C(J,K,L)

(g) A.B.C(I,J,K,L)

(h) A(I,J) . B.C(K,L)

(i) A(I) . B.C(J,K,L)

If B, but not A, is necessary for unique identification of this use of C, any of forms (d), (e), (f) or (g) may be used with the A. absent. If A, but not B, is necessary for unique identification of C, forms (g), (h), or (i) with B. omitted may be used. If neither A nor B is necessary for unique identification of C, form (g) with A.B. omitted may be used.

The basis for the elementary language structure of NPL is the basic character set described in Chapter 2. Single characters from the set, or strings of characters formed from the set, have specific properties or uses when they appear in an NPL program. In addition, certain items within the elementary language structure can have more than one use within the same program; but the different uses are recognized by context.

DELIMITERS

Delimiters used by the language fall into three classes, operators, brackets, and separators.

OPERATORS

Operators used by the language are divided into four types, arithmetic operators, comparison operators, bit string operators, and string operators.

The arithmetic operators are

```
+  denoting  addition
-  denoting  subtraction or negation
*  denoting  multiplication
/  denoting  division
** denoting  exponentiation
```

The comparison operators are

```
>  denoting  greater than
>= denoting  greater than or equal to
=  denoting  equal to
¬= denoting  not equal to
<= denoting  less than or equal to
<  denoting  less than
```

The bit string operators are

```
¬  denoting  not
&  denoting  and
|  denoting  or
```

The following table defines the bit string operators:

X	'1'B	'1'B	'0'B	'0'B
Y	'1'B	'0'B	'1'B	'0'B
¬X	'0'B	'0'B	'1'B	'1'B
X&Y	'1'B	'0'B	'0'B	'0'B
X|Y	'1'B	'1'B	'1'B	'0'B

Where bit string operands are of length greater than 1, the operations are performed bit by bit from left to right. Where operands are of different lengths, the shorter is extended on the right with zeros.

The string operator is

|| denoting concatenation

BRACKETS

Brackets used by the language are

(left parenthesis
) right parenthesis

Parentheses are used in expressions and to enclose lists.

SEPARATORS AND OTHER DELIMITERS

Separators and other delimiters are

,	comma		blank
;	semicolon	'	quote
=	assignment symbol	_	break character
:	colon	.	period

The comma is used for separating elements of lists.
The semicolon is used for terminating statements.
The assignment symbol is used in the assignment statement.
Quotes are used for enclosing string constants.
The break character is used in identifiers.
The colon is used in dimension specifications and to follow labels.
The period is used for separating items in qualified names, and as a decimal or binary point in constants.

KEYWORDS

A keyword is an identifier that has a special meaning. Keywords are not reserved words. They are classified as follows.

Statement Identifiers: A statement identifier is used to identify the nature of a statement. Some statement identifiers consist of more than one identifier, separated by blanks.

Attributes: Attributes specify identifier characteristics.

Separating Keywords: The separating keywords are THEN, ELSE, BY, TO, WHILE.

Built-in Function Names: A built-in function name is the name of an algorithm provided by the language and accessible to the programmer.

Options: An option is a specification which may be used by the programmer to influence the execution of a statement.

Conditions: A condition is used in the ON, SIGNAL, and REVERT statements. The programmer may specify special action on occurrence of the condition.

BLANKS

A word is an identifier, a constant, a picture specification (see "Arithmetic Attributes" in Chapter 15), or a sequence specification (see "The SEQUENCE Statement" in Chapter 17). Blanks are not allowed in words. In those cases in the language where two words lie adjacent, not

separated by an operator, an assignment sign, a parenthesis, a colon, a comma, or a semicolon, a blank is required to separate them. One or more blanks may appear freely between words and adjacent delimiters. Blanks are not permitted within composite operators like ** or >=.

```
CALLA         is not equivalent to CALL A
9.6E+2        is not equivalent to 9.6E +2
AB+BC         is equivalent to AB + BC
A TO B BY C   is not equivalent to ATOBBYC
```

COMMENTS

General Form:

```
/* character string */
```

A comment may appear anywhere that a blank is permitted. A comment may be replaced by a blank without changing the meaning of the program. The character string must not contain the characters */ in that sequence.

Example:

```
LABEL:/* THE BLOCK OF CODING BETWEEN BEGIN-END IS USED FOR COSH
CALCULATIONS */
          BEGIN;
            .
            .
            .
          END;
```

CHAPTER 6: FUNCTION REFERENCES AND EXPRESSIONS

Expressions are of three types: scalar, array, or structure. When expressions are evaluated, the type of the value returned is the type of the expression; i.e., the evaluation of a scalar expression returns a scalar result, etc.. However, an array or structure expression is evaluated on an element-by-element basis.
An expression can consist of a single operand, e.g., a function reference, or a constant, or multiple operands connected by operators, e.g., A+B/C or A||B. Operands in an expression need not have identical data attributes. If the characteristics of the operands are different, the necessary conversion is performed before evaluation.

FUNCTION REFERENCES

General Form:

function name (argument 1 , ... , argument n)

A function reference appearing in a program calls upon an algorithm to provide a value. In the language, the value of a function is represented by the function name followed by an optional list of arguments separated by commas. If arguments are not needed, the enclosing parentheses are omitted. (See Chapter 9 for a full discussion of functions.)

An argument may be a scalar expression, an array expression, a structure expression, a statement label designator, an entry name, an entry parameter, a file name, or a file parameter.

Examples of Function References:

```
SIN (X)
MAX (A,B,C)
PROFITCALCULATION (INFLATIONRATE,LOSS)
```

SCALAR EXPRESSIONS

Scalar expressions are expressions which return a scalar value. The type of the expression is the type of the scalar value. A scalar expression is evaluated by performing a sequence of operations, where an operation involves an operator and one or two operands. If one operand is involved, the operator is a prefix operator; otherwise, the operator is an infix operator.

Note: In this manual, the word 'expression' means 'scalar expression' unless explicitly qualified.

An operator has an associated type as follows:

** * / + -	operators of type arithmetic
¬ & |	operators of type bit string
>= > ¬= = < <=	comparison operators of context dependent type
||	string operator of context dependent type

Before the performance of every operation, the operands are converted to the type of the operator. In the case of the context dependent operators, the conversion is a function of the type of the operands (see "Arithmetic Operations" in Chapter 6).

The type of the value resulting from an arithmetic or a bit string operation is the type of the operation.

The type of the value resulting from a comparison operation is a bit string of length 1.

The type of the value resulting from a concatenation operation is a bit or character string.

The sequence in which the operations specified in a scalar expression are performed is described in "Type Conversion" in Chapter 6.

A scalar expression is

a constant
a scalar variable
a function reference
an expression enclosed in parentheses
any of the above preceded by a prefix operator
any two expressions connected by an infix operator

A scalar expression may not include statement label designators (see "Array Expressions" in this chapter).

BIT STRING OPERATIONS

General Forms:

¬ operand
operand & operand
operand | operand

These general forms specify the operations "not", "and", "or", respectively. The operands will be converted to type bit string before the operation is performed. The result will be of type bit. If the operands are of different lengths after conversion, the shorter is extended on the right with zeros to the length of the longer. The length of the result will be this extended length.

As an illustration of bit string operations, if Q, P, and R are bit strings whose values are, respectively, '010111'B, '111111'B and '101'B, then

```
      ¬Q yields '101000'B
    R&P yields '101000'B
   Q|¬R yields '010111'B
¬Q|¬R&P yields '111000'B
```

CONCATENATION OPERATIONS

General Form:

operand || operand

If the operands are both of type bit string, no conversion is performed and the result will also be of type bit. In all other cases, the operands are converted where necessary to type character string before the concatenation is performed. The result will be of type character.

As an illustration of concatenation, if Q and R are the same as above, and if W and X are character strings whose values are, respectively, 'AB,VV' and '?/PZ', then

```
   Q||R yields '010111101'B
R||R||Q yields '101101010111'B
   W||X yields 'AB,VV?/PZ'
   X||W yields '?/PZAB,VV'
```

ARITHMETIC OPERATIONS

General Forms:

```
+ operand
- operand
operand **  operand
operand  *  operand
operand  /  operand
operand  +  operand
operand  -  operand
```

These general forms specify the prefix operations, affirmation and negation, and the infix operations exponentiation, multiplication, division, addition and subtraction, respectively. Arithmetic operations require operands of type arithmetic. Thus, if necessary, the operands are converted to type arithmetic before the operation is performed.

Mixed Characteristics

The radix, scale, mode, or precision of the operands of an arithmetic infix operation may differ. When these characteristics are mixed, conversion is performed.

Consider an operation x op y, where x and y are operands, and op is an arithmetic operator. Let the result of the operation be z. Let the precision of x be (p,q) or (p). Let the precision of y be (r,s) or (r).

The rules for the conversion of operands and the characteristics of the results are as follows:

RADIX: If either operand is a constant, the radix of the constant is converted at compile time to the radix of the variable. If the two operands are both constants or both variables, and the radices differ, the binary operand is converted to decimal. The result is decimal.

SCALE: If either operand is a constant, the scale of the constant is converted at compile time to the scale of the variable. If the scales of the operands differ and the operation is not exponentiation, the fixed-point operand will be converted to floating point. The result will be floating point. If the operation is exponentiation, and the second operand (y) is a positive integer, the operation will be treated as repeated multiplication. The scale of the result will be the scale of the first operand (x). If the operation is exponentiation, and the second operand is not a positive integer, then the result will be floating point and will be found by an approximation method; the precision of the approximation method will be the precision of the first operand.

MODE: If either operand is complex, neither operand will be converted. The result will be complex.

PRECISION: This conversion depends on the scale of the operands, as follows:

Floating Point: Precision is defined for floating-point numbers as the number of digits carried in the representation of the fraction. The precision of a floating-point result will be the greater of the precisions of the two operands.

Fixed Point: The rules for fixed-point arithmetic use a symbol, N, which represents the length of the largest number in the implementation.

If the scale of the result (z) is FIXED, then the precision (m,n) of the result is related to the values t and u as follows:

Addition and Subtraction:
```
t = max (p - q, r - s) + u + 1
u = max (q,s)
```

Multiplication:
```
t = p + r
u = q + s
```

Division:
```
t = max (p,r)
u = q - s
```

Exponentiation:

If y is a positive integer constant
```
t = p* value (y)
u = q* value (y)
```

However, if (p*value(y)) > N or y is not a positive integer constant, conversion to FLOAT occurs.

The relation among t, u, m, and n is defined as follows:

If $t \leq N$, then $m = t$, $n = u$

If $t \leq N$, then $m = t$, $n = u$
If $t > N$, then $m = N$, $n = u$

The latter case implies that truncation is performed on the left.

The conversion from floating point to fixed point will occur only when a destination precision is known. The destination precision will define the conversion precision.

Arithmetic Mode Conversion

If a complex value is converted to a real value, the result is the magnitude of the complex value; i.e., the square root of the sum of the squares of the real and imaginary parts.

If a real value is converted to a complex value, the result is a complex value that has the real value as the real part and zero as the imaginary part.

Integer Conversion

Where conversion to integer is specified, as in the evaluation of subscript expressions, the conversion will be to coded FIXED BINARY (x,0), where x will be implementation defined.

Arithmetic Radix and Scale Conversion

The following table defines the precision resulting from radix and scale conversion. CEIL is a built-in function described in Appendix 1.

After	Before Conversion: Binary Fixed (p,q)	Decimal Fixed (p,q)	Binary Float (p)	Decimal Float (p)
Binary Fixed	(p,q)	(CEIL(p*3.32), CEIL(q*3.32))		
Decimal Fixed	(CEIL(p/3.32), CEIL(q/3.32))	(p,q)		
Binary Float	(p)	(CEIL(p*3.32))	(p)	(CEIL(p*3.32))
Decimal Float	(CEIL(p/3.32))	(p)	(CEIL(p/3.32))	(p)

COMPARISON OPERATIONS

General Form:

operand >= operand
operand > operand
operand ¬= operand
operand = operand
operand <= operand
operand < operand

The operation of comparison may be performed in three ways:

1. algebraic, implying comparison of signed numeric values.

2. character, implying left to right, pair by pair comparison of characters according to a collating sequence. If the operands are of different length, the shorter is extended with characters which compare low with all other characters. See "The SEQUENCE Statement" in Chapter 17 for details of collating sequence specification.

3. bit, implying left to right comparison of binary digits. If the strings are of different lengths, the shorter is extended on the right with zeros.

The result of a comparison is a bit string of length 1 that has the value '1'B if the relation is true and '0'B if it is false.

If the operands of a comparison are of different types, the operand of lower type is converted to the higher type. The priority of types is arithmetic, character string, bit string. As a result of conversion, both the operands will then be arithmetic, bit string or character string. Algebraic, bit, or character comparison, respectively, will then be performed.

TYPE CONVERSION

Bit String to Character String

The bit 1 becomes the character 1, and the bit 0 the character 0. The length is unchanged.

Character String to Bit String

The characters 1 and 0 become the bits 1 and 0. The conversion is illegal if the character string contains characters other than 0 and 1.

Character String to Arithmetic

The character string is interpreted according to the rules of list directed input; i.e., the contents of the string must be a valid constant, with optional sign prefixed, with optional surrounding blanks. The value is converted directly to an operand with the same radix, scale, mode, and precision that the constant designated by the string would have been converted to if it had appeared.

Bit String to Arithmetic

The bit string is interpreted as an unsigned binary integer, and converted to BINARY FIXED precision (S,0), where S will be implementation defined.

Arithmetic to Character String

CODED ARITHMETIC AND RADIX BINARY NUMERIC FIELDS: The arithmetic value is converted to a character string according to the rules of list directed output specified in Appendix 7.

NUMERIC FIELDS OF RADIX DECIMAL: The numeric field is interpreted as a character string (see Appendix 2).

Arithmetic to Bit String

CODED ARITHMETIC AND RADIX DECIMAL NUMERIC FIELDS: The magnitude of the arithmetic value is converted to BINARY FIXED precision (p,0), where p is related to the precision before conversion as follows (with ceilings of expressions used):

```
BINARY FIXED (r,s)      p = (r - s)
BINARY FLOAT (r)        p = (r)
DECIMAL FIXED (r,s)     p = (r - s) * 3.32
DECIMAL FLOAT (r)       p = r * 3.32
```

The sign is dropped and the resulting binary fixed value is interpreted as a bit string of length p.

NUMERIC FIELDS OF RADIX BINARY: The numeric field is interpreted as a bit string.

EVALUATION OF EXPRESSIONS

An expression may be enclosed in parentheses to force it to be considered as a single operand. The parenthesized expression is evaluated before the operation of which it is an operand is performed. If both operands of an operator are expressions, the left expression will be evaluated first. Thus parentheses modify the rules specifying the normal order of operations.

The priority of operations is

```
F   **  prefix +  prefix -          highest
E   *  /
D   infix +  infix -
C   >=  >  ¬=  <  <=
B   ¬
A   &
9   |
8   ||                              lowest
```

Subject to the rules associated with parentheses, operations within an expression are performed conceptually in order of decreasing priority. Thus, an exponentiation is effectively performed before an addition, and the latter before a string operation. The rules relating to abnormal functions and abnormal data should be noted (see "Abnormality of Procedures" in Chapter 9). If an interrupt resulting from an enabled ON condition (see Chapter 14) occurs while an expression is being evaluated, then the stage reached in the evaluation before the interrupt is undefined.

If an expression involves operations of the same priority, then, subject to the effect of parentheses, the operations **, prefix +, and - are performed from right to left and all others are performed from left to right.

Although operators * and + are associative, low order rounding errors will depend on the order of evaluation of an expression. Thus (A + B + C) is not necessarily equivalent to (A + C + B).

ARRAY EXPRESSIONS

The operands of an array expression are arrays or a mixture of scalars and arrays. The expression is then an array expression and returns an array result. All operations performed on arrays are performed on an element by element basis. Thus, all arrays appearing in any array expression must be of identical bounds. It is important to note that array expressions are not always expressions of conventional matrix algebra.

The result of the operation of a prefix operator or a built-in function upon an array is an array of identical bounds, each element of which is the result of the operator having operated on the corresponding element of the original array.

The appearance of a function reference (other than a built-in function) will imply a scalar operand. Thus, if A is an array, PROC(A) is a scalar function with an array argument.

Example:

```
                          5  3 -9
If A is the array         1 -2  7
                          6  3 -4

                         -5 -3  9
then -A is the array     -1  2 -7
                         -6 -3  4
```

The result of an operation in which a scalar and an array are connected by an infix operator is an array, of identical bounds to the original, each element of which is the result of the operation performed on the scalar and the corresponding element of the original array.

Example:

```
If A is the array          5 10  8
                          12 11  3

then 3*A is the array     15 30 24
                          36 33  9
```

The result of an operation in which two arrays of identical bounds are connected by an infix operator is an array of identical bounds to both original arrays, each element of which is the result of the operation performed on the corresponding elements of the two original arrays by the infix operator.

Example:

```
If A is the array          2  4
                           3  6
                           1  7
                           4  8

and if B is the array      1  5
                           7  8
                           3  4
                           6  3

then A+B is the array      3  9
                          10 14
                           4 11
                          10 11

and A*B is the array       2 20
                          21 48
                           3 28
                          24 24
```

STRUCTURE EXPRESSIONS

The operands of a structure expression are structures or a combination of structures and scalars. A structure expression yields a structure result. Array operands are not allowed in structure expressions. Note that the term 'expression', as used in this specification, does not include array expressions.

All operations performed on structures are performed on an element by element basis. Thus all structures appearing in a structure expression must have identical structuring. This implies that the structures must have the same number of contained scalars and arrays. The positioning of the scalars and arrays within the structure must be the same, and arrays similarly positioned must have identical dimensions and bounds. The data types need not be the same.

When an operation has one structure and one scalar operand, it is interpreted as many operations, one for each scalar element in the structure. Each suboperation involves a structure element and the scalar operand.

A structure expression may be thought of as a shorthand for the same form of the expression applied to each elementary item of the structure.

For example, consider the following structures:

```
1 A                    1 B
 2 PART1                3 PART1
    4 Q1                   5 Q1
    4 P1                   5 ALPHA
    4 W                    5 P1
 2 PART2                3 PART2
    3 Q2                   5 ALPHA
    3 P2                   5 Q2
    3 Z(3)                 5 Z(3)
```

Then the expression A-2*B is a shorthand for the following expressions:

```
A.Q1-2*B.Q1
A.P1-2*B.PART1.ALPHA
A.W-2*B.P1
A.Q2-2*B.PART2.ALPHA
A.P2-2*B.Q2
A.Z-2*B.Z
```

Note that the last expression is an array expression.

The value of a structure expression is a structure of identical structuring to the structure operands. The characteristics of the various elements are defined by the rules of scalar expression evaluation applied to each subexpression.

STRUCTURE EXPRESSIONS BY NAME

A structure expression by name may be formulated according to the rules for scalar expressions, but with all the operands either structures or scalars. The expression must have the appendage: , BY NAME. A structure expression by name may be used only as the right hand side of an assignment statement (see "BY NAME Structure Assignment" in Chapter 11). The evaluation of a structure expression by name involves the following steps:

Take each structure operand, and extract the names of all contained scalars and arrays.

Qualify these names by all the minor structure names which contain them, up to but not including the structure names specified in the structure expression by name.

The above processes will have generated n sets of qualified names, each set being associated with one of the n structure operands. Select a subset of these qualified names, such that it is a subset of all the n sets. Let this subset contain m names.

Construct m subexpressions, and associate one of the m names of the subset with each. The subexpression is identical in form to the original structure expression, except that where a structure name operand appeared, it is replaced by the qualified name associated with the subexpression, further qualified by the structure name which it replaces. Where the original expression involved a scalar, each subexpression also has the scalar.

The resulting subexpressions must be legal scalar or array expressions. At the termination of the above processes, the operands should all be unique names. Each subexpression is then evaluated to give a scalar or array value. The characteristics of the results are defined by the rules of scalar expression evaluation.

The result returned by a structure expression by name is a set of array or scalar values each having associated with it a qualified name, being the qualified name associated with the subexpression which gave as result the value.

If A and B are the sample structures described in "Structure Expressions", then the structure expression A-2*B, BY NAME is a shorthand for the following expressions:

```
A.PART1.Q1-2*B.PART1.Q1
A.PART1.P1-2*B.PART1.P1
A.PART2.Q2-2*B.PART1.Q2
A.PART2.Z-2*B.PART2.Z
```

Note that the last expression is an array expression.

Furthermore, the structure expression

```
B.PART1/B.PART2, BY NAME
```

is equivalent to the expression

```
B.PART1.ALPHA/B.PART2.ALPHA
```

CHAPTER 7: PROGRAM STRUCTURES

In NPL, the basic element of the language is a statement. The set of statements that is required to solve a particular problem constitutes a program. However, within the framework of an NPL program, a structure more complex than a single statement is used. Sets of statements are grouped to provide flexibility and control.

A set of statements called a group is delimited by DO...END and is used for control purposes. Other sets of statements called blocks are delimited by BEGIN...END or PROCEDURE...END. A begin block is activated in line and limits the scope of names; a procedure block or procedure is activated remotely and has the additional facilities of argument handling and return mechanisms.

STATEMENTS AND STATEMENT FORMAT

Statements are of two types simple statements and compound statements. The permitted statements are listed in Appendix 3.

SIMPLE STATEMENTS

General Form:

statement identifier statement body ;

The statement identifier is either a keyword or null.

The statement body is defined for each individual statement.

Where both the statement identifier and the statement body are null, the statement is a null statement. Where the statement identifier alone is null, the statement is an assignment statement.

COMPOUND STATEMENTS

A compound statement is a statement which may contain other statements. Also, compound statements can be nested. The statements contained in a compound statement are in the form of a group or begin block. Compound statements are not terminated by a semicolon. The final character of a compound statement is the semicolon of the last contained statement.

The two compound statements of the language are the IF compound statement (see Chapter 13) and the ON compound statement (see Chapter 14).

LABELS

Statements may be labeled to permit reference to them. Labeling is achieved by preceding a statement identifier by one or more labels. Each label is followed by a colon. Blanks may be placed as desired around the colon.

Example:

```
START: COMMENCE: BEGIN: A=B;
```

Multiple labels preceding a statement are synonyms and may be used interchangeably when referencing the statement. Labels appearing before PROCEDURE and ENTRY statements are entry names. All other labels are statement labels.

An entry name is used for identifying main or secondary entry points to a procedure (see "Blocks" in this chapter). An entry name preceding a PROCEDURE statement is called a procedure name.

Statement labels appearing before DECLARE, IMPLICIT, and SEQUENCE statements are ignored. Any reference to them is an error.

INITIAL VALUES FOR LABEL ARRAYS

If a label array is declared in a block (see "Blocks" in this chapter), then any statement (other than PROCEDURE or ENTRY) within that block may be preceded by a subscripted reference to the label array. The subscripts are optionally signed decimal integer constants.

The effect of preceding a statement with a subscripted reference is as follows: An INITIAL attribute (see Chapter 15) is constructed for the label array and added to the declaration. A label constant is constructed for the statement carrying the subscripted reference. This label constant is appropriately placed, with respect to the specified subscripts, in the INITIAL attribute. The subscripted reference preceding the statement is deleted.

It is not permitted to specify both the INITIAL attribute and the preceding form of initialization for a label array.

Example:

```
A: Z(3): X=4;
         .
         .
         .
   Z(2): Y=1;
         .
         .
         .
         GO TO Z(I);
```

GROUPS

General Forms:

```
label 1 : ... label m : DO statement
                        item 1
                        item 2
                        ...
                        item n
                        END; or END label;

label 1 : ... label m : statement
```

The statement is any NPL statement other than DO, PROCEDURE, BEGIN, DECLARE, IMPLICIT, FORMAT, SEQUENCE, or any compile-time statement.

The items may be a group, a block, or any statement.

The statement labels preceding the statement are optional. In the first form, one label may optionally be specified in the END statement.

The DO statement may specify iteration of the group (see discussion of DO in Chapter 13).

BLOCKS

General Form

```
label 1 : ... label m : block heading statement
                        item 1
                        item 2
                        ...
                        item n
                        END; or END label ;
```

The heading statement may be

a PROCEDURE statement; the block is then a procedure block or procedure.

a BEGIN statement; the block is then a begin block.

The items in the general form may be a block, a group, or any statement.

A begin block is activated by normal sequential program flow. A procedure is activated remotely by CALL statements (see Chapter 13), function references and I/O statements.

The PROCEDURE statement must be preceded by one or more labels. These labels are entry names which are used for referencing the primary entry point of the procedure. The labels preceding a BEGIN statement are optional.

Every procedure must logically end with either a RETURN, STOP, EXIT, or END statement, and physically with an END statement.

The inclusive text between the block heading statement and the keyword END, with the reservation explained below, is said to be contained in the block. A part of the text is called internal to a block B if it is contained in B, but not in any other block contained in B.

Labels preceding PROCEDURE and BEGIN statements internal to a block B, and ENTRY statements internal to a procedure block contained in B, are considered to be internal to the block B. The name of an external procedure and the names of all secondary entry points are said to be external.

Blocks may be nested, but partial overlap is not permitted.

Example:

```
A: PROCEDURE (X,Y);
   Y=Y-F(Y)-R(Y);
   IF Y>10 THEN BEGIN;
          Y=SIN (X);
          X=X+1;
          CALL C(X,Y);
          END;
D=E+D;
F: PROCEDURE (Q);
   P=Z*Q;
   SAVE (H);
   G: BEGIN;
      X=1;
      Y=2;
      Z=3;
      END G;
   RESTORE (H);
   END F;
END A;
```

THE PROCEDURE STATEMENT

General Form:

entry name 1 : ... entry name n : PROCEDURE
(formal parameter list) attribute 1 ... attribute m;

A PROCEDURE statement

heads a procedure.
defines the primary entry point to the procedure.
specifies the formal parameters for the primary entry point.
defines any special attributes of the procedure.
specifies the attributes of the value that will be returned if the procedure is invoked at the primary entry point.

The formal parameter list specifies the formal parameters of the entry point. The parameters are names and are separated by commas. When the procedure is invoked, a relationship is established between the arguments of the invocation and the parameter list. (Full details of this relationship are given in Chapter 10.) If the entry point requires no parameters, the parameter list and the enclosing parentheses are omitted.

The attributes are any of the following, separated by blanks where necessary.

OPTIONS (list)

The list is a list of implementation defined options separated by commas. The list may include options such as the following:

MAIN
REENTRANT
SECONDARY

The OPTIONS attribute may be specified only for an external procedure.

RECURSIVE

This attribute specifies that this procedure may be invoked

recursively. It does not apply to contained procedures which, if recursive, must also have the attribute.

FIXEDOVERFLOW
SUBSCRIPTRANGE
SIZE
MODULO

These options are related to the ON conditions with the same identifiers. They specify that the occurrence of the condition within the procedure should be signalled.

Data Attributes

Any of the data attributes , separated where necessary by blanks.

The data attributes specify the characteristics of the value returned by the procedure, when invoked as a function at the primary entry point. The value specified in the RETURN statement (see Chapter 13) will be converted to the characteristics specified.

If insufficient data attributes are specified at the entry point, the default rules of Chapter 17 will be applied, as determined by the name of the entry point.

If a procedure has multiple labels and no data attributes, there is potential ambiguity of the characteristics of the value to be returned. To avoid this ambiguity, succeeding labels are interpreted as if they were ENTRY names for successive ENTRY statements. For example,

```
A: I: PROCEDURE;
```

is equivalent to

```
A: PROCEDURE;
I: ENTRY;
```

If a RETURN (expression) is executed and no data attributes are specified, default rules will be applied to the label used in the invocation.

Example.

```
CLARET: PROCEDURE (AZURE,MAGENTA)  RECURSIVE FLOAT;
```

THE ENTRY STATEMENT

General Form:

entry name 1 : ... *entry name n* : ENTRY (*formal parameter list*)
attribute 1 ... *attribute m*;

The ENTRY statement specifies a secondary entry point to a procedure. The formal parameter list and the data attributes are formulated under the rules described for the PROCEDURE statement. There need not be any correlation between lists at primary and secondary entry points (see "Parameters, Dimensions and Length" in Chapter 10).

If a procedure has multiple labels and no data attributes, there is potential ambiguity of the characteristics of the value to be returned. To avoid this ambiguity, succeeding labels are interpreted as if they were ENTRY names for successive ENTRY statements. For example,

```
A: I: ENTRY;
```

is equivalent to

```
A: ENTRY;
I: ENTRY;
```

If a RETURN (expression) is executed and no data attributes are specified, default rules will be applied to the label used in the invocation.

For example, in the following illustration, CALC5 is the name of the secondary entry point of procedure CALC4.

```
CALC4:    PROCEDURE (X, Y, ERROR, Z) FLOAT(5) ;
          ALPHA = SQRT (X**2-Y**2) - ERROR;
          GO TO MEETING;
CALC5:    ENTRY (X, Y, Z) FLOAT(7) ;
          ALPHA = 2.5E-10 * SQRT(X-Y+1) ;
MEETING:  IF ALPHA > SQRT(ALPHA + 2) THEN Z = X;
          ELSE Z = Y;
          RETURN (ALPHA *X + Y - Z ** -5);
          END CALC4;
```

The ENTRY statement must be internal to the procedure block for which it defines a secondary entry point. It may not be internal to any block contained in this procedure. It may not be within a DO group which specifies iteration, nor in a DO group which is the unit of an ON statement.

THE BEGIN STATEMENT

General Form:

label 1 : ... label n : BEGIN ;

The labels are optional.

A BEGIN statement specifies the start of a begin block which is activated by normal sequential program flow.

PROGRAMS AND PROCEDURES

A program is a set of independent external procedures. Each external procedure is a complete nest of blocks. All blocks nested within an external procedure are internal.

DECLARATIONS

An identifier or qualified name may have more than one use in a program. Different uses are established by declarations, and references to different uses are distinguished by the rules of scope.

Declaration in a DECLARE Statement

The DECLARE statement is provided to enable the explicit specification of attributes of identifiers. Identifiers thus declared are said to be declared within the block to which the DECLARE statement is internal.

In the following example, X and V are declared in block B.

```
B: BEGIN;
   DECLARE (X,V) FLOAT(5);
   X = ALPHA - 4 + BETA ** 2;
   DELTA = X ** 2 - V ** 2 + SQRT (X - V + 1);
   END B;
```

Declaration in a Formal Parameter List

A name appearing in a formal parameter list is said to have been declared within the block to which the list is internal. The declaration of the same name with the same use in a formal parameter list and

in a DECLARE statement internal to the same block constitutes a single declaration. In the following example, A, I, and W are declared in block SINK.

```
SINK:    PROCEDURE (A, I, W );
         DECLARE W BIT(4) ;
              .
              .
              .
         END SINK;
```

Label Declarations

A statement label is said to have been declared in the block to which the associated statement is internal.

A label appearing before a PROCEDURE, BEGIN, or ENTRY statement is said to have been declared in the immediately containing block. However, if this latter block is the external procedure, the declaration is said to be internal to this external procedure, and is an external declaration.

In the following example, the first occurence of LOOP is said to be a declaration of the statement label in block P; the second occurence of LOOP is said to be a declaration of the statement label in the nested begin block. The name Q is a declaration in block P.

```
P:       PROCEDURE;
              .
              .
LOOP:     DO I = 1 TO N;
              .
              .
Q:        BEGIN;
LOOP:     DO J = 0 TO I;
              .
              .
          END LOOP;
          END Q;
              .
              .
          END LOOP;
          END P;
```

Implicit Declarations

An identifier which is referenced in a procedure, but which is not explicitly declared, is assumed to have an implicit declaration in the containing external procedure. In addition, such identifiers which can be recognized from the context as file names, entry names, or task identifiers are assumed to be external.

In the following example TEMP1 and TEMP2 are said to be implicitly declared in block Z.

```
Z:   PROCEDURE (PARAM1,PARAM2) COMPLEX;
     TEMP1=ABS(PARAM1-PARAM2) ;
     TEMP2=ABS(PARAM1+PARAM2) ;
     IF TEMP1=TEMP2 THEN RETURN (0) ;
      RETURN (COMPLEX(MAX(TEMP1,TEMP2)**2,MIN(TEMP1,TEMP2)**2)) ;
     END Z;
```

In the next example TEMP1 and TEMP2 are said to be implicitly declared in block ZZ.

```
ZZ:  PROCEDURE (ZA,ZB) ;
     TEMP1=ABS (ZA*2+ZB**2) ;
ZBZ: BEGIN;
     TEMP2= (TEMP1+ZB) **-2
     IF TEMP2>TEMP1 RETURN (TEMP2) ;
     END ZBZ;
     RETURN (TEMP1) ;
     END ZZ;
```

Scope of Declarations

The scope of a declaration is the block to which the declaration is internal, but excluding those contained blocks to which a redeclaration of the same name is internal. Distinct declarations of the same name may be linked by means of external declarations. This is achieved either explicitly by use of the attribute EXTERNAL, or implicitly as described above. (See "Scope Attributes" in Chapter 15 for details of 'headed' EXTERNAL.) The scope of an external declaration is the collected scopes of all declarations of the same name for which the name is declared as EXTERNAL, as described above.

Identifiers linked by external declarations must have the same attributes. It is an error to declare explicitly or implicitly an identifier as EXTERNAL with one set of attributes, and elsewhere in the same program to declare the same identifier as EXTERNAL with conflicting attributes. The external attribute may be used to obtain noncontinuous scopes (i.e. with holes) within an external procedure. A name or identifier use is said to be known or accessible within its scope. Identifiers which are not external are internal.

Examples:

```
A: PROCEDURE;
     DECLARE X FLOAT;
     ....
     B:    PROCEDURE (Y) ;
           DECLARE Y BIT (6) ;
           SAVE (X) ;
           ....
              C: BEGIN;
                 DECLARE A CHARACTER (10) ;
                 DECLARE X FIXED;
                 ....
              Y: RETURN;
                 END C;
           END B;
     D: PROCEDURE;
           DECLARE X FILE;
           Y = B (Y) ;
           ....
           END D;
     END A;
```

The following table illustrates the scope of each identifier appearing in the above example.

Identifier	Use	Scope (in terms of blocks)
A	external entry name	A,B,D
X	floating-point variable	A,B
B	internal entry name	A,B,C,D
Y	bit string	B
C	internal entry name	B,C
A	character string	C
Y	statement label	C
X	fixed-point variable	C
D	internal entry name	A,B,D,C
X	file name	D
Y	floating-point variable	D,A

Since file names and entry names are automatically EXTERNAL, the scope of the file name X and the entry name A may also include procedures in other external procedures of the program.

The following example illustrates interaction of scope and name qualification. All qualified names in the procedure are unique. Note that the reference to CARDOUT.RATE.NORMAL is legal, even though the qualifying identifiers CARDOUT and RATE have been redeclared in the internal procedure SICKPAY.

```
PAYROLL: PROCEDURE;
         DECLARE 1 CARDIN, 2 NAME, 2 RATE, 3 NORMAL, 3 OVER;
         DECLARE 1 CARDOUT, 2 NAME, 2 RATE, 3 NORMAL, 3 OVER;
                  .
                  .
         CARDOUT.RATE = CARDIN.RATE;
         CALL SICKPAY (CARDIN);
                  .
                  .
SICKPAY: PROCEDURE (CARD);
            DECLARE 1 CARD, 2 NAME, 2 RATE, 3 NORMAL, 3 OVER;
            DECLARE 1 CARDOUT, 2 NAME, 2 SICKRATE, 3 NORMAL, 3
               UNINSURED;
            WAGES = CARD.NORMAL*HOURS;
            SICKRATE.NORMAL = CARDOUT.RATE.NORMAL;
                     .
                     .
            END SICKPAY;
                  .
                  .
         END PAYROLL;
```

SEQUENCE OF CONTROL

Within a program, control normally passes sequentially from one statement to the next. However, sequential operation is modified by the following statements:

The GO TO statement	Chapter 13
The CALL statement	Chapter 13
The RETURN statement	Chapter 13
The END statement	Chapter 7
The PROCEDURE statement	Chapter 7
The SIGNAL statement	Chapter 14
The STOP statement	Chapter 13
The EXIT statement	Chapter 13

A GO TO statement transfers control to the specified statement label.

A CALL statement passes control to the specified entry point.

A RETURN statement returns control from a procedure to the invoking procedure.

An END statement logically terminating a procedure acts as a RETURN statement.

A PROCEDURE statement heads a procedure. Procedures may be considered as independent blocks, and placed anywhere within an external procedure consistent with desired identifier scopes. However, a procedure may be invoked only by a CALL statement or a function reference. Thus control passes around a nested procedure, from the statement before a PROCEDURE statement to the statement after the appropriate END statement.

A SIGNAL statement causes control to pass to the group specified within the associated ON statement.

The STOP and EXIT statements cause control to leave a program.

A function reference causes control to pass to the function procedure having the specified name.

The occurrence of a condition specified in an ON compound statement causes control to pass to the unit contained in the statement.

The flow of control through the IF and ON compound statements and through a DO group is not necessarily sequential (see Chapters 13 and 14).

In an appropriate environment, the asynchronous execution of several operations may involve transfer of control under the influence of external occurrences.

Example:

```
A:  PROCEDURE;
B:  X = Y + Z;
C:  CALL D;
E:  W = P* Q;
    D: PROCEDURE;
    G: S = T/P;
    H: RETURN;
    I: END D;
J:  U = V ** W;
K:  GO TO L;
    .....
    .....
L:  END;
```

Control passes in the following order

A,B,C,D,G,H,E,J,K,L.

PROCEDURE AND BLOCK TERMINATION

THE END STATEMENT

General Forms:

```
END;
END label ;
```

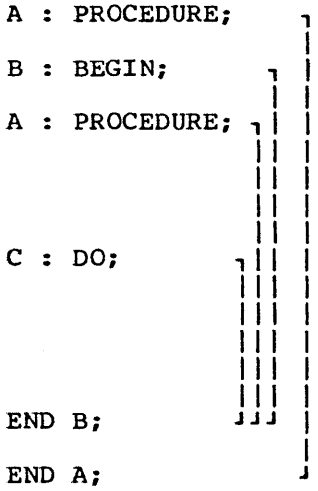
The END statement is used for terminating blocks and groups. The statement may optionally specify one of the labels declared before the heading statement of the group or block. If control passes to an END statement, it is treated as a RETURN statement (see Chapter 13).

MULTIPLE CLOSURE

When more than one block or group is terminated at the same point, it is unnecessary to provide multiple END statements. It is sufficient to supply one END statement that specifies the label of the outermost block to be terminated. The label specified must be known in the block which immediately contains the outermost block or group to be terminated.

An END statement causing multiple closure is equivalent to a sequence of END statements, equal in number to the number of groups or blocks being closed. If this statement is labeled, the label is on the last END statement in this equivalent sequence.

Example.



Storage allocation involves the association of storage with a particular variable. Storage must have been allocated for a variable before any reference is made to that variable. In choosing the class of storage to be associated with a given variable, the programmer has three alternatives:

1. He may specify that storage is to be allocated at the start of execution of the program and never released during execution. This is the static storage class.
2. He may specify that, during execution, storage is to be automatically allocated upon entry to a procedure and automatically freed upon return. This is the automatic storage class.
3. He may retain full control over the allocation and freeing of storage. This is the controlled storage class.

All variables must have a storage class. The storage class may be explicitly declared, using the storage class attributes STATIC, AUTOMATIC, and CONTROLLED; it may be given by default (see "Default Attribute" in Chapter 17) or it may be deduced from usage (for name parameters, see Chapter 10; for defined items see "DEFINED Attribute" in Chapter 15).

In addition, the qualifying scope attributes INTERNAL and EXTERNAL may be declared for STATIC and CONTROLLED data. If unspecified, the default scope is INTERNAL. AUTOMATIC data can only have the scope INTERNAL; AUTOMATIC EXTERNAL is not permitted. The qualifying scope attributes are described in Chapter 7.

THE STATIC STORAGE CLASS

Static storage is allocated at the start of execution and is not released until completion of program execution. STATIC variables may have internal or external scope.

Variables declared with adjustable size or dimensions (see "Allocation of Name Parameters" in Chapter 10) may not have the STATIC attribute. However, STATIC arguments can be passed by name as formal parameters with adjustable size or dimensions.

If a procedure involving static storage is invoked from within or as a separate task, then the static storage is common to all invocations. (See Chapter 18).

THE AUTOMATIC STORAGE CLASS

Automatic storage is allocated on each entry to the block to which its declaration is internal. This storage is released on leaving the block. If the block is a procedure which is invoked recursively, the previously allocated storage is pushed down on entry and popped up on return. AUTOMATIC variables have internal scope.

Label variables must be automatic; they may not be declared STATIC or CONTROLLED.

The following procedure illustrates the use of static and automatic storage.

```
P: PROCEDURE (X,Y);
   DECLARE I STATIC INITIAL(0),X(10),TEMP(10);
   I=I+1;
   TEMP=X**3/Y**2;
   DO J = 1 TO 10;
   IF (ABS(ABS(TEMP(J))-1) <= .3E-5) THEN RETURN(1); END;
   RETURN (SUM(TEMP**2));
   END P;
```

In this example the variable named I is of the static storage class (and, effectively, keeps count of how many times the procedure is invoked); TEMP is by default of the automatic storage class.

THE CONTROLLED STORAGE CLASS

The allocation and freeing of storage for variables declared as CONTROLLED are specified by the programmer by means of the statements ALLOCATE and FREE. CONTROLLED variables may have either internal or external scope.

If in the course of executing a statement any controlled data referenced by the statement is allocated or freed (by an abnormal function, for instance), then the effect is undefined.

THE ALLOCATE STATEMENT

General Form:

ALLOCATE allocation declaration 1,..., allocation declaration n;

An allocation declaration is of the form:

identifier attribute 1 ... attribute n

The identifier is an unqualified unsubscripted variable name. The variable must be of the controlled storage class. Attribute 1 may be a dimension attribute. Attribute i may be null or one of the attributes INITIAL, CHARACTER, or BIT. For a full discussion of attributes see Chapter 15.

A dimension or data attribute given in an allocation declaration must also be given in the corresponding declaration in a DECLARE statement. The number of dimensions in the dimension attribute must be the same in both declarations. If different upper or lower bounds or lengths are specified, those given in the allocation declaration override. The asterisk notation may be given in the DECLARE statement.

If any part of a structure is to be specified in this way, the entire major structure with all level numbers and identifiers must be included in the specification in the ALLOCATE statement. Only the attributes allowed in the ALLOCATE statement and which are desired to override those in the DECLARE statement may be specified. The form of the structure declaration is the same as that in the DECLARE statement (see Chapter 15).

A formal parameter passed by name may be specified in an ALLOCATE statement if the associated argument was of controlled storage class and not contained in a structure (see "Arguments Passed by Name" and "Allocation of Name Parameters" in Chapter 10).

The evaluations implied by the ALLOCATE statement are subject to the same rules as the evaluations involved in prologue activity (see "Prologues" in Chapter 16).

THE FREE STATEMENT

General Form:

FREE identifier 1 , ... , identifier n;

The identifier is an unqualified unsubscripted variable name. The variable must be of the controlled storage class.

The statement causes the storage most recently allocated for the variable to be freed. The next most recent allocation is popped up, and subsequent references to the identifier will reference that allocation.

In the case of asynchronous operation (see Chapter 18) the concept of most recent allocation is interpreted in the context of dynamically embracing tasks. Controlled storage allocated in a task after it has attached another task cannot be freed by the attached task.

If a specified identifier currently has no allocated storage, no action is taken.

The following example illustrates the use of controlled storage.

```
SUPER:PROCEDURE;
      DECLARE X(M) CONTROLLED, Y(M) AUTOMATIC;
      READ LIST (M,Y);
      ALLOCATE X;
      X=M*Y;
      CALL COMPUTE1;
      Y=X;
      WRITE LIST (X) ;
      FREE X;
      CALL COMPUTE2;
      END SUPER;
```

CHAPTER 9: CHARACTERISTICS OF PROCEDURES

A procedure is invoked by specifying the name of an entry point at which the execution of the procedure is to begin, together with a list of arguments which correspond to the list of formal parameters at that entry point. A procedure may be referenced by any procedure to which its name is known; however, an internal procedure may be referenced only if the immediately containing block is active.

Procedures may be either subroutine procedures or functions.

SUBROUTINE PROCEDURES

Subroutine procedures are programmer specified. They define an algorithm which may perform operations on the data known to the procedure. Subroutines may be invoked from CALL statements and from within I/O statements. Any arguments of the invocation are made available to the procedure. Values may be returned to the invoking procedure using arguments passed by name (see Chapter 10).

FUNCTIONS

Functions are invoked by function references which may include an argument list. These arguments are made available to the function, which returns a value.

Functions are of two types, function procedures and built-in functions.

FUNCTION PROCEDURES

A function is specified by writing a procedure. The name of the function is then an entry name of the procedure. Details of how a value is returned by a function procedure and rules relating to the type and precision of the value are specified in "The RETURN Statement" in Chapter 13.

BUILT-IN FUNCTIONS

Built-in functions are provided by the language and contain a subclass of generic functions. The built-in functions provided are listed in Appendix 1.

A generic function is a family of functions. A reference to a generic function causes the selection of a particular member of the family. The member chosen depends on the arguments provided.

The characteristics of the value returned by a generic function reference depend on the member of the family chosen.

Built-in functions other than generic functions have only a single member. Where necessary, the arguments provided are converted to the appropriate characteristics before the function is invoked. The characteristics of the value returned are invariant.

Built-in function names have the same scope rules as normal external identifiers. If undeclared built-in function names are referenced, they have implicit external declarations in the external procedure. If a built-in function name has been declared with another use in a block, then the built-in function is made accessible in contained blocks by declaring it with the attribute BUILTIN.

THE ENTRY ATTRIBUTE

General Forms:

```
ENTRY
ENTRY (parameter attribute list 1, ...
       parameter attribute list n)
```

The ENTRY attribute may be declared in a procedure for entry names referred to in that procedure. The first form is used to specify that the identifier being defined is an entry name. An entry name must be declared with the ENTRY attribute unless a reference is made in a CALL statement or in a function reference with arguments.

In the second form, each parameter attribute list is a succession of attributes describing the corresponding formal parameters of the entry point. Permissible attributes are those allowed for formal parameters (see Chapter 10). The attributes are separated by blanks. The number of parameter attribute lists must agree with the number of formal parameters required by the entry point. If a parameter attribute list is null, its place must be kept by a comma.

The second form of the ENTRY attribute need not be used unless the formal parameters of the entry are to be described. An ENTRY attribute of the first form specifies nothing about the number or nature of the parameters.

The dimension attribute may be specified for array parameters (see "The Dimension Attribute" in Chapter 15.) It must, however, be the first attribute specified. If expressions are included, they will be evaluated on entry to the declaring block. The * notation (see "Parameters Dimensions and Length" in Chapter 10) may be used.

If the argument is to be a structure, the structuring may be specified by a structure description using level numbers (see "Structures" in Chapter 16). This description does not involve identifiers, the level number being immediately followed by the attribute list. The first item in the structure description must be specified as being at level one. However, when argument checking is performed, importance will only be attached to relative levels; thus, it is not necessary for the argument to be a major structure so long as the structuring is the same. For instance, P is a proper argument to A in the following example.

```
DECLARE 1 O, 2 P, 3 Q, 4 R, 4 S, 3 T, 4 U,
        A ENTRY (1, 5, 6, 6, 4, 5);
CALL    A(P);
```

If the argument is to be an entry name, an ENTRY attribute may be specified for the argument. Consider, for example, passing a function without arguments as an argument to a procedure. If one wants to pass the entry name RANDOM as the second argument to a procedure named DETER, one declares

```
DETER ENTRY (BIT(6), ENTRY FLOAT)
```

and then invokes DETER:

```
CALL DETER (MASK, RANDOM)
```

The entry name RANDOM will be the second argument sent to DETER.

In the above example, if the declaration had been

```
DETER ENTRY (BIT(6), FLOAT)
```

then the function RANDOM would be invoked and its value sent as the (floating point) second argument to DETER.

If no attributes are given for a particular formal parameter, no assumptions are made about it. Otherwise, attributes required for full definition of a parameter but not specified in the ENTRY attribute, are deduced from default rules given in "Implicit and Default Attributes" in Chapter 17. The effect of the ENTRY attribute is described in Chapter 10. Expressions occuring in ENTRY attributes are evaluated on entering the block to which the ENTRY attribute is internal.

The use of NAME or VALUE attributes does not necessarily describe the way the corresponding formal parameter will be used in the invoked procedure, but describes the way the argument is presented. If neither a NAME nor a SETS attribute appears, the programmer should assume that a dummy argument may be created. If SETS appears, NAME is assumed. NAME, however, does not imply SETS. If NAME, VALUE and SETS do not appear, no uniform choice is made between NAME and VALUE: the choice will be made at each invocation according to the actual argument presented.

The following example illustrates the use of the ENTRY attribute:

```
Q: PROCEDURE;
   DECLARE A ENTRY FIXED(4),
           B FLOAT ENTRY (FLOAT, BIT),
           C ENTRY (FIXED(7), FLOAT(7), FIXED(7)),
           D FLOAT ENTRY (FIXED),
           M FIXED INITIAL (0) STATIC, (X,Y) EXTERNAL;
   M = M + 1 ;
   X = A*(1+D(M));
   CALL C(X**3, B, Y);
   IF Y > 100 THEN X = X+A;
   END Q;
```

In this example, A is a function procedure with no arguments and hence must be declared with the ENTRY attribute. The only reference to B is as an argument, and hence B must be similarly declared. C has been declared in order to effect conversion of the arguments. D has been declared to facilitate documentation.

ABNORMALITY OF PROCEDURES

Abnormality is a property of both external and internal procedures. Blocks invoking procedures which are abnormal must declare those names with the ABNORMAL, USES, or SETS attributes. This enables program optimization to be performed.

An external procedure is abnormal if it, or any procedures invoked by it

access, modify, allocate, or free external data;
modify, allocate, or free their arguments;
return inconsistent function values for identical argument values;
maintain any kind of history;
perform I/O operations; or
return control from the procedure by means of a GO TO statement.

An internal procedure is abnormal under the conditions listed for external procedures; it is also abnormal if it, or any procedures called by it, access, modify, allocate, or free variables declared in an outer block.

In the absence of ABNORMAL, NORMAL, USES, or SETS attributes, entry names invoked as functions are assumed to be normal and all other entry names are assumed to be abnormal.

If an expression contains a reference to an abnormal variable (see "The ABNORMAL, NORMAL and SECONDARY Attributes" in Chapter 15), or to an abnormal function that may alter another operand in the expression, then the order in which data is accessed within the expression becomes significant. (See "Evaluation of Expressions" in Chapter 6 for the hierarchy in which operations are performed.) This order is defined as follows:

> Consider an infix operator op with operands a, b of the form a op b in a scalar expression. Then either a, b or both may be a subscripted name, a function reference, or a subexpression of the form c op d. In the following discussion, the term 'elements' will denote the expressions that must be evaluated, such as subexpressions, arguments, and subscripts, and the functions that must be invoked before op can be applied.
>
> If a is an unsubscripted name or a constant, and b is neither an unsubscripted name nor a constant, then a will not be accessed until all the elements of b have been accessed.
>
> In all other cases, all elements of a are accessed before any elements of b are accessed.
>
> Subscript lists are evaluated and accessed left to right, and immediately before accessing the subscripted variable.
>
> Function argument lists are evaluated and accessed left to right, immediately before accessing (or invoking) the function.
>
> The order of assignment in multiple assignment is left to right.
>
> Array expressions are evaluated by performing a complete scalar evaluation of the expression in turn, for each position of the array in row major order. The result of the evaluation for an earlier position will not be altered by an evaluation of a later position. (See "Array Expressions" in Chapter 6).
>
> Structure expressions are evaluated by performing a complete scalar evaluation of the expression for each eligible field in the order in which the fields of the target structure were declared. The result of the evaluation for an earlier position will not be altered by an evaluation of a later position.

THE ABNORMAL ATTRIBUTE

Abnormal procedures, invoked as functions, must be declared in the invoking block with one or more of the attributes, ABNORMAL, USES, and SETS.

ABNORMAL used alone specifies complete abnormality. ABNORMAL used in combination with USES or SETS specifies that the function maintains a history, performs I/O, returns inconsistent function values, or contains an abnormal return. It is unnecessary to specify ABNORMAL for the built-in functions TIME and DATE.

The ABNORMAL attribute may also be specified for data (see Chapter 15). In particular, data which is changed by executing an ON unit is abnormal.

THE NORMAL ATTRIBUTE

This attribute specifies that the entry name is for a procedure which is not abnormal. The attribute may be used to override a factored or implicit ABNORMAL attribute.

THE USES AND SETS ATTRIBUTES

General Forms:

USES (item 1 , ... , item n)
SETS (item 1 , ... , item n)

The USES and SETS attributes may be declared in the invoking block for any entry name. If either is declared, complete information must be given about the abnormality of the specified entry name arising from data manipulation.

The items may be

a decimal integer n, specifying the n'th argument of the invocation.

an external identifier known to the invoking block.

an asterisk that implies all external identifiers known to the invoking block.

The appearance of an item in the SETS list specifies

that the procedure, or other procedures invoked by it, reassign that item.

that neither the procedure, nor procedures invoked by it, access that item other than to reassign it unless it is also specified in a USES attribute.

The appearance of an item in a USES list specifies

that the procedure, or other procedures invoked by it, access that item.

that neither the procedure nor procedures invoked by it reassign that item unless it is also specified in a SETS attribute.

When a procedure is invoked, a relationship is established between the arguments of the invocation and the formal parameters of the entry. Permissible arguments are listed in "Function References" in Chapter 6. An explicit ENTRY attribute may be specified for an invoked entry name (see "The ENTRY Attribute" in Chapter 9). This specifies the attributes of parameters of the entry. When a procedure is invoked, the data attributes of arguments passed must match those of the associated formal parameters. If the specified argument has different data attributes from the parameter, a dummy argument, with the value of the given argument, will be constructed, and converted to the characteristics of the parameter. The dummy argument is then passed to the entry. If the conversion is impossible, the program is in error (e.g. filename to bit). If no ENTRY attribute is specified, it is assumed that the attributes of the arguments given match those of the associated formal parameters.

If an argument is a label variable, a dummy argument having the current value of the label variable or array will be constructed and passed to the entry. If expressions involving operators or constants are specified as arguments, a dummy argument having the current value of the expression (or constant) is constructed and passed to the entry.

If the argument is a statement label constant, this value is qualified by an identification of the current invocation of the block containing the label and by the current task, before the invocation is performed and the value assigned to the parameter.

If a subscripted item is an argument, then the subscript is evaluated before the invocation. The specified element is then passed as the argument. Subsequent changes in the subscript during the execution of the invoked procedure do not influence the associated parameter. If a parameter is a scalar, the associated argument must also be a scalar.

If an argument is an array expression, the associated formal parameter must be declared as an array with identical dimensions and bounds to the argument (see "Parameters, Dimensions, and Length" in Chapter 10).

If an argument is a structure expression, the associated formal parameter must be declared as a structure with identical structuring.

Note that a scalar is a valid array or structure expression. Thus, a scalar argument may be passed to an array or structure parameter. However, it must be known before the invocation that the scalar constitutes an array or structure expression. Thus an explicit ENTRY attribute specifying this information must be declared for the entry name in the invoking procedure. An appropriate dummy array or structure will then be constructed and passed to the entry.

Formal parameters must be declared in the invoked procedure. They may not be declared in outer containing blocks. If no explicit declaration is given, an implicit declaration is assumed, internal to the invoked procedure, with default attributes. (See Chapter 17).

ARGUMENTS PASSED BY NAME

If a formal parameter has the attribute NAME, the associated argument is said to be passed by name. This means that during execution of the invoked procedure the formal parameter name is made synonymous with the name of the argument passed, and all references to the former are treated as references to the latter. Note, however, that where a dummy argument has been constructed as described previously, the parameter is synonymous with the dummy rather than with the actual argument.

If the argument is CONTROLLED, the name parameter is made synonymous with the most recent generation of the argument at the point of invocation, and this synonym normally remains fixed until the invoked procedure returns control to the caller. But if the name parameter also has the attribute CONTROLLED, then the parameter is always synonymous with the most recent generation at the point of reference, which may be different from the generation existing at the point of invocation. In this case, the invoked procedure may also ALLOCATE and FREE the name parameter (see "Allocation of Name Parameters" in Chapter 10).

NAME formal parameters may not be declared with the storage class attributes STATIC or AUTOMATIC, with the scope attributes, or with the DEFINED attribute.

This relationship between argument and name parameter is not established unless the invoked procedure is entered at an entry point where the parameter declared NAME appears in the formal parameter list. If the procedure is entered at an entry point where the parameter is not in the parameter list, it is inaccessible in the invoked procedure.

A formal parameter by name may be an unsubscripted unqualified variable name (including a label variable name), a file parameter, or an entry parameter.

A file parameter may be used within a procedure wherever a file name may be used; an entry parameter may be used wherever an entry name may be used.

If a formal parameter by name is a fixed-length string variable, the actual argument must be a fixed-length string. If the parameter is a variable-length string, the argument must also be one.

ARGUMENTS PASSED BY VALUE

If a formal parameter has the attribute VALUE, the associated argument is said to be passed by value. This means that when the procedure is invoked, the value of the argument is assigned to the associated parameter.

Value parameters may be scalar, array, or structure names. The parameters have explicit or implicit storage class and scope attributes which need not be the same as those of the corresponding argument.

Value parameters may only be CONTROLLED if storage has been allocated for them before the procedure is invoked. This implies that either of the following cases exist:

The parameter is CONTROLLED EXTERNAL and is declared so in the invoked procedure. Storage may then be allocated before the invocation by other procedures.

The parameter is CONTROLLED INTERNAL, and the procedure has an alternative entry point where the value parameter is not in the parameter list. The first invocation of the procedure can be

through this entry point and the CONTROLLED INTERNAL storage allocated. Subsequent invocations may use the other entry point where the value parameter is in the parameter list.

If a formal parameter by value is a variable-length string, the actual argument may be either a fixed-length or a variable-length string.

DEFAULT PARAMETER ATTRIBUTES

Formal parameters to be passed by value may be explicitly declared using the attribute VALUE. Formal parameters not declared NAME or VALUE are given the default attribute NAME.

ADJUSTABLE DIMENSIONS AND LENGTH

AUTOMATIC and CONTROLLED arrays and strings may be declared with adjustable dimensions and lengths, i.e. with expressions involving variables and function references as bounds or lengths. When storage is allocated for the array or string, these expressions are evaluated and converted to integer; thus, at this point, the variables in the expressions must have had storage allocated for them and must have been assigned a value. Full details of how such expressions may be formulated are given in the section headed "Prologues" in Chapter 16.

PARAMETERS, DIMENSIONS AND LENGTH

In general, the dimensions, bounds, and sizes of arguments must be the same as those of the corresponding formal parameter. (For the exception, see "Allocation of Name Parameters" in this chapter. This correspondence may be achieved by

declaring the values for the parameters as constants. This method of specification must be used for STATIC VALUE parameters.

specifying the length by an asterisk, or each and every dimension bound by an asterisk to indicate that the length or bounds are the same as those of the argument passed. Asterisk notation may not be used for STATIC VALUE parameters; if the asterisk notation is used for a CONTROLLED VALUE parameter, the ALLOCATE statement must specify the length or bounds.

declaring the bounds or length as any expression which, when evaluated, will give the appropriate value, i.e., adjustable bounds or lengths (see above).

NAME PARAMETERS, ADJUSTABLE LENGTHS AND DIMENSION BOUNDS

The expressions specified for dimension bounds or length must be formulated according to the rules stated under "Prologues" in Chapter 16.

VALUE PARAMETERS, ADJUSTABLE LENGTHS AND DIMENSION BOUNDS

The only difference between value parameters and variables that are not parameters is that the associated argument value is copied at entry points if the parameter is in the parameter list. Thus, storage is allocated for value parameters in exactly the same way as for other variables of the same storage class. The rules for adjustable dimensions and length of value parameters are exactly as described under "Default Parameter Attributes" in this chapter.

Storage is allocated for AUTOMATIC VALUE parameters on every entry to a procedure, whether the parameter is in the parameter list of the entry or not. Thus, if adjustable dimensions or length are specified for an AUTOMATIC VALUE array or string, the constituent expressions must be able to be evaluated at all entry points, i.e., not contain a name parameter which only appears in a subset of the parameter lists.

Value parameters may have the INITIAL attribute (see Chapter 15). Where both initial value assignment and argument copying should take place at the same point, the latter overrides the former.

ALLOCATION OF NAME PARAMETERS

Variables passed by name may be allocated and freed in an invoked procedure if the original argument was declared CONTROLLED, and the name parameter also has the attribute CONTROLLED.

If the variable is a string or an array, the length or dimension bounds must be declared in the invoked procedure. Either the asterisk notation may be used, or explicit bounds or length given.

If the asterisk notation is used, it means

> If storage has already been allocated for the argument, then in the invoked procedure the formal parameter will be assumed to have the length or bounds that were specified when the storage was allocated. Further allocations of the data will use these same values.
>
> If no storage has been allocated for the argument, then the program is in error. Bounds or lengths must be declared in the invoked procedure if the argument was passed unallocated.

If dimensions or length are explicitly specified in the invoked procedure, the following rules apply:

> If storage has already been allocated for the argument, then on entry to the invoked procedure the expressions specifying the parameter bounds or length are evaluated and must give values the same as those of the argument. If the parameter is subsequently re-allocated, these expressions will again be evaluated to give new bounds or length for the new allocation.
>
> If no storage has been allocated for the argument, then no requirements are made at the point of entry to the invoked procedure on the value of the expressions specified for bounds or length of the parameter. These expressions will only be evaluated at a subsequent point of allocation.

The initial value attribute may be specified in the invoked procedure for a name parameter which is allocated in that procedure.

The assignment statement is used for evaluation of expressions and assignment of values to scalars, arrays, and structures.

SCALAR ASSIGNMENT

General Form:

variable 1, ... , *variable n* = *scalar expression*, *option list*;

The items on the left of the equal sign may be a scalar variable name, or a pseudo variable (see below). These items may be of type arithmetic, bit, or character. The statement causes the following action:

Expressions on the left, in subscripts or pseudo variables, are evaluated from left to right.

The scalar expression is evaluated.

The value of the expression on the right is assigned to the scalars on the left. The value is converted, if necessary, to the characteristics of the variable on the left according to the rules stated under "Scalar Expressions" in Chapter 6.

One or more options may appear to the right of the scalar expression. The list items are separated by commas. The permitted options are:

```
FIXEDOVERFLOW
SUBSCRIPTRANGE
SIZE
MODULO
```

The first three options are related to ON conditions which have the same identifiers. They specify that the occurrence of the condition should be signalled when it occurs during execution of the statement, but excluding execution of any functions invoked by the statement. The treatment of the occurrence of the condition in such functions is determined by the function procedures themselves. In the absence of the option, the occurrence of the condition will not be signalled, unless the absence of the option is overruled by the presence of a procedure attribute with the same identifier.

The MODULO option specifies that replacement will be performed ignoring any SIZE error conditions.

PSEUDO VARIABLES

The following are permitted:

COMPLEX (a,b)	a and b are real arithmetic variables which need not have the same characteristics. On assignment, the real part of the expression on the right is assigned to a, the imaginary part to b.
REAL (c)	c is a complex variable. On assignment, the real value of the expression is assigned to the real part of c.

IMAG (c) — c is a complex variable. On assignment, the real value of the expression is assigned to the imaginary part of c.

SUBSTR (s,i,k) — s is a string. On assignment, the expression is assigned to the substring of s from the ith character or bit, k characters or bits long. If k is omitted, the expression will be assigned from the ith character or bit to the end of the string (see Appendix 1).

UNSPEC (v) — v is a scalar variable. The expression on the right is converted to a bit string and assigned to v without conversion.

ONCHAR — The expression on the right is converted to a character string of length 1. On assignment, the character which caused an I/O conversion error interrupt is replaced by the value assigned. This pseudo variable is only defined while such an interrupt is being processed.

ONFIELD — The expression on the right is converted to a character string. On assignment, the field that was being processed when an I/O interrupt occurred is

All pseudo variables are also built-in functions (see Appendix 1).

STRING ASSIGNMENT

When strings are assigned, the assignment is performed from left to right starting with the leftmost positions.

Assignment to Fixed-Length Strings

If the expression value is longer than the string on the left, the value is truncated. If it is shorter, it is extended on the right with zeros or blanks (bit or character).

Assignment to Variable-Length Strings

If the expression value is longer than the maximum length of the string on the left, the value is truncated. The new length of the string is the maximum length.

If the expression value is shorter than the maximum length of the string on the left, the value is assigned; and the new length of the string is the length of the value.

If the destination is the SUBSTR pseudo variable with a variable length string argument, the assignment is performed to this substring.

If the expression value is shorter than the substring, the rest of the substring is filled with blanks or zeros precisely as if the specified substring were in an assignment statement. If the expression value is longer than the substring, it will be truncated as if it were in an assignment statement. If no substring length is specified, truncation is only performed when the left hand string part before the substring and the assigned value exceed the maximum length for the argument.

To illustrate string assignment, suppose that:

```
A is a fixed-length string whose value is 'XZ/BQ'.
B is a variable-length string of maximum length 8 whose value is
    'MAFY'.
C is a fixed-length string of length 3.
D is a variable length string of maximum length 5.
Then:
If C=A; the value of C will be 'XZ/'.
If C='A'; the value of C will be 'A'.
If D=B; the value of D will be 'MAFY'.
If D=SUBSTR(A,2,3) || SUBSTR(A,2,3); the value of D will be
    'Z/BZ/'.
If SUBSTR(A,2,4)=B; the value of A will be 'XMAFY'.
If SUBSTR(A,4)=B; the value of A will be 'XZ/MA'.
If SUBSTR(B,2)=SUBSTR(A,2); the value of B will be 'MZ/BQ'.
If SUBSTR(B,2,2)=SUBSTR(A.3); the value of B will be 'M/BY'.
```

ARRAY ASSIGNMENT

General Form:

array 1 , ... , *array n* = *array expression* ;

The items on the left of the equal sign may be an array variable name or a pseudo array.

All the arrays on the left, and the array expression must have the same number of dimensions and identical dimension bounds. The action caused by the statement is identical to that described for scalar assignment, except array values are used and assigned on an element by element basis.

The permitted pseudo arrays have a syntax the same as the permitted pseudo variables, except that the first argument must be an array of appropriate dimensions and bounds. The meaning, taken on an element by element basis, is the same.

To illustrate array assignment, suppose that:

```
A is the array              2  4
                            3  6
                            1  7
                            4  8

and B is the array          1  5
                            7  8
                            3  4
                            6  3

Then, if A=(A+B)**2-A(1,1);

A will have the value       7  79
                           98  194
                           14  119
                           98  119
```

SIMPLE STRUCTURE ASSIGNMENT

General Form:

structure name = structure expression, options ;

The options that can be used on scalar assignment may also be used on structure assignments. There is an option, BY NAME, that is permitted only on a structure assignment. In the absence of the BY NAME option, the structure on the left must have identical structuring to the expression on the right. The action caused is identical to that described for scalar assignment, except that structure values are used and result in element by element assignment of corresponding elements.

If the BY NAME option is used, the structure assignment statement causes the following action:

Subscript expressions on the left are evaluated.

The structure expression BY NAME is evaluated. (See "Structure Expressions BY NAME" in Chapter 6). This gives a set of array and/or scalar values, each with an associated qualified name.

All names of elementary scalars and arrays of the structure on the left are qualified with all appropriately containing minor structure names up to but not including the name specified in the by name assignment statement. This results in a set of qualified names.

Pairs of identical qualified names from the two lists are selected.

Values from the right are assigned to items on the left for the pairs of identical qualified names. These assignments must be legal, e.g., arrays may not be assigned to arrays of different dimensions or bounds.

In by name structure assignment, it is unnecessary for the structuring of all participating structures to be identical. Names defined on structures appearing in by name assignment take no part in the name pairing.

STATEMENT LABEL ASSIGNMENT

General Form:

label variable 1 , ... , label variable n =
statement label designator, options;

The above form of the assignment statement causes the assignment of the value of the statement label designator on the right to the label variables on the left. The options that may be used are the same as those for scalar assignment.

When a statement label is assigned to a label variable, the value is qualified by an identification of the current invocation of the block containing the label and by the current task (see Chapter 18).

The qualification information is used when a GO TO specifies the label variable in order to make the identified invocation current and to check that control does not cross task boundaries.

The SAVE and RESTORE statements provide means for holding data by name in auxiliary storage.

THE SAVE STATEMENT

General Forms:

```
SAVE ( item 1 , ... , item n ) ;
SAVE (item 1, ... ,item n) , (expression) ;
```

The items may be variable names, subscripted names, qualified names, or subscripted qualified names.

The first form of the statement is exactly equivalent to the series of simple SAVE statements

```
SAVE (item 1) ;
  .
  .
SAVE (item n) ;
```

The second form is equivalent to

```
temp = expression;
SAVE (item 1) , (temp) ;
  .
  .
SAVE (item n) , (temp) ;
```

A simple SAVE statement causes the data encompassed by the specified name to be placed in auxiliary storage. This data is identified by the data name qualified by the allocation of data and details of the current task (see Chapter 18.)

If no expression is specified, and items of the same name and allocation are repeatedly stored, the values are stacked. If an expression is specified, only one value for a given name (qualified) and given expression value will be saved at any one time, and subsequent execution of a SAVE statement with matching identification will cause the previously saved value to be overridden.

THE RESTORE STATEMENT

General Forms:

```
RESTORE ( item 1 , ... , item n ) ;
RESTORE (item 1, ... ,item n) , (expression) ;
```

The permitted items, and the breakdown of the statements into simple RESTORE statements is the same as that described for the SAVE statement. The name specified in a simple RESTORE statement is qualified as described for the SAVE statement. This identification specifies a value previously saved; this value is assigned to the associated scalar, array, or structure.

Once a value has been restored, it may not be restored again. Thus if the same item has been repeatedly saved with no qualifying expression, the action of restoring the data causes the top item of the stacked information to be deleted. Therefore the stacked information is treated in a first in last out manner.

A value may be saved in one external procedure and restored in another if the data name is EXTERNAL and if the SAVE and RESTORE statements refer to the same allocation of the data name.

Data saved cannot be restored in part. Thus, if an array is saved, an element cannot be restored; if a structure is saved, an array element cannot be restored, etc.

THE GO TO STATEMENT

General Form:

GO TO statement label designator ;

The GO TO statement transfers control to the statement specified by the statement label designator. The designator may be a statement label or a scalar label variable. For example, the designator may be a subscripted label variable, giving the effect of a multiway switch. A GO TO may not pass control from outside a DO group to a statement inside, if the DO group specifies iteration.

A GO TO from one block to another has the effect of terminating all blocks dynamically descendant from the block implied by the destination. Conditions are reinstated and automatic variables freed in the same way as if the blocks terminated normally. When this form of termination is used to terminate a procedure that was invoked as a function, the evaluation of the expression that contained the corresponding function reference will be discontinued and control transferred to the designated label.

Control may not be passed to an inactive block.

A GO TO may not terminate a procedure invoked by a CALL from a statement allowing the CALL option.

The following example serves to illustrate some uses of statement label constants and variables in GO TO statements:

```
TSET:   PROCEDURE (TO, TF, TBAR, ERROR) ;
        DECLARE ERROR LABEL VALUE, (R,X,W,V) EXTERNAL,
        SWITCH (5) LABEL INITIAL (TERROR, HIGHT, FINET, LOWT,
                                                  TERROR) ;
        ON SUBSCRIPTRANGE GO TO TERROR;
        IF (TBAR>TO) & (TBAR<TF) THEN TEMP=TBAR/2;
        GO TO SWITCH (ROUND (2*SIN (TEMP) *COS (TEMP) +TO) ) ;
TERROR: X,W,V=0; GO TO ERROR;
HIGHT:  IF V>W THEN GO TO TERROR; RETURN (TF) ;
FINET:  TEMP=TBAR-TO;
        RETURN (TEMP/ (R**3+X**2+W) +2*COS (TEMP) *SIN (TEMP) ;
LOWT:   IF V<W THEN GO TO TERROR; RETURN (TO) ;
        END TSET;
```

THE IF COMPOUND STATEMENT

General Forms:

IF expression THEN unit

IF expression THEN unit 1 ELSE unit 2

The unit appearing in the general forms may be a group or a begin block.

In the first form, the scalar expression is evaluated and, if necessary, converted to a bit string. If any bit in the resulting string has a value '1', the unit is executed; and control passes to the

next statement following the IF compound statement. If all bits have the value '0', the unit is not executed; and control passes to the next statement.

In the second form, the expression is similarly evaluated. If any bit is '1', unit 1 is executed; and control passes to the next statement following the IF compound statement. If all bits have the value '0', unit 2 is executed and control passes to the next statement.

The units may contain statements which specify transfer of control, and so override these normal sequencing rules.

The IF compound statement is not itself terminated with a semicolon. The last character is the semicolon of the last contained statement.

'IF' compound statements may be nested and an ELSE clause is always associated with the innermost preceding IF. Null ELSE clauses may be required to specify the desired effect.

Examples:

```
IF QUEUE = EMPTY THEN CALL COMPILE;
ELSE GO TO MULTIPROCESS;

A:   IF X > Y THEN
                IF Z = W THEN
                C:DO;
                IF W > P THEN Y = 1;
                ELSE Y = 2;
                IF P = Q THEN X = 3;
                END C;
                ELSE;
          ELSE X = 4;
     J :   Z = 5 ;
```

THE DO STATEMENT AND ITERATION OF A DO GROUP

General Forms:

```
DO;
DO WHILE expression;
DO variable = specification list ;
          or DO pseudo variable = specification list ;
```

The DO statement delimits the start of a DO group (see "Groups" in Chapter 7), and, in the first general form, performs this function alone. The DO statement, however, may also specify iteration of the group which it heads.

The iteration specified by the second general form is defined by the following expansion:

```
LABEL:   DO WHILE expression;
         statement 1;
         .
         .
         statement n;
         END;
NEXT :   statement;
```

is exactly equivalent to

```
LABEL:   IF ¬ (expression) THEN GO TO NEXT;
         statement 1;
         .
         .
         statement n;
         GO TO LABEL;
NEXT :   statement;
```

The third general form specifies controlled iteration. The variable is a subscripted or unsubscripted scalar variable. The specification list is a list of specifications separated by commas.

In general each specification involves three expressions,giving the starting value of the scalar, the increment to be added to the value of the scalar after each iteration of the loop, and the terminating value of the scalar. Iteration is terminated as soon as the value of the scalar passes its terminating value. The iteration for the next specification is then begun. When the last specification is complete, control passes to the statement following the DO group.

Each specification may be one of the following forms:

expression 1 TO expression 2 BY expression 3
or expression 1 BY expression 3 TO expression 2

expression 1 TO expression 2

expression 1 BY expression 3

expression 1

or any of the specifications followed by

WHILE expression 4

The second form is the same as the first, with the 'BY' expression understood to be the integer 1.

The third form is the same as the first, with expression 2 infinite.

The fourth form is the same as the first, with all three expressions equal to the specified expression, and implies a single execution of the group with the control variable having the value of the expression.

The fifth form specifies that before each associated execution of the group expression 4 will be evaluated, and, if necessary, converted to give a bit string value. If any bit in the resulting string has a value '1', the iterations continue uninterrupted. If all bits have value '0', the iterations associated with the current specification are terminated.

All the expressions specified must lead to legal statements in the language, when they are substituted in the following expansion:

```
LABEL:    DO variable = expression 1 TO
          expression 2 BY expression 3
          WHILE expression 4;
          statement 1;
          .
          .
          statement n;
          END;
NEXT:     statement;
```

is exactly equivalent to

```
LABEL:    variable = expression 1;
LABEL1:   IF (expression2 - variable)
          * SIGN (expression 3) < 0
          THEN GO TO NEXT;
          IF ¬ (expression 4) THEN GO TO NEXT;
          statement 1;
          .
          .
          statement n;
          variable = variable + expression 3;
          GO TO LABEL1;
NEXT:     statement;
```

LABEL1 and NEXT are introduced statement labels. If more than one specification had been given, NEXT would refer to the initialization for the next specification. If the WHILE clause is omitted, the IF compound statement involving expression 4 is replaced by a null statement.

Some examples of DO statements are:

```
DO INDEX=CTR WHILE A<B, 5 TO 10 WHILE A=B, 100;
DO I=J TO K BY I, I+1 TO N BY 1;
```

THE CALL STATEMENT

General Forms:

```
CALL entry name ( argument list ) , task option ;
CALL ( expression ) ( argument list ) , task option ;
```

The first form causes the invocation of the specified entry name by activating the containing procedure and passing control to the entry point.

On execution of the second form of the CALL statement, the scalar expression is evaluated and, if necessary, converted to a character string. This string specifies a program name which must have been specified previously in a FETCH statement, and must not have been specified subsequently in a DELETE statement. The specified program is invoked and the listed arguments are passed. No conversion is performed for the arguments; those specified must have characteristics that match the associated formal parameters. The arguments passed cannot be entry names or built-in function names.

The argument list is a list of arguments separated by commas. Permissible arguments are specified under "Function references" in Chapter 6. A relationship is established on entry between the items in the argument list and the formal parameters specified at the entry point to the invoked procedure (see Chapter 10). If the entry point does not specify any formal parameters, the argument list and its enclosing brackets must be omitted from the CALL statement.

The TASK option may be specified if the environment allows and it is desired that the invoked procedure be executed asynchronously with the invoking procedure (see "Data Allocation Across Tasks" in Chapter 18). If the option is omitted, the preceding comma is omitted.

Examples:

```
CALL CRITICAL_PATH (A,B*C,D);

AEIPROFIT: CALL SCRIP_ISSUE (BULLS, BEARS, BUTTERFLIES);

CALL TRANSMIT('LAB',TIME),TASK (QTAB);

CALL RANDOM_GENERATE;

CALL ('PROCTL');

CALL (A||B) (C,D,E);
```

THE RETURN STATEMENT

General Forms:

```
RETURN;
RETURN ( expression ) ;
```

The RETURN statement causes termination of execution of the containing procedure, and returns control to the invoking procedure.

The first form terminates all but function procedures and returns control to the first executable statement logically following the statement which invoked the procedure. This is the only form which may be used to terminate a procedure invoked with the TASK option.

The second form must be used to terminate a procedure invoked as a function procedure. The value returned by the function is the value of the expression specified in the RETURN. If the entry point at which the procedure was invoked specified data attributes, the value of the expression is converted to these characteristics before it is returned.

THE DISPLAY STATEMENT

General Form:

```
DISPLAY (scalar expression);
DISPLAY (scalar expression), task option;
DISPLAY (scalar expression) (character variable);
DISPLAY (scalar expression) (character variable),task option;
```

The statement causes the evaluation of the expression and, where necessary, its conversion to a character string. This string is displayed to the operator as a message. The task option specifies asynchronous operation (see Chapter 18).

The third and fourth forms specify a character string which will receive a message from the operator. The third form will cause the program to wait until the operator's message has been received; other forms do not cause the program to wait.

THE WAIT STATEMENT

General Forms:

WAIT (wait specification 1, ..., wait specification n);
WAIT (wait specification 1, ..., wait specification n) (scalar expression);

A wait specification has one of these two forms :

task identifier

task identifier (scalar expression)

In either form of the wait specification, the task identifier must reference a task which has been attached by the task executing the WAIT statement. In the second form, the scalar expression is evaluated on execution of the WAIT statement and converted, where necessary, to give an integer p.

Each attachment of a task with a given identifier causes an entry to be made in an attachment list for that identifier. The first entry in the attachment list indicates the oldest attachment of the task, and the last entry in the list indicates its newest attachment. A wait specification causes interrogation of the status of the attachment list relevant to the specified task identifier.

With a wait specification of the first form, the attachment list for the specified identifier is scanned from the oldest to the newest entry, as many times as is necessary, until one attachment of the identifier task is found to have completed. This entry is then marked so that it will not be considered by further wait specifications (of the first form) for that identifier, and the wait specification is said to be satisfied. If the tasks indicated by the attachment list have all completed and have all been marked by earlier wait specifications, or if the attachment list is null, the wait specification is also said to be satisfied.

With a wait specification of the second form, only the p'th entry (the oldest entry is numbered 1) of the attachment list for the specified identifier is considered. When the task which it indicates is found to have completed, regardless of whether earlier wait specifications of either form may have interrogated the same entry, the wait specification is said to be satisfied. If the attachment list contains fewer than p entries, or if p is zero or negative, the wait specification is also said to be satisfied.

When a WAIT statement of the first form is encountered, the program flow is suspended until each of its wait specifications has been satisfied; program flow then passes to the statement following the WAIT statement. When a WAIT statement of the second form is encountered, the expression is evaluated to give an integer m. The program flow is suspended until any m of the n wait specifications have been satisfied. If m is zero or negative, program flow continues. If m is greater than n, all wait specifications will be satisfied before program flow continues.

The following is a simple illustration of the use of the WAIT statement:

```
CRANK:  PROCEDURE;
        DECLARE (N, M, A(N, M), B(N, M)) AUTOMATIC;
        READ LIST (N,M,A,B) , TASK (TAU) ;
        DECLARE (KAPPA, BETA) EXTERNAL;
        TEMP1=SIN(KAPPA) **3;
        TEMP2=COS(BETA-1) ;
        TEMP3= (TEMP1-TEMP2) /100+MOD(MAX(TEMP2,TEMP1) ,3) ;
        WAIT(TAU) ;
        A=TEMP3*A; B=TEMP3*B;
        WRITE DATA (A,B) ;
        END CRANK;
```

THE STOP STATEMENT

General Form:

```
STOP;
```

The STOP statement causes immediate termination of a program.

THE EXIT STATEMENT

General Form:

```
EXIT;
```

The EXIT statement causes termination of the task that contains the statement and all tasks attached by this task. (See Chapter 18.)

THE DELAY STATEMENT

General Form:

DELAY (scalar expression) ;

The execution of the DELAY statement causes evaluation of the expression and conversion to an integer n, followed by suspension of execution of the controlling task for n milliseconds. Execution will continue immediately after n milliseconds only if the controlling task is of sufficiently high priority to cause selection of this task in preference to all other ready tasks.

Example:

```
PROO1: DELAY (1000) ;
```

THE FETCH STATEMENT

General Form:

FETCH (scalar expression) , task option;

On execution of the FETCH statement, the scalar expression is evaluated and, where necessary, converted to a character string. This string specifies a program name. The specified program is fetched and made accessible. It is assumed that the program was inaccessible before the FETCH.

After execution of the FETCH, the program may be invoked by a form of the CALL statement, which is discussed earlier in this chapter.

Data declared EXTERNAL, task identifiers, and file names are not shared between programs made accessible by a FETCH statement.

Initial values for STATIC data in a program will be established at the time of fetching.

Example:

```
FETCH ('PROCTL');
```

THE DELETE STATEMENT

General Form:

DELETE (scalar expression) ;

On execution of the DELETE statement, the scalar expression is evaluated and, where necessary, converted to a character string. This string specifies a program name which must appear in a previously executed FETCH statement.

The statement causes the specified program to be made inaccessible. After execution of a DELETE statement, the program name may not be specified in a CALL statement before the execution of a subsequent FETCH.

Deletion of a program includes deletion of its STATIC data areas and of all storage allocated by it. Care should be taken not to delete an active program.

Example:

```
DELETE ('PROCTL');
```

When an interrupt occurs during program execution, standard system action is taken; however, NPL provides the facility to override this system action on most machine and system interrupts. A programmer can specify the particular action to be taken when an interrupt occurs and can record the status of the program at the point of interrupt. In addition, a programmer can initiate programmed interrupts and can simulate machine interrupts to facilitate debugging.

THE ON COMPOUND STATEMENT

General Form:

```
ON condition SNAP unit
ON condition SYSTEM;
```

SNAP may be omitted. The unit may be a group or a begin block.

Execution of an ON statement enables a condition. The occurrence of an enabled condition is called an interrupt. The permitted conditions are listed in Appendix 5. In the case of FIXEDOVERFLOW, SUBSCRIPTRANGE, and SIZE, the occurrence of the condition will only signal the condition, if signalling is specified. (See the "PROCEDURE Statement" IN Chapter 7 and "Scalar Assignment" in Chapter 11.) For other conditions, the occurence always results in signalling. When the signal occurs, the action specified by the enabling statement is performed.

In no way other than activation from such an interrupt may flow of program control be transferred to statements within the ON unit.

In the first form, if SNAP has been specified and an interrupt occurs, then information relevant to the status of the program at the time of interrupt is listed on a debugging file. Control is then passed to the first statement of the unit.

In the second form, when an interrupt occurs, standard system action for the condition is performed as described in Appendix 5.

Subject to restrictions specified for particular conditions, enabling carries down into descendant blocks (i.e., blocks activated as a result of normal sequencing through or invocations from the block containing the enabling ON statement).

If a condition is enabled in a block where the same condition has been previously enabled by an ON statement executed during the same activation of the block, the first enabling is overridden.

If a condition is enabled in a block where the same condition has been previously enabled by an ON statement executed in a dynamically embracing block, the former enabling is stacked and replaced by the latter. Upon termination of execution of a block, all conditions are enabled as they were in the dynamically embracing block.

When an interrupt occurs in a descendant block, control is passed to the unit in the enabling ON statement. Descendant blocks are temporarily suspended. The ON unit is executed as if it were called as a procedure block at the time of interupt; this concept, in particular, determines the generations of data available in the ON unit and the effect of further ON statements within the ON unit. Controlled data which has subsequently been freed is inaccessible.

When execution of the unit is completed normally, control returns to the point following interrupt, the environment that existed before the interrupt being re-established.

If transfer of control out of the unit is specified, all descendant blocks are synthetically released (e.g., appropriate storage is released) before the transfer takes place.

At the start of execution of a program all conditions are enabled for system action. ON conditions enabled when a task is attached (see Chapter 18) never carry over to the attached task. At the start of execution of a new task, all conditions are enabled for system action.

THE REVERT STATEMENT

General Form:

```
REVERT condition;
```

The condition is as described for the ON statement (see Appendix 5).

Execution of this statement causes the named condition to be enabled as it was in the closest dynamically embracing block.

Examples:

```
REVERT OVERFLOW;
REVERT ENDFILE (MASTER);
```

THE SIGNAL STATEMENT

General Form:

```
SIGNAL condition;
```

The condition is as described for the ON statement.

Execution of this statement simulates the raising of the condition, i.e., causes the action specified by the currently enabled ON (for this condition) to be performed.

Examples:

```
SIGNAL OVERFLOW;
SIGNAL ENDFILE (MASTER_FILE);
SIGNAL CONDITION(X);
```

Attributes are characteristics that are associated with identifiers in an NPL program. However, while some attributes are characteristics of the identifier itself, e.g., the scope attribute; other attributes are characteristics of the item that the identifier represents, e.g., the mode attribute.

ATTRIBUTE CLASSES

Data attributes
Dimension attributes
ABNORMAL, NORMAL, and SECONDARY attributes
Entry name attributes
Scope attributes
Storage class attributes
DEFINED attribute
INITIAL attribute
Symbol table attributes
Parameter attributes
Structure attributes
File attributes

DATA ATTRIBUTES

ARITHMETIC ATTRIBUTES

Arithmetic data may be declared to have the following attributes. (See also "Labels" in Chapter 7.)

Radix: Arithmetic data may be declared to have the attribute BINARY or the attribute DECIMAL. If radix is unspecified, the default attribute depends on the first letter in the name; if I-N, BINARY is assumed; otherwise DECIMAL is assumed.

Scale: Arithmetic data may be declared to have the attribute FIXED or the attribute FLOAT. If scale is unspecified, the default attribute depends on the first letter of the name; if I-N, FIXED is assumed; otherwise FLOAT is assumed.

Mode: Arithmetic data may be declared to have the attribute REAL or the attribute COMPLEX. If mode is unspecified REAL is assumed.

Precision Appendage: The precision of arithmetic data may be declared (w) or (w,d), where w and d are decimal integer constants. The first form (w) is used for data of scale FLOAT and specifies that at least w decimal or binary digits of significance are to be maintained. Both forms may be used for data of scale FIXED. If d is omitted, it is assumed to be zero. w gives the total number of decimal or binary digits to be maintained and d the scale factor for the data. The precision appendage immediately follows a scale, radix, or mode attribute; it may not appear alone or separated from one of these attributes. If precision is unspecified, an implementation-defined default is assumed.

Numeric Field Representation: Arithmetic data to be represented as a numeric field is declared with the picture attribute, which has the form,

PICTURE (picture specification)

The picture attribute describes the format of the associated numeric field. The picture specification is a series of picture characters (see Appendix 2). Assignment of a value to numeric field causes the value to be edited to the appropriate format.

The picture describing a numeric field defines the radix, scale, mode, and precision of the associated arithmetic data.

BIT STRING ATTRIBUTES

Bit string data may be declared to have either or both of the following attributes.

BIT (length)
VARYING

The length specifies the actual length of fixed-length strings and the maximum length of variable-length strings. The length may be an expression or *. The latter may be declared only for a name or value formal parameter, specifying that the length is the same as that of the corresponding argument, or for NAME CONTROLLED formal parameters or CONTROLLED data, specifying that the length is to be taken from the last allocation. The length of strings declared STATIC must be a decimal integer constant.

If the length is an expression, it will be evaluated and converted to an integer at the point of allocation, or on entry to the declaring block for name parameters. See "Prologues" in Chapter 16 for rules relating to the formulation of these expressions.

CHARACTER STRING ATTRIBUTES

Character string data can have one or more of the attributes.

CHARACTER (length)
VARYING
PICTURE (picture specification)

LABEL VARIABLE ATTRIBUTES

A label variable must be declared to have one of the following attributes.

LABEL
LABEL (statememt label constant 1,...,statement label constant n)

The attribute specifies that the associated identifier is a label variable. The optional bracketed list gives a list of statement label constants known at the point of the LABEL declaration. It specifies that during the execution of the program the value of the label variable will always be one of the listed constants. This information is given to aid the optimization that can be done on the program.

THE DIMENSION ATTRIBUTE

The dimension and bounds of an array are declared using the attribute,

(1st bounds, ... , nth bounds)

The bounds may be either all asterisks or all of the following form.

expression
or expression 1: expression 2

The asterisk notation may be used for CONTROLLED variables, NAME parameters, and AUTOMATIC VALUE parameters.

The expressions are evaluated and converted to integer (see "Integer Conversion" in Chapter 6) when storage is allocated for the array (or when linkage is established for name parameters). The first form gives the upper bound, the lower bound being assumed to be 1. The second form specifies both lower and upper bounds in that order. The lower bound must be algebraically less than the upper bound. The dimension attribute must always immediately follow the array identifier in an array declaration. The bounds of arrays declared STATIC must be decimal integer constants. See "Prologues" in Chapter 16 for the rules relating to expressions used in the dimension attribute.

The dimension attribute, if present, must always appear as the first attribute in a name declaration (see "Name Declaration" in Chapter 16). Hence, the dimension attribute must either follow the array name; or, if factored, the dimension attribute must follow the right parentheses.

THE ABNORMAL, NORMAL, AND SECONDARY ATTRIBUTES

ABNORMAL

The ABNORMAL attribute may be declared for any variable. It specifies that the data may be unpredictably altered during the execution of the program (e.g., by asynchronous operation or the execution of an ON unit as described under "The ON Compound Statement" in Chapter 14) and that every time ABNORMAL data is referenced its associated storage must be accessed for its current value.

NORMAL

The NORMAL attribute may be used to override a factored or implicit ABNORMAL attribute.

SECONDARY

This attribute specifies that where possible and necessary, less than normally efficient storage may be allocated for the variable. The attribute may be declared only for major structure names and variables not contained in structures or arrays.

ENTRY NAME ATTRIBUTES

An entry may be declared to have any of the attributes, ENTRY,ABNORMAL, NORMAL, SETS, USES, GENERIC, and BUILTIN. (The first five attributes have been discussed under "The ENTRY Attribute" and "Abnormality of Procedures" in Chapter 9.) An entry name may be declared with any of the data attributes in this Chapter. These attributes specify the characteristics of the value returned when the entry name is invoked as a function. If the data attributes are not specified, default or implicit characteristics (see Chapter 17) will be assumed for the value returned.

The SETS, USES, GENERIC, and BUILTIN attributes all imply ENTRY.

THE GENERIC ATTRIBUTE

The programmer may define a family of entry names using the attribute,

GENERIC (entry name declaration 1, ... , entry name declaration n)

Each entry name declaration corresponds to one member of the family. All the entry name declarations must have an ENTRY attribute. They may optionally have ABNORMAL and data attributes. They may not have the GENERIC attribute. These ENTRY attributes must specify attributes for every parameter of the associated entry name. Attributes unspecified but required for full definition will be deduced from default rules (see Chapter 17).

When a generic entry name is referenced, the attributes of the arguments specified must match exactly the list in the ENTRY attribute of one and only one member of the family. The reference will be interpreted as a reference to this member.

The choice of entry name may be based on the number of arguments in the reference to the name.

Generic entry names (as opposed to references) may be specified as arguments if the invoked entry name is declared with the ENTRY attribute (explicit or implicit for internal procedures). This ENTRY attribute must specify that the appropriate parameter is an entry name and specify by means of a further ENTRY attribute the attributes of all its parameters. This enables a choice to be made of which family member is to be passed.

When arrays are involved the choice is based only on the dimensionality of the array, not on extents. When strings are involved, the lengths do not participate.

Example:

```
DECLARE BESSEL GENERIC(FXBESS ENTRY(FIXED) FIXED,
                       FLBESS ENTRY(FLOAT) FLOAT,
                       XLBESS ENTRY(FLOAT) FIXED),
        X ENTRY(FLOAT, FIXED ENTRY(FLOAT));
Y = Y + BESSEL(Y);
CALL X(Y, BESSEL);
```

The assignment statement results in the invocation of the procedure FLBESS. The CALL statement results in the function name XLBESS being passed as an argument to procedure X.

THE BUILTIN ATTRIBUTE

BUILTIN

This attribute specifies that references to the associated identifier within the scope of this declaration be taken as references to the built-in function of the same name.

This attribute is used when it is desired to reference a built-in function in a block, but the block is contained in another block where the built-in function name has been declared to have another use.

SCOPE ATTRIBUTES

INTERNAL
EXTERNAL or EXTERNAL (identifier)

The second form of the EXTERNAL attribute specifies that the scope of the declared identifier is to be limited to the union of the scopes of declarations of the same identifier with the same specified heading. For a full discussion of scope, see the section headed "Declarations" in Chapter 7.

STORAGE CLASS ATTRIBUTES

AUTOMATIC
STATIC
CONTROLLED

For a full discussion of storage class, see Chapter 8.

These attributes may not be specified for entry names and file names, or with the DEFINED attribute. The first two attributes may not be specified for NAME parameters. The last two attributes may not be specified for label variables.

THE DEFINED ATTRIBUTE

DEFINED base identifier position option

The DEFINED attribute may be declared for scalar, array, and structure identifiers. It specifies that the defined item should occupy the same storage as the base.

In general, the characteristics of the defined item must be the same as those of the base. However, a certain amount of mixed defining is allowed within the following two classes:

1. The bit class, comprised of
   a. numeric fields of radix binary
   b. fixed-length bit strings
   c. arrays of either (a) or (b)
   d. structures of either or both of (a) and (b)
2. The character class, comprised of
   a. numeric fields of radix decimal
   b. fixed-length character strings
   c. arrays of either (a) or (b)
   d. structures of either or both of (a) and (b)

Coded arithmetic data may be defined only on coded arithmetic data of the same radix, scale, mode and precision. Label data may be defined only on label data. The base may be a variable length string; the defined item may never be.

The defined item must always be specified as a subset (including the full set) of the base.

Expressions specified in base subscript lists are evaluated when the defined item is referenced, not declared. Use of a defined item in an argument list is interpreted as a reference.

Data declared with the DEFINED attribute may not have any of the following attributes:

storage class

scope
NAME
INITIAL
VARYING
SYMBOL

In addition, if a defined variable is declared with the ABNORMAL or SECONDARY attribute, and these attributes conflict with the respective attributes of the base, this contributes an error.

The base identifier must always be known within the block where the defined identifier is declared and may not have been declared with the DEFINED attribute.

If the attributes of the defined data involve expressions (other than in a defining subscript list; see "Subscripted Array Defining" in Chapter 15), these expressions are evaluated on entry to the declaring block, regardless of the storage class of the base. The base, however, is always taken as the generation existant (current generation) at each reference to the defined variable.

SCALAR DEFINING

Both defined item and base are scalars. The base may be subscripted, specifying a scalar element of an array. The defined scalar may not be an element of a structure or an array.

The permitted forms are:

Defined Item	Base
Coded arithmetic	Coded arithmetic of same radix, scale, mode, and precision.
Label	Label
Binary numeric field or bit string	Binary numeric field or bit string
Decimal numeric field	Decimal numeric field or character string

Where the base is a string, the POSITION option of the form

POSITION (decimal integer constant)

may be declared. It specifies an offset (n) from the start of the base where the defined item commences. If omitted, POSITION(1) is implied.

ARRAY DEFINING

Both defined item and base are arrays. The defined item must have a dimension specification, and may not be an element of a structure. The permitted forms are the same as for scalar defining. In array defining, there is a relation between each element of the defined array and a corresponding element of the base.

The POSITION option may be given when the base is an array of strings. It specifies that each element of the defined array commences at the nth bit or character of the corresponding element of the base array.

Two classes of array defining are permitted: simple array defining and subscripted array defining.

Simple Array Defining

The base must be an unsubscripted array name having the same number of dimensions as the defined array. The dimension bounds of the defined array must be a subset of the bounds of the base array. A subsequent subscripted reference to the defined array is interpreted as a reference to the base array with identical subscripts.

A subsequent unsubscripted reference to the defined array is interpreted as a reference to the declared subset of the base array specified by the dimension bounds.

Subscripted Array Defining

The base must be an unsubscripted array name followed by a defining subscript list. The base need not have the same number of dimensions as the defined array.

The defining subscript list defines the relation between the elements of the defined array and the base array. It must have as many subscripts as the base array has dimensions. The defining subscripts may be any expressions, including dummy variables of the form

iSUB

where i is a decimal integer constant in the range 1 to n; n is the dimensionality of the defined array. The subscript expressions must be of the form

$$a \pm a_1*1SUB \pm a_2*2SUB \pm \ldots \pm a_n*nSUB$$

where a is any scalar expression involving variables known within the block containing the DEFINED declaration. The integer value of the expression will be used. If any a is zero, that iSUB may be omitted. The iSUB's must appear in the order shown above.

A subsequent subscripted reference to the defined array is interpreted as follows:

Each iSUB in the defining subscript is replaced by the integer value of the ith subscript given for the defined array. Before replacement, the subscript is conceptually enclosed in parentheses.

The reference to the defined array elements is interpreted as a reference to the base array element specified by the generated subscript.

A subsequent unsubscripted reference to the defined array is interpreted as a reference to the array defined by the mapping.

If a defined array name is specified as an argument to an invoked procedure, the expressions in the defining subscript list are evaluated before the invocation. The invoked procedure can still reassign values to elements of the defined array by a name parameter; but the relation between the defined array elements and the base elements is frozen on entry.

MIXED DEFINING

Major structures, and arrays not contained in structures, having elements all of the same class as described under "The Defined Attribute" in this chapter may be defined on scalar strings of the same class and on structures having elements all of the same class.

Scalar strings may be defined on, i.e., have as a base

major structures

minor structures not contained in dimension structures

structure elements with the base being specified as a subscripted structure name

unsubscripted arrays not contained in dimensioned structures

All the elements must be of the same class as the defined string.

When the base is a scalar string, the POSITION option may be specified to indicate that the defined array or structure is offset from the start of the string. It may not be specified with mixed defining when the base is an array or structure.

Defining subscript lists may not be used with mixed defining.

Some examples of defining are illustrated below:

```
DECLARE A(M,N), AT(N,M) DEFINED A(2SUB,1SUB),
          D DEFINED A(I,I),B(0:M*N-1),
          BROWN(M,N) DEFINED B(-(M+1) + 1SUB + M*2SUB);
DECLARE 1 P, 2 Q CHARACTER (10), 2 R CHARACTER (100),
          PSTRING1 CHARACTER(110) DEFINED P;
          LIST CHARACTER(40),
          ALIST CHARACTER(10) DEFINED LIST,
          BLIST CHARACTER(20) DEFINED LIST POSITION(20),
          CLIST CHARACTER(10) DEFINED LIST POSITION(10);
```

THE INITIAL ATTRIBUTE

INITIAL (item1,...,item n)
INITIAL CALL entry name (argument list)

In the following discussion, the term constant denotes either a constant or a complex expression of the form
real constant + imaginary constant
or real constant - imaginary constant

The INITIAL attribute specifies either constant values to be assigned to data when storage is allocated to it, or a procedure to be invoked to perform initialization at allocation. In the second form, the entry name and the arguments passed must satisfy the conditions stated in "Prologues" in Chapter 16. The second form must not be used to initialize static data. The INITIAL attribute is specified in the external procedure which will allocate storage for the data. For example, CONTROLLED data initial values must be specified in each procedure which will allocate the data.

The first form lists constant values. Only one is required for a scalar; more may be given for an array. In the latter case the constants specified are assigned to successive elements of the array in row major order (final subscript varying most rapidly). If too many values are specified, excess ones are ignored. If insufficient are supplied the remainder of the array is not initialized. The items in the list may be an asterisk (*) indicating no initialization for that element, an optionally signed constant, or a replication.

A replication has one of the following forms:

(replication factor) optionally signed constant

(replication factor) (item 1, ... , item n)
(replication factor) *

The items in the second form are as defined above, i.e., replication may be nested.

The replication factor may be any expression that satisfies the rules stated in "Prologues" in chapter 16. When storage is allocated, the expression is evaluated to give an integer specifying the number of repetitions. Replication factors for STATIC data however, must be constants. (See "Labels" in Chapter 7 for an alternative method of specifying initial values for label arrays.) Thus (10) (7) '1'B indicates ten seven-bit constants and (6) 'A' indicates one six-character constant 'AAAAAA'.

The INITIAL attribute may not be given for

entry names
file names
DEFINED data
structures Formal parameters passed by value.

Some examples of the INITIAL attribute are illustrated below:

```
DECLARE A(10,10) INITIAL ((20) 0, (20) ((3) 5,10)),
IDTY(N,N) INITIAL(1, (N-1) ((N) 0,1)),
SWITCH INITIAL('1'B),
NINES CHARACTER(N) INITIAL((N) '9'),
STIR(*) INITIAL CALL STIRRER EXTERNAL CONTROLLED;
```

SYMBOL TABLE ATTRIBUTES

SYMBOL
SYMBOL (identifier)
NOSYMBOL

The first form specifies that the declared identifier, which does not require name qualification for unique identification, should appear in the symbol table.

The second form, used when the declared identifier requires name qualification for unique identification, specifies that the identifier appearing in the parentheses should appear in the symbol table as a synonym for the qualified name. A variable whose name or synonym appears in the symbol table may have its values transmitted under data-directed input.

The third form specifies that the declared identifier should not appear in the symbol table.

PARAMETER ATTRIBUTES

NAME
VALUE

For a full discussion of parameters see Chapter 10.

THE LIKE ATTRIBUTE

LIKE structure name

The structure name may be qualified or unqualified but not subscripted. It must be known to the block containing the LIKE attribute. The structure named may not itself be declared with the LIKE attribute.

The LIKE attribute specifies that the identifier having the attribute should be taken to have a structure description identical to that declared for the named structure.

If the structure description of the named structure has been declared, and if a direct application of the description to the structure declared LIKE would cause an incorrect discontinuity in level numbers, then the level numbers will be modified by a constant subtrahend before application.

For example,

```
DECLARE 1 A,
          2 FIELD1,
             3 DTL1 PICTURE ($ZZ.99),
             3 DTL2 CHARACTER (10),
          2 FIELD2 BIT (50),
        1 X,
          2 FIELD1,
             3 SUBFLD1 LIKE A.FIELD1,
             3 TABLE (3),
          2 FIELD2 LIKE A.FIELD1;
```

is equivalent to

```
DECLARE 1 A,
          2 FIELD1,
             3 DTL1 PICTURE ($ZZ.99),
             3 DTL2 CHARACTER (10),
          2 FIELD2 BIT (50),
        1 X, 2 FIELD1,
             3 SUBFLD1,
                4 DTL1 PICTURE ($ZZ.99),
                4 DTL2 CHARACTER (10),
             3 TABLE (3),
          2 FIELD2,
             3 DTL1 PICTURE ($ZZ.99),
             3 DTL2 CHARACTER (10);
```

Also, in the following example,

```
DECLARE 1 A EXTERNAL, 2 (B,C,D), 1 E LIKE A;
```

is equivalent to writing

```
DECLARE 1 A EXTERNAL, 2 (B,C,D), 1 E EXTERNAL, 2 (B,C,D);
```

FILE DESCRIPTION ATTRIBUTES

File Type Attributes

The following attributes are used to describe data files (see Chapter 21).

File Attribute

An identifier which refers to a file has the attribute FILE.

Standard Attributes

The attribute is written STANDIN or STANDOUT and declares this file name to be a synonym for the standard input or output file, respectively.

Specific Medium Attribute

This attribute is written

MEDIUM (option list)

The options will be implementation defined and will be used to indicate the nature of the input/output media that are used for files at execution time.

Usage Attribute

This attribute is written

USAGE (option list)

The options will be implentation defined and will be used to indicate the desired destination, such as punched cards or printed page, for the data of output files at execution time.

Parity Attribute

This attribute is written

USAGE (option list)

The options will be implementation defined and will be used to indicate the parity of data recording on files at execution time.

Storage Equivalence Attribute

This attribute is written

POOL (file name)

The file name is the name of another file which may share, with the file currently being described, the storage necessary for transmission of data to and from external media. It is the responsibility of the programmer to avoid confusion of usage.

BLOCK Attribute

This attribute is written

BLOCK (x,y,z)

where: x is VARIABLE, FIXED, or SPECIAL
y is a decimal integer constant indicating the maximum length in characters of the block. (See Chapter 20.)
z is a decimal integer constant indicating the blocking factor i.e., the number of records per block.

This attribute enables the programmer to specify blocking details.

GROUP Attribute

This attribute is written

GROUP (n)

where n is the number of records per group.

DYNAMIC CONTROL ATTRIBUTES

The following attributes may be given in the file description or they may appear in OPEN, or CLOSE statements, in which case the file description is overridden.

Disposition Attributes

These attributes specify the final disposition of data. They are written

```
DISCARD
KEEP
STAY
```

For a description of these attributes, see "The OPEN and CLOSE Statements" in Chapter 21

Function Attribute

This attribute indicates the function of a file, and is written

```
INPUT
OUTPUT
INOUT
```

The attribute specifies that the function is input, output, or is to be used for direct access replacement for both input and output.

ACCESS ATTRIBUTE

The ACCESS attribute specifies both the file organisation and the manner of accessing that file. This attribute is written

```
ACCESS (x)
```

where x may be one of the following:

```
SEQUENTIAL
DIRECT
INDEXSEQUENTIAL
INDEXDIRECT
COMMUNICATIONS
```

The terms define the manner in which the "next record" is accessed as follows:

SEQUENTIAL	The next record is determined by the current physical position within the file.
DIRECT	The next record is determined by precomputing its position within the file.
INDEXSEQUENTIAL	The next record is determined by the current logical position within an ordered index which reflects physical positions within the file.
INDEXDIRECT	The next record is determined through an index which reflects physical positions within the file.
COMMUNICATIONS	The next record is determined by installation-defined teleprocessing conventions.

The KEY option of various input/output statements is implemented for a given file according to the access method of that file. The KEY value for DIRECT access must be specified as a decimal integer constant; while for INDEXDIRECT, the value is converted to a character string, if necessary. SEQUENTIAL record is determined by a pre-established sequence. However, the KEY option may be specified for a file organized for INDEXSEQUENTIAL access in order to specify where the "sequential" accessing within the index is to begin.

THE ZERO ATTRIBUTE

ZERO

This attribute specifies that trailing blank characters in data input fields are to be treated as numeric zeros. It has no effect on output. Examples:

```
DECLARE INPUT FILE INPUT BLOCK (FIXED,200,5);
DECLARE BINARY OUTPUT FILE ODD;
DECLARE CUP FILE STANDIN CARD;
```

NAME DECLARATION

General Form:

level number identifier attribute 1 ... attribute n

The identifier is the name of the item. The level number is a decimal integer and must be used when the identifier is a structure or is contained in a structure (see "Structures" in this Chapter).

DECLARATIONS AND FACTORING OF ATTRIBUTES

General Form:

name declaration 1 , ... , name declaration n

level number (declaration 1 , ... , declaration n)
attribute 1 ... attribute n

A declaration is a list of name declarations separated by commas.

From this list, attributes common to several name declarations can be factored. This is achieved by enclosing the subset of name declarations in parentheses and following it by the list of factored attributes, separated by blanks. Factoring may be nested to any desired level.

If a factored attribute is in conflict with an attribute specified in the region of factoring then the inner attribute overrides the factored attribute. Attributes on the same factoring level may not be in conflict with one another. If a factored attribute cannot legally be applied to a name declaration in the region of factoring, then it is not taken to apply to that name declaration.

A level number may be factored before a parenthesized list of declarations. If the name declarations in the region of factoring have level numbers, these override the factored number.

Examples:

```
JOE FLOAT, JIM FIXED (3,5), JACK BIT (10)
1A,2 (B,3 C CHARACTER (5), 3 D BINARY, E) STATIC
```

THE DECLARE STATEMENT

General Form:

DECLARE declaration list ;

The DECLARE statement is a nonexecutable statement used for the specification of attributes of identifiers and names. All the attributes given for a particular name must be declared together in one DECLARE statement. Specification of contradictory or additional attributes for a particular name in more than one name declaration or DECLARE statement in the same procedure is illegal.

Attributes of external identifiers, declared in separate blocks and external procedures, must not conflict or supply explicit information that was not explicit or implicit in other declarations.

STRUCTURES

STRUCTURE DESCRIPTION BY LEVEL NUMBER

A major structure is a structure not contained in a structure and at level one. All other structures are minor structures. All elements contained in a structure at level n are at a level greater than n. They need not be at level n + 1.

Structuring is specified by following the declaration of the structure identifier by declarations for the contained items. The structure identifier and the identifiers of all contained items must be preceded by a level number.

If a minor structure is at level n, the minor structure contains all items with level numbers greater than n declared before the next item with a level number not greater than n. A major structure description is terminated by

the next item being at level 1,
the next item having no level number,
the end of a declaration list.

STRUCTURES AND THE DIMENSION ATTRIBUTE

A dimension attribute (see "The Dimension Attribute" in Chapter 15.) may be given for a structure name. The structure is then an array of structures and all contained items are arrays or arrays of structures. Contained scalar items must be referenced by subscripted names. Contained structure elements must be referenced by subscripted qualified names; cross sections of contained arrays must be referenced by using the asterisk notation. (See "Cross Sections of Arrays" in Chapter 4.)

Example:

```
1 A(2), 2 B, 2 C(2), 3 E(2), 3 F
```

has the form

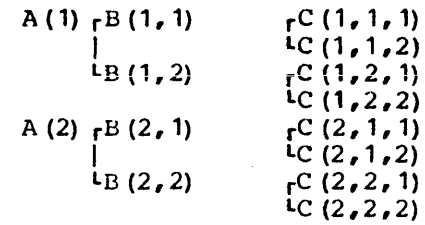
```
A(1) ┌ B(1)
     │ C(1,1)    ┌E(1,1,1)
     │           │E(1,1,2)
     │           └F(1,1)
     │ C(1,2)    ┌E(1,2,1)
     │           │E(1,2,2)
     │           └F(1,2)
     └
A(2) ┌ B(2)
     │ C(2,1)    ┌E(2,1,1)
     │           │E(2,1,2)
     │           └F(2,1)
     │ C(2,2)    ┌E(2,2,1)
     │           │E(2,2,2)
     │           └F(2,2)
     └
```

It should be noted that if the dimension attribute is factored from a structure declaration, then it applies to each level in the structure.

Example:

```
DECLARE (1 A,2 B,2 C) (2);
```

has the form



DATA ATTRIBUTES AND STRUCTURES

Structure names may not be given data attributes. Data attributes factored from regions containing structure declarations are not taken to apply to the structures.

SCOPE ATTRIBUTES AND STRUCTURES

Major structure names may be declared EXTERNAL. Items contained in structures may not be declared EXTERNAL, and if INTERNAL is unspecified are assumed to be INTERNAL.

STORAGE CLASS ATTRIBUTES

All items in a structure must be of the same storage class.

A major structure name may be given a storage class which will be assumed to apply to all elements of the structure. If a structure is controlled, only the major structure may be allocated and freed not the elements (see "The CONTROLLED Storage Class" in Chapter 8).

STRUCTURES AND THE DEFINED ATTRIBUTE

For a full discussion on the use of the DEFINED attribute see The DEFINED Attribute in Chapter 15.

PROLOGUES

On entering a block, certain actions are performed, e.g., allocation of storage for automatic variables. The initiation of a block is known as a prologue.

On entry to the prologue, the following items are available for computation:

(a) variables declared outside the block and known within it.

(b) variables declared STATIC and known within the block.

(c) formal parameters passed to the block by name.

(d) variables defined on items belonging to (a), (b), or (c) provided that all items involved in any defining subscripts belong to (a), (b), (c) or (d)

The prologue makes available for computation all the other variables known within the block:

(e) formal parameters passed by value to the block

(f) automatic variables declared in the block

(g) variables defined on items belonging to (e) or (f) with defining subscripts which depend on items of (e), (f) or (g)

In making these items available, the prologue may need to evaluate expressions defining lengths, bounds, replication factors, and initial values.

The evaluations necessary to make an item available may reference other items being made available. In such circumstances, the items referenced must not require for their allocation reference to the first item, either directly or by reference to another item that does; e.g., the following statement is illegal

```
DECLARE ((A(M) INITIAL(3,2,1), M INITIAL(A(1)))AUTOMATIC;
```

The following restrictions are imposed to ensure that the total effect of the prologue does not depend on the order in which the expressions are evaluated.

The evaluations must not invoke abnormal functions, except in the case of evaluations for initializing. These latter must be abnormal only because they set the variable being initialized. The sequence in which the evaluations reference any abnormal data is not defined.

Function calls within the evaluations must not reference items being made available by the prologue.

CHAPTER 17: IMPLICIT AND DEFAULT FACILITIES

As discussed in the previous chapter, when attributes are declared in a DECLARE statement, they are associated with a particular identifier. However, other methods exist in the language for this association of attributes with identifiers. For example, in an IMPLICIT statement, attributes are declared but they are associated with a specified set of identifiers. In addition, when attributes are not declared for an identifier, they are assumed from default rules.

THE IMPLICIT STATEMENT

General Form:

IMPLICIT *implicit declaration 1* , ... , *implicit declaration n* ;

The format of the IMPLICIT statement is identical to that of the DECLARE statement except that identifiers are replaced by letters or "letter ranges" and that level numbers are not permitted. A "letter range" is of the form

letter 1 - *letter 2*

A letter range is a shorthand representation of all the letters, letter 1 through letter 2 separated by commas and enclosed in parentheses. Letter 1 must appear earlier in the alphabet (or extended alphabet) than letter 2 (see Appendix 8).

If the dimension attribute is specified, it must immediately follow a letter or letter range; it may not be factored, and it must specify constant bounds. As in the DECLARE statement, attributes which are factored are overridden by conflicting attributes factored or unfactored at deeper levels. Attributes on the same factoring level must not be in conflict with one another.

The IMPLICIT statement specifies that all undeclared or partially declared identifiers commencing with a letter appearing in the IMPLICIT statement have the attributes associated with the letter as specified in the implicit declaration. A partially declared identifier is an identifier that has been declared with insufficient attributes for it to be fully defined. In this case, explicitly declared attributes override conflicting implicit ones.

The scope of an IMPLICIT statement is an external procedure. Any given letter may thus appear only once in an external procedure in IMPLICIT statements.

The IMPLICIT statement may not involve any identifiers other than keywords. Thus the attributes DEFINED and LIKE are not permitted because their use involves the appearance of variable names.

Example:

```
IMPLICIT C BINARY COMPLEX, (B-G EXTERNAL, U-W STATIC,
          I EXTERNAL, J) INITIAL (0) ;
IMPLICIT P-R CHARACTER (20) INITIAL (20) ,S-U (100, 100) EXTERNAL;
```

IMPLICIT AND DEFAULT ATTRIBUTES

It is unnecessary for all or any of the attributes of identifiers to be explicitly declared. The following process is performed for associating attributes with identifiers.

IMPLICIT ATTRIBUTES

Explicitly declared identifiers are classified by analyzing DECLARE statements (see "Declarations and Factoring of Attributes" in this Chapter) and statement labels (see "Labels" in Chapter 7) using the rules of scope to distinguish between different uses of the same name (see "Declarations" in Chapter 7) The contextual usage of identifiers is then examined.

Identifiers occuring where only a file name is allowed are classified as file names, viz.,

File option (READ, WRITE, SPACE, GROUP, SKIP, PAGE, LAYOUT, SEARCH, SORT)
File specification (OPEN, CLOSE)
Giving option (SORT)
Pool option (DECLARE)
File conditions (ON, REVERT)

Identifiers occuring in statements of the form CALL identifier, with or without an argument list, are classified as external entry names; however, if the occurrence of the entry name lies within the scope of the same identifier used to label a PROCEDURE or ENTRY statement, the identifier is classified as an internal entry name. Identifiers appearing in CALL options are similarly treated.

Identifiers occurring where only task identifiers are allowed (TASK option, WAIT statement, COMPLETE built-in function) are classified as task identifiers.

Only the foregoing contextual usages are examined. Contextual usages of an identifier must be consistent with each other and with the explicit declaration, if any, within the same scope. (Note: For identifiers with no explicit declaration, the scope is the entire external procedure.)

Default attributes, as modified by IMPLICIT statements, are then given to all identifiers. However, those identifiers which have already been classified as specified in the preceding paragraphs, are not given default attributes which would conflict with that classification.

Finally, if an identifier appears in an expression with an immediately following parenthesized list, and that identifier has no dimension attribute, it is further classified as an entry name. The classification as an external or an internal entry name is made in the same manner as for identifiers occurring in the CALL statement. This classification by usage as a function reference must not conflict with the attributes already attached to the identifier as specified in preceding paragraphs.

DEFAULT ATTRIBUTES

Data Type: If none of the attributes, CHARACTER, BIT, or LABEL, is specified, then arithmetic data type is assumed.

Arithmetic Variables: If the radix, scale, or mode, are not specified, the default attributes depend on the first letter of the identifier. If I through N, FIXED REAL BINARY is assumed; otherwise, FLOAT REAL DECIMAL

is assumed. If any, but not all of radix, scale, or mode, are supplied, the omitted attributes are assumed according to the following table:

Specified	Assumed		
	Radix	Scale	Mode
BINARY		FLOAT	REAL
DECIMAL		FLOAT	REAL
FIXED	DECIMAL		REAL
FLOAT	DECIMAL		REAL
REAL	DECIMAL	FLOAT	
COMPLEX	DECIMAL	FLOAT	
BINARY FIXED			REAL

The preceding table also implies the other combinations that may be specified.

If precision is not specified, an implementation-defined default precision will be assumed.

Entry Names:

If an external entry name appears as a function reference, the entry is assumed to be NORMAL; otherwise, the entry is assumed to be ABNORMAL. External entry labels invoked in CALL statements and internal entry labels (however they are invoked) are assumed to be ABNORMAL.

File Name: The EXTERNAL attribute is assumed.

Scope: If the scope is unspecified for variable names, INTERNAL is assumed.

Storage Class Attributes: If EXTERNAL scope is declared and the storage class is unspecified, STATIC IS assumed. If INTERNAL scope is declared and the storage class is unspecified, or if neither storage class nor scope is specified, AUTOMATIC is assumed.

Label Variables: The range of a label variable is assumed to be all statement labels known within the scope of the variable.

Parameter Attributes: Formal parameters without parameter attributes are assumed to be NAME, unless of type label, in which case VALUE is assumed.

Symbol Table Attributes: If no symbol table attribute is specified, NOSYMBOL is assumed.

THE SEQUENCE STATEMENT

General Form:

SEQUENCE decimal integer (character constant) ;

The SEQUENCE statement is a nonexecutable statement. Only one may appear in each external procedure

The decimal integer constant specifies the number of characters in the character constant. Any character may appear only once in the string. The characters may be any permitted data character (whether or not it has an associated graphic).

The statement specifies the collating sequence, in ascending order, for the external procedure. When character string comparison is performed during subsequent execution of the external procedure, the specified collating sequence will be used. Characters not specified in the string compare high with those specified. The result of comparing two unspecified characters is undefined. If a SEQUENCE statement is not given, an implementation defined collating sequence is used.

CHAPTER 18: ASYCHRONOUS OPERATION OF TASKS

The language allows specification of asynchronous operation of tasks. The specification is expressed by means of the TASK option that is applied to a CALL, FETCH, DISPLAY, and certain input/ output operations.

Synchronization of asynchronous operations may be effected with the WAIT statement.

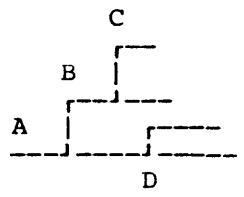
The main purpose of asynchronous operation is to share the computer resources among various tasks which may be performed asynchronously within a program. The actual sharing, or scheduling, of the resources is accomplished by the operating system within the bounds allowed by the specifications in the program.

TASK RELATIONSHIPS

Task relationships may be illustrated by a time chart. Assume that a horizontal increment in the figure below represents a unit of time, and that a single horizontal line represents a task. A vertical line connecting two tasks indicates that the attaching task (lower line) is initiating the attached task (upper line) at that point in time. This initiating of a task is accomplished when either of the following occur:

A procedure is invoked by a CALL with a TASK option. In this case, the execution of the invoked procedure, together with the execution of all procedures it invokes, etc., comprise the attached task. The CALL is a member of the attaching task.

A DISPLAY, FETCH or input/output statement with a TASK option attached is executed.



THE TASK OPTION

General Forms:

TASK (task identifier , priority)
TASK (task identifier)

The TASK option specifies that the operation to which it is appended is to be invoked asynchronously and attached as a task. This task is identified by the specified task identifier which may later be specified in a WAIT statement or as an argument of the COMPLETE built-in function.

An optional priority may be included. As part of the task option, the priority is specified by any scalar expression which is evaluated and converted to an integer (positive or negative) when the statement carrying the TASK option is executed. The integer indicates a priority relative to the attaching task. If the priority is omitted, zero (relative to the attaching task) is assumed.

If, during program execution, the control system is required to pass control to one of several tasks, the decision will be reached on the basis of their priorities. If the priority is omitted, a system standard priority will be provided.

DATA ALLOCATION ACROSS TASKS

The rules of scope of identifiers hold across task boundaries. It is therefore the responsibility of the programmer, by use of the WAIT statement, to avoid freeing storage allocated in the attaching task and accessed in the attached task.

An attached task has almost the same access to the attaching task's data as it would have if it were executed synchronously; that is, when it is attached all allocations of controlled variables known to the attaching task are passed to the attached task. However, subsequent allocations in the attached task are known only within the attached task and its subsequent descendants; subsequent allocations in the attaching task are known only within the attaching task and its subsequent descendants. Thus, the stack of allocations for a controlled variable splits into separate stacks, with a common part preceding task initiation and separate parts after task initiation. A task may only free storage which it allocated. All storage allocated within a task is destroyed when that task is completed.

The statements in the preceding paragraph apply as well to allocations of CONTROLLED NAME parameters.

TERMINATION OF TASKS

A task is terminated in one of the four following ways:

1. The execution of an EXIT statement at any time within the task.
2. The execution of a RETURN statement in the "top" procedure of a task, i.e., a procedure that has been invoked asynchronously.
3. The execution of an END statement that terminates the "top" procedure of a task.
4. The execution of a STOP statement in any task of the program.

STACKING OF TASK IDENTIFIERS

When a task is attached, the task identifier is recorded in a list of active tasks within the attaching task. This list contains the identifiers of all active tasks which have been attached by the task but have not yet been recognised as complete by a WAIT statement (see Chapter 13). When a task is recursively attached, the list element is a stack which operates on a last-in-first-out basis.

The list of task identifiers is only known to the task which generated the list. The list is not copied into the storage of attached tasks which initially have null lists. Thus, there can be no reference by one task to a task that was attached by some other task. A task

cannot wait on, or test with the COMPLETE built-in function, any task other than its own immediate descendants.

Program parameterization, modification and augmentation may be achieved using macro variables and procedures. Conditional and iterative program generation can be specified using compile-time statements.

MACRO VARIABLES

Macro variables are represented in the language by identifiers which must be declared in a macro DECLARE statement. A macro variable can only be declared either FIXED or CHARACTER.

FIXED specifies an integer variable with an implementation-defined precision.

CHARACTER specifies a character string. For a fixed-length string, an explicit length must be declared. For a variable-length string, the attribute VARYING must be declared; but no length specification is allowed. An implementation defined maximum length will be assumed.

Macro variables may be given an INITIAL attribute. The constant specified must be of the same type as the macro variable, i.e., either a decimal integer constant or character constant.

The appearance of a macro variable in the source program causes the replacement of the variable by its current "value". In the case of character variables, this is the contents of the string (with no enclosing quote marks) . In the case of fixed macro variables, the value is a character representation of the decimal integer.

MACRO PROCEDURES

Macro procedures are represented in the language by macro entry names that are invoked by macro function references. A macro procedure is programmer defined by an external procedure that has the option MACRO specified in the PROCEDURE statement.

All formal parameters of a macro procedure must be declared CHARACTER VARYING with a maximum size. A macro procedure must be a function procedure and logically end with the statement

RETURN (expression) ;

The value returned by a macro procedure will always be a variable-length character string. The option MACRO in the PROCEDURE statement implies the procedure attributes CHARACTER VARYING with an implementation defined maximum length.

Macro procedures may not perform input/output operations (including SAVE and RESTORE) , or use the ON statement. They may not invoke procedures which use these operations. Otherwise macro procedures may use the full facilities of the language.

Macro procedures must be declared with the attribute ENTRY in a macro DECLARE statement in the invoking procedure. It is assumed for such procedures that the value returned is a variable-length character string. In addition, when a macro procedure requires no arguments, the attribute NOARGS must be declared.

THE OPERATION OF MACRO PROCEDURES

The appearance of a macro function reference in the source program causes the macro procedure to be invoked. The procedure returns a character string result. This character string then replaces the function reference in the source text (with no enclosing quote marks).

Macro procedure invocations may have as arguments any expressions. These expressions may contain macro variables and macro function references. The replacement associated with these items will, as described previously, be performed before the macro procedure is invoked. The resulting expressions will then be conceptually enclosed in quote marks and passed as variable-length character string arguments to the macro procedure. The arguments to a macro procedure are delimited by the parentheses and commas of the argument list, so may include non-significant blanks.

MACRO EXPRESSIONS

Macro expressions may have as operands

character string constants
decimal integer constants
macro variables

Macro expressions may take one of the following forms:

(a) operand
(b) ±operand
(c) operand arith operator operand
(d) ±operand arith operator operand
(e) operand || operand ... operand || operand

The operands of forms (a) and (e) may be any of the permitted operands. The operand of forms (b), (c), and (d) must be decimal integer constants or fixed macro variables. Expressions of form (e) may involve conversion from type integer to type character. The conversion is performed according to list-directed transmission rules (see Appendix 7). The arithmetic operator in forms (c) and (d) may be + - / * .

Macro expressions are subdivided into fixed and character expressions. Macro expressions involving macro variables which have not been assigned a value are in error.

COMPILE-TIME STATEMENTS

THE MACRO DECLARE STATEMENT

General Form:

% DECLARE macro declaration list ;

The macro declaration list is a normal declaration list, except that only the attributes described in this section may be used, i.e., FIXED, CHARACTER, VARYING, size, INITIAL, ENTRY, NOARGS, and only one level of attribute factoring is allowed.

Only one macro DECLARE statement is allowed in an external procedure. The DECLARE statement must immediately follow the PROCEDURE statement. All macro variables and procedures referenced in an external procedure must be explicitly declared.

The scope of a macro identifier is the entire external procedure. Macro identifiers cannot be overridden by redeclarations. All appearances of the identifier (outside character constants and comments) are taken as referring to macro identifiers.

THE COMPILE-TIME ASSIGNMENT STATEMENT

General Form:

% label : macro variable = macro expression ;

The label is optional.

The statement causes the value of the macro expression to be assigned to the macro variable. If the expression is of type fixed, the variable on the left may be either of type fixed or character. If the expression is of type character, the variable on the left must also be of type character unless the value of the expression is a string which contains a decimal integer constant. In this latter case, the variable may be of type fixed.

All conversions implied in assignment are performed according to list directed transmission rules (see Appendix 7).

THE COMPILE-TIME NULL STATEMENT

General Form:

% Label:

Compile-time control may be passed to compile-time null statements.

THE COMPILE-TIME IF STATEMENT

General Form:

% label : IF macro expression comparison operator macro expression
THEN unit ELSE unit

The label is optional. A unit is a begin block or a group.

The statement causes conditional selection of source program text. If the macro relation has a true value, the first unit is selected and the second rejected, if false, the second unit is selected. The ELSE clause may be omitted, in which case a false result merely causes the rejection of the first unit.

The subsequent text selected is that following the IF statement. Each unit may be a compile-time statement or a unit containing compile-time statements and noncompile-time statements.

THE COMPILE-TIME GO TO STATEMENT

General Form:

% label : GO TO label ;

The first label is optional and the second label must precede a compile-time statement.

Control is transferred to the compile-time statement carrying the label appearing in the GO TO statement. The action associated with the compile-time statement is performed; and unless the statement specifies another transfer, the source text following it is selected.

COMPILE-TIME ACTIVITY

In general, the source text need not be syntactically correct prior to the execution of compile-time statements and macro variable and function replacement. However, the PROCEDURE statement heading the external procedure must commence with the key word PROCEDURE and be properly terminated with a semicolon. This statement must not contain improperly formed comments or character string constants. This does not prejudice the legal appearance of comments before the PROCEDURE statement.

The macro DECLARE statement following the PROCEDURE statement must be syntactically correct. Further requirements on the syntax of the source text are explained by describing the process involved. Three classes of text will be referred to in the following exposition:

1. source text, which is that presented by the programmer,
2. generated text, which is that introduced by replacement of macro variable names and function references, and
3. program text, which is the result of macro and compile-time activity.

Program text must be a syntactically and logically correct external procedure.

The first compile-time action performed is the construction of a list of macro variable names and function names. The list is ordered according to appearance in the macro DECLARE statement. If initial values are specified, they are associated with the list item.

The source text, including the PROCEDURE statement, but excluding the macro DECLARE statement, is scanned from left to right, top to bottom. The scan looks for

```
comments
string constants
macro identifiers
compile-time statements
```

A comment is recognized as a string contained by /* and */. A string constant is recognized as a string delimited by quotes and contained by blanks or special characters other than quotes. The constant may contain double quotes. An identifier is recognized as a valid identifier delimited by blanks or special characters other than quotes.

A compile time statement is recognized by the occurrence of

```
;    %
THEN %
ELSE %
```

In the latter two cases, the program text preceding the THEN or ELSE must be, up to this point, a valid IF statement (compile-time or otherwise). However, these requirements may have been generated by prior macro replacement.

When the scan reaches the %, the text following the % must constitute a syntactically correct compile-time statement. This text may be prior generated text, source text, or a combination of the two. Note that a

syntactically correct compile-time IF statement, at this point, imposes no restrictions on the text constituting the units following THEN and ELSE, unless they are themselves single compile-time statements.

Source and generated text passed over by the scan are added to the program text. When the scan locates an identifier, other than in a compile-time statement, it is compared with the list of macro identifiers, sequentially from the top. If no match is found, the scan passes over the identifier. If a match is found, the following action occurs:

1. If the identifier is a macro variable name, the current value of the variable is taken as generated text. This generated text is then scanned according to 3. The macro variable must have been assigned a value (by the INITIAL attribute or by compile-time assignment).
2. If the identifier is a macro function name, the function is invoked. The argument list, if present, is scanned and any macro replacement is performed. The arguments are passed to the macro procedure. The value returned by the function constitutes generated text. This text is scanned according to 3. If the function requires an argument list, the leading parenthesis must be present in the text being scanned when the function name is recognized. On completion of macro replacemement within the argument list, the text must be a valid invocation of the function.
3. The generated text is scanned normally, except that macro variable names and function references are not replaced. On completion of the scan of a given portion of generated text, the scan returns to the source text at the point following the variable or function reference which caused the generation.

When the scan locates comments or character constants, it passes over them without any analysis.

When the scan locates a compile-time statement carrying a label, a record is made of the position of this label in the source text so that subsequent compile-time transfers of control to this label may be performed.

When the scan locates a compile-time null statement, the scan continues from the point logically following that statement.

When the scan locates a compile time assignment statement, the macro expression is evaluated, and its value associated with the macro variable in the list. This value replaces any previous value and is used for subsequent reference. The scan continues with the text following the compile-time statement.

When the scan locates a compile-time IF statement, the macro relation is evaluated. If a true value is obtained, the scan continues following the THEN. When the scan reaches the semicolon terminating the unit following the THEN, the preceding program text must constitute a syntactically correct unit. If there is an ELSE clause, this is skipped by the scan; no source text is moved to program text. However to achieve this skip, the unit following the ELSE must be partially scanned. This partial scan generates pseudo program text by skipping compile-time statements and comments, recognizing string constants, and performing macro variable and function reference replacement. This pseudo program text must constitute a valid unit because it is analyzed to identify the end of the unit. The pseudo text is discarded on recognition of the end of the unit. If this is generated text, and further generated text follows it, then the normal scan continues with this generated text. Otherwise, the normal scan continues at the point in the source text following either the macro variable or function reference which generated the END; or following the END; itself in the source text. If the macro relation returns a false value, the unit

following the THEN is skipped. This skip is achieved by a partial scan as described above. If no ELSE clause is present, the normal scan continues with the source text following the THEN clause. If an ELSE clause appears, the keyword ELSE must appear in the source text. The scan continues normally with the source text following the ELSE.

When the scan locates a compile-time GO TO statement, a transfer is made to the statement carrying the specified label. If this is a forward skip, a partial scan, as described above, is performed to locate the label. The pseudo program text resulting from this scan must be syntactically correct. If a backward skip is specified, no scanning activity takes place. The action associated with the compile-time statement transferred to is performed; and if this involves no further transfer, the normal scan continues with the source text following the statement. Compile time activity is completed when the scan reaches the end of the source text. The program text at this point constitutes the external procedure.

Processing of data may be considered to consist of three operations: acquisition of data, data manipulation, and disposition of data results. This section deals with the first and the final operations, commonly termed input/output activity.

All input/output activity is transacted with named collections of data called files. The name of a file is a file name. Files may be subdivided into smaller collections of data called records. Furthermore, records may be ordered within a file so that the data conceptually constitute a single stream upon which the record structure has been superimposed. Records in such ordered files may be collected into groups. Records are variously defined according to the medium on which a file resides and the ultimate source or destination of its contents.

The natural record or group structure of all or part of a data file may be inappropriate to some applications. For such applications, significant divisions of data may be indicated by arbitrary symbols, called segment delimiters, into segments. A record or group boundary within a segment may be disregarded entirely. Record or group boundaries may be made significant and insignificant by turns within one data file.

CHAPTER 21: OPENING AND CLOSING FILES

Before data can be transmitted between internal storage and a file, certain preparations must be made, such as checking for and ensuring the availability of the data medium, and allocation of appropiate programming support and storage areas. These preparations are called "opening" the file. Similarly, when usage of a file is completed, it must be "closed," in order to permit release of the facilities allocated for "opening," and to cause proper disposition of the file. The programmer may accomplish these two actions by writing the statements OPEN and CLOSE. If he elects to do this, many of the attributes of the file may be specified dynamically at execution of these statements. He may, however, omit either statement. In this case the file is opened during the first READ or WRITE statement which references it, and is closed at the completion of the program.

THE OPEN AND CLOSE STATEMENTS

General Forms:

OPEN file group 1 , ... , file group n , task option ;
CLOSE file group 1 , ... , file group n , task option ;

A file group may be either of the following:

file specification
(file specification 1, ... , file specification n) attribute 1....
...attribute n

A file specification is:

file name attribute 1 ... attribute n

File groups and specifications are separated by commas. File attributes are separated by blanks. The TASK option is described in Chapter 18. It specifies that the opening or closing activity be performed asynchronously. It may be omitted with the preceding comma.

The following file attributes may be given in the OPEN statement:

INPUT, OUTPUT, or INOUT

This specifies the direction of data transmission that will be permitted for the file. INOUT may be given for direct access files, stating that both INPUT and OUTPUT is permitted. If this attribute is not specified in the OPEN statement, and has not been declared for the file, then no assumption is made at the time of execution of the OPEN. If the file is subsequently READ then it will be assumed to be INPUT, and OUTPUT if it is specified in a WRITE. References to the file in PAGE and LAYOUT statements before INPUT or OUTPUT is established force no assumptions. References to the file in GROUP, SPACE, SKIP, or SEGMENT statements before INPUT or OUTPUT is established force the default assumption INPUT. It should be noted that INPUT files cannot be written on and OUTPUT files cannot be read.

TITLE (expression)

A file name may be associated with more than one set of data. The

choice of the desired set may be delayed until the OPEN statement is executed. At this point the expression in the TITLE option is evaluated, converted to a character string, and used to identify the data set. If the TITLE option is omitted, the file name is taken as the data set name.

STAY

This attribute specifies that the file be commenced at the current logical position of the associated external device, rather than at its logical beginning.

ACTIVITY (expression)

The expression is evaluated and converted to an integer specifying the relative activity of the file in implementation defined units.

The following attributes may be given in the CLOSE statement:

DISCARD

This attribute indicates that the specified data files are to be removed from the operating system and will not be required again. The external media, the buffer areas, and support routines may all be re-allocated.

KEEP

This attribute indicates that the operating system may re-allocate support programs and buffer areas for the specified data files, but that the actual data media of the file should remain available for later use.

STAY

This attribute indicates that the data media should remain logically positioned wherever it is when it is closed. Normally the data file is repositioned to its logical beginning.

The following attribute may be used on either the OPEN or CLOSE statement:

IDENT

The IDENT attribute has four forms.

```
IDENT (data list) (format list)
IDENT (output list)
IDENT entry name (argument list)
IDENT
```

Used on the OPEN for an output file, the IDENT attribute specifies a label to be placed on the external medium. For an input file it provides information for label checking.

Used on the CLOSE, it provides the same function for trailer labels.

The first form provides an explicit label record. The second form provides arguments for a system label construction or checking procedure. The third form causes the invocation of a procedure which may construct or check the label. This procedure is responsible for performing its own reading and writing. The fourth form specifies that the file has, or is to have, no label. This is also assumed if this attribute is unspecified.

Normally, data is transmitted between the external medium and storage as a record. The record may be thought of as a continuous string of characters or bits, with the string subdivided into contiguous substrings called fields. A field may be empty or contain one and only one datum, called an item. Following one or more contiguous fields, there may be a mark which defines the preceding fields to be a segment.

The number of fields in a record, the size of those fields, the nature of the datum in each field, and the segment marks, if any, is called the format of the record. The order of items is specified by a list of elements. On input, the elements are variables or pseudo variables to which are assigned the values of the corresponding items of data. On output, the elements are expressions whose values are given to the corresponding items of output data. As data is transmitted, a field pointer moves across the record in synchronism with the processing of the list elements. The positioning of the pointer is governed by format specifications given for the record. The list element and the format may be specified in the record or may be specified in a list of elements and a format specification in the program.

MODES OF DATA TRANSMISSION

There are three types of data transmission: format directed transmission, list directed transmission, and data directed transmission.

FORMAT DIRECTED TRANSMISSION

INPUT: The form of the data on the external media is defined by a format list. The program storage areas that the data is to be assigned to is specified by a data list.

OUTPUT: The data values to be transmitted are defined by a data list. The form that the data is to have on the external medium is defined by a format list.

LIST DIRECTED TRANSMISSION

INPUT: The data on the external medium is in the form of valid constants. The program storage areas to which the data is to be assigned is specified by a data list.

OUTPUT: The data values to be transmitted are specified by a data list. The form of the data on the external medium is a function of the data value (see Appendix 7).

DATA DIRECTED TRANSMISSION

INPUT: The data on the external medium is in the form of valid constants and also includes information defining the program storage areas that the data is to be assigned to.

OUTPUT: The data values to be transmitted are specified by a data list. The data on the external medium has the form of valid constants and also includes the name of the data being transmitted (see Appendix 7).

FORMAT AND LIST DIRECTED DATA LISTS

General Form:

(element 1 , ... , element n)

On input, the elements may be a scalar name, an array name, a structure name, a pseudo variable or pseudo array, or a repetitive specification of such elements.

On output, the elements may be a scalar expression, an array expression, a structure expression, or a repetitive specification involving any of these elements.

A repetitive specification has the following form:

(element 1,...,element n item = specification list)

The item may be a variable or pseudo variable. The elements are as described above for the appropriate mode of transmission. The specification list is as described in "The DO Statement..." in Chapter 13.

The repetitive specification designates that the element list is to be repeated according to the specification list. The control variable is set to the initial value, and the elements transmitted. The control variable is then modified and the elements again transmitted. This process is repeated until the specification list is exhausted and the iterations complete. Repetitive specifications may be nested to any depth. A repetitive specification involving m elements repeated n times is equivalent to n * m elements.

For example the list element

```
((A(I,J) I=1 TO 2) J=3 TO 4)
```

will provide elements of the array A in the order

A(1,3), A(2,3), A(1,4), A(2,4).

If a list element is of mode complex, the real part is transmitted first. If a list element is an array name, the elements of the array are transmitted in row major order. If a list element is a structure name, the elements of the structure are transmitted in the order specified by the structure declaration. If an input variable is assigned in a list, its new value will be used in all later references within the list.

FORMAT LISTS

General Form:

(format element 1 , ... , format element n)

The format elements may be a format item, a format item preceded by a replication factor, or a list of format elements separated by commas, enclosed in parentheses and preceded by a replication factor.

Permissible format items are listed in Appendix 6.

A replication factor is either a decimal integer constant, or an expression enclosed in parentheses. The replication factor is evaluated and converted to an integer (n) every time the following element is used. It specifies that the format element is to be used repeatedly n times. A zero replication factor specifies that the following item or

list is to be skipped and not used. A negative replication factor will be considered zero.

THE FORMAT STATEMENT

General Form:

label 1: ... label m: FORMAT format list;

The FORMAT statement specifies a format list. It must carry a statement label. The statement label may be used in a remote format specification (see Appendix 6) to specify that the list should be used at that point. The remote format specification and the FORMAT statement must be internal to the same block.

MODES OF DATA SPECIFICATION

DATA SPECIFICATION FOR FORMAT DIRECTED TRANSMISSION

General Form:

data list format list

The data items to be transmitted are specified by the data list.

Format items are subdivided into two classes, data format items and control format items (see Appendix 6). A format item involving n-fold repetition is equivalent to n sequential format items.

A repetitive specification involving m-fold repetition of p scalar data items is equivalent to m * p data items. An array or a structure in a data list is equivalent to q data items, where q is the number of scalar elements of the array or structure. The first scalar data list item is associated with the first data format item, the second scalar item, with the second data format item, etc. Suppose the format list effectively contains j data format items, and the data list effectively contains k scalar items. Then, if j < k, after j scalar data items have been transmitted the format list is reused, the (j+1)th scalar item being associated with the first data format item, etc. This reuse will be performed as many times as required. If j > k, redundant data format items are ignored.

The actions associated with the control format items encountered in the format list during transmission are performed at the appropriate point. The specified transmission is complete when the last data list item has been processed using the corresponding data format item. Subsequent format items are ignored.

Examples

The first of the following examples is a format directed input specification and the second an output specification.

```
(NAME, DATE, SALARY) (A(COLA_COLB), X(2), A(6), F(M+2,2))
('RESULT('||I||')=', A(I) I=1 TO 20) (A, F(8,3))
```

DATA SPECIFICATION FOR LIST DIRECTED TRANSMISSION

General Form:

LIST (data list) (scalar expression)

Input

The data on the medium is a list of constants. Where the list item is an array name and the data a scalar constant, the constant is assigned to the first element of the array, the following constant to the second element (row major order), etc. A character constant containing a real value is valid as a constant. Thus '2.6' is a valid constant.

Complex data items require two constants, the first a real constant; the second, imaginary. Sterling constants are not allowed in complex data.

A structure name in the data list represents a list of the contained scalar variables and arrays in the order specified in the structure description.

A scalar expression, enclosed in parentheses, may optionally follow the list directed data list. If the expression is not present, data items on the external medium must be separated by a comma or a blank. If present, the expression will be evaluated and converted, if necessary, to a character string; the resultant character string will be recognized as the separator of data items on the external medium.

Output

The values of the scalar variables in the data list are converted to a character representation of the value (as described in Appendix 7) and transmitted to the external medium.

A scalar expression, enclosed in parentheses, may optionally follow the list directed data list. If the expression is not present, a blank will be used to separate data items to be transmitted. If present, the expression will be evaluated and converted, if necessary, to a character string; the resultant character string will be used to seperate data items to be transmitted.

Examples

The first two examples are list directed input specifications and the latter are output specifications.

```
LIST(CARD.RATE, DYNAMIC_FLOW)
LIST(THICKNESS(DISTANCE)
     DISTANCE=1 TO 1000)
LIST(P,Z,M,R)
LIST(A*B/C, (X+Y)**2)  (',')
```

DATA SPECIFICATION FOR DATA DIRECTED TRANSMISSION

General Forms:

```
DATA
DATA (element_1,...,element_n)
```

The first form specifies data directed input. The second form may be used to specify either input or output.

Input

The data on the external medium is in the form of a list of assignments, separated by commas, and terminated by a semicolon. An assignment is of the form:

scalar variable name = constant

The scalar variable may be a subscripted name with decimal integer constant subscripts. DEFINED variables may not be used.

If the first form of specification (with no list) is used, the names on the external medium may be any unqualified name known at the point of transmission and declared with the attribute SYMBOL.

If the second form is used, the elements of the list may be unsubscripted scalar, array or structure names. The names on the external media must appear in the list; however they need not be in the same order and the list may include redundant names. If the list includes an array name, subscripted references to that array may appear on the external medium. The list may include qualified names; qualified names of identical form may then appear on the external medium. Subscripted qualified names may have interleaved subscripts in the data list, but not on the external medium.

Output

The second general form must be used. The list elements may be a scalar, structure or array name, or a repetitive specification involving only these elements, or further repetitive specifications.

The data specified in the list will be transmitted in the form of scalar assignments. Array names in the list are interpreted as a list of the contained subscripted elements. Thus array values as described above are not generated by data directed output.

Qualified names appearing in data directed output lists will be transmitted with the same qualification, but with subscripts following rather than interleaved. Structure names in the list are interpreted as a list of the contained scalar elements.

The READ and WRITE statements cause, as required, the transmission of data from external media to storage, or from storage to external media, respectively. Each READ or WRITE statement normally processes one record, and only one record, completely. However, the programmer may process part of one record, or several records, with one READ or WRITE statement, if he so specifies. Also, he may process a segment of a file without reference to records.

The READ and WRITE statements may also be used for internal editing and moving of strings. If the name of a string variable, or an element of a string array, is specified in place of a file name in the statements, then the READ causes transmission from the string to the data list, and the write from the data list to the string.

THE READ AND WRITE STATEMENTS

General Form:

READ option list ;
WRITE option list ;

The option list consists of one or more of the options described below. Except for the data specification and CALL options, any one option may not appear more than once. Commas are used to separate the options in a list.

FILE (filename) or FILE (filename 1, ..., filename n)

This option specifies the name of the file from which the data is to be acquired. If neither the FILE nor the STRING option is specified, the standard system input file is assumed for READ, and the standard output file for WRITE. In the WRITE a list of file names may be given, specifying simultaneous writing on more than one file. It is illegal to specify a file name in a READ or WRITE statement if that file is in the stack of current files.

STRING (name)

This option gives the name of the string either from which the data is to be acquired (READ), or to which the data is to be transmitted (WRITE). Only the data specification options may appear with the string option.

Data Specification

This option is described under "Modes of Data Specification" in Chapter 22. Only forms permitted for input may be given in the READ statement and forms permitted for output in the WRITE statement. Either a data specification or the CALL option must be always specified. This option may appear as many times as required, and all modes of transmission may be arbitrarily specified together. They may appear with the CALL option. The transmissions associated with each data specification and edit procedure are performed in the order that the options appear.

HOLD

This option causes the record pointer position to be "remembered" on completion, so that the next READ or WRITE will begin its data scan at the point where the prior operation ceased scanning. If HOLD is not specified and the record is of fixed length, the remaining part of the record is padded on output or skipped on input.

CROSS

The CROSS option can be specified using either of two forms,

```
CROSS
CROSS (expression)
```

In the second form, the value of the expression is converted to an integer before use. The normal READ or WRITE operation will operate upon a single record and produce an error condition if the data specification causes the record boundary to be crossed. The CROSS option permits data acquisition to proceed through any number of records (optionally limited by the integer value of the expression) in order to satisfy the data requirements specified. The MARGIN qualifications for the data file, if present, are still valid while in the CROSS mode (see "The LAYOUT Statement" in Chapter 25).

PRINT

This option may only be used on the READ statement. It specifies that data transmitted will, at the same time as the read, be written in the same format on the standard output file.

SEGMENT (expression)

The expression is converted, if necessary, to a character string. This string serves as a segment delimiting mark, This option permits the data input stream to be synchronized not at the record boundary, but at the mark. Upon satisfying the data requirements for the READ operation, the record is further scanned for the specified mark, such that the next READ of the file will proceed from the mark. All prior unused data is lost to the program. Should the mark be encountered while transmitting data, transmission ends. At completion of record construction for the WRITE operation (through one or possibly more records), the specified mark is added to the last record. Emission of the record is not thereby implied. Subsequent data is appended to this record until the maximum record length is met. The SEGMENT option implies both the HOLD and the CROSS options.

ITEM (scalar variable name)

This option specifies that a count of the scalar items transmitted for the READ or WRITE operation is to be kept and assigned to the scalar variable.

KEY (expression)

This option is used when direct access to a particular record is required. The expression is converted to integer, if it is of arithmetic or bit string type, or to character, if it is of type character string. The expression is evaluated whenever a new record transmission is required during execution of the data specifications for a READ or WRITE statement. The value of the expression provides a 'key' to the record. This selects the record to be read. On output, the key is appended to the data record to be transmitted. A key which is identical to an existent key causes overwriting of that record.

CALL entry name (argument list)

This option causes the invocation of the specified procedure, which may perform further action on the file using GET or PUT statements (discussed later in this chapter) or positioning statements (see Chapter 24).

TASK

This option specifies that the operations associated with the READ or WRITE statements (including invocation of a procedure specified by the CALL option) be performed asynchronously with the procedure containing the statement (see "The TASK Option" in Chapter 18).

ZERO

This option specifies that trailing blanks in numeric data input fields are to be treated as zeros.

FROM (filename)

This option specifies that the last record read from the named file is to be written on the output file. If the last record was read using a KEY, the key must not be respecified, as the old key is used.

Examples:

```
READ FILE (INVENTORY) , (ITEM.NAME,ITEM.COST)
(A(20) , F(5,2)) ;
```

The file named INVENTORY is read for one record. The first 20 characters of the record are placed into the character string variable ITEM.NAME, the next 5 are converted from fixed decimal external format to the internal form of the variable, ITEM.COST. A subsequent READ of the data file will be synchronized to the next record boundary.

```
READ FILE (TABLES) , (TABLE.POOL) (F(5)) , KEY (Q) ;
```

The file named TABLES is read for the record identified by the character string Q (the record key). This record is composed of 5-digit fixed-point integers. The record is converted to internal integer representation and each item is assigned to the array TABLE.POOL.

```
READ FILE (FILEZ) , (AB) (A(10)) , SEGMENT ('*') , ITEM (NAMETOT) ;
```

The file FILEZ is read for alphabetic data items, each 10 characters in length, which are assigned to the character string array AB. Assignment ceases when either the complete array is satisfied or the SEGMENT mark, the asterisk, is encountered (in the former case, the input data stream is subsequently synchronized to the next occurrence of the segment terminator). The number of 10 character items assigned may then be checked by inspection of the scalar variable NAMETOT.

```
WRITE FILE (INVENTORY) , (ITEM.NAME,ITEM.COST) (A(20) ,F(5,2)) ;
```

This is similar to the first example. The WRITE causes the construction of a 25 character record.

```
WRITE FILE (TABLES) , (TABLE.POOL) (F(5)) , KEY (Q) ;
```

This is similar to the second example. A record is constructed from the contents of the TABLE.POOL array, and a key, the value of the variable Q, is appended. Should the key be identical to an existent key within the data file, the WRITE causes replacement of that keyed record.

```
WRITE FILE (FILEZ), ('FINAL DATA',X,Y) (A), F(3,2), E(5,2));
```

Three data items are transmitted to FILEZ (assuming X and Y are scalar variables). The first is the character string of length 10, FINAL DATA, then the fixed-point form of the value of X, then the floating-point form of the value of Y.

```
WRITE DATA (X,Y,Z);
```

The values of the three variables X, Y and Z are transmitted to the standard output file in the data-directed format. If X is floating-point value 3.1141593, Y is fixed-point value 347, and Z is character string value bbMATH, then the output data stream would appear as

```
X = 3.1141593, Y = 347, Z = '  MATH'
```

and would be in a form suitable for data directed input.

THE GET AND PUT STATEMENTS

The GET and PUT statements have meaning in a procedure invoked by the CALL option in a READ or WRITE statement (or a procedure invoked by such a procedure, etc.).

When a procedure is invoked by the CALL option in a READ or WRITE statement, the file(s) specified in that statement become the "current" file(s). At the completion of the READ or WRITE statement, such file(s) are no longer the "current" file(s). This concept of "current" files permits the execution of GET, PUT, PAGE, LAYOUT, and various positioning statements in which no file is specified. When there is a current file immediately prior to a READ or WRITE statement, that file becomes current again at the completion of the READ or WRITE statement. Thus the current file is chosen from the top of a stack. A GET or PUT executed when there is no file in the stack is in error and will cause job termination.

THE GET STATEMENT

General Form:

```
GET options;
```

This statement causes data to be fetched from the current file, converted from external data form, and assigned to variables as specified. The options are separated by commas and are:

Data Specification

This option is identical to the forms specified in the READ statement. One or more data specifications may be given.

KEY (expression)

This option is identical to the form specified in the READ statement. The expression will be evaluated every time a new record is required. This option is necessary only on a direct access (keyed) file, and then only if the GET statement may cross record boundaries.

As data is fetched from the file, a "pointer" moves across the records as demanded by the data specifications. This pointer may be repositioned within the record (see the POSITION statement).

THE PUT STATEMENT

General Form:

PUT options;

The options are as stated for the GET statement, except that the data specifications are identical to the forms specified in the WRITE statement.

This statement causes data to be fetched from variables as specified, and moved to the record being constructed for the current file. As in the GET statement, a pointer moves across the record as it is being formed. The pointer may be repositioned by the POSITION statement. The character count of the record is dependant upon the rightmost sweep of the pointer, if the records of the file are defined as variable length. If fixed length, the size is predetermined.

CHAPTER 24: POSITIONING STATEMENTS

Positioning within and between records or segments may be accomplished with the POSITION, TAB, SKIP, SPACE, GROUP, and SEGMENT statements. The first two of these are for positioning within records only, and apply only to current files. The remainder are interrecord, intergroup, or intersegment and may apply either to the current file or to an explicitly designated file.

THE POSITION STATEMENT

General Form:

POSITION (format list) ;

The format list is as described under "Format Lists" in Chapter 22. This statement manipulates the pointer mentioned under the GET and PUT statements. When the POSITION statement is executed, the pointer is first reset to the beginning of the current record. The format list elements are then used to determine the movement of the pointer as if there were associated data list elements corresponding to the format items. Since no data list exists, all format items must have an explicit or implicit width (precision) specification.

The following format items are not allowed in the POSITION format list: GROUP, SEGMENT, SKIP, SPACE, and Remote formats.

If the POSITION statement moves the pointer across parts of an output record which have no information edited into them, the record is assumed to be initially blank.

THE REPOSITION STATEMENT

General Form:

REPOSITION;

The execution of this statement resets the pointer to the position immediately before the data item causing an error condition. The statement may appear only in ON units as described in Chapter 14.

THE TAB STATEMENT

General Forms:

TAB;
TAB (scalar expression) ;

The expression is evaluated when the statement is executed and converted to an integer n. If omitted, TAB(1) is implied. The statement causes the pointer to be aligned on the nth TAB of the record or line (see "The LAYOUT Statement" in Chapter 25). The intervening data will be skipped.

INTERRECORD POSITIONING

The following statements allow positioning between records, groups, or segments. In each of these statements, the options may be null, or may be either or both of the following, separated by commas:

FILE (file name)

This option specifies that the action is to be taken on the indicated file. If this option is not used, the action is taken on the current file.

KEY (expression)

This option specifies that the expression is to be evaluated to form a key for each access of a new record. If this option is not present, the records are accessed sequentially.

THE SKIP STATEMENT

General Forms:

```
SKIP options;
SKIP (expression) , options;
```

The value of the expression is converted to an integer (n) when the statement is executed. When used with print files, lines and pages are considered, otherwise records and groups. On input, the statement causes a skip to the nth record of the group. If the current record is greater than n it causes a skip to the nth record of the next group. On output, the statement causes the creation of sufficient empty records to cause alignment on the record as above.

If the expression is omitted, SKIP(1) is implied.

THE SPACE STATEMENT

General Forms:

```
SPACE options;
SPACE ( scalar expression ) , options;
```

Execution of the SPACE statement causes evaluation of the expression and conversion of the result to an integer, n. The value of n determines the line or record spacing. Absence of the expression implies SPACE (1).

THE GROUP STATEMENT

General Forms:

```
GROUP options;
GROUP (expression) , options;
```

The expression is converted to an integer (n) when the statement is executed. If omitted, GROUP (1) is assumed.

This statement causes the group currently being processed to be released from the program. A group is defined as the records delimited by group terminators. Input records are effectively skipped through until a group-terminator is encountered, with synchronization occurring at the next group, or, if n is the value of the expression, at the nth

subsequent group. Output records are terminated with a group delimiter and released. "Empty" records are supplied if the data file is defined to have a fixed number of records per group.

THE SEGMENT STATEMENT

General Forms:

```
SEGMENT options;
SEGMENT (expression), options;
```

The expression is evaluated and converted to an integer (n) when the statement is executed. If omitted, SEGMENT (1) is assumed. The statement causes the buffer pointer to be positioned at the point following the nth segment delimiter after the current position (see SEGMENT in Chapter 23.)

On input, sufficient records are skipped to effect this positioning, if necessary. On output, empty segments are constructed, if necessary, as in the SPACE and GROUP statements. SEGMENT positioning need not, however, cross record boundaries.

The PAGE and LAYOUT statements are provided to facilitate preparation of files for printing and to describe the format of print files so that they may be subsequently READ. The statements may, however, be used for any non-print file. Groups and records will then be considered in place of pages and lines. The statements refer explicitly, or in the case of a procedure invoked by the CALL option in a READ or WRITE statement, implicitly, to a particular file and each applies to that file until overridden by another statement of the same type. Until such statements are encountered, system standards are assumed to apply.

The execution of a PAGE or LAYOUT statement for a file destroys all options established by previously executed PAGE or LAYOUT statements for the same file.

THE PAGE STATEMENT

General Form:

PAGE option list ;

The statement specifies the pagination of a file (or files). Execution of the PAGE statement causes a skip to the start of the next page or group, dummy records or blank lines being generated on output files. The specified options are then established.

The options are separated by commas. Permitted options are listed below.

FILE (file name 1, ..., file name n)

This option specifies the files to be operated on. If the option is omitted in a procedure invoked by the CALL option in a READ or WRITE statement, the files specified in the invoking procedure are used. If the option is omitted elsewhere, the standard output file is assumed.

NUMBER (expression)

The value of the expression is converted to integer when the PAGE statement is executed. This option specifies that the pages or groups are (on input) or are to be (on output) numbered on the right of the heading, starting at the number which is the value of the expression. If the NUMBER option is not specified, no numbering will be expected on input or generated on output.

HEAD (expression)

The expression is evaluated and, if necessary, converted to type character string. This character string is used either as the page title, left adjusted, or as the first record of each group.

FOOT (expression)

The expression is evaluated and, if necessary, converted to a character string. This is used either as a line at the foot of each page (left adjusted), or as the last record of each group (left adjusted).

SIZE (expression)

The value of the expression is converted to integer when the PAGE statement is executed. The value of the expression specifies the number of lines per page, or records per group, including heading, footing and blank lines or records. If this option is unspecified, system standards apply.

SPACE (expression)

The value of the expression is converted to integer when the PAGE statement is executed. The value specifies the line or record spacing. Thus, if two is specified, one blank line or empty record will automatically be generated on output or skipped on input between each line or record explicitly specified. Absence of this option implies SPACE (1).

AT (expression 1) (expression 2)
AT (expression 1) CALL entry name (argument list)

Expression 1 is evaluated and converted to an integer n when the PAGE statement is executed. Subsequently when the nth record or line of each group or page is reached:

a) In the first form, the character string resulting from evaluating expression 2 at the time the PAGE was executed is understood to appear (on input) or will be output.
b) In the second form, the arguments are evaluated and the entry name invoked. The scope of the arguments is that of the block containing the PAGE statement. Since the arguments are evaluated at each call, the block containing the PAGE must still be active at each call.

THE LAYOUT STATEMENT

General Form:

LAYOUT option list ;

The statement specifies the horizontal layout of data on input and output. Execution of the LAYOUT statement causes the establishment of the specified options. The options are separated by commas. Permitted options are listed below:

FILE (file name 1,..... file name n)

This option specifies the files to be operated on. If the option is omitted in a procedure invoked by the CALL option in a READ or WRITE statement, the files specified in the invoking procedure are used. If the option is omitted elsewhere, the standard output file is assumed.

MARGIN (expression 1, expression 2)

The values of both expressions are converted to integer when the LAYOUT statement is executed. These values are interpreted as the positions of the left and right margins of the record and line, respectively. On input, data before the left margin or after the right margin is ignored. On output, the first data item of a record or line is aligned on the left margin, with blanks before, and data is not placed beyond the right margin (blanks inserted if fixed-length record).

TAB (expression 1, ..., expression n)

The expressions are converted to integers when the LAYOUT statement is executed. The values are used to indicate positions from the left end of the line or record. During list and data directed output, successive items are aligned on successive free tabs. During format directed transmission, alignment on a tab can be achieved by use of the TAB format item. In other cases, alignment on a tab can be achieved by using the TAB statement.

CHAPTER 26: ASYNCHRONOUS LOCATION OF DATA

Facilities to acquire, manipulate, and dispose of data in a random order are available through the SEARCH statement. This statement is useful when data manipulation from one or more data file may be performed in any order. Operations upon the files may be synchronized in various ways by means of the WAIT statement.

Use of the SEARCH statement causes a specific record to be requested from a data file. A number of search requests may be issued by the program before any requested record is located, thereby establishing a queue of data requests, any one of which may be satisfied momentarily. When a particular record is located, its processing may proceed while other records are being located to satisfy outstanding requests. Alternatively, processing of any record can be delayed until all requested records have been located.

THE SEARCH STATEMENT

General Form:

SEARCH option list ;

The options permitted in the option list are listed below. The options are separated by commas.

FILE (file name)

This specifies the file to be scanned. This information must always be provided in the SEARCH statement.

KEY (expression)

This option causes the record having the specified key to be located. (See the discussion of the KEY option under "The READ and WRITE Statements" in Chapter 23.)

CONTENT Data Specification

The data specification is of the form used for format directed transmission. This option permits unkeyed records to be found by analysis of record content. The data format indicates the areas within the record which are to be checked for comparison with the values of the items in the data list.

If the KEY option is specified also, the keyed record is first found. The search by content then commences at this point. If the KEY option is not specified the search commences at the current position in the file, advances to the end of the file, and then wraps around to the beginning, etc.

LIMIT (expression)

This option may be specified with either the CONTENT option, or both the KEY and the CONTENT option. The expression is evaluated to give an integer n. In the first case it limits the search by content to the n records following and including the record where the file is currently positioned. In the second case it limits the search by content to the n records following and including the keyed record.

CALL entry name (argument list)

This option causes the specified procedure to be invoked on finding the requested record. Within the procedure will be a READ or WRITE statement for actual transmission of the data located. The expressions in the argument list will be evaluated before the search is commenced.

Task Option

This option specifies that the SEARCH, and any procedures invoked by it, should be performed asynchronously with the procedure containing the SEARCH (see "The TASK Option" in Chapter 18).

Examples of the SEARCH Statement

```
SEARCH FILE (FILEAAA) , KEY ('JOE') , CALL PAYCHECK;
```

This example causes FILEAAA to be searched for the record keyed by the characters JOE, location of which will cause the PAYCHECK procedure to be invoked.

```
SEARCH FILE (FILEAAA) , CONTENT ('JOE') (X (20),
A (10)), CALL HIRE;
```

This example causes FILEAAA to be searched for a record whose first 20 characters are ignored, but whose next 10 characters are the three characters JOE and 7 blanks, the finding of which will cause invocation of the procedure HIRE. Note that the record so located might not be unique, and further SEARCH requests on this file may be used to locate the remaining records with JOE in the 21st through 23rd character positions.

```
SEARCH FILE (FILEAAA) , KEY (NAME (I)),
CALL ALPHA, TASK (A) ;

SEARCH FILE (FILEBBB) , KEY (NAME (I)),
CALL BETA, TASK (B) ;

SEARCH FILE (FILECC) , KEY (NAME (I)),
CALL GAMMA, TASK (C) ;
```

This example shows a series of SEARCH statements being used to set up three completely asynchronous program flows, data acquisition, manipulation, and possible disposition being performed for each data file in an optimal time sequence.

General Form:

SORT option list ;

The SORT statement specifies that the records on the specified file are to be sorted. The sorting is performed on fields which are specified in the statement. The sorting may be in ascending (UP) order or descending (DOWN) order.

The size of the records to be sorted must either be specified in the BLOCK attribute for the file name or be implied by a record description. The size may be variable, in which case the BLOCK attribute must be declared with a maximum size.

The options are separated by commas. The following are permitted:

FILE (file name 1, ..., file name n)
This specifies the files to be sorted. If more than one file name is specified, a merge is also performed.

RECORD (format list)

The format list is as described under "Format Lists" in Chapter 22. This option describes either the format of the whole record or merely an initial portion of the record. In the latter case the BLOCK attribute must be declared for the file name giving the actual, or maximum if variable, length of the record. The format list defines "fields" on the record. The nth format item describes the nth field. If a format item has a replication factor of m, this constitutes m fields. Of the "additional" format items specified in Appendix 6 only POSITION is permitted. This item does not constitute a field.

sort specification 1 , ... , sort specification n

Each sort specification may have one of the forms:

UP (integer 1, ..., integer n)
DOWN (integer 1, ..., integer n)

The integers are decimal integer constants and specify the fields, with respect to the record specification, to be sorted. UP specifies an ascending sort, DOWN a descending sort. The fields are taken from left to right from the sort specification list. The file is sorted on the left-most specified field first, then the next field, and so on. The sort comparisons are performed using the character collating sequence for character string fields, bit comparison for bit string fields, and algebraic comparison for arithmetic fields (see "Arithmetic Operations" in Chapter 6).

GIVING (file name)

This option is used to specify the file on which the sorted output is to be written. If omitted, the standard output file is used.

If stated and unique from the standard output file, and unique from any of the files to be sorted, the file must not be currently open. The system will open it for output, produce the sorted file, and close it. If stated, and identical to one of the files to be

sorted, then after reading the file, the system will close it, open it for output, produce the sorted file, and close it.

TASK

This option permits asynchronous execution of the SEARCH statement. See "The TASK Option" in Chapter 18.

APPENDIX 1: BUILT-IN FUNCTIONS

ARITHMETIC GENERIC FUNCTIONS

The generic functions listed in this section return a value of type arithmetic. The arguments may, unless otherwise specified, be any expressions. If nonarithmetic they will be converted to type arithmetic before the function is invoked according to the rules stated under "Scalar Expressions" in Chapter 6. Where reference is made to an argument, it should be taken to mean the converted argument when a nonarithmetic argument has been specified. The magnitude of a complex number is the positive square root of the sum of the squares of the real and imaginary parts.

Name — Arguments and Function Value.

ABS
- Arguments: One is permitted.
- Function value = absolute value of argument, i.e. positive value of real argument, positive magnitude of complex. Radix, scale, mode, and precision are those of argument.

MAX
- Arguments: Two or more are permitted.
- Function value = value of maximum argument, converted to highest characteristics of all arguments specified. The magnitude of complex numbers is used for comparison.

MIN
- Arguments: Two or more are permitted.
- Function value = value of minimum argument, converted to highest characteristics of all arguments specified. The magnitude of complex numbers is used for comparison.

MOD
- Arguments: Two are permitted, x and y.
- Function value = x-FLOOR(x/y) *y. The rules of expression evaluation give characteristics of result. If the value obtained by this formula is negative the absolute value of the modulus y is added to give a positive result.

SIGN
- Arguments: One is permitted.
- Function value = integer 1 if argument > 0; = 0 if argument = 0; = -1 if argument < 0.

FIXED
- Arguments: Three are permitted. The second and third are optional decimal integers specifying the number of digits after the decimal or binary point. If omitted the second argument assumes a value specified by each implementation, the third assumes zero.
- Function value = first argument converted to scale FIXED with precision as specified but radix and mode unchanged.

FLOAT
- Arguments: Two are permitted. The second is an optional decimal integer specifying the precision of the result. If omitted a value specified by each implementation will be assumed.
- Function value = first argument converted to scale FLOAT with precision as specified but radix and mode unchanged.

FLOOR

Arguments: One is permitted, x. If complex specified it will be converted to real.

Function value = largest integer not exceeding x. Radix, scale, mode, and precision are that of converted argument.

CEIL

Arguments: One is permitted, x. If complex specified it will be converted to real.

Function value = smallest integer not exceeded by x. Radix, scale, mode, and precision are that of converted argument.

TRUNC

Arguments: One is permitted, x. If complex specified it will be converted to real.

Function value = FLOOR(x) if $x \geq 0$, = CEIL(x) if $x < 0$. Radix, scale, mode, and precision are that of converted argument.

BINARY

Arguments: Three are permitted. The second and third are optional decimal integers specifying the binary precision of the result. If scale FIXED all three are required, if scale FLOAT the third is not required.

Function value = first argument converted to radix binary with scale and mode unchanged. If unspecified the precision is that of the first argument (see "Scalar Expressions" in Chapter 6).

DECIMAL

Arguments: Three are permitted. The second and third are optional decimal integers specifying the decimal precision of the result. If scale FIXED all three are required, if scale FLOAT the third is not required.

Function value = first argument converted to radix decimal with scale and mode unchanged. If unspecified the precision is that of the first argument (see "Scalar Expressions" in Chapter 6).

PRECISION

Arguments: Three are permitted. The second and third are decimal integers specifying the precision of the result. If scale FIXED all three are required, if scale FLOAT the third is not required.

Function value = first argument converted to specified precision. Radix, scale, and mode are unchanged.

ADD

Arguments: Four are permitted. The third and fourth are decimal integer constants specifying the precision of the result. If the scale of the result is FIXED , all four are required; if the scale is FLOAT, the fourth is not required.

Function values = the sum of the first and second arguments. Radix and scale of the result are the higher of those of the first two arguments. Precision is as specified.

MULTIPLY

Arguments: Four are permitted. The third and fourth are decimal integer constants specifying the precision of the result. If the scale of the result is FIXED, all four are required; if the scale is FLOAT, the fourth is not required.

Function value = the product of the first and second arguments. Radix and scale of the result are the higher of those of the first two arguments. Precision is as specified.

DIVIDE

Arguments: Four are permitted. The third and fourth are decimal integer constants specifying the precision of the result. If

the scale of the result is FIXED, all four are required; if the scale is FLOAT, the fourth is not required.

Function value = the result of dividing the first argument by the second. Radix and scale of the result are the higher of those of the first two arguments. Precision is as specified.

COMPLEX

Arguments: Two are permitted. The first is the real part, the second is the imaginary part.

Function value = complex number formed from the two arguments. Radix, scale, and precision of result is the higher of those of the arguments.

REAL

Arguments: One is permitted, complex value.

Function value = real part of argument. Radix, scale, and precision are unchanged.

IMAG

Arguments: One is permitted, complex value.

Function value = imaginary part of argument. Radix, scale, mode, and precision are unchanged.

CONJG

Arguments: One is permitted, complex value.

Function value = conjugate of the argument. Radix, scale, mode, and precision are unchanged.

FLOAT ARITHMETIC GENERIC FUNCTIONS

The following generic functions may have as arguments any expression. This expression will be converted to floating point before the function is invoked. The result will be of scale float with the precision and radix of the converted argument.

Function Reference	Function Value
EXP (x)	e^x
LOG (x)	log (x)
LOG 10 (x)	$\log_{10}$ (x)
LOG 2 (x)	$\log_2$ (x)
ATAND (x)	arctan (x) in degrees.
ATAN (x)	arctan (x) in radians.
TAND (x) degree argument	tan (x)
TAN (x) radian argument	tan (x)
SIND (x) degree argument	sin (x)
SIN (x) radian argument	sin (x)
COSD (x) degree argument	cos (x)
COS (x) radian argument	cos (x)
TANH (x) radian argument	tanh (x)
ERF (x)	Two divided by square root of pi, multiplied by the integral from 0 to x of EXP ($-t^2$) with respect to t.
SQRT (x)	The positive square root of x.
ERFC (x)	1 - ERF (x)
COSH (x) radian argument	cosh (x)
SINH (x) radian argument	sinh (x)
ATANH (x)	arctanh (x)
ATAN (x,y)	arctan (x/y). The arguments are converted to the highest characteristics of the pair.

STRING GENERIC FUNCTIONS

The generic functions listed in this section may be used for manipulation of strings. The arguments specified as strings may be any expression. If the argument is arithmetic it will be converted to bit string (if radix binary) or character string (if radix decimal) before the function is invoked.

Name Arguments and Function Value

BIT
Arguments: Two are permitted. The second is an optional decimal integer specifying size of result.
Function value = first argument converted to type bit string. If the size is unspecified the size of the result will be a function of the first argument characteristics (see "Scalar Expressions" in Chapter 6).

CHAR
Arguments: Two are permitted. The second is an optional decimal integer specifying size of result.
Function value = first argument converted to type character string. If the size is unspecified, the size of the result will be a function of the first argument characteristics (see "Scalar Expressions" in Chapter 6).

SUBSTR
Arguments: Three are permitted. The first is a string, the second is any expression having value i when converted to integer, the third is optionally any expression having value j when converted to integer.
Function value = substring of first argument from ith character or bit, j characters or bits long. Let first argument have length k. If j < 1 the substring is null. If i < 1 the substring length becomes j'=i+j-1 and i becomes i'=1. The values i and j, or i' and j' if i < 1, are used to determine whether (i+j-1) > k. If greater, the substring length j'=k-i+1. Finally, if i > k the substring is null.

INDEX
Arguments: Two are permitted. If both arguments are bit strings no conversion occurs, otherwise conversion to character string is performed.
Function value = decimal integer with implementation defined precision giving:
(a) The index of the first element of the first argument such that starting at this element the second argument appears as a substring.
(b) Zero, if no such index satisfying (a) exists, or if either of the arguments are of zero length.

LENGTH
Arguments: One is permitted, a string.
Function value = decimal integer of implementation defined precision giving current length of argument.

HIGH
Arguments: One is permitted, a decimal integer constant.
Function value = character string of the length specified and composed of the highest characters of the data character set.

LOW
Arguments: One is permitted, a decimal integer constant.
Function value = character string of the length specified and composed of the lowest characters of the data character set.

REPEAT

Arguments: Two are permitted. The first is a string and the second a decimal integer constant n.

Function value = string argument concatenated with itself n times.

UNSPEC

Arguments: One is permitted.

Function value = bit string which is the internal coded representation of the argument. The length is an implementation defined function of the argument characteristics.

BF$n^1n^2n^3n^4$ where each term n is either 0 or 1.

Arguments: Two are permitted, bit strings X and Y.

Function value = bit string Z where if X and Y are of different lengths the shorter is extended with zeros, and Z is of the longer length. The following table relates the jth bit of Z to the jth bits of X and Y.

Xj	Yj	Zj
0	0	n^1
0	1	n^2
1	0	n^3
1	1	n^4

S = SUM(SQRT(X))

BUILT-IN FUNCTIONS FOR MANIPULATION OF ARR YS

The following built-in functions have array expression arguments and return scalar values. In the following functions X is any array expression unless otherwise specified.

Function Reference	Function Value

SUM (X)

A scalar value equal to the sum of all the elements of X. Precision, scale, mode and radix that of argument elements. (The argument is converted to arithmetic before the function is invoked.)

PROD (X)

As above but product.

ALL (X)

The argument is converted to bit string. The result is a bit string of the length (or max length if variable) of the elements of X. The ith bit of the result is 1 if the ith bits of all the elements of X are 1. Otherwise 0.

ANY (X)

As above, ith bit of result 1 if any of the ith bits of elements of X are 1. If all 0, then result bit 0.

POLY (X,Y)

X (M : N) and Y (P : Q) are vectors. Result is

$$\sum_{J=0}^{N-M} \left[X(M+J) * \prod_{I=J}^{N-M} Y(P+I) \right]$$

If (P - Q) < (M - N) then Y(I) = Y(P) when I < (P - Q). This definition permits a scalar as second argument, when both P and Q are taken as 1. The characteristics of the result are the higher of the arguments after conversion to arithmetic.

LBOUND (X,S)

S is a scalar expression which is converted to an integer n. The function value is an integer giving the current lower bound of the nth dimension of X.

HBOUND (X,S)

As above but higher bound.

DIM (X,S)

S is as above. The function value is an integer giving the current extent of the nth dimension of X.

SCAN (A,I,'operator')

A is any array expression; I is a decimal integer constant. The third argument may be any operator in quotes. The function value is defined by the value of TEMP on exit from the following loop:

```
TEMP = A (*,....,*, LBOUND (A,I),*,....,*);
DO J = LBOUND (A,I) + 1 TO HBOUND (A,I);
TEMP = TEMP operator A (*,....,*,J, *,....,*);
END;
```

TEMP has dimensions N-1 where A has N. The bounds of TEMP are the first (I - 1) and the last (N - I) of A. TEMP has the radix, scale, mode and precision of A if arithmetic, and the length of elements of A if string.

ARRAY BUILT-IN FUNCTIONS

All the built-in functions listed under "Arithmetic Generic Functions" and "String Generic Functions" in this appendix may have array expressions as main argument. They yield an array of the same dimensions and bounds as the argument, the function being performed on each element. The rules are the same as those for the scalar functions.

CONDITION BUILT-IN FUNCTIONS

The following built-in functions (with no arguments) are available to allow investigation of interrupts arising from enabled ON conditions. They may only be referenced in ON units.

Function Reference	Function Value

ONPOINT

An integer, being the value of the I/O buffer pointer when the I/O condition arose.

ONLOC

A character string of variable length, being the name of the procedure in which the condition arose.

ONFIELD

A character string of variable length, being the contents of the field being processed when the I/O condition arose.

ONCHAR

A character string of length 1, being the character which caused an I/O conversion error.

ONCODE

An integer whose value is dependant on a detected error. Each of the following error categories has a set of contiguous code values:

I/O errors
Conversion errors
Control program errors

OTHER BUILT-IN FUNCTIONS

Function Reference	Function Value

DATE

Character string of length six of the form YYMMDD, where YY is year, MM is month DD is day.

TIME

Character string of length nine of the form HHMMSSTTT, where HH is hours, MM is minutes, SS is seconds, TTT is milliseconds.

ALLOCATION (X)

X is a CONTROLLED major structure or unsubscripted array or scalar variable not in a structure. The function value is '1'B if storage has been allocated for X and '0'B if not.

POINT (Filename)

The value of this function is a decimal integer precision (Y) where Y is implementation defined. It specifies the current position of the pointer relative to the start of the current logical record for the named file.

COMPLETE (Task Identifier)

The task identifier refers to the task most recently attached by the task containing the COMPLETE function reference, but not waited on. If the task specified has been completed, or is unknown (i.e. because it has never been attached, or has already been successfully waited on) the function value is '1'B. otherwise it is '0'B.

ROUND (Expression, Decimal Integer Constant)

The expression may be scalar array, or structure. The function value is the expression value rounded on the n'th digit after the point where n is the value of the integer. (Binary digits if radix binary, decimal if radix decimal). Nonarithmetic elements of the expression argument are unmodified. Floating point rounding is a bias removal rather than systematic rounding. Radix, scale, mode and precision of value that of argument. For FIXED point, digits after the rounded digit are set to zero.

STRING (Structure Name)

The argument must be a structure composed either of all bit strings and numeric fields of radix binary, or character strings and numeric fields of radix decimal. The function value is a string, being the concatenation of all the structure elements.

APPENDIX 2: PICTURE SPECIFICATIONS

Picture specifications are used to define the format of external numeric data and internal numeric fields, and to define the format of external and internal character string data. This appendix describes the composition of picture specifications for numeric and character data, and defines the meaning of each character which may be used within a picture specification.

NUMERIC FIELD DATA AND THE PICTURE DESCRIPTION

The format of numeric fields is described by a picture specification: A picture specification is a string of picture characters enclosed in quote marks. These characters define the position of digits within the field, digit positions involving zero suppression , sign positions, editing characters, conditional editing characters, drifting editing characters, decimal or binary point positions, the start of exponent subfields, the start of imaginary part subfields, and representation of sterling fields.

PICTURE CHARACTERISTICS

Numeric fields have the characteristics radix, scale, and mode.

REAL BINARY FIXED fields may only use the picture characters S 1 2 3 and V.

REAL BINARY FLOAT fields may only contain the picture characters S 1 2 3 V and K.

REAL DECIMAL FIXED fields may contain any picture characters except E K W 1 2 or 3.

REAL DECIMAL FLOAT fields may contain any picture characters except W 1 2 or 3.

COMPLEX fields are constructed of two REAL fields of the same radix separated by the picture character W. The scale of the field is the scale of higher priority of the real and imaginary parts.

Real numeric fields contain subfields. A FIXED field has one subfield, a FLOAT field has two subfields, the "fraction" part and the exponent.

The two subfields of a REAL FLOAT field are separated by either of the picture characters E or K. The former specifies that the character E be included in the field, the latter that no E is to appear in the field but that the exponent subfield be assumed to start after this position. Only one E or K may appear in each REAL FLOAT picture.

Decimal and binary points are represented in the picture by V. Exactly one V must appear in each REAL picture.

Decimal digit positions in picture specifications are represented by the picture character 9, binary digit positions by 1 2 or 3.

GENERAL FORM OF PICTURE SPECIFICATIONS

The general forms of picture specifications for numeric fields of various characteristics are as follows:

Real Binary Fixed

```
11 ... 1V11 ... 1

22 ... 2V22 ... 2

33 ... 3V33 ... 3
```

One sign character S may precede the first 1 of the first form of the binary fixed picture. In this position the field will contain a binary 1 if the value is negative or 0 if zero or positive. The picture involving 2's specifies a binary value in 2's complement notation, and that involving 3's a value in 1's complement notation. The sign character S may not appear in these fields. REAL binary fixed pictures may not mix the characters 1, 2 and 3.

Real Binary Float

```
11 ... 1V11 ... 1K11 ... 1

22 ... 2V22 ... 2K22 ... 2

33 ... 3V33 ... 3K33 ... 3
```

The first form specifies normal binary notation. A sign character S may precede the first 1 and immediately follow the K. The sign will represent, respectively the sign of the fraction part, and the sign of the exponent. The second two forms specify 2's and 1's complement notation for the subfields. The sign character S may not appear in these fields. Real binary float pictures may not mix the characters 1, 2, and 3.

Real Decimal Fixed

```
99 ... 9V99 ... 9
```

The V or . may be omitted. A V will then be assumed to appear following the last digit. Sign, editing, and zero suppression picture characters may be included. These are described below.

Real Decimal Float

```
99 ... 9(V or .)99 ... 9(E or K)99 ... 9
```

Sign and editing picture characters may be included. These are described below. A sign character will be taken to refer to the sign of the subfield in which the character appears (except CR or DB).

Complex

```
real picture W real picture
```

Only one W may appear in a picture. The two real pictures must specify fields of the same radix. The real pictures may not specify sterling fields.

DIGIT, POINT AND SUBFIELD DELIMITING CHARACTERS

9	Specifies that the associated field position will contain any decimal digit.
1	Specifies that the associated field position contains a binary digit. This character may not appear in a REAL picture with either 2 or 3.
2	Specifies that the associated field position contains a binary digit, being part of a binary value in 2's complement notation. This character may not appear in a REAL picture with either 1, 3 or 5.
3	Specifies that the associated field position contains a binary digit, being part of a binary value in 1's complement notation. This character may not appear in a REAL picture with either 1, 2 or 5.
V	Specifies that a decimal or binary point should be assumed to appear at this point in the associated field. It does not specify a character in the field.
K	Specifies that the exponent subfield should be assumed to follow the point in the field associated with the K. It does not specify a character in the field.
E	Specifies that the associated field position will contain the letter E, indicating the start of the exponent subfield.
W	Specifies that the imaginary field of the complex value should be assumed to follow the point associated with the W. It does not specify a character in the field.

ZERO SUPPRESSION CHARACTERS

A leading zero in a numeric subfield is a zero to the left of any of the digits 1 to 9 in the subfield. The leftmost of these latter digits and all digits in the subfield following it, are significant digits (including any zeros). Picture characters are provided for zero suppression, leading zero suppression , and the replacement of these zeros by blanks or asterisks.

Zero suppression characters are not permitted in floating point pictures.

Z	Specifies a conditional digit position. If the associated field position involves a leading zero it will be represented in the field by a blank, otherwise the digit will appear. The character may not appear to the right of 9 T I R or a drifting string in a subfield. It may not appear with * in a subfield.
*	Specifies a conditional digit position. If the associated field position involves a leading zero it will be represented in the field by *, otherwise the digits will appear. The character may not appear to the right of 9 T I R or a drifting string in a subfield. It may not appear with Z in a subfield.
Y	Specifies a conditional digit position. If the associated field position involves a zero (leading or otherwise) it will be represented in the field by a blank, if it involves a digit other than zero that digit will appear.

DRIFTING EDITING SYMBOLS

The picture characters S + - or $ may be static or drifting. The former use specifies that there is a field position where the sign, blank, or $ always appears. The latter use specifies that leading zeros are suppressed and the suppressed positions contain blanks. In addition the rightmost suppressed position associated with the picture character will contain the sign, $ or blank (see + and -).

A drifting character is specified by multiple use of that character in a picture subfield. Thus if a subfield contains one $ it is interpreted as static, if more than one, as drifting.

Drifting characters must appear in strings. A string is a sequence of the same drifting character, optionally containing interspersed characters V./, B. Picture characters / , . B following the last drifting symbol of the string are considered part of the string, however a following V terminates the string and is not part of it. A subfield may only contain one drifting string. The picture characters * and Z may not appear to the right of a drifting string in a subfield.

The field position associated with the characters / , . B appearing in a drifting string will contain

/ comma, or point blank if a significant digit has appeared to the left.

the drifting symbol, if the next position to the right contains the leftmost significant digit of the subfield.

blank, if the leftmost significant digit of the subfield is more than one position to the right.

If a drifting string contains the drifting character n times, then the string is associated with n - 1 conditional digit positions. The field position associated with the leftmost drifting character may only contain the drifting character blank, never a digit. If a drifting string is specified for a subfield the other potentially drifting

characters may only appear once to the left of the string in the subfield i.e. the other characters represent a static $ or sign.

If a drifting string contains a V, then all digit positions of the subfield following the point or V must also be part of the drifting string.

If one of the characters Z or * follows the V in a subfield, then all digit positions in the subfield following the point must be Z or *.

In the case where all digit positions after the V contain suppression characters, suppression will only occur where all the fraction digits are zero. The resulting field will then be all blanks or *'s. If there are any significant fraction digits they will all appear unsuppressed.

Drifting Characters.

$	If this character appears more than once in a subfield it is a drifting character, otherwise it is a static character. The static character specifies that the character $ be placed in the associated field position. The static character must appear either to the left of all digit positions in a subfield, or to the right of all digit positions in a subfield. See details above for the drifting use of the character.
S	Specifies the sign character + if field value ≥0, otherwise -. The character may be drifting or static. The details are identical to $.
+	Specifies the sign character + if field value ≥0, otherwise blank. The character may be drifting or static. The details are identical to $
-	Specifies the sign character - if field value <0, otherwise blank. The character may be drifting or static. The details are identical to $.

Editing Character

B	Specifies that a blank appears in the associated field position.

Conditional Editing Characters

,	If the subfields in which the comma appears involves no zero suppression that character specifies that a , will appear in the associated field position. If zero suppression is involved the comma will only appear if there is an unsuppressed digit to the left of the comma position in the subfield. If there is no such unsuppressed digit the associated field position will contain a character which depends on the first picture character preceding the comma. (a) The preceding character is *. The field position will contain a *. (b) The preceding character is a drifting $ S + or - the action taken will be identical to that which would have occurred if the picture specification had contained the drifting character in place of the comma. (c) If the preceding picture character is anything other than the above, the field position associated with the comma will contain a blank.
/	Exactly as , but /
.	Exactly as , but .

Sign Characters

It is possible for digit characters in numeric fields also to contain a sign. This is called overpunching. The picture characters which are provided to specify this ar T, I, and R:

T	specifies that the associated field position will contain a digit overpunched with the sign of the containing subfield.
I	specifies that the associated field position will contain a digit over-punched with + if the containing subfield is ≥ 0; otherwise the digit with no overpunching.
R	specifies that the associated field position will contain a digit overpunched with - if the containing subfield is < 0; otherwise the digit with no overpunching.

The two-character picture items CR and DB may be used to reflect the sign of REAL numeric fields.

CR	specifies that the associated field positions will contain the letters CR if the containing REAL field value (as opposed to containing subfield) is < 0. Otherwise the positions will contain two blanks. The characters CR may only appear to the right of all digit positions of a REAL field.
DB	As CR, but if the value is>=0

STERLING PICTURES

Picture specifications may describe numeric fields which represent sterling values in pounds, shillings, and pence. Such specifications must always commence with G indicating the start of a sterling field. Sterling fields are REAL FIXED DECIMAL and when involved in arithmetic operations will be converted to a value representing fixed point pence. The picture characters 1 2 3 K E X A W may not be used in sterling picture specifications.

The following additional characters are provided for use in sterling pictures.

8	specifies the position of a shilling digit in BSI single character representation.
7	specifies the position of a pence digit in BSI single character representation.
6	specifies the position of a pence digit in IBM single character representation.
G	specifies the start of a sterling picture. It does not specify a character in the numeric field.
H	specifies that the associated field position contains the shilling character S
D	specifies that the associated field position contains the pence character D

The general form of a sterling picture is

editing characters 1 pounds field
separator 1 shillings field separator 2
pence field editing characters 2

Editing characters 1 may be one or more of the characters $ + - S.

The pounds field may contain characters Z Y * 9 T I R, $ + - S. The first three specify zero suppression. The last four must be drifting characters specified more than once according to the rules described before. The comma may be used as a break character.

Separator 1 may be one or more of the characters / . B V.

The shillings field may be:

99 YY ZZ

Y9 8

Z9 ZY

The 9's may be replaced by T, I or R.

Zero suppression associated with the picture character Z only occurs if the whole of the field to the left of this character (including the pounds field) is zero and has also been suppressed.

Separator 2 may be one or more of the characters / . B V H.

The pence field may commence with

99	YY	ZZ
Y9		7
Z9	ZY	6

optionally followed by . or V (specifying the decimal point position) and any number of 9's, Z's, or Y's. Any of the 9's may be replaced by T

99	YY	ZZ
Y9		7
Z9	ZY	6

optionally followed by . or V (specifying the decimal point position) and any number of 9's, Z's, or Y's. Any of the 9's may be replaced by T

B . D CR DB

PICTURE SPECIFICATIONS AND PRECISION

The following picture characters represent actual or conditional digit positions (hereafter in this section called digit positions):

1 2 3 9 Z * Y Drifting $ and S and + and - T I R

The precision of a REAL FIXED numeric field is (m, n) where m is the total number of digit positions in the field, and n the number of digit positions following the . or V.

The precision of a REAL FLOAT field is (p) where p is the total number of digit positions before the E or K.

The precision of a COMPLEX field is determined as follows. Obtain the precisions of the real and imaginary parts that would be obtained from converting them to the higher scale of the two. If this scale is float the precision of the complex value is the greater of these two precisions. If the scale is fixed and the precision of the parts are (s,t) and (u, v) then the precision of the complex value is:

$$\max(s-t,\ u-v) + \max(t,v),\ \max(t,v)$$

Decimal fixed point pictures may have a scaling factor. This may be achieved by placing:

F (optionally signed integer)

at the extreme right hand of the picture subfield. This specifies that the decimal point should be assumed to be q (where q is the integer value) places to the right (or left if negative) of the position assumed in absence of the scaling factor. The precision of the numeric field is then:

(m,n-q) if $(n-q) \geq 0$ and $q > 0$
(m-q-n,0) if $(n-q) < q$ and $q > 0$
(m,n-q) if $(n-q) \leq m$ and $q < 0$
(n-q,n-q) if $(n-q) > m$ and $q < 0$

These precisions may not exceed the implementation defined limits imposed on DECIMAL FIXED values.

PICTURE SPECIFICATIONS AND SIZE

Under certain circumstances numeric fields are interpreted as strings. Then the following picture characters are taken to represent string positions, and the string size is the number of these characters which appear in the picture specification.

1 2 3 9 . E Z * Y $ S + - B , / T I R C 8 7 6 H D

REPETITION OF PICTURE CHARACTERS

Repetition of a character may be achieved by preceding it by a decimal integer constant n in parentheses. The result is n of character.

PICTURES FOR CHARACTER STRINGS

A form of picture may be given for character strings.

A	The associated field position may contain any letter
X	The associated field position may contain any character

Editing characters are provided as follows:

,	when a value is assigned to the string a , will be inserted.
.	as above, but .
/	as above, but /
B	as above, but "blank"

When a string is referenced, and that string has had editing characters as described above inserted, the value includes these editing characters.

Pictures for character strings must always contain at least one X or A. A character picture as an input format item specifies that the external data is in the form defined by the picture. A conversion error will be raised if it is not. No insertion or removal of editing characters will be performed. A character picture as an output format causes insertion of the editing characters and checking for X and A.

This appendix list all statements of the New Programming Language. The chapter number for the principal discussion of each statement is shown in parentheses.

Non-Executable Statements

Compile Time Statements	(19)
DECLARE	(16)
FORMAT	(22)
IMPLICIT	(17)
SEQUENCE	(17)

Executable Statements;

Null	(7)
Assignment	(11)
CALL	(13)
DISPLAY	(13)
END	(7)
DO	(13)
GO TO	(13)
IF Compound	(13)
ON Compound	(14)
RESTORE	(12)
RETURN	(13)
REVERT	(14)
SAVE	(12)
SIGNAL	(14)
WAIT	(13)
EXIT	(13)
ALLOCATE	(8)
FREE	(8)
ENTRY	(7)
PROCEDURE	(7)
BEGIN	(7)
STOP	(13)
OPEN	(21)
CLOSE	(21)
READ	(23)
WRITE	(23)
GET	(24)
PUT	(24)
POSITION	(24)
REPOSITION	(24)
SPACE	(24)
GROUP	(24)
SKIP	(24)
TAB	(24)
PAGE	(25)
LAYOUT	(25)
SEARCH	(26)
SORT	(27)
DELAY	(13)
FETCH	(13)
DELETE	(13)

APPENDIX 4: PERMISSIBLE KEY-WORD ABBREVIATIONS

Abbreviations are provided for certain keywords. The abbreviations themselves are keywords and will be recognized as synonomous in every respect with the full keywords. The abbreviated keywords are shown to the right of the full keywords in the following list.

PROCEDURE	PROC
DECLARE	DCL
DECIMAL	DEC
BINARY	BIN
COMPLEX	CPLX
COMPLETE	CPLT
CHARACTER	CHAR
VARYING	VAR
POSITION	POS
INITIAL	INIT
INTERNAL	INT
EXTERNAL	EXT
AUTOMATIC	AUTO
CONTROLLED	CTL
DEFINED	DEF
ABNORMAL	ABNL
PRECISION	PREC
OVERFLOW	OFL
UNDERFLOW	UFL
FIXEDOVERFLOW	FOFL
SUBSCRIPTRANGE	SUBRG
ZERODIVIDE	ZDIV
CONVERSION	CONV

The condition names which may be used in ON statements are listed and described in this appendix. For each condition name, the description consists of an identification of the event or events which "raise" the condition, such that the action specified by an ON statement with that condition name would be executed. The description also includes, if applicable, the "system action" which would be taken if the condition were raised in the absence of a programmer supplied ON statement referencing that condition. If the dominating action were that of the statement ON condition SYSTEM; the system action would be taken.

The ON conditions fall logically into five groups. The first includes those associated with data handling, expression evaluation, and computation. The second group consists of the conditions relevant to input/output activity. The third group includes ON conditions used for program checkout; there are no system actions for these conditions. The fourth group consists of the one condition CONDITION (identifier), which the programmer may employ to introduce conditions of his own naming. The fifth group includes the conditions FINISH and ERROR; these two conditions may affect the system action for other conditions.

COMPUTATIONAL CONDITIONS

OVERFLOW

This condition is caused by floating point overflow. The result will be set to the maximum value before executing the specified ON block. The system action is to comment and terminate execution.

UNDERFLOW

This condition is caused by floating point underflow. The result will be set to the smallest non-zero floating point number in the machine's representation, before executing the specified ON block. The system action is to comment, set the result to zero, and continue.

ZERODIVIDE

This condition is caused by an attempt to divide by zero. The result will be set to the maximum value before executing the specified ON block. The system action is to comment and terminate execution.

FIXEDOVERFLOW

This condition is caused by fixed point overflow as the result of a calculation in a procedure in which this condition is mentioned. The result will be set to the maximum value before executing the specified ON block. The system action is to comment and terminate execution.

SIZE

This condition is caused by assignment to a datum whose field definition is not large enough to accomodate the value being assigned. The value will be truncated and assigned. The system action is to comment and terminate execution.

INPUT/OUTPUT CONDITIONS

CONVERSION (filename)
This condition is caused by an illegal character in the input data from a specified file. The system action is to comment and terminate execution.

TRANSMIT (filename)
This condition is caused by a transmission error on the specified file. The system action is to comment and retry, and if unsuccessful after the standard number of retries, to comment and terminate execution.

LIST (filename)
This condition is caused by an unrecognisable identifier on data directed input. The system action is to comment and terminate execution.

SEARCH (filename)
This condition is caused by the inability to find the requested keyed record from the specified file. The system action is to comment and terminate execution.

FIELDOVERFLOW (filename)
This condition is caused by an output item which is too large for the output field width specified. If numeric, the leading zeros are ignored. The system action is to comment and terminate execution.

ENDRECORD (filename)
This condition is caused by an illegal attempt to read past a record delimiter from the specified file. The system action is to comment and terminate execution.

ENDGROUP (filename)
This condition is caused by an attempt to read past a group delimiter from the specified file. The system action is to comment and terminate execution.

ENDFILE (filename)
This condition is caused by an attempt to read past a file delimiter from the specified file. The system action is to comment and terminate execution.

UNDEFINEDFILE (filename)
This condition is raised when the specified file is not available. The system action is to comment and terminate execution.

PROGRAM CHECKOUT CONDITIONS

SUBSCRIPTRANGE
This condition is caused by an attempt to use a subscript outside its specified bounds in a procedure in which this condition is mentioned.

(identifier 1, ... , identifier n)
In the identifier list, each identifier is either a statement label, data (an unsubscripted variable, array, or structure name), or an entry label. The identifiers in the list are separated by commas. Each name is, in effect, enabled independantly; hence a subsequent enable of a subset of the list

overrides the subset, but not the remainder, of the list. The action for the three types of identifier is as follows:

For statement label identifiers the condition is raised prior to the execution of the statement carrying the labels.

For data identifiers the condition is raised whenever a value has, or may have, been assigned to any generation of any part of the listed data internal to a block in which the condition is mentioned. Possible assignment is assumed to occur when the data item is passed as a name argument (without dummy argument construction); when the data item is (internal), or may be (external), known to an invoked procedure; or when a data directed read is executed and the data item is either in the list or declared SYMBOL.

For entry label identifiers the condition is raised prior to the invocation of any of the entry labels by the procedure in which this condition is mentioned.

In each of the three cases the appropriate identifier will be listed on a debugging output file when the condition is raised.

THE CONDITION CONDITION

CONDITION (identifier)

The identifier is specified by the programmer, and is external. The condition is raised by the execution of a SIGNAL statement with the same condition.

THE FINISH AND ERROR CONDITIONS

FINISH

This condition is raised after normal system action by completion of the program by any means. The system action is to comment and return control to the system.

ERROR

This condition is raised after normal system action by any condition which causes the program to abort. The system action is to comment and terminate execution.

APPENDIX 6: FORMAT ITEMS

Format items are involved in format directed data transmission (see Chapter 22). There are two types of format items: data format items and control format items.

Data format items specify the form of data on an external medium. Under format directed transmission each scalar data item is associated with one format item. Control format items specify control over records and groups being read or constructed.

Data format items may describe data representation in two modes, external and internal. The former is designed to be more readable and uses character representation. The latter is a binary representation and is primarily used for compact intermediate storage. Arithmetic internal format items other than P specify 'coded' internal form.

External format items use the following quantities:

w, being the length if the field in characters used by the external representation (including signs, decimal points, and E's when present).

d, being the number of positions after the decimal or binary point.

s, being the number of significant digits (binary or decimal) to appear.

The above quantities may be specified by any expression. When the format item is used the expression is evaluated and converted to an integer. Internal format items may specify precision and length. This is given in exactly the same way as the precision attribute. The radix of the precision is that of the format item, or where this is indeterminate, that of the associated List item. If size or precision is omitted it is assumed to be that of the associated list item, converted where necessary to the characteristics specified by the format item. The type, radix, scale, mode and precision of a list item may differ from its associated format item. Where this occurs conversion will be performed.

On input the external data will be converted to the characteristics of the list item. Rules for the conversion are given under "Scalar Expressions" in Chapter 6.

DATA FORMAT ITEMS

This section describes the format items relating to arithmetic data, bit string data, character string data, and data of external form.
In the 'External Form' of arithmetic data, decimal digits are represented by the characters 0 through 9.

FIXED POINT FORMAT ITEMS

Two external forms are provided for fixed point format items:

F (w,d)

F (w)

On input, the external data is the character representation of decimal fixed point number anywhere in a field of length w. If the point is omitted from the data and d is specified, the point is assumed to be before the last d digits. If the point is omitted and d unspecified, it is taken as zero. If both d and the point are specified the latter overrides the former.

On output, the external data is a decimal fixed point number right adjusted in field of width w. If d is specified, a point is inserted before the last d digits and the value appropriately positioned. Trailing zeros are supplied if necessary. Truncation, if required, may be performed on right or left. If d is omitted, and the decimal fixed point precision of the associated list element is (p, r), then p digits are placed right adjusted in the field. No point is inserted. Leading zeros are suppressed. If p is greater than w, truncation is performed on the left. If the data is less than zero a minus sign is inserted before the first significant digit.

Four internal forms are provided for fixed point format items. The first two are used for decimal data, and the second two for binary data:

IF (Precision)

IF

IFB (Precision)

IFB

FLOATING POINT FORMAT ITEMS

Two external forms are provided for floating point format items:

E (w,d,s)

E (w,d)

On input, the data is an optionally signed character representation of a decimal floating point number anywhere within field of width w. The form is thus:

± Integer or fixed number E ± Decimal Integer

Both signs may be omitted. If the second sign is present the E may be omitted. The point may be omitted and will be assumed to be before the d'th fraction digit. If the point is specified it overrides the implied point indicated by the value of d. Both E and the exponent may be omitted. A zero exponent will be assumed.

On output the external form will be:

- or blank s-d digits.d digits E ± exponents

The exponent will be a decimal integer of P digits, where P will be defined for a particular implementation. The exponent will be adjusted so that the leading digit of the characteristic is non zero.

If the above form does not fill the field of width w, it will be right adjusted and blanks inserted on the left. If s is omitted it will be taken as equal to d. (S + P + 3) for non negative values and (S + P + 4) for negative values must not exceed w.

Four internal forms are provided for floating point format items. The first two are used for decimal data, and the second two for binary data:

IE (Precision)

IE

IEB (Precision)

IEB

COMPLEX FORMAT ITEMS

Two external forms are provided for complex format items:

C(Real Format Spec, Real Format Spec)

C(Real Format Spec)

On input the external data is the real and imaginary parts of the complex number in adjacent fields described by the two contained format specifications. If the second specification is omitted it is assumed to be the same as the first.

On output, the form of the real and imaginary parts is specified by enclosed real format items.

One internal form is provided for complex format items:

IC (Internal Real Format Item)

The internal real format item specifies the form of both real and imaginary parts.

ARITHMETIC FORMAT SPECIFICATION BY PICTURE

The external form of arithmetic data may be described by a numeric PICTURE. The format item involving such a description is

P '<u>picture specification</u>'

For details associated with the picture specification see Appendix 2. The internal form of numeric field representation of arithmetic data is equivalent to the external form. The P format item may describe data to be transmitted without editing from the external medium to an internal form. Subsequent arithmetic operations on such data may, however, require editing.

BIT STRING FORMAT ITEMS

Two external forms are provided for bit string format items:

A (W)

A

A format specification describes the external representation of a bit string using characters 0 and 1. If w is omitted, it is taken to be the length of the associated list element.

On input, the external data is a character representation of bit string anywhere within the field of width w.

On output the character representation of bit string is left adjusted in field of width w. Truncation, if necessary is performed on the right. Blanks are used for padding.

Two internal forms are provided for bit string format items:

B (Length)

B

Length is the length of the string in bits. If omitted it is taken as the current length of the associated bit string list element. The external representation is the coded form for a bit string. If S bits are encoded in one character, the width of the external field is:

TRUNC ((Length-1) /S) +1.

On input the coded string is interpreted as a bit string and truncated if necessary to the specification length.

On output the string is extended with zeros to length TRUNC (SIZE-1)-*W+S and the external form is this string interpreted as a character string.

CHARACTER STRING FORMAT ITEMS

Three external forms are provided for character string format items:

A (W)

A

P (Character String Picture Specification)

The external representation is a string of w characters. On output, truncation, if necessary, is performed on the right. If the associated list element is too short it is extended on the right with blanks. If wis omitted it is taken as the current length of the associated list element. If the picture form is used w is implied. Checking and editing will be performed.

The above format items may be used to describe data to be transmitted in internal form.

GENERAL FORMAT SPECIFICATION

A "General" format item may be used to specify an external form of data.

Three external forms are provided for general format specification:

G (w, d, s)

G (w, d)

G (w)

The type of the external character representation of the data is assumed to be that of the associated list element. Coded bit string external representation may not be described by a general format item.

In the case of strings the effect of the general format item is identical to A (w),d and s, if specified, are ignored.

On input, in the case of type arithmetic, the scale of the external character representation is deduced. The effect of the general format item is then identical to F (w), F (w, d) for fixed point numbers and E (w, d, s) for floating point numbers.

On output, in the case of type arithmetic, the data is analysed in the light of the specified field width w.

If the data may be represented without loss of accuracy as a fixed point number the external form is that specified by F (w), or F (w, d) if d is specified. If the data cannot be suitably represented by an F format item, it is necessary that d be specified in the general format item.

The effect will then be identical to

E (w,d) or E (w,d,s)

if s is specified.

A format item of the form

IG

specifies that the format of the data on the external media is to be identical to its internal form.

SPACING FORMAT ITEMS

For external data, the following format item may be used:

X (w)

On input, the format item specifies that the next w characters of the external data are to be ignored.

On output, the format specifies that w characters of blanks are to be inserted into the external data.

FURTHER CONTROL FORMAT ITEMS

The following may also be used as format items. Their effect is identical to the statements of the same name, described in Chapter 24.

SPACE (expression)

SPACE

SKIP (expression)

SKIP

GROUP (expression)

GROUP

POSITION (format list)

TAB

TAB (expression)

Only the POSITION item of the above, may be used in FORMAT lists intended for internal string editing.

REMOTE FORMAT SPECIFICATION

If it is desired to locate format items remotely from a format list, the following form may be used:

R (Statement Label Designator)

The above format item specifies that the statement label is attached to a FORMAT statement (see "The FORMAT Statement" in Chapter 22). The statement includes a list of format items which should be taken to replace the remote format item. The remote format specification and the FORMAT statement must be internal to the same procedure.

APPENDIX 7: LIST AND DATA DIRECTED OUTPUT

LIST DIRECTED OUTPUT

Data may be output under list direction without specific format instruction. The field length on the external medium is a function of the precision or length of the data and the value of the data.

CODED ARITHMETIC DATA

The external form will be a valid decimal constant.

Coded Real Fixed Decimal (P,Q) Data

The values s,t,u are involved in the field width leading zero suppression will be performed on the first (p-q-1) digits of the value. Let this cause the suppression of s zeros. If q is non zero an explicit decimal point will appear on the external medium; t is then 1. If q is zero no point will appear and t is zero. q digits will always appear after the decimal point.

If the data value is less than zero a minus sign will be placed before the first digit in the field; u is then 1. Otherwise no sign will appear, and u is zero. The field width will be (p-s+t+u).

Coded Real Fixed Binary (R,S) Data

The data will be converted to FIXED DECIMAL and output as above.

Coded Real Float Decimal (P)

The data is converted according to rules for F(W,D) where, if P is the declared precision of the item, W = P+2 and D=P-3. If this conversion causes either a digit overflow into the sign position or a significant zero digit in the position immediately to the right of the decimal point, then the data item is converted according to the rules for E(W,D,S) where,if P is the declared precision of the item, W=P+6,D=P-3, and S=P. Otherwiswe, field of four blank characters is appended to the right, such that the total field width is w+4; the effect is similar to that of the pair of format items F(W,D), X(4).

Coded Real Binary Float (R)

The data will be converted to DECIMAL FLOAT and output as above.

Coded Complex Data

The external representation is the same as two immediately adjacent REAL fields, being the real and imaginary parts of the data. However, a sign will always precede the imaginary part; it will be + if imaginary part 0, - otherwise. The imaginary part will be immediately followed by the letter I.

The field width will be the sum of the widths of the subfields for the real and imaginary parts (as described above) + 2 if imaginary part 0, or + 1 otherwise.

NUMERIC FIELD DATA

Numeric Fields of Radix Decimal

The format and field length of the external form will be that specified by the picture.

Numeric Fields of Radix Binary

The format and field length of the external form will be that specified by the picture. The binary digits 0 and 1 will be represented by the characters 0 and 1.

CHARACTER STRING DATA

The contents of the character string are output. No enclosing quotes are supplied. Contained quotes are unmodified. The field width is p where p is the current length of the string.

BIT STRING DATA

The format of the data on the external medium is that of a bit string constant, i.e. the value will be enclosed in quotes and followed by the letter B. The field width will be (p + 3), where p is the current length of the string.

LIST DIRECTED OUTPUT FORMAT

List directed output items are tabbed, i.e. aligned in vertical columns. This tabbing will be implementation defined. System tabbing may be overridden by a TAB option on a LAYOUT statement for the appropriate file.

Each item of list directed data specification except the last will be immediately followed by the separating character. If the data is to be re-read under list direction care should be taken to avoid including character data or numeric fields in the output, and to avoid ambiguity resulting from choice of separating or terminating characters.

Data items will be output on successive free tab position. An item may span several tabs. Items will not span page lines or cards etc.

DATA DIRECTED OUTPUT

The form of each scalar data directed output item is:

name = value

The name is as specified in the data directed output list, with subscript expressions evaluated and replaced by integer constants, and interleaved subscripts moved to the right. The value is as defined for list directed output, except for the case of character data where the string is enclosed in quotes and contained quotes represented as two quote characters.

Items output under data direction are tabbed as described above for list directed output. The separating character will always be a blank.

If an array name is an item in a item in a data directed output list, it is interpreted as a list of the subscripted elements of the array in row major order. Thus if the array has n elements, n subscripted items will be output.

Data directed output is suitable for data directed input only if it includes no numeric fields of radix binary, or numeric fields of radix decimal which do not have the form of valid arithmetic constants.

For implementation of NPL for IBM System/360 Data Processing Systems, certain language features will be limited to a definite size, length, or order. Many of these features are identified in earlier sections of this publication as ones which "will be implementation defined." Others arise from practical requirements in the translation of NPL programs to machine-sensible and machine-executable form.

CHARACTER SETS

Either of two character sets will be used for writing NPL source programs. They differ as to syntactic character set, one allowing the use of 60 characters and the other the use of 48 characters. They are identical as to data character set and collating sequence.

Syntactic Character Set : 60 Characters

The 60 characters making up this set are described in "Language Character Set" in Chapter 2.

Syntactic Character Set : 48 Characters

The 48 characters making up this set are identical to those of the 60 character set, with restrictions and changes as described in following paragraphs.

The following characters are not used :

Percent	%
Colon	:
Not	¬
Or	|
And	&
Greater than	>
Less than	<
Break character	_
Semicolon	;
Number sign	#
Commercial at	@
Question mark	?

The following operators as used in the 60 character set are replaced in the 48 character set by alphabetic operators as indicated :

60 Character Set	48 Character Set
>	GT
>=	GE
¬=	NE
<=	LE
<	LT
¬	NOT
|	OR
&	AND
||	CAT

In each case, one or more blanks must immediately precede the alphabetic operator if the preceding character would otherwise be alphanumeric, and

one or more blanks must immediately follow if the following character would otherwise be alphanumeric. Thus, to indicate the comparison of the variables A6 and BQ2Y for inequality, one would write A6 NE BQ2Y, but not A6NEBQ2Y, A6 NEBQ2Y, or A6NE BQ2Y. As the equals symbol is usable, however, the comparison of these two variables for equality may be written A6=BQ2Y.

The word NOT is "reserved" in the 48 character set; that is, it must not be used as a programmer-specified identifier.

The break character, commercial at sign, and number sign are not used, and consequently may not be employed in identifiers.

The following three characters are replaced as indicated:

60 Character Set	48 Character Set
:	..
;	,.
%	//

The two periods which replace the colon must be immediately followed by a blank if the otherwise following character is itself a period.

Data Character Set

Any character which will result in a unique pattern of the eight binary digits making up a byte of IBM System/360 storage is a valid character in the data character set, and may be used in source programs to construct character string constants, comments, and SEQUENCE statement character strings.

Collating Sequence

In the execution of NPL programs, comparisons of character data will observe the collating sequence resulting from the representations of involved characters in bytes of System/360 storage. The SEQUENCE statement may appear in source programs, but will be ignored.

For "letter ranges" in the IMPLICIT statement, the sequencing is, low to high, currency symbol, number sign, commercial at sign, and the twenty-six letters A to Z. The number sign and the commercial at sign are not used in the 48-character syntactic character set.

LENGTH OF IDENTIFIERS

Most identifiers which a programmer constructs in writing an NPL program must be composed of not more than 31 characters.

Certain identifiers, however, must be composed of not more than 7 characters. They are:

- All EXTERNAL data identifiers
- External PROCEDURE and ENTRY labels.
- File names.
- Task identifiers as used in the TASK option, the WAIT statement, and the COMPLETE built-in function.

REPRESENTATION OF DATA

The following paragraphs specify the representation of data in System/360 storage, and state the various permitted precisions and lengths of such data.

FIXED BINARY data are represented in binary fixed point form, with maximum precision of 31 binary digits. The default precision is 15,0. The minimum precision which will be assumed for binary fixed point constants, when involved in expression evaluation, is 15.

FIXED DECIMAL data are represented in packed decimal form, with maximum precision of 15 decimal digits. The default precision is 5,0. The minimum precision which will be assumed for decimal fixed point constants, when involved in expression evaluation, is 5.

FLOAT BINARY data are represented in hexadecimal floating point form (see "IBM System/360 Principles of Operation", Form A22-6821). Maximum precision is 53 binary digits. If the specified precision is equal to or less than 21, short floating point form is used. If the specified precision is 22 or more, long floating point form is used. The default precision is 21. The minimum precision which will be assumed for binary floating point constants, when involved in expression evaluation, is 21.

FLOAT DECIMAL data are represented in hexadecimal floating point form. The maximum precision is 16 decimal digits. If the specified precision is equal to or less than 6, short floating point form is used. If the specified precision is 7 or more, long floating point form is used. The default precision is 6. The minimum precision which will be assumed for decimal floating point constants, when involved in expression evaluation, is 6.

CHARACTER data are represented with one byte per character. The maximum length for CHARACTER data of specified or VARYING length is 32,767 characters. The default length is 1 character.

BIT data are represented with one binary digit per bit. The maximum length for BIT data of specified or VARYING length is 32,767 bits. The default length is 1 bit.

In evaluation of expressions involving FIXED data, the maximum field width for internal results (which is termed N under "Arithmetic Operations" in Chapter 6) is 31 for BINARY data and 15 for DECIMAL data.

The default precision for FIXED macro variables is 31,0. The default length for CHARACTER VARYING macro variables will be later specified.

ARRAY BOUNDS

Arrays are limited, for each dimension, to a lower bound of -32,768 and to an upper bound of 32,767.

IBM

International Business Machines Corporation
Data Systems Division
Development Laboratory
Poughkeepsie, New York

Printed in U.S.A. 320-0908