

Table of Contents

TYPE STATEMENTS 6-7
 INTEGER, REAL, LOGICAL Types 6-7
 CHARACTER Type Statement 6-8
 COMMON STATEMENT 6-9
 EXTERNAL STATEMENT 6-10
 INTRINSIC STATEMENT 6-11
 SAVE STATEMENT 6-11
 EQUIVALENCE STATEMENT 6-12
 Statement Restrictions 6-13
 DATA STATEMENT 7-3
 EXPRESSIONS 8-3
 ARITHMETIC EXPRESSIONS 8-3
 Integer Division 8-5
 Type Conversions and Result Types 8-5
 CHARACTER EXPRESSIONS 8-6
 RELATIONAL EXPRESSIONS 8-6
 LOGICAL EXPRESSIONS 8-8
 PRECEDENCE OF OPERATORS 8-9

Table of Contents

PAUSE 10-15
 END 10-15
 I/O SYSTEM 11-3
 I/O SYSTEM OVERVIEW 11-4
 Records 11-4
 Files 11-5
 File Properties 11-6
 File Name 11-6
 File Position 11-6
 Formatted and Unformatted Files 11-7
 Sequential/Direct Access 11-7
 Internal Files 11-8
 Special Properties of Internal Files 11-9
 Units 11-9
 CONCEPTS AND LIMITATIONS 11-10
 The FORTRAN I/O System 11-10
 Common I/O Operations 11-11
 Less Common File Operations 11-12
 Limitations of I/O System 11-14
 Direct Files with Blocked Devices 11-14
 BACKSPACE Use 11-14
 BACKSPACE Restrictions 11-14
 I/O Statement Side Effects 11-15

Table of Contents

EVALUATION OF EXPRESSIONS	8-10
ASSIGNMENT STATEMENTS	9-3
COMPUTATIONAL STATEMENTS	9-3
LABEL ASSIGNMENT STATEMENT	9-4
CONTROL STATEMENTS	10-3
UNCONDITIONAL GOTO	10-3
COMPUTED GOTO	10-4
ASSIGNED GOTO	10-5
ARITHMETIC IF	10-5
LOGICAL IF	10-6
BLOCK IF THEN ELSE	10-7
Block IF	10-9
ELSEIF	10-10
ELSE	10-11
ENDIF	10-11
DO	10-12
CONTINUE	10-14
STOP	10-14

Table of Contents

I/O STATEMENTS	11-15
Elements of I/O Statements	11-16
The Unit Specifier ('u')	11-16
The Format Specifier ('f')	11-17
The Input/Output List ('iolist')	11-17
Input Entities	11-18
Output Entities	11-18
Implied DO lists	11-19
I/O Statements	11-19
OPEN Statement	11-20
CLOSE Statement	11-22
READ Statement	11-23
WRITE Statement	11-25
BACKSPACE Statement	11-26
ENDFILE Statement	11-26
REWIND Statement	11-27
I/O Side Effects Restriction	11-27
FORMATTED I/O	12-3
FORMAT SPECIFICATIONS	12-3
FORMAT AND I/O LIST	12-5
EDIT DESCRIPTORS	12-7
Nonrepeatable Edit Descriptors	12-7
'xxxx' (Apostrophe Editing)	12-7
H (Hollerith Editing)	12-8
X (Positional Editing)	12-8

MANUAL OVERVIEW

This manual is intended as a user reference manual for the SofTech Microsystems FORTRAN 77 language system. SofTech Microsystems FORTRAN 77 is a dialect of FORTRAN which is closely related to the ANSI Standard FORTRAN 77 Subset language defined in ANSI X3.9-1978. Readers familiar with the ANSI standard will find a concise description of the differences between SofTech Microsystems FORTRAN 77 and the standard in Appendix A; in general, this manual does not presume that the reader is familiar with the standard.

SofTech Microsystems FORTRAN 77 runs on the p-machine architecture, which is available on a variety of host machines as a language system integrated into the UCSD Operating System. The reader is assumed to be somewhat familiar with the use of the Operating System and Text Editor, although the specifics of how to compile, link, and execute a FORTRAN program in the environment are covered in this manual. Refer to the Operating System manual for more details.

This manual is intended primarily as a reference manual for the FORTRAN system and contains all of the information necessary to fully utilize it. The reader is assumed to have some prior knowledge of some dialect of FORTRAN, although someone familiar with another high level language should be able to learn FORTRAN from this manual. The manual is not a tutorial in the sense that it does not teach the reader, step by step, the concepts necessary to write successively more complex programs in FORTRAN; rather, each

CHAPTER 1

INTRODUCTION

Introduction

section of the manual fully explains one part of the FORTRAN language system.

The manual is organized as follows: Chapters 1, 2, and 3 are general, and describe the manual and basic information required to successfully use FORTRAN in even a trivial way. Chapters 4, 5, and 6 describe the data types available in the language and how a program assigns a particular data type to an identifier or constant. Chapter 7 deals with the DATA statement, which is used for initialization of memory. Chapters 8, 9, 10, and 11 define the executable parts of programs and the meanings associated with the various executable constructs. I/O statements are presented in Chapter 11, and the associated FORMAT statement and formatted I/O are described in Chapter 12. The subroutine structure of a FORTRAN compilation, including parameter passing and intrinsic (system provided) functions, is the topic of Chapter 13. Finally, Chapter 14 discusses the rather sophisticated means which exist for compiling FORTRAN subroutines separately, overlaying, and linking in subroutines which are written in other languages.

NOTATIONAL CONVENTIONS

These are the notational conventions used throughout this manual:

Upper Case and Special Characters - are written as they would be in a program.

Lower Case Letters and Words - indicate generalizations which must be replaced by actual FORTRAN syntax in a program, as described in the text. The reader may assume that once a lowercase entity is defined, it retains its meaning for the entire context of discussion.

Example of Upper and Lower Case: The format which describes editing of integers is denoted 'Iw', where w is a nonzero, unsigned integer constant. Thus, in an actual statement, a program might contain I3 or I44. The format which describes editing of reals is 'Fw.d', where d is an unsigned integer constant. In an actual statement, F7.4 or F22.0 are valid. Notice that the period, as a special character, is taken literally.

Brackets - indicate optional items.

Example of Brackets: 'A[w]' indicates that either A or A12 are valid (as a means of specifying a character format).

... - is used to indicate ellipsis. That is, the optional item preceding the three dots may appear one or more times.

Example of ...: The computed GOTO statement is described by 'GOTO (s [, s] ...) [, i]' indicating that the syntactic item denoted by s may be repeated any number of times with commas separating them.

Introduction

Blanks normally have no significance in the description of FORTRAN statements. The general rules for blanks, covered in Chapter 3, govern the interpretation of blanks in all contexts.

CHAPTER 2
SOFTTECH MICROSYSTEMS
FORTRAN 77

This chapter describes how to use SofTech Microsystems FORTRAN 77. It assumes that the reader is familiar with the basic operation of the p-System. The mechanics of preparing, compiling, linking, and executing a FORTRAN program are outlined, and an explanation of the Compiler listing file is given.

A FORTRAN PROGRAM

Compiling a FORTRAN program

The SofTech Microsystems FORTRAN 77 Compiler is invoked as the Pascal Compiler would be invoked: by typing 'C' at the command level. The R(un) command, which will compile and execute a program, may also be used. If the file has already been compiled, the R(un) command will simply execute the code file. For these commands to call FORTRAN, the FORTRAN Compiler must be named SYSTEM.COMPILER. When your disk is shipped, the FORTRAN Compiler is named FORTRAN.CODE. To make it SYSTEM.COMPILER, type 'F' to enter the Filer, C(hange) SYSTEM.COMPILER to PASCAL.CODE, and C(hange) FORTRAN.CODE to SYSTEM.COMPILER. To start using Pascal again, reverse the renaming process.

Typing 'C' or 'R' at the command level causes the compiler to use the workfiles SYSTEM.WRK.TEXT and SYSTEM.WRK.CODE. If no workfile is present, the Operating System will prompt for the name of a .TEXT file to use.

The FORTRAN Compiler will prompt for a listing file. If a <return> is typed, no listing will be generated.

Once the prompts are all answered, the actual compilation begins. The progress of the compilation will be shown on the console by a successive display of dots. Each dot represents one line of source code.

Remember that anything which applies to the Pascal SYSTEM.COMPIILER will now apply to FORTRAN. For more information, refer to the Operating System manual.

Providing Runtime Support

To run any program on the p-System, some runtime support is needed. The package of routines which do this for FORTRAN is distinct from the package which does this for Pascal and is originally shipped in the file RTUNIT.CODE. When you change FORTRAN.CODE to SYSTEM.COMPIILER, you must also change SYSTEM.LIBRARY to PASCAL.LIBRARY (or some other name you will remember) and RTUNIT.CODE to SYSTEM.LIBRARY. After this is done, you may run your FORTRAN programs.

It may be that you have placed programs of your own in SYSTEM.LIBRARY. In this case, you will be familiar with the use of the Librarian. RTUNIT.CODE should be added to the

SYSTEM.LIBRARY file. The library text file facility described in Chapter 2 of the Operating System manual is also available to FORTRAN programmers.

Executing a FORTRAN program

A compiled, linked FORTRAN program is executed in the same manner as any other user program, i.e., by typing an 'X' at the command level, followed by the name of the file containing the linked program.

FORM OF INPUT PROGRAMS

All input source files read by FORTRAN must be .TEXT files. This allows the Compiler to read large blocks of text from a disk file in a single operation, increasing the compile speed significantly. The simplest way to prepare .TEXT files is to use the Screen Oriented Editor. For a more precise description of the fields in a FORTRAN 77 source statement, see Chapter 3, which explains the basic structure of a FORTRAN program.

\$INCLUDE Statement

To facilitate the manipulation of large programs, the SofTech Microsystems Compiler has extended the FORTRAN 77 standard with an \$INCLUDE Compiler directive. The format of the directive is:

\$INCLUDE file.name

with the \$ appearing in column 1 (see Chapter 3 for an explanation of Compiler directives in general). The meaning is to compile the contents of the file 'file.name' and insert the code into the current code file, before continuing with compilation of the current file. The included file may contain additional \$INCLUDE directives, up to a maximum of five levels of files (four levels of \$INCLUDE directives). It is often useful to have the description of a COMMON block kept in a single file and to include it in each subroutine that references that COMMON area, rather than making and maintaining many copies of the same source, one in each subroutine. There is no limit to the number of \$INCLUDE directives that can appear in a source file.

COMPILER LISTING

The Compiler listing, if requested, contains various information that may be useful to the FORTRAN programmer. The listing consists of the user's source code as read, along with line numbers, symbol tables, error messages, and optional cross-reference information.

The following is a sample listing:

```

FORTRAN Compiler IV.0 (0.0)
 0. 0 C
 1. 0 C --- Example Program #1234
 2. 0 C
 3. 0
 4. 0 SXREP
 5. 0
 6. 0 PROGRAM EX1234
 7. 0
 8. 0 INTEGER A(10,10)
 9. 0 CHARACTER*4 C
10. 0
11. 0 CALL INIT(A,C)
12. 6 I = 1
13. 9 200 A(I) = 1
**** Error number: 57 in line: 13
14. 20 I = I + 1
15. 26 IF (ABS(10-I) .NE. 0) GOTO 200
16. 37
17. 37 END

A INTEGER 3 8 11 13
C CHAR* 4 103 9 11
EX1234 PROGRAM 6
I INTEGER 105 12 13 13 14
14 15
IABS INTRINSIC 15
INIT SUBROUTINE 2,FWD 11

18. 0 SUBROUTINE INIT(B,D)
19. 0 INTEGER B(10,10)
20. 0 CHARACTER*4 D
21. 0
22. 0 RETURN
23. 2 END

B INTEGER 2* 18 19
D CHAR* 4 1* 18 20
INIT SUBROUTINE 2 18

EX1234 PROGRAM
INIT SUBROUTINE 2,7

24 lines. 1 errors.

```

The first line indicates which version of the Compiler was used for this compilation. In the example it is version 0.0 for p-System version IV.0. The leftmost column of numbers is the source-line number. The next column indicates the procedure-relative instruction counter that the corresponding line of source code occupies as object code. It is only meaningful for executable statements and data statements. To the right of the instruction counter is the source statement.

Errors are indicated by a row of asterisks followed by the error number and line number, as appears in the example between lines 13 and 14. In this case it is error number 57, "Too few subscripts", indicating that there are not enough subscripts in the array reference A(I).

At the end of each routine (function, subroutine, or main program), a local symbol table is printed. This table lists all identifiers that were referenced in that program unit, along with their definition. If the \$XREF Compiler directive has been given, a table of all lines containing an instance of that identifier in the current program unit is also printed. If the identifier is a variable, it is accompanied by its type and location. If the variable is a parameter, its location is followed by an asterisk, such as the variables B and D in the SUBROUTINE INIT. If the variable is in a common block, then the name of the block follows enclosed by slashes. If the identifier is not a variable, it is described appropriately. For subroutines and functions, the unit-relative procedure number is given. If it resides in a different segment, then the segment number follows. If the Compiler assumes that it will reside in the same segment, but has not yet appeared, it is listed as a forward program unit by the notation 'FWD'.

At the end of the compilation, the global symbol table is printed. It contains all global FORTRAN symbols referenced in the compilation. No cross-reference is given. The number of source lines compiled and the number of errors encountered follow. If there were any errors, then no object file is produced.

THE CODEFILE

The object codefile generated by the FORTRAN Compiler is compatible with the p-System Linker and Librarian. Indeed, it is hard to tell by examining a codefile whether it was created by the FORTRAN Compiler or the Pascal Compiler. For a description of the binary format of a codefile, see the Operating System Reference manual.

CHAPTER 3

BASIC STRUCTURE OF A FORTRAN
PROGRAM

In the most fundamental sense, a FORTRAN program is a sequence of characters which, when fed to the Compiler, are understood in various contexts as characters, identifiers, labels, constants, lines, statements, or other (possibly overlapping) syntactic substructure groupings of characters. The rules which the Compiler uses to group the character stream into certain substructures, as well as various constraints on how these substructures may be related to each other in the source program character stream, are the topic of this chapter.

CHARACTER SET

A FORTRAN source program consists of a stream of characters, originating in a .TEXT file, consisting of:

Letters - The 52 upper and lower case letters A through Z and a through z.

Digits - 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9.

Special Characters - The remaining printable characters of the ASCII character set.

The letters and digits, treated as a single group, are called the alphanumeric characters. FORTRAN interprets lower case letters as upper case letters in all contexts except in character constants and Hollerith fields. Thus, the following user-defined names are all indistinguishable to the FORTRAN Compiler:

ABCDE abcde AbCdE aBcDe

In addition to the above, actual source programs given to the FORTRAN Compiler contain certain hidden (nonprintable) control characters inserted by the Text Editor which are invisible to the user. FORTRAN uses these control characters in exactly the same way as the Text Editor and transforms them, using the rules of .TEXT files, into the FORTRAN character set.

The collating sequence for the FORTRAN character set is the ASCII sequence.

LINES

A FORTRAN source program may also be considered a sequence of lines, corresponding to the normal notion of line in the Text Editor. Only the first 72 characters in a line are treated as significant by the Compiler, with any trailing characters in a line ignored. Note that lines with fewer than 72 characters are possible and, if shorter than 72 columns, the Compiler does treat as significant the length of a line. For an illustration of this, see the paragraph entitled, "The Character Type," in Chapter 4.

COLUMNS

The characters in a given line fall into columns, with the first character being in column 1, the second in column 2, etc. The column in which a character resides is significant in FORTRAN, with columns 1 through 5 being reserved for statement labels, column 6 for continuation indicators and other column conventions, and columns 7 through 72

for actual statements.

BLANKS

The blank character, with the exceptions noted below, has no significance in a FORTRAN source program and may be used for the purpose of improving the readability of FORTRAN programs. The exceptions are:

Blanks within string constants are significant.

Blanks within Hollerith fields are significant.

Blanks on Compiler directive lines are significant.

A blank in column 6 is used in distinguishing initial lines from continuation lines.

Blanks count in the total number of characters the Compiler processes per line and per statement.

COMPILER DIRECTIVE LINES

A line is treated as a Compiler directive if the \$ character appears in column 1 of an input line. Compiler directives are used to transmit various kinds of information to the Compiler. A Compiler directive line may appear any place that a comment line can appear, although certain directives are restricted to appear in certain places. Blanks are significant on Compiler directive lines and are used to delimit keywords and filenames. The set of directives is described below:

\$INCLUDE

\$INCLUDE filename

Include textually the file 'filename' at this point in the source. Nested includes are implemented to a depth of nesting of five files. Thus, for example, a program may include various files with subprograms, each of which includes various files which describe COMMON areas (which would be a depth of nesting of three files).

\$USES

\$USES ident
 [IN filename]
 [OVERLAY]

This is similar to the USES command in the Pascal Compiler. The already compiled FORTRAN subroutines or Pascal procedures contained in the .CODE file 'filename' (or in the

file '**SYSTEM.LIBRARY' if no file name is present) become callable from the currently compiling code. This directive must appear before the initial non-comment input line. For more details, see Chapter 14.

\$XREF

\$XREF

This produces a cross-reference listing at the end of each procedure compiled.

\$EXT

\$EXT SUBROUTINE name #params
 or
\$EXT [type] FUNCTION name #params

The subroutine or function called 'name' is an Assembly Language routine. The routine has exactly '#params' reference parameters.

COMMENT LINES

A line is treated as a comment if any one of the following conditions is met:

A 'C' (or 'c') in column 1.

A '**' in column 1.

Line contains all blanks.

Comment lines do not effect the execution of the FORTRAN program in any way. Comment lines must be followed immediately by an initial line or another comment line. They must not be followed by a continuation line. Note that extra blank lines at the end of a FORTRAN program result in a compile time error, since the system interprets them as comment lines, but they are not followed by an initial line.

STATEMENTS, LINES, AND LABELS

The following paragraphs define a FORTRAN statement in terms of the input character stream. The Compiler recognizes certain groups of input characters as complete statements according to the rules specified here. The remainder of this manual will further define the specific statements and their properties. When it is necessary to refer to specific kinds of statements here, they are simply referred to by name.

Labels

A statement label is a sequence of from one to five digits. At least one digit must be nonzero. A label may be placed anywhere in columns 1 through 5 of an initial line. Blanks and leading zeros are not significant.

Initial Lines

An initial line is any line which is not a comment line or a Compiler directive line and contains a blank or a 0 in column 6. The first five columns of the line must be either all blank or contain a label. With the exception of the statement following a logical IF, FORTRAN statements all begin with an initial line.

Continuation Lines

A continuation line is any line which is not a comment line or a Compiler directive line and contains any character in column 6 other than a blank or a 0. The first five columns of a continuation line must be blanks. A continuation line is used to increase the amount of room to write a given statement. If it will not fit on a single initial line, it may be extended to include up to 9 continuation lines.

Statements

A FORTRAN statement consists of an initial line, followed by up to 9 continuation lines. The characters of the statement are the up to 660 characters found in columns 7 through 72 of these lines. The END statement must be wholly written on an initial line, and no other statement may have an initial line which appears to be an END statement.

STATEMENT ORDER

The FORTRAN language enforces a certain ordering among statements and lines which make up a FORTRAN compilation. In general, a compilation consists of some number of subprograms (possibly zero) and, at most, one main program (see sections on compilation units and subroutines). The various rules for ordering statements appear below.

Program Units

A subprogram begins with either a SUBROUTINE or a FUNCTION statement and ends with an END statement. A main program begins with a PROGRAM statement or any other, but not a SUBROUTINE or FUNCTION statement, and ends with an END statement. A subprogram or the main program is referred to as a program unit.

Statement Ordering

Within a program unit, whether a main program or a subprogram, statements must appear in an order consistent with the following rules:

A SUBROUTINE or FUNCTION statement, or PROGRAM statement, if present, must appear as the first statement of the program unit.

FORMAT statements may appear anywhere after the SUBROUTINE or FUNCTION statement, or PROGRAM statement if present.

All specification statements must precede all DATA statements, statement function statements, and executable statements.

All DATA statements must appear after the specification statements and precede all statement function statements and executable statements.

All statement function statements must precede all executable statements.

Within the specification statements, the IMPLICIT statement must precede all other specification statements.

These rules are illustrated in the following chart:

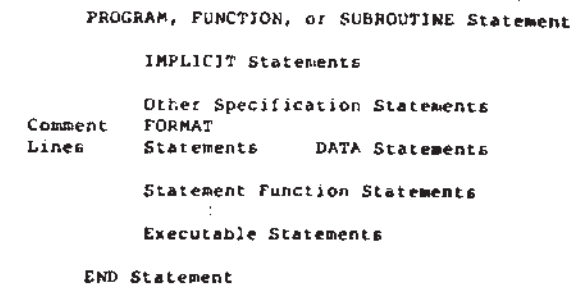


Table 3.1, Order of Statements within Program Units.

Basic Structure of a FORTRAN Program

The chart is to be interpreted as follows:

Classes of lines or statements above or below other classes must appear in the designated order.

Classes of lines or statements may be interspersed with other classes which appear across from one another.

Source Program Final Statement

When creating FORTRAN programs with the Editor, the final END statement must be entered as a complete line. That is, there must be a "return" character following the statement. Otherwise, the Compiler will not find the END statement and will issue an error message. In addition, that "return" character must be the final character in the program source file. Any further characters, even blanks, might be considered part of a subsequent subprogram by the Compiler.

CHAPTER 4 DATA TYPES

There are four basic data types in SofTech Microsystems FORTRAN 77: integer, real, logical, and character. This chapter describes the properties of each type, the range of values for each type, and the form of constants for each type.

THE INTEGER TYPE

The integer data type consists of a subset of the integers. An integer value is an exact representation of the corresponding integer. An integer variable occupies one word (two bytes) of storage and can contain any value in the range -32768 to 32767. Integer constants consist of a sequence of one or more decimal digits preceded by an optional arithmetic sign, + or -, and must be in range. A decimal point is not allowed in an integer constant. The following are examples of integer constants:

123	+123	-123	0
00000123	32767	-32768	

THE REAL TYPE

The real data type consists of a subset of the real numbers. A real value is normally an approximation of the real number desired. A real variable occupies two consecutive words (4 bytes) of storage. The range of real values is approximately:

-1.7E+38 ...	-5.8E-39	0.0
5.8E-39 ...	1.7E+38	(LSI-11)

The actual range depends upon which computer is being used. The precision is greater than 6 decimal digits.

A basic real constant consists of an optional sign followed by an integer part, a decimal point, and a fraction part. The integer and fraction parts consist of 1 or more decimal digits, and the decimal point is a period, '.'. Either the integer part or the fraction part may be omitted, but not both. Some sample basic real constants follow:

-123.456	+123.456	123.456
-123.	+123.	123.
-.456	+456	.456

An exponent part consists of the letter 'E' followed by an optionally signed integer constant. An exponent indicates that the value preceding it is to be multiplied by 10 to the value of the exponent part's integer. Some sample exponent parts are:

E12	E-12	E+12	E0
-----	------	------	----

A real constant is either a basic real constant, a basic real constant followed by an exponent part, or an integer constant followed by an exponent part. For example:

+1.000E-2	1.E-2	1E-2
+0.01	100.0E-4	.0001E+2

all represent the same real number, one one-hundredth.

THE LOGICAL TYPE

The logical data type consists of the two logical values, true and false. A logical variable occupies one word (two bytes) of storage. There are only two logical constants, .TRUE. and .FALSE., representing the two corresponding logical values. The internal representation of .FALSE. is a word of all zeros; and the representation of .TRUE. is a word of all zeros, but a one in the least significant bit. If a logical variable contains any other bit values, its logical meaning is undefined.

THE CHARACTER TYPE

The character data type consists of a sequence of ASCII characters. The length of a character value is equal to the number of characters in the sequence. The length of a particular constant or variable is fixed, and must be between 1 and 127 characters. A character variable occupies one word of storage for each two characters in the sequence, plus one word if the length is odd. Character variables are always aligned on word boundaries. The blank character is allowed in a character value and is significant.

A character constant consists of a sequence of one or more characters enclosed by a pair of apostrophes. Blank characters are allowed in character constants and count as one character each. An apostrophe within a character constant is represented by two consecutive apostrophes with no blanks in between. The length of a character constant is equal to the number of characters between the apostrophes, with doubled apostrophes

Data Types

counting as a single apostrophe character. Some sample character constants are:

```
'A' ' ' 'Help!' ''  
'A very long CHARACTER constant'
```

Note the last example, '', that represents a single apostrophe, '.

FORTRAN allows source lines with up to 72 columns. Shorter lines are not padded out to 72 columns, but left as input. When a character constant extends across a line boundary, its value is as if the portion of the continuation line beginning with column 7 is juxtapositioned immediately after the last character on the initial line. Thus, the FORTRAN source:

```
200      CH = 'ABC<cr>  
        X DEF'
```

(where the '<cr>' indicates a carriage return or the end of the source line) is equivalent to:

```
200      CH = 'ABC DEF'
```

with the single space between the C and D being the equivalent to the space in column 7 of the continuation line. Very long character constants can be represented in this manner.

CHAPTER 5 FORTRAN NAMES

A FORTRAN name, or identifier, consists of an initial alphabetic character followed by a sequence of 0 through 5 alphanumeric characters. Blanks may appear within a FORTRAN name, but have no significance. A name is used to denote a user- or system-defined variable, array, function, subroutine, etc. Any valid sequence of characters may be used for any FORTRAN name. There are no reserved names as in other languages. Sequences of alphabetic characters used as keywords are not to be confused with FORTRAN names. The Compiler recognizes keywords by their context and in no way restricts the use of user chosen names. Thus, a program can have, for example, an array named IF, READ, or GOTO, with no error indicated by the Compiler (as long as it conforms to the rules that all arrays must obey). Using such names, however, is not a recommended practice.

SCOPE OF FORTRAN NAMES

The scope of a name is the range of statements in which that name is known, or can be referenced, within a FORTRAN program. In general, the scope of a name is either global or local, although there are several exceptions. A name can only be used in accordance with a single definition within its scope. The same name, however, can have different definitions in distinct scopes.

A name with global scope may be used in more than one program unit (a subroutine, function, or the main program) and still refer to the same entity. In fact, names with global scope can only be used in a single, consistent manner within the same program. All subroutine, function subprogram, and common names, as well as the program name,

have global scope. Therefore, there cannot be a function subprogram that has the same name as a subroutine subprogram or as a common data area. Similarly, no two function subprograms in the same program can have the same name.

A name with local scope is only visible (known) within a single program unit. A name with a local scope can be used in another program unit with a different meaning, or with a similar meaning, but is in no way required to have similar meanings in a different scope. The names of variables, arrays, parameters, and statement functions all have local scope. A name with a local scope can be used in the same compilation as another item with the same name, but a global scope as long as the global name is not referenced within the program unit containing the local name. Thus, a function can be named FOO, and a local variable in another program unit can be named FOO, without error, as long as the program unit containing the variable FOO does not call the function FOO. The Compiler detects all scope errors and issues an error message when they occur, so the user need not worry about undetected scope errors causing bugs in programs.

One exception to the scoping rules is the name given to common data blocks. It is possible to refer to a globally scoped common name in the same program unit that an identical locally scoped name appears. This is allowed because common names are always enclosed in slashes, such as /NAME/, and are therefore always distinguishable from ordinary names by the Compiler.

Another exception to the scoping rules is made for parameters to statement functions. The scope of statement function parameters is limited to the single statement forming that statement function. Any other use of those names within that statement function is not allowed, and any other use outside that statement function is allowed.

UNDECLARED FORTRAN NAMES

When a user name that has not appeared before is encountered in an executable statement, the Compiler infers from the context of its use how to classify that name. If the name is used in the context of a variable, the Compiler creates an entry into the symbol table for a variable of that name. Its type is inferred from the first letter of its name. Normally, variables beginning with the letters I, J, K, L, M, or N are considered integers, while all others are considered reals. These defaults can be overridden by an IMPLICIT statement (see Chapter 6). If an undeclared name is used in the context of a function call, a symbol table entry is created for a function of that name, with its type being inferred in the same manner as that of a variable. Similarly, a subroutine entry is created for a newly encountered name that is used as the target of a CALL statement. If an entry for such a subroutine or function name exists in the global symbol table, its attributes are coordinated with those of the newly created symbol table entry. If any inconsistencies are detected, such as a previously defined subroutine name being used as a function name, an error message is issued.

In general, one is encouraged to declare all names used within a program unit, since it helps to assure that the Compiler associates the proper definition with that name. Allowing the Compiler to use a default meaning can sometimes result in logical errors that are quite difficult to locate. Indeed, most modern programming languages require the programmer to declare all names, to avoid any such potential difficulties.

CHAPTER 6
SPECIFICATION
STATEMENTS

This chapter describes the specification statements in SofTech Microsystems FORTRAN 77. Specification statements are non-executable. They are used to define the attributes of user defined variable, array, and function names. There are eight kinds of specification statements:

IMPLICIT
DIMENSION
Type Statements
COMMON
EXTERNAL
INTRINSIC
SAVE
EQUIVALENCE

Specification statements must precede all executable statements in a program unit. If present, any IMPLICIT statements must precede all other specification statements in a program unit as well. Otherwise, the specification statements may appear in any order within their own group.

IMPLICIT STATEMENT

An IMPLICIT statement is used to define the default type for user-declared names. The form of an IMPLICIT statement is:

IMPLICIT type (a [,a]...) [,type (a [,a]...)]...

The 'type' is one of INTEGER, LOGICAL, REAL, or CHARACTER[*nnn]

The 'a' is either a single letter or a range of letters. A range of letters is indicated by the first and last letters in the range separated by a minus sign. For a range, the letters must be in alphabetical order.

The 'nnn' is the size of the character type that is to be associated with that letter or letters. It must be an unsigned integer in the range 1 to 127. If *nnn is not specified, it is assumed to be *1.

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

Implicit types can be overridden or confirmed for any specific user-name by the appearance of that name in a subsequent type statement. An explicit type in a FUNCTION statement also takes priority over an IMPLICIT statement. If the type in question is a character type, the user-name's length is also overridden by a latter type definition.

The program unit can have more than one IMPLICIT statement, but all implicit statements must precede all other specification statements in that program unit. The same letter cannot be defined more than once in an IMPLICIT statement in the same program unit.

DIMENSION STATEMENT

A DIMENSION statement is used to specify that a user-name is an array. The form of a DIMENSION statement is:

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

where each 'var(dim)' is an array declarator. An array declarator is of the form:

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

'name' is the user defined name of the array.

'd' is a dimension declarator.

DIMENSION Declarators

The number of dimensions in the array is the number of dimension declarators in the array declarator. The maximum number of dimensions is three. A dimension declarator can be one of three forms:

An unsigned integer constant.

A user-name corresponding to a non-array integer formal argument.

An asterisk.

A dimension declarator specifies the upper bound of the dimension. The lower bound is always one. If a dimension declarator is an integer constant, then the array has the corresponding

number of elements in that dimension. An array has a constant size if all of its dimensions are specified by integer constants. If a dimension declarator is an integer argument, then that dimension is defined to be of a size equal to the initial value of the integer argument upon entry to the subprogram unit at execution time. In such a case, the array is called an adjustable-sized array. If the dimension declarator is an asterisk, the array is an assumed-sized array, and the upper bound of that dimension is not specified.

All adjustable- and assumed-sized arrays must also be formal arguments to the program unit in which they appear. Additionally, an assumed-size dimension declarator may only appear as the last dimension in an array declarator.

The order of array elements in memory is column-major order. That is to say, the leftmost subscript changes most rapidly in a memory-sequential reference to all array elements.

Array Element Name

The form of an array element name is:

```
arr(sub [,sub]... )
```

'arr' is the name of an array.

'sub' is a subscript expression.

A subscript expression is an integer expression used in selecting a specific element of an array. The number of subscript expressions must match the number of dimensions in the array declarator. The value of a subscript expression must be between 1 and the upper bound for the dimension it represents.

TYPE STATEMENTS

Type statements are used to specify the type of user-defined names. A type statement may confirm or override the implicit type of a name. Type statements may also specify dimension information.

A user-name for a variable, array, external function, or statement function may appear in a type statement. Such an appearance defines the type of that name for the entire program unit. Within a program unit, a name may not have its type explicitly specified by a type statement more than once. A type statement may confirm the type of an intrinsic function, but is not required. The name of a subroutine or main program cannot appear in a type statement.

INTEGER, REAL, LOGICAL Types

The form of an INTEGER, REAL, or LOGICAL type statement is:

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


'type' is one of INTEGER, REAL, or LOGICAL.

'var' is a variable name, array name, function name, or an array declarator. For a definition of an array declarator, see the section on the DIMENSION statement in this chapter.

CHARACTER Type Statement

The form of a CHARACTER type statement is:

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

'var' is a variable name, array name, or an array declarator. For a definition of an array declarator, see the section on the DIMENSION Statement.

'nnn' is the length in number of characters of a character variable or character array element. It must be an unsigned integer in the range 1 to 127.

The length nnn following the type name CHARACTER is the default length for any name not having its own length specified. If not present, the default length is assumed to be one. A length immediately following a variable or array overrides the default length for that item only. For an array, the length specifies the length of each element of that array.

COMMON STATEMENT

The COMMON statement provides a method of sharing storage between two or more program units. Such program units can share the same data without passing it as arguments. The form of the COMMON statement is:

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

'cname' is a common block name. If a 'cname' is omitted, then the blank common block is specified.

'nlist' is a comma separated list of variable names, array names, and array declarators. Formal argument names and function names cannot appear in a COMMON statement.

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

Any common block name can appear more than once in COMMON statements in the same program unit. All elements in all nlists for the same common block are sequentially allocated storage in that common storage area in the order that they appear.

All elements in a single common area must be either all of type CHARACTER or none of type character. Furthermore, if two program units reference the same named common area containing

character data, association of character variables of different lengths is not allowed. Two variables are said to be associated if they refer to the same actual storage.

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

EXTERNAL STATEMENT

An EXTERNAL statement is used to identify a user-defined name as an external subroutine or function. The form of an EXTERNAL statement is:

EXTERNAL name [,name]...

'name' is the name of an external subroutine or function.

Appearance of a name in an EXTERNAL statement declares that name to be an external procedure. Statement function names cannot appear in an EXTERNAL statement. If an intrinsic function name appears in an EXTERNAL statement, then that name becomes the name of an external procedure, and the corresponding intrinsic function can no longer be called from that program unit. A user-name can only appear once in an EXTERNAL statement.

INTRINSIC STATEMENT

An INTRINSIC statement is used to declare that a user-name is an intrinsic function. The form of an INTRINSIC statement is:

INTRINSIC name [,name]...

'name' is an intrinsic function name.

Each user-name may only appear once in an INTRINSIC statement. If a name appears in an INTRINSIC statement, it cannot appear in an EXTERNAL statement. All names used in an INTRINSIC statement must be system-defined INTRINSIC functions. For a list of these functions, see Chapter 13.

SAVE STATEMENT

A SAVE statement is used to retain the definition of a common block after the return from a procedure that defines that common block. Within a subroutine or function, a common block that has been specified in a SAVE statement does not become undefined upon exiting from the subroutine or function. The form of a SAVE statement is:

SAVE /name/ [,/name/]...

where: 'name' is the name of a common block.

NOTE: In SofTech Microsystems FORTRAN 77 all common blocks are statically allocated, so the SAVE statement is not necessary. Common blocks are never disposed on exiting a procedure. The

SAVE statement is implemented here for the sake of program portability.

EQUIVALENCE STATEMENT

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

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

where: 'nlist' is a list of at least two variable names, array names, or array element names. Argument names may not appear in an EQUIVALENCE statement. Subscripts must be integer constants and must be within the bounds of the array they index.

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

Statement Restrictions

An EQUIVALENCE statement cannot specify that the same storage location is to appear more than once, such as:

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

which forces the variable R to appear in two distinct memory locations. Furthermore, an EQUIVALENCE statement cannot specify that consecutive array elements are not stored in sequential order. For example:

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

is not allowed.

When EQUIVALENCE statements and COMMON statements are used together, several further restrictions apply. An EQUIVALENCE statement cannot cause storage in two different common blocks to become equivalenced. An EQUIVALENCE statement can extend a common block by adding storage elements following the common block, but not preceding the common block. For example:

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

is not allowed because it extends the common block by adding storage preceding the start of the block.

CHAPTER 7
DATA STATEMENT

The DATA statement is used to assign initial values to variables. A DATA statement is a non-executable statement. If present, it must appear after all specification statements and prior to any statement function statements or executable statements. The form of a DATA statement is:

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

'nlist' is a list of variable, array element, or array names.

'clist' is a list of constants or constants preceded by an integer constant repeat factor and an asterisk, such as:

5*3.14159 3*'Help' 100*0

A repeat factor followed by a constant is the equivalent of a list of all constants of that constant's value repeated a number of times equal to the repeat constant.

There must be the same number of values in each clist as there are variables or array elements in the corresponding nlist. The appearance of an array in an nlist is the equivalent to a list of all elements in that array in storage sequence order. Array elements must be indexed only by constant subscripts.

The type of each non-character element in a clist must be the same as the type of the corresponding variable or array element in the accompanying nlist. Each character element in a clist must (1) correspond to a character variable or array element in the nlist and (2) have a length that is less than or equal to the length of that variable or array

Data Statement

element. If the length of the constant is shorter, it is extended to the length of the variable by adding blank characters to the right. Note that a single character constant cannot be used to define more than one variable or even more than one array element.

Only local variables and array elements can appear in a DATA statement. Formal arguments, variables in common, and function names cannot be assigned initial values with a DATA statement.

CHAPTER 8 EXPRESSIONS

This chapter describes the four classes of expressions found in the FORTRAN language. They are:

- Arithmetic Expressions.
- Character Expressions.
- Relational Expressions.
- Logical Expressions.

ARITHMETIC EXPRESSIONS

An arithmetic expression produces a value which is either of type integer or of type real. The simplest forms of arithmetic expressions are:

- Unsigned integer or real constant.
- Integer or real variable reference.
- Integer or real array element reference.
- Integer or real function reference.

The value of a variable reference or array element reference must be defined for it to appear in an arithmetic expression. Moreover, the value of an integer variable must be defined with an arithmetic value, rather than a statement label value previously set in an ASSIGN statement.

Other arithmetic expressions are built up from the above simple forms using parentheses and these arithmetic operators:

CHAPTER 8 EXPRESSIONS

<u>Operator</u>	<u>Representing Operation</u>	<u>Precedence</u>
**	Exponentiation	Highest
/	Division	Intermediate
*	Multiplication	
-	Subtraction or Negation	Lowest
+	Addition or Identity	

Table 8.1. Arithmetic Operators.

All of the operators are binary operators, appearing between their arithmetic expression operands. The + and - may also be unary, preceding their operand. Operations of equal precedence are left-associative, except exponentiation, which is right-associative. Thus, $A / B * C$ is the same as $(A / B) * C$; and $A ** B ** C$ is the same as $A ** (B ** C)$. Arithmetic expressions may be formed in the usual algebraic sense, as in most programming languages, except that FORTRAN prohibits two operators from appearing consecutively. Thus, $A ** -B$ is prohibited, although $A ** (-B)$ is permissible. Note that unary minus is also of lowest precedence, so that $- A * B$ is interpreted as $-(A * B)$. Parentheses may be used in a program to control the associativity and the order of operator evaluation in an expression.

Integer Division

The division of two integers results in a value which is the quotient of the two values, truncated toward 0. Thus, $7 / 3$ evaluates to 2, $(-7) / 3$ evaluates to -2, $9 / 10$ evaluates to 0, and $9 / (-10)$ evaluates to 0.

Type Conversions and Result Types

Arithmetic expressions may involve operations between operands which are of different type. The general rules for determining type conversions and the result type for an arithmetic expression are:

An operation between two integers results in an expression of type integer.

An operation between two reals results in an expression of type real.

For any operator except **, an operation between a real and an integer converts the integer to type real and performs the operation, resulting in an expression of type real.

For the operator **, a real raised to an integer power is computed without conversion of the integer and results in an expression of type real. An integer raised to a real power is converted to type real, and the operation results in an expression of type real. Note that for integer I and negative integer J, $I ** J$ is the same as $1 / (I ** IABS(J))$ which is subject to the rules of integer division; so, for

example, $2^{**(-4)}$ is $1/16$ which is 0.

Unary operators result in the same result type as their operand type.

The type which results from the evaluation of an arithmetic operator is not dependent on the context in which the operation is specified. For example, evaluation of an integer plus a real results in a real, even if the value obtained is to be immediately assigned into an integer variable.

CHARACTER EXPRESSIONS

A character expression produces a value which is of type character. The forms of character expressions are:

- Character constant.
- Character variable reference.
- Character array element reference.
- Any character expression enclosed in parenthesis.

There are no operators which result in character expressions.

RELATIONAL EXPRESSIONS

Relational expressions are used to compare the values of two arithmetic expressions or two character expressions. It is not allowed to compare an arithmetic value with a character value. The result of a relational expression is of type logical.

Relational expressions may use any of these operators to compare values:

<u>Operator</u>	<u>Representing Operation</u>
.LT.	Less than
.LE.	Less than or equal to
.EQ.	Equal to
.NE.	Not equal to
.GT.	Greater than
.GE.	Greater than or equal to

Table 8.2. Relational Operators.

All of the operators are binary operators, appearing between their operands. There is no relative precedence or associativity among the relational operands, since an expression of the form $A .LT. B .NE. C$ violates the type rules for operands. Relational expressions may only appear within logical expressions.

Relational expressions with arithmetic operands may have an operand of type integer and one of type real. In this case, the integer operand is converted to type real before the relational expression is evaluated.

Relational expressions with character operands compare the position of their operands in the ASCII collating sequence. An operand is less than another if it appears earlier in the collating

sequence, etc. If operands of unequal length are compared, the shorter operand is considered as if it were blank extended to the length of the longer operand.

LOGICAL EXPRESSIONS

A logical expression produces a value which is of type logical. The simplest forms of logical expressions are:

- Logical constant.
- Logical variable reference.
- Logical array element reference.
- Logical function reference.
- Relational expression.

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

<u>Operator</u>	<u>Representing Operation</u>	<u>Precedence</u>
.NOT.	Negation	Highest
.AND.	Conjunction	
.OR.	Inclusive disjunction	Lowest

Table 8.3. Logical Operators.

The .AND. and .OR. operators are binary operators, appearing between their logical expression operands. The .NOT. operator is unary, preceding its operand. Operations of equal precedence are left associative; so, for example, A .AND. B .AND. C is equivalent to (A .AND. B) .AND. C. As an example of the precedence rules, .NOT. A .OR. B .AND. C is interpreted the same as (.NOT. A) .OR. (B .AND. C). It is not permitted to have two .NOT. operators adjacent to each other, although A .AND. .NOT. B is an example of an allowable expression with two operators being adjacent.

The meaning of the logical operators is their standard semantics, with .OR. being "nonexclusive"; that is, .TRUE. .OR.

PRECEDENCE OF OPERATORS

When arithmetic, relational, and logical operators appear in the same expression, their relative precedences are:

Expressions

<u>Operator</u>	<u>Precedence</u>
Arithmetic	Highest
Relational	
Logical	Lowest

Table 8.4. Relative Precedence
of Operator Classes.

EVALUATION OF EXPRESSIONS

Any variable, array element, or function referenced in an expression must be defined at the time of the reference. Integer variables must be defined with an arithmetic value, rather than a statement label value as set by an ASSIGN statement.

Certain arithmetic operations are prohibited if they cannot be evaluated (e.g., dividing by zero). Other prohibited operations are raising a zero valued operand to a zero or negative power and raising a negative valued operand to a power of type real.

CHAPTER 9 ASSIGNMENT STATEMENTS

An assignment statement is used to assign a value to a variable or an array element. There are two basic kinds of assignment statements: computational assignment statements and label assignment statements.

COMPUTATIONAL STATEMENTS

The form of a computational assignment statement is:

`var = expr`

'var' is a variable or array element name, and

'expr' is an expression.

Execution of a computational assignment statement evaluates the expression and assigns the resulting value to the variable or array element appearing on the left. The type of the variable or array element and the expression must be compatible. They must both be either numeric, logical, or character, in which case the assignment statement is called an arithmetic, logical, or character assignment statement.

If the type of the elements of an arithmetic assignment statement are not identical, automatic conversion of the value of the expression to the type of the variable is done. The following table gives the conversion rules:

Assignment Statements

Type of variable or <u>array element</u>	Type of expression	
	<u>integer</u>	<u>real</u>
integer	expr	INT(expr)
real	REAL(expr)	expr

Table 9.1. Type conversion for arithmetic assignment statements.

If the length of the expression does not match the size of the variable in a character assignment statement, it is adjusted so that it does. If the expression is shorter, it is padded with enough blanks on the right to make the sizes equal before the assignment takes place. If the expression is longer, characters on the right are truncated to make the sizes the same.

LABEL ASSIGNMENT STATEMENT

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

ASSIGN label TO var

'label' is a format label or statement label, and

'var' is an integer variable.

Execution of an ASSIGN statement sets the integer variable to the value of the label. The label can be either a format label or a statement label, and it must appear in the same program unit as the

Assignment Statements

ASSIGN statement. When used in an assigned GOTO statement, a variable must currently have the value of a statement label. When used as a format specifier in an I/O statement, a variable must have the value of a format statement label. The ASSIGN statement is the only way to assign the value of a label to a variable.

**CHAPTER 10
CONTROL
STATEMENTS**

Control statements are used to control the order of execution of statements in the FORTRAN language. This chapter describes the following control statements:

Unconditional GOTO.
Computed GOTO.
Assigned GOTO.
Arithmetic IF.
Logical IF.
Block IF THEN ELSE.
Block IF.
ELSEIF.
ELSE.
ENDIF.
DO.
CONTINUE.
STOP.
PAUSE.
END.

The two remaining statements which control the order of execution of statements are the CALL statement and the RETURN statement, both of which are described in Chapter 13.

UNCONDITIONAL GOTO

The syntax for an unconditional GOTO statement is:

GOTO s

where s is a statement label of an executable statement that is found in the same program unit as the GOTO statement. The effect of executing a GOTO statement is that the next statement executed is the statement labeled s. It is not legal

to jump into a DO, IF, ELSEIF, or ELSE block from outside the block (see the various sections for an explanation of the kinds of blocks).

COMPUTED GOTO

The syntax for a computed GOTO statement is:

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

where *i* is an integer expression, and each *s* is a statement label of an executable statement that is found in the same program unit as the computed GOTO statement. The same statement label may appear repeatedly in the list of labels. The effect of the computed GOTO statement can be explained as follows: Suppose that there are *n* labels in the list of labels. If $i < 1$ or $i > n$ then the computed GOTO statement acts as if it were a CONTINUE statement, otherwise, the next statement executed will be the statement labeled by the *i*th label in the list of labels. It is not allowed to jump into a DO, IF, ELSEIF, or ELSE block from outside the block (see the various sections for an explanation of the kinds of blocks).

NOTE: computed GOTOs are often used to implement a CASE construct.

ASSIGNED GOTO

The syntax for an assigned GOTO statement is:

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

where *i* is an integer variable name, and each *s* is a statement label of an executable statement that is found in the same program unit as the assigned GOTO statement. The same statement label may appear repeatedly in the list of labels. When the assigned GOTO statement is executed, *i* must have been assigned the label of an executable statement that is found in the same program unit as the assigned GOTO statement. The effect of the statement is that the next statement executed will be the statement labelled by the label last assigned to *i*. If the optional list of labels is present, a runtime error is generated if the label last assigned to *i* is not among those listed. It is not legal to jump into a DO, IF, ELSEIF, or ELSE block from outside the block (see the various sections for an explanation of the kinds of blocks).

ARITHMETIC IF

The syntax for an arithmetic IF statement is:

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

where *e* is an integer or real expression, and each of *s1*, *s2*, and *s3* are statement labels of executable statements found in the same program unit as the arithmetic IF statement. The same statement label may appear more than once among the three labels. The effect of the statement is to evaluate the expression and then select a label based on the

Control Statements

value of the expression. Label s1 is selected, if the value of e is less than 0; s2 is selected, if the value of e equals 0; and s3 is selected, if the value of e exceeds 0. The next statement executed will be the statement labeled by the selected label. It is not legal to jump into a DO, IF, ELSEIF, or ELSE block from outside the block (see the various sections for an explanation of the kinds of blocks).

LOGICAL IF

The syntax for a logical IF statement is:

```
IF (e) st
```

where e is a logical expression; and st is any executable statement, except a DO, block IF, ELSEIF, ELSE, ENDIF, END, or another logical IF statement. The statement causes the logical expression to be evaluated; and if the value of that expression is .TRUE., then the statement, st, is executed. Should the expression evaluate to .FALSE., the statement st is not executed, and the execution sequence continues as if a CONTINUE statement had been encountered.

Control Statements

BLOCK IF THEN ELSE.

The following paragraphs describe the block IF statement and the various statements associated with it. These statements are new to FORTRAN 77 and can be used to dramatically improve the readability of FORTRAN programs and to reduce the number of GOTOs of the various forms. As an overview of these sections, the following three code skeletons illustrate the basic concepts:

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

```
IF(I.LT.10)THEN
    . Some statements executed only if I.LT.10
ENDIF
```

Skeleton 2 - Block IF with a series of ELSEIF statements:

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

Skeleton 3 - Illustrates that the constructs can be nested and that an ELSE statement can follow a block IF without intervening ELSEIF statements (indentation solely to enhance readability):

```

IF(I.LT.100)THEN
  . Some statements executed only if I.LT.100
  .
  . IF(J.LT.10)THEN
  .   . Some statements executed only if I.LT.100
  .   . and J.LT.10
  .   . ENDIF
  .
  . Some statements executed only if I.LT.100
ELSEIF(I.LT.1000)THEN
  . Some statements executed only if I.GE.100 and
  . I.LT.1000
  . IF(J.LT.10)THEN
  .   . Some statements executed only if I.GE.100
  .   . and I.LT.1000 and J.LT.10
  .   . ENDIF
  .
  . Some statements executed only if I.GE.100 and
  . I.LT.1000
ENDIF

```

In order to understand, in detail, the block IF and associated statements, the concept of an IF-level is introduced. For any statement, its IF-level is

$$n1 - n2$$

where $n1$ is the number of block IF statements from the beginning of the program unit that the statement is in, up to and including that statement, and $n2$ is the number of ENDIF statements from the beginning of the program unit up to, but not including, that statement. The IF-level of every statement must be greater than or equal to 0; and the IF-level of every block IF, ELSEIF, ELSE, and ENDIF must be greater than 0. Finally, the IF-level of every END statement must be 0. The IF-level will be used to define the nesting rules for the block IF and associated statements and to

define the extent of IF blocks, ELSEIF blocks, and ELSE blocks.

Block IF

The syntax for a block IF statement is:

IF (e) THEN

where e is a logical expression. The IF block associated with this block IF statement consists of all of the executable statements, possibly none, that appear following this statement. This is up to, but not including, the next ELSEIF, ELSE, or ENDIF statement that has the same IF-level as this block IF statement. (The IF-level defines the notion of "matching" ELSEIF, ELSE, or ENDIF.) Executing the block IF statement first causes the expression to be evaluated. If it evaluates to .TRUE., and there is at least one statement in the IF block, the next statement executed is the first statement of the IF block. Following the execution of the last statement in the IF block, the next statement to be executed will be the next ENDIF statement at the same IF-level as this block IF statement. If the expression in this block IF statement evaluates to .TRUE., and the IF block has no executable statements, the next statement executed is the next ENDIF statement at the same IF level as the block IF statement. If the expression evaluates to .FALSE., the next statement executed is the next ELSEIF, ELSE, or ENDIF statement that has the same IF-level as the block IF statement. Note that transfer of control into an IF block from outside that block is not allowed.

ELSEIF

The syntax of an ELSEIF statement is:

ELSEIF (e) THEN

where e is a logical expression. The ELSEIF block associated with an ELSEIF statement consists of all of the executable statements, possibly none, that follow the ELSEIF statement. This is up to, but not including, the next ELSEIF, ELSE, or ENDIF statement that has the same IF-level as this ELSEIF statement. The execution of an ELSEIF statement begins by evaluating the expression. If its value is .TRUE., and there is at least one statement in the ELSEIF block, the next statement executed is the first statement of the ELSEIF block. Following the execution of the last statement in the ELSEIF block, the next statement to be executed will be the next ENDIF statement at the same IF-level as this ELSEIF statement. If the expression in this ELSEIF statement evaluates to .TRUE., and the ELSEIF block has no executable statements, the next statement executed is the next ENDIF statement at the same IF level as the ELSEIF statement. If the expression evaluates to .FALSE., the next statement executed is the next ELSEIF, ELSE, or ENDIF statement that has the same IF-level as the ELSEIF statement. Note that transfer of control into an ELSEIF block from outside that block is not allowed.

ELSE

The syntax of an ELSE statement is:

ELSE

The ELSE block associated with an ELSE statement consists of all of the executable statements, possibly none, that follow the ELSE statement up to, but not including, the next ENDIF statement that has the same IF-level as this ELSE statement. The "matching" ENDIF statement must appear before any intervening ELSE or ELSEIF statements of the same IF-level. Note that transfer of control into an ELSE block from outside that block is not allowed.

ENDIF

The syntax of an ENDIF statement is:

ENDIF

There is no effect of executing an ENDIF statement. An ENDIF statement is required to "match" every block IF statement in a program unit in order to specify which statements are in a particular block IF statement.

DO

The syntax of a DO statement is:

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

where s is a statement label of an executable statement. The label must follow this DO statement and be contained in the same program unit. In the DO statement, i is an integer variable; and e1, e2, and e3 are integer expressions. The statement labeled by s is called the terminal statement of the DO loop. It must not be an unconditional GOTO, assigned GOTO, arithmetic IF, block IF, ELSEIF, ELSE, ENDIF, RETURN, STOP, END, or DO statement. If the terminal statement is a logical IF, it may contain any executable statement except those not permitted inside a logical IF statement.

A DO loop is said to have a "range" beginning with the statement which follows the DO statement and ending with (and including) the terminal statement of the DO loop. If a DO statement appears in the range of another DO loop, its range must be entirely contained within the range of the enclosing DO loop, although the loops may share a terminal statement. (This is not recommended.) If a DO statement appears within an IF block, ELSEIF block, or ELSE block, the range of the associated DO loop must be entirely contained in the particular block. If a block IF statement appears within the range of a DO loop, its associated ENDIF statement must also appear within the range of that DO loop. The DO variable, i, may not be set by the program within the range of the DO

loop associated with it. It is not allowed to jump into the range of a DO loop from outside its range.

The execution of a DO statement causes the following steps to happen in order:

The expressions e1, e2, and e3 are evaluated. If e3 is not present, e3 defaults to 1 (e3 must not evaluate to 0).

The DO variable, i, is set to the value of e1.

The iteration count for the loop is computed to be $\text{MAX}0(((e2 - e1 + e3)/e3), 0)$ which may be zero (Note: unlike FORTRAN 66) if either $e1 > e2$ and $e3 > 0$, or $e1 < e2$ and $e3 < 0$.

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

Following the execution of the terminal statement of a DO loop, the following steps occur in order:

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

The iteration count is decremented by one.

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

The value of the DO variable is well-defined after execution of the loop; no matter if the DO loop exits as a result of the iteration count becoming

zero, as the result of a transfer of control out of the DO loop, or as the result of a RETURN statement.

Example of the final value of a DO variable:

```
C This program fragment prints the number 1 to 11 on
C the CONSOLE:
DO 200 I=1,10
200 WRITE(*,'(I5)')I
WRITE(*,'(I5)')1
```

CONTINUE

The syntax of a CONTINUE statement is:

CONTINUE

There is no effect associated with execution of a CONTINUE statement. The primary use for the CONTINUE statement is a convenient statement to label, particularly as the terminal statement in a DO loop.

STOP

The syntax of a STOP statement is:

STOP [n]

where n is either a character constant or a string of not more than 5 digits. The effect of executing a STOP statement is to cause the program to terminate. The argument, n, if present, is displayed on CONSOLE: upon termination.

PAUSE

The syntax of a PAUSE statement is:

PAUSE [n]

where n is either a character constant or a string of not more than 5 digits. The effect of executing a PAUSE statement is to cause the program to be suspended, pending an indication from the CONSOLE: that it is to continue. The argument, n, if present, is displayed on the CONSOLE: as part of the prompt requesting input from the CONSOLE:. If the indication from the CONSOLE: is received to continue execution of the program, execution resumes as if a CONTINUE statement had been executed.

END

The syntax of an END statement is:

END

Unlike other statements, an END statement must wholly appear on an initial line and contain no continuation lines. No other FORTRAN statement, such as the ENDIF statement, may have an initial line which appears to be an END statement. The effect of executing the END statement in a subprogram is the same as execution of a RETURN statement, and the effect in the main program is to terminate execution of the program. The END statement must appear as the last statement in every program unit.

CHAPTER 11 I/O SYSTEM

Chapters 11 and 12 of this manual describe the FORTRAN I/O System. Chapter 11 describes the basic FORTRAN I/O concepts and statements, and Chapter 12 describes the FORMAT statement. The four major Sections of these chapters are:

I/O System Overview - Provides an overview of the FORTRAN file System. Defines the basic concepts of I/O records, I/O units, and the various kinds of file access available under the System.

General Discussion of I/O System Concepts and Limitations - The definitions made in the I/O System Overview are related to how to accomplish various simple, as well as complex, tasks using the I/O System. There is a general discussion of I/O System limitations.

I/O Statements - The statements of the I/O System are presented with the exception of the FORMAT statement.

Formatted I/O and the FORMAT Statement - The FORMAT statement and formats in general are described.

NOTE: Refer to "Concepts and Limitations" in this chapter for a brief discussion of the most commonly used forms of files and I/O statements, and for a complete sample program that illustrates the most commonly used forms of I/O.

I/O SYSTEM OVERVIEW

In order to fully understand the I/O statements, it is necessary to be familiar with a variety of terms and concepts related to the structure of the FORTRAN I/O System. Most I/O tasks can be accomplished without a complete understanding of this material, thus, the reader is encouraged to skip "Concepts and Limitations" on the first reading and subsequently use the "I/O System Overview" primarily for reference.

Records

The building block of the FORTRAN file system is the Record. A Record is a sequence of characters or a sequence of values. There are three kinds of records:

- Formatted.
- Unformatted.
- Endfile.

A formatted record is a sequence of characters terminated by the character value which corresponds to the "return" key on a terminal (character value 13). Formatted records are processed on input consistent with the way that the Operating System and Text Editor process characters. Thus, reading characters from formatted records in FORTRAN is identical to reading characters in other System programs and other languages on the System. Formatted files are normally transportable across p-System implementations on different processors.

An unformatted record is a sequence of values, with no System alteration or processing; no physical representation for the end of record exists. Unformatted files generated on different processors are not generally interchangeable, since the internal representations of integers and reals differ among processors.

The System makes it appear as though an endfile record exists after the last record in a file, although no physical endfile mark ever exists.

Files

A FORTRAN file is a sequence of records. FORTRAN files are one of two kinds:

- External.
- Internal.

An external FORTRAN file is a file on a device or the device itself. An internal FORTRAN file is a character variable which serves as the source or destination of some I/O action. From this point on, both FORTRAN files and the notion of a file as known to the Operating System and the Editor will be referred to simply as files, with the context determining which meaning is intended. (The OPEN statement provides the linkage between the two notions of files; and in most cases the ambiguity disappears, since after a file has been opened, the two notions are one and the same.)

File Properties

A file which is being acted upon by a FORTRAN program has a variety of properties. These properties are described in the following paragraphs.

File Name

A file may have a name. If present, a name is a character string identical to the name by which it is known to the File System. There may be more than one name for the same file, such as SYS:A.TEXT and #4:A.TEXT.

File Position

A file has a position property which is usually set by the previous I/O operation. There is a notion of the initial point in the file, the terminal point in the file, the current record, the preceding record, and the next record of the file. It is possible to be between records in a file, in which case the next record is the successor to the previous record and there is no current record. The file position after sequential writes is at the end of file, but not beyond the endfile record. Execution of the ENDFILE statement positions the file beyond the endfile record, as does a read statement executed at the end of file (but not beyond the endfile record). Reading an endfile record may be trapped by the user using the END= option in a READ statement.

Formatted and Unformatted Files

An external file is opened as either formatted or unformatted. All internal files are formatted. Files which are formatted consist entirely of formatted records, and files which are unformatted consist entirely of unformatted records. Files which are formatted obey all the structural rules of .TEXT files, so that they are fully compatible with the System Text Editor.

Sequential/Direct Access

An external file is opened as either sequential or direct. Sequential files contain records with an order property determined by the order in which the records were written (the normal sequential order). These files must not be read or written using the REC= option which specifies a position for direct access I/O. The System will attempt to extend sequential access files, if a record is written beyond the old terminating boundary of the file; but the success of this depends on the existence of room on the physical device at the appropriate location.

Direct access files may be read or written in any order (they are random access files). Records in a direct access file are numbered sequentially, with the first record numbered one. All records in a direct access file have the same length, which is specified at the time the file is opened. Each record in the file is uniquely identified by its record number, which

was specified when the record was written. It is entirely possible to write the records out of order, including, for example, writing record 9, 5, and 11 in that order without the records in between. It is not possible to delete a record once written, but it is possible to overwrite a record with a new value. It is an error to read a record from a direct access file which has not been written; but the System will not detect this error, unless the record which is being read is beyond the last record written in the file (a non-written record before the end-of-file contains garbage). Direct access files must reside on block-structured peripheral devices such as the diskette, so that it is meaningful to specify a position in the file and reference it. The System will attempt to extend direct access files, if an attempt is made to write to a position beyond the previous terminating boundary of the file. However, the success of this depends on the existence of room on the physical device at the appropriate location.

Internal Files

Internal files provide a mechanism for using the formatting capabilities of the I/O System to convert values to and from their external character representations, within the FORTRAN internal storage structures. That is, reading a character variable converts the character values into numeric, logical, or character values; and writing into a character variable allows values to be converted into their (external) character representation.

Special Properties of Internal Files

An internal file is a character variable or character array element. The file has exactly one record, which has the same length as the character variable or character array element. Should less than the entire record be written by a WRITE statement, the remaining portion of the record is filled with blanks. The file position is always at the beginning of the file prior to I/O statement execution. Only formatted, sequential I/O is permitted with internal files; and only the I/O statements READ and WRITE may specify an internal file.

Units

A unit is a means of referring to a file. A unit specified in an I/O statement is one of:

- External unit specifier.
- Internal file specifier.

External unit specifiers are either integer expressions which evaluate to non-negative values; or the character *, which stands for the CONSOLE: device. In most cases, external unit specifier values are bound to physical devices (or files resident on those devices) by name (using the OPEN statement). Once this binding of value to System file name occurs, FORTRAN I/O statements refer to the unit number as a means of referring to the appropriate external entity. Once opened, the external unit specifier value is uniquely associated with a particular external entity until an explicit CLOSE occurs or until

the program terminates. The only exception to the above binding rules is that the unit value 0 is initially associated with the CONSOLE: device for reading and writing, and no explicit OPEN is necessary. The character * is interpreted by the System as specifying unit 0.

An internal file specifier is a character variable or character array element which directly specifies an internal file.

CONCEPTS AND LIMITATIONS

The FORTRAN I/O System

FORTRAN provides a rich combination of possible file structures. Choosing from among these many structures may at first seem somewhat confusing. However, two kinds of files will suffice for most applications.

* - CONSOLE:, a sequential, formatted file, also known as unit 0 - This particular unit has the special property that an entire line terminated by the return key must be entered when reading from it, and the various backspace and line delete keys familiar to the System user serve their normal functions. Note that a READ from any other unit will not have these properties, even if that unit is bound to CONSOLE: by an explicit OPEN statement.

Explicitly opened external, sequential, formatted files - These files are bound to a System file by name in an OPEN statement. They can be read and written in the System Text Editor compatible format.

Common I/O Operations

Here is a sample program which uses the kinds of files discussed in the previous paragraph for reading and for writing. The various I/O statements are explained in detail later in this chapter in the "I/O Statements" Section.

```

C Copy a file with three columns of integers, each 7
C columns wide, from a file whose name is input by the
C user to another file named OUT.TEXT, reversing the
C positions of the first and second column.
PROGRAM COLSWP
CHARACTER*23 FNAME
C Prompt to the CONSOLE: by writing to *
WRITE(*,900)
900 FORMAT('input file name - ')
C Read the file name from the CONSOLE: by reading from *
READ(*,910) FNAME
910 FORMAT(A)
C Use unit 3 for input, any unit number except 0 will do
OPEN(3,FILE=FNAME)
C Use unit 4 for output, any unit number except 0 and 3
C will do
OPEN(4,FILE='OUT.TEXT',STATUS='NEW')
C Read and write until end of file
100 READ(3,920,END=200)I,J,K
WRITE(4,920)J,I,K
920 FORMAT(BN,3I7)
GOTO 100
200 WRITE(*,910)'Done'
END

```


Less Common File Operations

The less commonly used file structures are appropriate for certain classes of applications. A very general indication of the intended usages for them are as follows: If the I/O is to be random access, such as in maintaining a database, direct access files are probably necessary. If the data is to be written by FORTRAN and reread by FORTRAN (on the same brand of processor), unformatted files are more efficient both in file space and in I/O overhead. The combination of direct and unformatted is ideal for a database to be created, maintained, and accessed exclusively by FORTRAN. If the data must be transferred without any System interference, especially if all 256 possible bytes will be transferred, unformatted I/O will be necessary, since .TEXT files are constrained to contain only the printable character set as data. An example of a usage of unformatted I/O would be in the control of a device which has a single byte, binary interface. Formatted I/O would, in this example, interpret certain characters, such as the ASCII representation for carriage return, and fail to pass them through to the program unaltered. Internal files are not I/O in the conventional sense but rather provide certain character string operations and conversions within a standard mechanism.

Use of formatted direct files requires special caution. FORTRAN formatted files attempt to comply with the Operating System rules for .TEXT files (for a discussion of .TEXT files, see the Operating System manual). FORTRAN I/O is able to enforce these rules for sequential files, but it cannot always enforce them for direct

files. Direct files are not necessarily legal .TEXT files, since any unwritten records contain undefined values which do not follow .TEXT file constraints. Direct files do, of course, obey FORTRAN I/O rules.

A file opened in FORTRAN is either "old" or "new", but there is no concept of "opened for reading" as distinguished from "opened for writing". Therefore, you may open "old" (existing) files and write to them, with the effect of modifying existing files. Similarly, you may alternately write and read to the same file (providing that one avoids reading beyond end of file or reading unwritten records in a direct files). A write to a sequential file effectively deletes any records which had existed beyond the freshly written record. Normally, when a device is opened as a file (such as CONSOLE: or PRINTER:), it makes no difference whether the file is opened as "old" or "new". With diskette files, opening "new" creates a new temporary file. If that file is closed using the "keep" option, or if the program is terminated without doing a CLOSE on that file, a permanent file is created with the name given when the file was opened. If a previous file existed with the same name, it is deleted. If closed using the "delete" option, the newly created temporary file is deleted, and any previous file of the same name is left intact. Opening a diskette file as "old" will generate a run time error if the file does not exist and alter the existing file if written.

In addition, there is an I/O intrinsic function EOF, presented in Chapter 13, which returns a logical value indicating whether the file associated with the unit specifier passed to it is at end-of-file. A familiarity with the FORTRAN file system, units, records, and access methods as described in the previous Sections is assumed.

Elements of I/O Statements

The various I/O statements take certain parameters and arguments which specify sources and destinations of data transfer, as well as other facets of the I/O operation. The abbreviations used throughout this Section are defined in the following three subsections.

The Unit Specifier ('u')

The unit specifier, 'u', can take one of these forms in an I/O statement:

* - refers to the CONSOLE:.

integer expression - refers to external file with unit number equal to the value of the expression (* is unit number 0).

name of a character variable or character array element - refers to the internal file which is the character datum.

The Format Specifier ('f')

The format specifier, 'f', can take one of these forms in an I/O statement:

statement label - refers to the FORMAT statement labeled by that label.

integer variable name - refers to the FORMAT label which that integer variable has been assigned to using the ASSIGN statement.

character expression - the format which is specified is the current value of the character expression provided as the format specifier.

The Input/Output List ('iolist')

The input/output list, 'iolist', specifies the entities whose values are transferred by READ and WRITE statements. An iolist is a possibly empty list, separated by commas, of items which consist of:

Input or Output entities - see the following two subsections.

Implied DO lists - see "Implied DO Lists" in this section.

Input Entities

An input entity may be specified in the iolist of a READ statement and is of one of these forms:

Variable name.

Array element name.

Array name - this is a means of specifying all of the elements of the array in storage sequence order.

Output Entities

An output entity may be specified in the iolist of a WRITE statement and is of one of these forms:

Variable name;

Array element name;

Array name - this is a means of specifying all of the elements of the array in storage sequence order;

Any other expression not beginning with the character '(' - to distinguish implied DO lists from expressions.

Implied DO lists

Implied DO lists may be specified as items in the I/O list of READ and WRITE statements and are of the form:

(iolist, i = e1, e2 [, e3])

where the iolist is as above (including nested implied DO lists); and i, e1, e2, and the optional e3 are as defined for the DO statement. That is, i is an integer variable; and e1, e2, and e3 are integer expressions. In a READ statement, the DO variable i (or an associated entity) must not appear as an input list item in the embedded iolist, but may have been read in the same READ statement outside of the implied DO list. The embedded iolist is effectively repeated for each iteration of i with appropriate substitution of values for the DO variable i.

I/O Statements

The following I/O statements are supported by FORTRAN. The possible form for each statement is specified first, with an explanation of the meanings for the forms following. Certain items are specified as required, if they must appear in the statement; and are specified as optional, if they need not appear. Typically,

optional items have defaults. Examples are provided.

OPEN Statement

OPEN(

u,

Required; must appear as the first argument. Must not be internal unit specifier.

FILE=fname,

The file name, 'fname', is a character expression. This argument to OPEN is required and must appear as the second argument.

The following arguments are all optional and may appear in any order. The options are character constants with optional trailing blanks (except RECL=). Defaults are indicated.

STATUS='OLD'

Default, for reading or writing existing files.

STATUS='NEW'

For writing new files.

ACCESS='SEQUENTIAL' (Default)

ACCESS='DIRECT'

FORM='FORMATTED' (Default)

FORM='UNFORMATTED'

RECL=r1)

The record length, 'r1', is an integer expression. This argument to OPEN is for DIRECT access files only, for which it is required.

The OPEN statement binds a unit number with an external device or file on an external device by specifying its file name. If the file is to be direct, the RECL=r1 option specifies the length of the records in that file.

REC=rn

For direct access only; otherwise, error. Positions to record number rn, where rn is a positive integer expression. If omitted for direct access file, reading continues from the current position in the file.

END=s)

Optional, statement label. If not present, reading end of file results in a run time error. If present, encountering an end of file condition results in the transfer to the executable statement labeled s which must be in the same program unit as the READ statement.

iolist

The READ statement sets the items in iolist (assuming that no end of file or error condition occurs). If the read is internal, the character variable or character array element specified is the source of the input; otherwise, the external unit is the source.

Example program fragment:

```
C Need a two dimensional array for the example
  DIMENSION IA(10,20)
C Read in bounds for array off first line, hopefully less
C than 10 and 20. Then read in the array in nested
C implied DO lists with input format of 8 columns of width
C 5 each.
  READ(3,990)I,J,((IA(I,J),J=1,J),I=1,I,1)
990 FORMAT(2I5/, (8I5))
```

WRITE Statement.

WRITE(

u,

Required; must be first argument.

f,

Required for formatted write as second argument; must not appear for unformatted write.

REC=rn)

For direct access only; otherwise, error. Positions to record number rn, where rn is a positive integer expression. If omitted for direct access file, writing continues at the current position in the file.

iolist

The WRITE statement transfers the iolist items to the unit specified. If the write is internal, the character variable or character array element specified is the destination of the output; otherwise, the external unit is the destination.

Example program fragment:

```

C Place message: "One = 1, Two = 2, Three = 3" on the
C CONSOLE:, not doing things in the simplest way!
  WRITE(*,980)'One = ',1,1+1,'ce = ',+(1+1+1)
980  FORMAT(A,I2,',', Two = ',1X,I1,',', Thr',A,I1)

```

BACKSPACE Statement

BACKSPACE u

Unit is not internal unit specifier. Can only be issued on units which are bound to blocked devices. Can only be issued on units which are direct or sequential formatted (i.e., not on sequential unformatted).

BACKSPACE causes the file connected to the specified unit to be positioned before the preceding record. If there is no preceding record, the file position is not changed. Note that if the preceding record is the endfile record, the file becomes positioned before the endfile record.

ENDFILE Statement

ENDFILE u

Unit is not an internal unit specifier.

ENDFILE "writes" an end of file record as the next record of the file connected to the specified unit. The file is then positioned after the end of file record, so further sequential data transfer is prohibited until either a BACKSPACE or REWIND is executed. An ENDFILE on a direct access file makes all records written beyond the position of the new end of file disappear.

REWIND Statement

REWIND u

Unit is not an internal unit specifier.

Execution of a REWIND statement causes the file associated with the specified unit to be positioned at its initial point.

I/O Side Effects Restriction

Any function referenced in an expression within any I/O statement must not cause any I/O statement to be executed.

This chapter describes formatted I/O and the FORMAT statement. A familiarity with the FORTRAN file system, units, records, access methods, and I/O statements as described in the previous chapters is assumed.

FORMAT SPECIFICATIONS

If a READ or WRITE statement specifies a format, it is considered a formatted, rather than an unformatted I/O statement. Such a format may be specified in one of three ways, as explained in the previous chapter. Two ways refer to FORMAT statements, and one is an immediate format in the form of a character expression containing the format itself. The following are all valid and equivalent means of specifying a format:

```
WRITE(*,990)I,J,K
990  FORMAT(2I5,13)

ASSIGN 990 TO 1FMT
990  FORMAT(2I5,13)
WRITE(*,1FMT)I,J,K

WRITE(*,'(2I5,13)')I,J,K

CHARACTER*8 FMTCH
FMTCH = '(2I5,13)'
WRITE(*,FMTCH)I,J,K
```

The format specification itself must begin with "(", possibly following initial blank characters, and end with a matching ")". Characters beyond the matching ")" are ignored.

FORMAT statements must be labelled, and like all nonexecutable statements, may not be the target of a branching operation.

CHAPTER 12 FORMATTED I/O AND THE FORMAT STATEMENT

Between the initial "(" and terminating ")" is a list of items, separated by commas, each of which is one of:

[r] ed - repeatable edit descriptors

ned - nonrepeatable edit descriptors

[r] fs - a nested format specification. At most 3 levels of nested parenthesis are permitted within the outermost level.

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

The repeatable edit descriptors, explained in detail below, are:

Iw
Fw.d
Ew.d
Ew.dEe
Lw
A
Aw

where I, F, E, L, and A indicate the manner of editing; w and e are nonzero, unsigned, integer constants; and d is an unsigned integer constant.

The nonrepeatable edit descriptors (also explained in detail below) are:

'xxxx' - character constants of any length; see special rules below

nHxxxx - another means of specifying character constants; see rules below

nX
/
\
kP
BN
BZ

where apostrophe, H, X, slash, backslash, P, BN, and BZ indicate the manner of editing; x is any ASCII character; n is a nonzero, unsigned, integer constant; and k is an optionally signed integer constant.

FORMAT AND I/O LIST

Before describing in greater detail the manner of editing specified by each of the above edit descriptors, it must be explained how the format specification interacts with the input/output list (iolist) in a given READ or WRITE statement.

If an iolist contains at least one item, at least one repeatable edit descriptor must exist in the format specification. In particular, the empty edit specification, (), may be used only if no items are specified in the iolist (in which case the only action caused by the I/O statement is the implicit record skipping action associated with formats). Each item in the iolist will become associated with a repeatable edit descriptor during the I/O

statement execution in turn. In contrast to this, the other format control items interact directly with the record and do not become associated with items in the iolist.

The items in a format specification are interpreted from left to right. Repeatable edit descriptors act as if they were present *r* times (omitted *r* is treated as a repeat factor of 1). Similarly, a nested format specification is treated as if its items appeared *r* times.

The formatted I/O process proceeds as follows: The "format controller" scans the format items in the order indicated above. When a repeatable edit descriptor is encountered, either

a corresponding item appears in the iolist, in which case the item and the edit descriptor become associated, and I/O of that item proceeds under format control of the edit descriptor; or

the "format controller" terminates I/O.

If the format controller encounters the matching final) of the format specification; and there are no further items in the iolist, the "format controller" terminates I/O. If, however, there are further items in the iolist, the file is positioned at the beginning of the next record; and the "format controller" continues by rescanning the format, starting at the beginning of the format specification terminated by the last preceding right parenthesis. If there is no such preceding right parenthesis, the "format controller" will rescan the format from the beginning. Within the portion of the format rescanned, there must be at least one

repeatable edit descriptor. Should the rescan of the format specification begin with a repeated nested format specification, the repeat factor is used to indicate the number of times to repeat that nested format specification. The rescan does not change the previously set scale factor or BN or BZ blank control in effect. When the "format controller" terminates, the remaining characters or an input record is skipped or an end of record is written on output, except as noted under the edit descriptor.

EDIT DESCRIPTORS

Here are the detailed explanations of the various format specification descriptors, beginning with the nonrepeatable edit descriptors:

Nonrepeatable Edit Descriptors

'xxxx' (Apostrophe Editing)

The apostrophe edit descriptor has the form of a character constant. Embedded blanks are significant, and double " are interpreted as a single '. Apostrophe editing may not be used in a READ statement. It causes the character constant to be transmitted to the output unit.

H (Hollerith Editing)

The nH edit descriptor causes the following n characters, with blanks counted as significant, to be transmitted to the output. Hollerith editing may not be used in a READ.

Examples of Apostrophe and Hollerith editing:

```
C Each write outputs characters between the
C slashes: /ABC'DEF/
      WRITE(*,970)
970  FORMAT('ABC'DEF')
      WRITE(*,('ABC'DEF'))
      WRITE(*,(7HABC'DEF'))
      WRITE(*,960)
960  FORMAT(7HABC'DEF')
```

X (Positional Editing)

On input (a READ), the nX edit descriptor causes the file position to advance over n characters; thus, the next n characters are skipped. On output (a WRITE), the nX edit descriptor causes n blanks to be written, providing that further writing to the record occurs; otherwise, the nX descriptor results in no operation.

/ (Slash Editing)

The slash indicates the end of data transfer on the current record. On input, the file is positioned to the beginning of the next record. On output, an end of record is written; and the file is positioned to write on the beginning of the next record.

\ (Backslash Editing)

Normally when the "format controller" terminates, the end of data transmission on the current record occurs. If the last edit descriptor encountered by the "format controller" is the backslash, this automatic end of record is inhibited. This allows subsequent I/O statements to continue reading (or writing) out of (or into) the same record. The most common use for this mechanism is to prompt to the CONSOLE: and read a response off the same line as in:

```
WRITE(*, '(A\)' 'Input an integer -> ')
READ(*, '(BN,I6)' I)
```

The backslash edit descriptor does not inhibit the automatic end of record generated when reading from the * unit. Input from the CONSOLE: must always be terminated by the return key. This permits the backspace character and the line delete key to function properly.

P (Scale Factor Editing)

The kP edit descriptor is used to set the scale factor for subsequent F and E edit descriptors, until another kP edit descriptor is encountered. At the start of each I/O statement, the scale factor equals 0. The scale factor affects format editing in the following ways:

On input, with F and E editing, providing that no explicit exponent exists in the field, and F output editing, the externally represented number equals the internally represented number multiplied by 10^{**k} .

On input, with F and E editing, the scale factor has no effect, if there is an explicit exponent in the input field.

On output, with E editing, the real part of the quantity is output multiplied by 10^{**k} ; and the exponent is reduced by k (effectively altering the column position of the decimal point, but not the value that is output).

BN and BZ (Blank Interpretation)

These edit descriptors specify the interpretation of blanks in numeric input fields. The default, BZ, is set at the start of each I/O statement. This makes blanks, other than leading blanks, identical to zeros. If a BN edit descriptor is processed by the "format controller", blanks in subsequent input fields will be ignored, unless, and until, a BZ edit descriptor is processed. The effect of ignoring blanks is to take all the nonblank characters in the input field and treat them as if they were right justified in the field with the number of leading blanks equal to the number of ignored

blanks. For instance, the following READ statement shown accepts the characters shown between the slashes as the value 123 (where <cr> indicates hitting the return key):

```
100 READ(*,100) I
    FORMAT(BN,16)

    /123      <cr>/,
    /123 456<cr>/,
    /123<cr>/, or
    / 123<cr>/.
```

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

Repeatable Edit Descriptors

I, F, and E (Numeric Editing)

The I, F, and E edit descriptors are used for I/O of integer and real data. The following general rules apply to all three of them:

On input, leading blanks are not significant. Other blanks are interpreted differently; depending on the BN or BZ flag in effect; but all blank fields always become the value 0. Plus signs are optional.

On input, with F and E editing, an explicit decimal point appearing in the input field overrides the edit descriptor specification of the decimal point position.

On output, the characters generated are right justified in the field with padding leading blanks, if necessary.

On output, if the number of characters produced exceeds the field width; or the exponent exceeds its specified width, the entire field is filled with asterisks.

I (Integer Editing)

The edit descriptor Iw must be associated with an iolist item which is of type integer. The field width is w characters in length. On input, an optional sign may appear in the field. The general rules in the preceding paragraphs apply to the I edit descriptor.

F (Real Editing)

The edit descriptor Fw.d must be associated with an iolist item which is of type real. The width of the field is w positions, the fractional part of which consists of d digits. The input field begins with an optional sign, followed by a string of digits, optionally containing a decimal point. If the decimal point is present, it overrides the d specified in the edit descriptor; otherwise, the rightmost d digits of the string are interpreted as following the decimal point (with leading blanks converted to zeros if necessary). Following this is an optional exponent which is one of:

plus or minus, followed by an integer; or

E or D, followed by zero or more blanks, followed by an optional sign, followed by an integer (E and D are treated identically).

The output field occupies w digits, d of which falls beyond the decimal point, and the value output is controlled both by the iolist item and the current scale factor. The output value is rounded rather than truncated.

The preceding general rules apply to the F edit descriptor.

E (Real Editing)

An E edit descriptor either takes the form Ew.d or Ew.dEe. In either case, the field width is w characters. The e has no effect on input. The input field for an E edit descriptor is identical to that described by an F edit descriptor with the same w and d. The form of the output field depends on the scale factor (set by the P edit descriptor) which is in effect. For a scale factor of 0, the output field is a minus sign (if necessary), followed by a decimal point, followed by a string of digits, followed by an exponent field for exponent, exp, of one of the following forms:

Ew.d -99 <= exp <= 99

E, followed by plus or minus, followed by the two digit exponent.

Ew.d -999 <= exp <= 999

Plus or minus, followed by three digit exponent.

Ew.dEe -(10**e) - 1) <= exp <= (10**e) -1

E, followed by plus or minus, followed by e digits which are the exponent with possible leading zeros.

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

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

The preceding general rules apply to the E edit descriptor.

L (Logical Editing)

The edit descriptor is Lw, indicating that the field width is w characters. The iolist element which becomes associated with an L edit descriptor must be of type logical. On input, the field consists of optional blanks, followed by an optional decimal point, followed by a T

(for .TRUE.) or and F (for accepted on input, so that .TRUE. and .FALSE. are valid inputs. On output, w - 1 blanks are followed by either T or F as appropriate.

A (Character Editing)

The forms of the edit descriptor are A or Aw. If w is not present, the number of characters in the iolist item, with which it becomes associated, determines the length (an implicit w). The iolist item must be of character type if it is to be associated with an A or Aw edit descriptor. On input, if w exceeds or equals the number of characters in the iolist element, the rightmost characters of the input field are used as the input characters; otherwise, the input characters are left justified in the input iolist item, and trailing blanks are provided. On output, if w should exceed the characters produced by the iolist item, leading blanks are provided; otherwise, the leftmost w characters of the iolist item are output.

CHAPTER 13
PROGRAMS
AND
SUBROUTINES

This chapter describes the format of program units. A program unit is either a main program, a subroutine, or a function program unit. The term procedure is used to refer to either a function or a subroutine. This chapter also describes the CALL and RETURN statements as well as function calls.

MAIN PROGRAM

A main program is any program unit that does not have a FUNCTION or SUBROUTINE statement as its first statement. It may have a PROGRAM statement as its first statement. The execution of a program always begins with the first executable statement in the main program. Consequently, there must be precisely one main program in every executable program. The form of a PROGRAM statement is:

PROGRAM pname

where: 'pname' is a user defined name that is the name of the main program.

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

SUBROUTINES

A subroutine is a program unit that can be called from other program units by a CALL statement. When invoked, it performs the set of actions defined by its executable statements and then returns control to the statement immediately following the statement that called it. A subroutine does not directly return a value, although values can be passed back to the calling program unit via parameters or common variables.

SUBROUTINE Statement

A subroutine begins with a SUBROUTINE statement and ends with the first following END statement. It may contain any kind of statement other than a PROGRAM statement or a FUNCTION statement. The form of a SUBROUTINE statement is:

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

'sname' is the user defined name of the subroutine.

'farg' is a user defined name of a formal argument.

The name 'sname' is a global name, and it is also local to the subroutine it names. The list of argument names defines the number and (with any subsequent IMPLICIT, type, or DIMENSION statements) the type of arguments

to that subroutine. Argument names cannot appear in COMMON, DATA, EQUIVALENCE, or INTRINSIC statements. The following is an example of a subroutine:

```

Program TEST
irec=3000
b=1.0
h=2.0
a=3.0
t=4.0

call prndat (irec,b,h,a,t)
end

subroutine prndat (i,u,v,x,y)

open(6,file='printer:')
write(6,200)i,u,v,x,y
format(14,4F10.2)
return
end
    
```

3000	1.00	2.00	3.00	4.00
------	------	------	------	------

CALL Statement

A subroutine is executed as a consequence of executing a CALL statement in another program unit that references that subroutine. The form of a CALL statement is:

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

'sname' is the name of a subroutine.

'arg' is an actual argument.

An actual argument may be either an expression or the name of an array. The actual arguments in the CALL statement must agree in type and number with the corresponding formal arguments specified in the SUBROUTINE statement of the referenced subroutine. If there are no arguments in the SUBROUTINE statement, then a CALL statement referencing that subroutine must not have any actual arguments, but may optionally have a matched pair of parentheses following the name of the subroutine. Note that a formal argument may be used as an actual argument in another subprogram call.

Execution of a CALL statement proceeds as follows: All arguments that are expressions are evaluated. All actual arguments are associated with their corresponding formal arguments, and the body of the specified subroutine is executed. Control is returned to the statement following the CALL statement upon exiting the subroutine, by executing either a RETURN statement or an END statement in that subroutine.

A subroutine specified in any program unit may be called from any other program unit within the same executable program. Recursive subroutine calls, however, are not allowed in FORTRAN. That is, a subroutine cannot call itself directly, nor can it call another subroutine that will result in the first subroutine being called again before it returns control to its caller.

FUNCTIONS

A function is referenced in an expression and returns a value that is used in the computation of that expression. There are three kinds of functions: external functions, intrinsic functions, and statement functions. This section describes the three kinds of functions.

A function reference may appear in an arithmetic expression. Execution of a function reference causes the function to be evaluated, and the resulting value is used as an operand in the containing expression. The form of a function reference is:

```
fname ( [arg [,arg]...] )
```

'fname' is the name of an external, intrinsic, or statement function.

'arg' is an actual argument.

An actual argument may be an arithmetic expression or an array. The number of actual arguments must be the same as in the definition of the function, and the corresponding types must agree.

External Functions

An external function is specified by a function program unit. It begins with a FUNCTION statement and ends with an END statement. It may contain any kind of statement other than a PROGRAM statement or a SUBROUTINE

statement. The form of a FUNCTION statement is:

```
[type] FUNCTION fname ( [farg [, farg]...] )
```

'type' is one of INTEGER, REAL, or LOGICAL.

'fname' is the user defined name of the function.

'farg' is a formal argument name.

The name 'fname' is a global name, and it is also local to the function it names. If no type is present in the FUNCTION statement, the function's type is determined by default and any subsequent IMPLICIT or type statements that would determine the type of an ordinary variable. If a type is present, then the function name cannot appear in any additional type statements. In any case, an external function cannot be of type character. The list of argument names defines the number and, with any subsequent IMPLICIT, type, or DIMENSION statements, the type of arguments to that subroutine. Neither argument names nor 'fname' can appear in COMMON, DATA, EQUIVALENCE, or INTRINSIC statements.

The function name must appear as a variable in the program unit defining the function. Every execution of that function must assign a value to that variable. The final value of this variable, upon execution of a RETURN or of an END statement, defines the value of the function. After being defined, the value of this variable can be referenced in an expression, exactly like

any other variable. An external function may return values in addition to the value of the function by assignment to one or more of its formal arguments.

Intrinsic Functions

Intrinsic functions are functions that are predefined by the FORTRAN compiler and are available for use in a FORTRAN program. Table 13.1 gives the name, definition, number of parameters, and type of the intrinsic functions available in p-System FORTRAN 77. An IMPLICIT statement does not alter the type of an intrinsic function. For those intrinsic functions that allow several types of arguments, all arguments in a single reference must be of the same type.

All intrinsic functions used in a program unit must appear in an INTRINSIC statement.

An intrinsic function name may appear in an INTRINSIC statement, but only those intrinsic functions listed in Table 13.1 may do so. An intrinsic function name also may appear in a type statement, but only if the type is the same as the standard type of that intrinsic function.

Arguments to certain intrinsic functions are limited by the definition of the function being computed. For example, the logarithm of a negative number is undefined and, therefore, not allowed.

Statement Functions

A statement function is a function that is defined by a single statement. It is similar in form to an assignment statement. A statement function can only appear after the specification statements and before any executable statements in the program unit in which it appears. A statement function is not an executable statement, since it is not executed in order as the first statement in its particular program unit. Rather, the body of a statement function serves to define the meaning of the statement function. It is executed, as any other function, by the execution of a function reference. The form of a statement function is:

```
fname ( [arg [, arg]...] ) = expr
```

'fname' is the name of the statement function.

'arg' is a formal argument name.

'expr' is an expression.

The type of the 'expr' must be assignment compatible with the type of the statement function name. The list of formal argument names serves to define the number and type of arguments to the statement function. The scope of formal argument names is the statement function. Therefore, formal argument names may be used as other user defined names in the rest of the program unit enclosing the statement function definition. The name of the statement function, however, is local to the enclosing program unit and must not be otherwise used; **except** as the name of a common block or as the

name of a formal argument to another statement function. The type of all such uses, however, must be the same. If a formal argument name is the same as another local name, then a reference to that name within the statement function defining it always refers to the formal argument, never to the other usage.

Within the expression 'expr', references to variables, formal arguments, other functions, array elements, and constants are allowed. Statement function references, however, must refer to statement functions that have been defined prior to the statement function in which they appear. Statement functions cannot be recursively called, either directly or indirectly.

A statement function can only be referenced in the program unit in which it is defined. The name of a statement function cannot appear in any specification statement, except in a type statement which may not define that name as an array, and in a COMMON statement as the name of a common block. A statement function cannot be of type character.

RETURN STATEMENT

A RETURN statement causes return of control to the calling program unit. It may only appear in a function or subroutine. The form of a RETURN statement is:

```
RETURN
```

Execution of a RETURN statement terminates the execution of the enclosing subroutine or function. If the RETURN statement is in a function, then the value of that function is equal to the current value of the variable with the same name as the function. Execution of an END statement in a function or subroutine is equivalent to execution of a RETURN statement.

PARAMETERS

This section discusses the relationship between formal and actual arguments in a function or subroutine call. A formal argument refers to the name by which the argument is known within the function or subroutine; and an actual argument is the specific variable, expression, array, etc., passed to the procedure in question at any specific calling location.

Arguments are used to pass values into and out of procedures. Variables in common can be used to perform this task as well. The number of actual arguments must be the same as formal arguments, and the corresponding types must agree.

On entry to a subroutine or function, the actual arguments become associated with the formal arguments, much as an EQUIVALENCE statement associates two or more arrays or variables, and COMMON statements in two or more program units associate lists of variables. This association remains in effect until execution of the subroutine or function is terminated. Thus, assigning a value to a formal argument during execution of a subroutine or function may alter the value of the

corresponding actual argument. If an actual argument is a constant, function reference, or an expression other than a simple variable, assigning a value to the corresponding formal argument is not allowed, and may have some strange side effects. In particular, assigning a value to a formal argument of type character, when the actual argument is a literal, can be disastrous.

If an actual argument is an expression, it is evaluated immediately prior to the association of formal and actual arguments. If an actual argument is an array element, its subscript expression is evaluated just prior to the association and remains constant throughout the execution of the procedure, even if it contains variables that are redefined during the execution of the procedure.

A formal argument that is a variable may be associated with an actual argument that is a variable, an array element, or an expression.

A formal argument that is an array may be associated with an actual argument that is an array or an array element. The number and size of dimensions in a formal argument may be different than those of the actual argument, but any reference to the formal array must be within the limits of the storage sequence in the actual array. While a reference to an element outside these bounds is not detected as an error in a running FORTRAN program, the results are unpredictable.

<u>Intrinsic Function</u>	<u>Definition</u>	<u>No. Args</u>	<u>Name</u>	<u>Type of Argument Function</u>	
Type Conversion	Conversion to Integer int(a)	1	INT IFIX	Real Real	Integer Integer
	See Note 1				

	Conversion to Real See Note 2	1	REAL FLOAT	Integer Integer	Real Real
	Conversion to Integer See Note 3	1	ICHAR	Character	Integer
	Conversion to Character	1	CHAR	Integer	Character
Truncation	int(a) See Note 1	1	AINI	Real	Real
Nearest Whole Number	int(a.5) a>=0 int(a.5) a<0	1	ANIM	Real	Real
Nearest Integer	int(a.5) a>=0 int(a.5) a<0	1	NINT	Real	Integer
Absolute Value	a	1	IABS ABS	Integer Real	Integer Real
Remaindering	a1int(a1/a2)*a2 See Note 1	2	MOD AMOD	Integer Real	Integer Real
Transfer of Sign	a1 if a2>=0 a1 if a2<0	2	ISIGN SIGN	Integer Real	Integer Real
Positive Difference	a1a2 if a1>a2 0 if a1<=a2	2	DDIM DIM	Integer Real	Integer Real
Choosing Largest Value	max(a1,a2,...)	>=2	MAX0 AMAX1	Integer Real	Integer Real
			AMAX0 MAX1	Integer Real	Real Integer
Choosing Smallest Value	min(a1,a2,...)	>=2	MIN0 AMIN1	Integer Real	Integer Real
			AMIN0 MIN1	Integer Real	Real Integer
Square Root	a**0.5	1	SQRT	Real	Real
Exponential	e**a	1	EXP	Real	Real
Natural Logarithm	log(a)	1	ALOG	Real	Real
Common Logarithm	log10(a)	1	ALOG10	Real	Real
Sine	sin(a)	1	SIN	Real	Real
Cosine	cos(a)	1	COS	Real	Real
Tangent	tan(a)	1	TAN	Real	Real
Arcsine	arcsin(a)	1	ASIN	Real	Real
Arccosine	arccos(a)	1	ACOS	Real	Real
Arctangent	arctan(a)	1	ATAN	Real	Real
	arctan(a1/a2)	2	ATAN2	Real	Real
Hyperbolic Sine	sinh(a)	1	SINH	Real	Real

Hyperbolic Cosine	cosh(a)	1	COSH	Real	Real
Hyperbolic Tangent	tanh(a)	1	TANH	Real	Real
Lexically Greater Than or Equal	a1 >= a2 See Note 4	2	LGE	Character	Logical
Lexically Greater Than	a1 > a2 See Note 4	2	LGT	Character	Logical
Lexically Less Than or Equal	a1 <= a2 See Note 4	2	LLE	Character	Logical
Lexically Less Than	a1 < a2 See Note 4	2	LLT	Character	Logical
End of File	End_of_File(a) See Note 5	1	EOF	Integer	Logical

Table 13.1 Intrinsic Functions

Table 13.1 Notes

- 1) For a of type real, if a >= 0 then int(a) is the largest integer not greater than a, if a < 0 then int(a) is the most negative integer not less than a. IFIX(a) is the same as INT(a).
- 2) For a of type integer, REAL(a) is to the greatest possible precision. This varies from processor to processor. FLOAT(a) is the same as REAL(a).
- 3) ICHAR converts a character value into an integer value. The integer value of a character is the ASCII internal representation of that character, and is in the range 0 to 127. For any two characters, c1 and c2, (c1 .LE. c2) is .TRUE. if and only if (ICHAR(c1) .LE. ICHAR(c2)) is
- 4) LGE(a1,a2) returns the value .TRUE.. if a1 = a2 or if a1 follows a2 in the ASCII collating sequence. Otherwise it returns .FALSE..

Programs and Subroutines

LGT(a1,a2) returns .TRUE. if a1 follows a2 in the ASCII collating sequence, otherwise it returns .FALSE..

LLE(a1,a2) returns .TRUE. if a1 = a2 or if a1 precedes a2 in the ASCII collating sequence, otherwise it returns .FALSE..

LLT(a1,a2) returns .TRUE. if a1 precedes a2 in the ASCII collating sequence, otherwise it returns .FALSE..

The operands of LGE, LGT, LLE, and LLT must be of the same length.

- 5) EOF(a) returns the value .TRUE. if the unit specified by its argument is at or past the end of file record, otherwise it returns .FALSE.. The value of a must correspond to an open file, or to zero (which indicates CONSOLE:).
- 6) All angles are expressed in radians.
- 7) All arguments in an intrinsic function reference must be of the same type.

CHAPTER 14 COMPI LATION UNITS

This chapter describes the relationship between FORTRAN and the Pascal segment mechanism. In normal use, the user need not be aware of such intricacies. However, if the user desires to interface FORTRAN with Pascal, to create overlays, or to take advantage of separate compilation or libraries, the details contained here are helpful. This chapter consists of the following sections:

- Units, Segments, Partial Compilation, and FORTRAN.
- The \$USES Compiler Directive.
- Linking Pascal and FORTRAN.
- The \$EXT Compiler Directive.

The first section discusses the general form of a FORTRAN program in terms of the operating system object code structure. The next section describes the \$USES compiler directive. This directive provides access libraries or already compiled procedures, and provides overlays in FORTRAN. The next section describes how one links FORTRAN with Pascal. The final section explains the \$EXT compiler directive.

PARTIAL COMPILATION

If a FORTRAN compilation contains no main procedure, then it is output as if it were a Pascal unit compilation. The unit is given the name 'U' followed by the name of its first procedure. For example:

```
C --- No PROGRAM statement present
SUBROUTINE X
...
END
```

```

SUBROUTINE Y
...
END
...
SUBROUTINE Z
...
END
    
```

would be compiled into a single unit named 'UX'. (Assume for later examples that the object code is output to file 'X.CODE'.) All procedures called from within unit UX must be defined within unit UX, unless a \$USES or a \$EXT statement has shown them to reside in another unit. Similarly, procedures in unit UX cannot be called from other units unless the other units contain a \$USES UX statement. Thus, a typical main program that would call X might be:

```

C
C -- This is the main program BIGGIE
C
$USES UX IN X.CODE

PROGRAM BIGGIE
...
CALL X
...
END
SUBROUTINE W
...
CALL Y
...
END
    
```

If the \$USES statement were not present, the FORTRAN compiler would expect subroutines X and Y to appear in the same compilation, somewhere after subroutine W. Assume that the object code for this compilation is output to the file 'BIGGIE.CODE'.

Thus, the user can create libraries of functions, partial compilations, etc., and save compilation time and disk space, by a simple use of the \$USES

statement. For more information on the \$USES statement, see the section on the \$USES statement.

THE \$USES COMPILER DIRECTIVE

The \$USES compiler directive provides several distinct functions to the user. It allows procedures and functions in separately compiled units, such as the system library, to be called from FORTRAN. It provides the user a relatively secure form of separate compilation for FORTRAN compilations. It allows the user to call Pascal routines that have been compiled into Pascal units.

The format of the \$USES control statement is:

\$USES unitname [IN filename] [OVERLAY]

where: 'unitname' is the name of a unit.

'filename' is a valid file name.

As with all \$ control statements, the \$ must appear in column one. This compiler directive directs the compiler to open the .CODE file 'filename', locate the unit 'unitname', and process the INTERFACE information associated with that unit, generating a reasonable FORTRAN equivalent declaration for the FORTRAN compilation in progress. All \$USES commands must appear before any FORTRAN statements, specification or executable, but they are allowed to follow comment lines and other \$ control lines. If the optional 'IN filename' is present, the name 'filename' is used as the file to process. If it is not, the file '*SYSTEM.LIBRARY' is used as a default. The optional field OVERLAY has no effect on program execution and is included

in version IV.0 only for compatibility with version II.0.

WARNING: If a FORTRAN main program \$USES a Pascal unit, any global variables in the INTERFACE part of that unit will not be accessible from FORTRAN. See the next section, "Linking Pascal and FORTRAN," in this chapter, for further information.

Separate Compilation

Separate compilation is accomplished by compiling a set of subroutines and functions without any main program. Each such compilation creates a code file containing a single unit. Then, when the main program is compiled, possibly along with many subroutines or functions, it \$USES the separately compiled units. The routines compiled with the main program obtain the correct definition of each externally compiled procedure through the \$USES directive.

In the simplest form, when no \$USES statements appear in any of the separate compilations, the user simply \$USES all separately compiled FORTRAN units in the main program. However, this limits the procedure calls in each of the separately compiled units to procedures defined in the same unit. If there are calls to procedures in unit A from unit B, then unit B must contain a \$USES A statement. The main program must then contain a \$USES A statement as its first \$USES statement, followed by a \$USES B statement. This is necessary for the compiler to get the unit numbers allocated consistently.

In more complicated cases, the user must ensure that all references to procedures in outside units are preceded by the proper \$USES statement in the same order and are not missing any units. If unit B \$USES unit A, and unit C \$USES unit B, then unit C must first \$USES unit A. Likewise, if units D and E both \$USES unit F, they both must contain exactly the same \$USES statements prior to the \$USES F statement.

LINKING PASCAL AND FORTRAN

In order to call Pascal routines from FORTRAN, the Pascal routines must first be compiled into a Pascal unit. The FORTRAN program can then \$USES that unit. Unfortunately, the exceedingly rich type structure present in Pascal is not present in FORTRAN. Also, the I/O systems of FORTRAN and Pascal are not compatible. Therefore, it is not possible to do everything one might desire. This section does, however, help the user do what is possible in interfacing the two languages.

There are several precautions that the user must take for FORTRAN I/O to work from Pascal programs. The FORTRAN I/O procedures use the heap for the allocation of file related storage, so the user should not force the deallocation of heap memory via calls to MARK and/or RELEASE. Other restrictions may apply in special cases.

Since there are Pascal types that have no FORTRAN equivalent, the way FORTRAN looks at Pascal parameters is somewhat limited. FORTRAN does recognize both reference and value parameters when calling Pascal subroutines. The following

table shows how FORTRAN views Pascal declarations:

Pascal Declaration:	FORTRAN's View:
CONST anything ... ;	Ignored.
TYPE anything ... ;	Ignored.
VAR anything ... ;	Ignored.
PROCEDURE X(arg-list);	SUBROUTINE X(arg-list)
FUNCTION X(arg-list): type;	type FUNCTION X(arg-list)
Note: type must be INTEGER, LOGICAL, or REAL.	
types:	
REAL	REAL
BOOLEAN	LOGICAL
CHAR	CHARACTER*1
ALFAN	CHARACTER*n 1 <= n <= 127
any other identifier	INTEGER
arg-list:	
(VAR I,J: type)	(I,J)
type I,J	
(I,J: type)	*** There is no proper FORTRAN equivalent to value parameters, but the FORTRAN compiler does generate the correct calling sequence for Pascal routines with value parameters.

Examples of using Pascal from FORTRAN:

```
ALFA5->CHARACTER*5
ALFA120->CHARACTER*120
```

Likewise, when the INTERFACE information for a FORTRAN program is output, it must be mapped onto Pascal declarations. The following table gives the corresponding declarations:

FORTRAN Declaration:	Pascal's View:
SUBROUTINE X(arg-list)	PROCEDURE X(arg-list);
type FUNCTION X(arg-list)	FUNCTION X(arg-list): type;
types:	
INTEGER	INTEGER
REAL	REAL
LOGICAL	BOOLEAN
CHARACTER*n	CHAR n = 1
	PACKED ARRAY of CHAR
	2 <= n <= 127
arg-list:	
(I) (VAR I: type)	
type I	

NOTE: When a Pascal compilation USES a FORTRAN unit, it is the responsibility of the Pascal program to make sure that any needed type declarations for the ALFAN types are properly defined. This cannot consistently be done by FORTRAN as it would lead to duplicate type definitions should a user use two FORTRAN units in which each declares the same type. There is another point that must be made for Pascal programs that call FORTRAN subroutines. If the subroutine has a REAL parameter that is in actuality an array, the Pascal program must pass a scalar instead of an array. This should not be a problem. Since the Pascal program can pass the first element of the array, and all FORTRAN parameters are reference parameters, the FORTRAN subroutine has access to the whole array. The user is cautioned to remember that Pascal stores its arrays in row-major order, while FORTRAN stores them in column-major order.

When a FORTRAN program USES a Pascal unit, the interface section variables in that Pascal unit are not accessible from FORTRAN.

Here are two examples which illustrate how interfacing is accomplished between FORTRAN and Pascal. In the first example, a Pascal unit is used by a FORTRAN host. The ALFAN construct is employed. When the FORTRAN compiler parses the interface section of the Pascal unit, it ignores the type declaration where ALFA25 is defined. When procedure APROC is parsed, however, FORTRAN recognizes that parameter A is a 25 element packed array of characters. This corresponds to the FORTRAN declaration CHARACTER*25 as shown in the host FORTRAN program below.

```

PASCAL UNIT

unit pascal;
interface
  type alfa25=packed array[1..25] of char;
  procedure aprocv(var a:alfa25);
implementation
  procedure aprocv(var a:alfa25);
  begin
  end;
end.

```

```

FORTRAN HOST PROGRAM

uses pascal in pascal.code
subroutine frtran
character*25 array
call aprocv (array)

```

In the next example, a Pascal program uses a FORTRAN unit. A host Pascal program which defines any ALFA's must do so in a unit. In this example, the unit is called ALFAS. Unit ALFAS is required because the Pascal program must use the FORTRAN unit (with a USES statement) before any Pascal types can be declared. The unit can define the necessary ALFA types before the Pascal compiler parses the FORTRAN interface text.

One thing that you may notice is that there isn't any ALFA25 type in the FORTRAN interface text. (There is only the standard FORTRAN CHARACTER*25 statement.) This may lead you to wonder why such a type has to be declared in order for the Pascal compiler to parse that interface text. You should note that the Pascal compiler doesn't actually parse the original FORTRAN text. Instead, the FORTRAN compiler creates a Pascal interface section whenever a FORTRAN unit is compiled. This Pascal-translated interface section is what the Pascal compiler sees and it contains an ALFA25 type.

Another point about this example concerns the fact that when the Pascal program calls the FORTRAN procedure FWRITE (in the second to last line), an indexed array is passed. Since the Pascal-translated FORTRAN interface section contains text which makes this parameter type ALFA25, you must call FWRITE with A[1] rather than simply A. Passing A[1] loads the starting address of the array A onto the stack which enables FORTRAN to access it properly.

```

FORTRAN UNIT

subroutine funit
return
end

subroutine fwrite (a)
character*25 a
dimension a(25)
do 10 i=1,15
  write(*,'(A)') a(i)
10 continue
return
end

PASCAL HOST PROGRAM

Program pascal2;

unit alfas;
interface
  type alfa25 = packed array[1..25] of char;
implementation
end;

uses alfas,
      ($U for .ran2.code) ofunit;
var
  i,:integer;
  a:array[1..15] of alfa25;
begin
  for j:=1 to 15 do
    for i:=1 to 25 do
      a[j,i]:=chr(i+ord('0'));
    write(a[j]);
  end.

```

THE \$EXT COMPILER DIRECTIVE

The \$EXT compiler directive is used when one desires to call assembly language routines, or routines in \$SEPARATE FORTRAN or Pascal units, from a FORTRAN 77 routine. The form of the \$EXT directive is:

```
      ( SUBROUTINE )  
$EXT ( ) procname (*params  
      ( type ) FUNCTION )
```

where: 'type' is either INTEGER, LOGICAL, or REAL.

'procname' is the name of the subroutine or function, and

'*params' is an integer equal to the number of parameters that this procedure requires.

This directive must appear before any FORTRAN statements, either specification or executable, but may follow comment lines or other \$ compiler directives. All parameters are passed by reference (called VAR parameters if Pascal) to procedures defined by the \$EXT directive. It is up to the user to follow this convention, as the linker does not enforce it. The linker does, however, check the number of parameters.

A P P E N D I C E S

APPENDIX A ANSI FORTRAN DIFFERENCES

This appendix is directed to the reader who is familiar with the ANSI Standard FORTRAN 77 Subset language as defined in ANSI X3.9-1978. It concisely describes how SofTech Microsystems FORTRAN 77 differs from the standard language. The differences fall into three general categories:

- Unsupported Features
- Full-Language Features
- Extensions to Standard

Unsupported Features

There are two significant places where SofTech Microsystems FORTRAN 77 does not comply with the standard. One is that procedures cannot be passed as parameters and the other is that INTEGER and REAL data types do not occupy the same amount of storage. Both differences are due to limitations of the p-machine architecture.

Parametric procedures are not supported simply because there is no practical way to do so in the p-machine. The instruction set does not allow the loading of a procedure's address onto the stack, and more significantly, does not provide for the calling of a procedure whose address is on the stack.

REAL variables require 4 bytes of storage while INTEGER and LOGICAL variables only require 2 bytes. This is due to the fact that the p-machine supported operations on those types are implemented in those sizes.

Full-Language Features

There are several features from the full language that have been included in this implementation for a variety of reasons. Some were done at either minimal or zero cost, such as allowing arbitrary expressions in subscript calculations. Others were included because it was felt that they would significantly increase the utility of the implementation, especially in an engineering or laboratory application. An example is the generalized I/O that allows easier control of peripherals. In all cases, a program which is written to comply with the subset restrictions will compile and execute properly, since the full language properly includes the subset constructs. A short description of full language features included in the implementation follows.

Subscript Expressions - The subset does not allow function calls or array element references in subscript expressions, but the full language and this implementation do.

Do Variable Expressions - The subset restricts expressions that define the limits of a DO statement, but the full language does not. SofTech Microsystems FORTRAN also allows full integer expressions in DO statement limit computations. Similarly, arbitrary integer expressions are allowed in implied DO loops associated with READ and WRITE statements.

Unit I/O Number - SofTech Microsystems FORTRAN allows an I/O unit to be specified by an integer expression, as does the full language.

Expressions in I/O list - The subset does not allow expressions to appear in an I/O list, whereas the full language does allow expressions in the I/O list of a WRITE statement. SofTech Microsystems FORTRAN allows expressions in the I/O list of a WRITE statement, providing that they do not begin with an initial left parenthesis.

NOTE: The expression $(A+B)*(C+D)$ can be specified in an output list as $+(A+B)*(C+D)$ which, incidently, does not generate any extra code to evaluate the leading +.

Expression in computed GOTO - SofTech Microsystems FORTRAN allows an expression for the value of a computed GOTO, consistent with the full language rather than the subset language.

Generalized I/O - SofTech Microsystems FORTRAN allows both sequential and direct access files to be either formatted or unformatted. The subset language restricts direct access files to be unformatted and sequential files to be formatted. SofTech Microsystems FORTRAN also contains an augmented OPEN statement which takes additional parameters that are not included in the subset. There is also a form of the CLOSE statement, which is not included at all in the subset. I/O is described in more detail in Chapters 11 and 12.

Extensions to Standard

The language implemented has several minor extensions to the full language standard. These are briefly described below:

Compiler Directives - Compiler directives have been added to allow the programmer to communicate certain information to the Compiler. An additional kind of line, called a Compiler directive line, has been added. It is characterized by a dollar sign '\$' appearing in column 1. A Compiler directive line may appear any place that a comment line can appear, although certain directives are restricted to appear in certain places. A Compiler directive line is used to convey certain compile-time information to the System about the nature of the current compilation. The set of directives is briefly listed below:

`$INCLUDE filename`

Include textually the file 'filename' at this point in the source. Nested includes are implemented to a depth of nesting of five files. Thus, for example, a program may include various files with subprograms, each of which includes various files which describe common areas (which would be a depth of nesting of three files).

`$USES ident`
 `[IN filename]`
 `[OVERLAY]`

This is similar to a USES command in the UCSD Pascal Compiler. The already compiled FORTRAN subroutines or Pascal procedures contained in the .CODE file 'filename', or in the file '*SYSTEM.LIBRARY' (if no file name is present), become callable from the currently compiling code. This directive must appear before the initial noncomment input line. For more details, see Chapter 14.

`$XREF`

Produce a cross-reference listing at the end of each procedure compiled.

`$LX? SUBROUTINE name #parms`
 or
`$EXT [type] FUNCTION name #parms`

The subroutine or function named 'name' is either an assembly language routine or a routine in a \$SEPARATE unit (either FORTRAN or Pascal). The routine has exactly '#parms' reference parameters.

Backslash Edit Control - The edit control character '\ ' can be used in formats to inhibit the normal advancement to the next record which is associated with the completion of a READ or a WRITE statement. This is particularly useful when prompting to an interactive device, such as CONSOLE:, so that a response can be on the same line as the prompt.

End of File Intrinsic Function - An intrinsic function, EOF, has been provided. The function accepts a unit specifier as an argument and returns a logical value which indicates whether the specified unit is at its end of file.

Lowercase Input - Upper and lowercase source input is allowed. In most contexts, lowercase characters are treated as indistinguishable from their uppercase counterparts. Lowercase is significant in character constants and Hollerith fields.

APPENDIX B SAMPLE PROGRAM

This program demonstrates how some of the FORTRAN subroutines and files are used.

There are three FORTRAN subroutines and one Pascal procedure called from the FORTRAN host program. The Pascal procedure INIT displays the description and instructions of the program. Given the radius, the FORTRAN subroutines compute the area, surface area, and volume of a circle.

This program opens a sequential, formatted file that is saved after program termination. Input to the program is via the console, which is opened by default. Both exponential and floating point outputs are represented.

```

SUBROUTINE PASCINIT IN PASCINIT.CODE
PROGRAM DEMO

      INTEGER COUNT

      OPEN(1, FILE='circle.data', STATUS='new', FORM='formatted')
      OPEN(6, FILE='printer:')

C      Print program instructions via a Pascal unit
      CALL INIT

      P* = 3.14159
      COUNT = 1

50  CONTINUE
      WRITE(*, '(A)Radius for circle')
      WRITE(*, '(I3)COUNT')
      WRITE(*, '(A) ')

C      Input radius ith blanks being ignored
      READ(*, '(BN,F4.0)') R
      IF (R .EQ. 0.0) GOTO 100

```

```

c      Compute the Area, Surface Area, and Volume
      call ACOMP(P1,R,AREA)
      call SCOMP(P1,R,SAREA)
      call VCOMP(P1,R,VOLUME)

      write(1,200) COUNT,R,AREA,SAREA,VOLUME
      COUNT = COUNT + 1
      goto 50

c      Set end of file marker and start at the first
c      record of the file.
100    endfile 1
      rewind 1

      write(6,'(//)')
      write(6,220)

c      Print the data using exponential notation
      do 110 I=1,COUNT
        read(1,200,end=113) I,R,AREA,SAREA,VOLUME
        write(6,200) I,R,AREA,SAREA,VOLUME
110    continue

c      Go to the beginning of the file
113    rewind 1

c      Print the data using floating point notation

      write(6,'(//)')
      write(6,220)

      do 115 I=1,COUNT
        read(1,200,end=120) I,R,AREA,SAREA,VOLUME
        write(6,230) I,R,AREA,SAREA,VOLUME
115    continue

120    close(1,STATUS='keep')

200    format(15,4E16.5)
220    format(' CIRCLE',6X,'RADIUS',10X,'AREA',8X,'SURFACE AREA',
1      7X,'VOLUME'//)
230    format(15,4F16.5)

      end

      subroutine ACOMP(P1,RAD,AREA)
      AREA = PI* RAD**2
      return
      end

      subroutine SCOMP(P1,RAD,SAREA)

      SAREA = 4 * PI * (RAD**2)
      return
      end

      subroutine VCOMP(P1,RAD,VOLUME)

      VOLUME = (4 * PI * (RAD**3))/3
      return
      end

```

The following is the Pascal unit used by the preceding program:

```

unit PASCINIT;

interface

  procedure INIT;

implementation

  procedure INIT;
  var I: integer;

  begin
    gotoxy 15,07;
    writeln('This program computes the area, surface area,');
    gotoxy(15,08);
    writeln('and volume of a circle given the radius. To');
    gotoxy(15,09);
    writeln('end the input, enter 0 for the radius. Output');
    gotoxy(15,10);
    writeln('will be represented in two ways: ');
    gotoxy(15,12);
    writeln('      a) Exponential');
    gotoxy(15,13);
    writeln('      b) Floating Point');
    gotoxy(0,15);
  end;
end.

```

This appears on the screen when the program is run:

```

This program computes the area, surface area,
and volume of a circle given the radius. To
end the input, enter 0 for the radius. Output
will be represented in two ways:

```

```

      a) Exponential
      b) Floating Point

```

```

Radius for circle 1: 4
Radius for circle 2: 10
Radius for circle 3: 20
Radius for circle 4: 2
Radius for circle 5: 6
Radius for circle 6: 8
Radius for circle 7: 0

```


The FORTRAN program prints the following:

CIRCLE	RADIUS	AREA	SURFACE AREA	VOLUME
1	.40000E+01	.50265E+02	.20106E+03	.26808E+03
2	.10000E+02	.31416E+03	.12556E+04	.41888E+04
3	.20000E+02	.12566E+04	.50265E+04	.33510E+05
4	.30000E+02	.12566E+04	.50265E+04	.33510E+05
5	.60000E+01	.11310E+03	.45239E+03	.90478E+03
6	.80000E+01	.20106E+03	.80425E+03	.21447E+04

CIRCLE	RADIUS	AREA	SURFACE AREA	VOLUME
1	4.00000	50.26500	201.06000	268.08000
2	10.00000	314.16000	1256.60000	4188.80000
3	20.00000	1256.60000	5026.50000	33510.00000
4	2.00000	12.56600	50.26500	33.51000
5	6.00000	113.10000	452.39000	904.78000
6	8.00000	201.06000	804.25000	2144.70000

APPENDIX C FORTRAN ERROR MESSAGES

Compile-Time Error Messages

- 1 Fatal error reading source block
- 2 Nonnumeric characters in label field
- 3 Too many continuation lines
- 4 Fatal end of file encountered
- 5 Labeled continuation line
- 6 Missing field on \$ compiler directive line
- 7 Unable to open listing file specified on \$ compiler directive line
- 8 Unrecognizable \$ compiler directive
- 9 Input source file not valid textfile format
- 10 Maximum depth of include file nesting exceeded
- 11 Integer constant overflow
- 12 Error in real constant
- 13 Too many digits in constant
- 14 Identifier too long

15	Character constant extends to end of line	32	Incompatible arguments to EQUIVALENCE
16	Character constant zero length	33	Variable appears more than once in a type specification statement
17	Illegal character in input	34	This identifier has already been declared
18	Integer constant expected	35	This intrinsic function cannot be passed as an argument
19	Label expected	36	Identifier must be a variable
20	Error in label	37	Identifier must be a variable or the current FUNCTION
21	Type name expected (INTEGER, REAL, LOGICAL, or CHARACTER[*n])	38	'/' expected
22	Integer constant expected	39	Named COMMON block already saved
23	Extra characters at end of statement	40	Variable already appears in a COMMON block
24	'(' expected	41	Variables in two different COMMON blocks cannot be equivalenced
25	Letter IMPLICIT'ed more than once	42	Number of subscripts in EQUIVALENCE statement does not agree with variable declaration
26)' expected	43	EQUIVALENCE subscript out of range
27	Letter expected	44	Two distinct cells EQUIVALENCE'd to the same location in a COMMON block
28	Identifier expected	45	EQUIVALENCE statement extends a COMMON block in the negative direction
29	Dimension(s) required in DIMENSION statement		
30	Array dimensioned more than once		
31	Maximum of 3 dimensions in an array		

- | | | | |
|----|--|----|--|
| 46 | EQUIVALENCE statement forces a variable to two distinct locations, not in a COMMON block | 60 | Size of EQUIVALENCE'd CHARACTER items must be the same |
| 47 | Statement number expected | 61 | Illegal assignment - types do not match |
| 48 | Mixed CHARACTER and numeric items not allowed in same COMMON block | 62 | Can only call SUBROUTINES |
| 49 | CHARACTER items cannot be EQUIVALENCE'd with non-character items | 63 | Dummy parameters cannot appear in COMMON statements |
| 50 | Illegal symbol in expression | 64 | Dummy parameters cannot appear in EQUIVALENCE statements |
| 51 | Can't use SUBROUTINE name in an expression | 65 | Assumed-size array declarations can only be used for dummy arrays |
| 52 | Type of argument must be INTEGER or REAL | 66 | Adjustable-size array declarations can only be used for dummy arrays |
| 53 | Type of argument must be INTEGER, REAL, or CHARACTER | 67 | Assumed-size array dimension specifier must be last dimension |
| 54 | Types of comparisons must be compatible | 68 | Adjustable bound must be either parameter or in COMMON prior to appearance |
| 55 | Type of expression must be LOGICAL | 69 | Adjustable bound must be simple integer variable |
| 56 | Too many subscripts | 70 | Cannot have more than 1 main program |
| 57 | Too few subscripts | 71 | The size of a named COMMON must be the same in all procedures |
| 58 | Variable expected | 72 | Dummy arguments cannot appear in DATA statements |
| 59 | '=' expected | | |

100	Statement out of order	118	ELSE statement forbidden in this context
101	Unrecognizable statement	119	No matching IF for ELSE
102	Illegal jump into block	120	Unconditional GOTO forbidden in this context
103	Label already used for FORMAT	121	Assigned GOTO forbidden in this context
104	Label already defined	122	Block IF statement forbidden in this context
105	Jump to format label	123	Logical IF statement forbidden in this context
106	DO statement forbidden in this context	124	Arithmetic IF statement forbidden in this context
107	DO label must follow DO statement	125	',' expected
108	ENDIF forbidden in this context	126	Expression of wrong type
109	No matching IF for this ENDIF	127	RETURN forbidden in this context
110	Improperly nested DO block in IF block	128	STOP forbidden in this context
111	ELSEIF forbidden in this context	129	END forbidden in this context
112	No matching IF for ELSEIF	131	Label referenced but not defined
113	Improperly nested DO or ELSE block	132	DO or IF block not terminated
114	'(' expected	133	FORMAT statement not permitted in this context
115)' expected	134	FORMAT label already referenced
116	THEN expected		
117	Logical expression expected		

- | | | | |
|-----|--|-----|---|
| 135 | FORMAT must be labeled | 149 | Unrecognizable option in OPEN statement |
| 136 | Identifier expected | 150 | Direct access files must specify RECL= in OPEN statement |
| 137 | Integer variable expected | 151 | Adjustable arrays not allowed as I/O list elements |
| 138 | 'TO' expected | 152 | End of statement encountered in implied DO, expressions beginning with '(' not allowed as I/O list elements |
| 139 | Integer expression expected | 153 | Variable required as control for implied DO |
| 140 | Assigned GOTO but no ASSIGN statements | 154 | Expressions not allowed as reading I/O list elements |
| 141 | Unrecognizable character constant as option | 155 | REC= option appears twice in statement |
| 142 | Character constant expected as option | 156 | REC= expects integer expression |
| 143 | Integer expression expected for unit designation | 157 | END= option only allowed in READ statement |
| 144 | STATUS option expected after ',' in CLOSE statement | 158 | END= option appears twice in statement |
| 145 | Character expression as filename in OPEN | 159 | Unrecognizable I/O unit |
| 146 | FILE= option must be present in OPEN statement | 160 | Unrecognizable format in I/O statement |
| 147 | RECL= option specified twice in OPEN statement | 161 | Options expected after ',' in I/O statement |
| 148 | Integer expression expected for RECL= option in OPEN statement | 162 | Unrecognizable I/O list element |

163	Label used as format but not defined in format statement	208	There is no such unit in this \$USES file
164	Integer variable used as assigned format but no ASSIGN statements	209	Missing UNIT name in \$USES statement
165	Label of an executable statement used as a format	210	Extra characters at end of \$USES directive
166	Integer variable expected for assigned format	211	Intrinsic units cannot be overlaid
167	Label defined more than once as format	212	Syntax error in \$EXT directive
169	Function with no parameters needs null parameter list () even when there are no arguments	213	A SUBROUTINE cannot have a type
200	Error in reading \$USES file	214	SUBROUTINE/FUNCTION name in \$EXT directive has already been defined
201	Syntax error in \$USES file	400	Code file write error
202	SUBROUTINE/FUNCTION name in \$USES file has already been declared	401	Too many entries in JTAB
203	FUNCTIONS cannot return values of type CHARACTER	402	Too many SUBROUTINES/FUNCTIONS in segment
204	Unable to open \$USES file	403	Procedure too large (code buffer too small)
205	Too many \$USES statements	404	Insufficient room for scratch file on system disk
206	No .TEXT info for this unit in \$USES file	405	Read error on scratch file
207	Illegal segment kind in \$USES file		

Run-Time Error Messages

600	Format missing final ')'	615	Hollerith field in format requires repetition factor
601	Sign not expected in input	616	X field in format requires repetition factor
602	Sign not followed by digit in input	617	P field in format requires repetition factor
603	Digit expected in input	618	Integer appears before '+' or '-' in format
604	Missing N or Z after B in format	619	Integer expected after '+' or '-' in format
605	Unexpected character in format	620	P format expected after signed repetition factor in format
606	Zero repetition factor in format not allowed	621	Maximum nesting level for formats exceeded
607	Integer expected for w field in format	622)' has repetition factor in format
608	Positive integer required for w field in format	623	Integer followed by ',' illegal in format
609	'.' expected in format	624	'.' is illegal format control character
610	Integer expected for d field in format	625	Character constant must not appear in format for reading
611	Integer expected for e field in format	626	Character constant in format must not be repeated
612	Positive integer required for e field in format	627	'/' in format must not be repeated
613	Positive integer required for w field in A format	628	'\' in format must not be repeated
614	Hollerith field in format must not appear for reading		

629 BN or BZ format control must not be repeated

630 Attempt to perform I/O on unknown unit number

631 Formatted I/O attempted on file opened as unformatted

632 Format fails to begin with '('

633 I format expected for integer read

634 F or E format expected for real read

635 Two '(' characters in formatted real read

636 Digit expected in formatted real read

637 L format expected for logical read

639 T or F expected in logical read

640 A format expected for character read

641 I format expected for integer write

642 w field in F format not greater than d field + 1

643 Scale factor out of range of d field in E format

644 E or F format expected for real write

645 L format expected for logical write

646 A format expected for character write

647 Attempt to do unformatted I/O to a unit opened as formatted

648 Unable to write blocked output, possibly no room on device for file

649 Unable to read blocked input

650 Error in formatted textfile, no <cr> in last 512 bytes

651 Integer overflow on input

652 Too many bytes read out of direct access unit record

653 Incorrect number of bytes read from a direct access unit record

654 Attempt to open direct access unit on unblocked device

655 Attempt to do external I/O on a unit beyond end of file record

656 Attempt to position a unit for direct access on a nonpositive record number

657 Attempt to do direct access to a unit opened as sequential

658 Attempt to position direct access unit on unblocked device

- 659 Attempt to position direct access unit beyond end of file for reading
- 660 Attempt to backspace unit connected to unblocked device
- 661 Attempt to backspace sequential, unformatted unit
- 662 Argument to ASIN or ACOS out of bounds (ABS(X) .GT. 1.0)
- 663 Argument to SIN or COS too large (ABS(X) .GT. 10E6)
- 664 Attempt to do unformatted I/O to internal unit
- 665 Attempt to put more than one record into internal unit
- 666 Attempt to write more characters to internal unit than its length
- 667 EOF called on unknown unit
- 697 Integer variable not currently assigned a format label
- 698 End of file encountered on read with no END= option
- 699 Integer variable not ASSIGNED a label used in assigned goto
- 1000+ Compiler debug error messages - should never appear in correct programs

INDEX

- A -

- A (character editing)..... 1
- ALFAn types..... 1
- Apostrophe editing..... 1
- Arithmetic expressions..... 1
- Arithmetic IF statement..... 1
- Arithmetic operators..... 1
- Array..... 1
- Array element name..... 1
- Assigned GOTO statement..... 1
- Assignment statements
 - computational..... 1
 - computational, automatic conversion..... 1
 - general..... 1
 - label..... 1

- B -

- Backslash edit control..... 1
- BACKSPACE..... 11-
- BACKSPACE, not used..... 11-
- BACKSPACE statement..... 11-
- Blanks, source program..... 11-
- Block IF control statement..... 10

- C -

Calling Pascal routines	14-7
CALL statement	13-3, 13-5
Character expressions	8-6
CHARACTER type statement	6-8
CLOSE statement	11-22
Codefile	2-9
Comment lines	3-7
COMMON statement	6-9
COMMON statements	13-12
Compilation units	14-3
Compiler directives	
\$EXT	3-7, 14-12
extensions to	A-6
\$INCLUDE	2-6, 3-6
lines	3-6
\$USES	3-6
\$XREF	2-8, 3-7
Compiler listing	
errors	2-7
global symbol table	2-8
local symbol table	2-8
sample	2-6
Compiling	2-3
Computational assignment statements	9-3
Computed GOTO statement	10-4
CONTINUE control statement	10-14
Control statements	
arithmetic IF	10-5
assigned GOTO	10-4
block IF	10-6
block IF syntax	10-9
computed GOTO	10-4
CONTINUE	10-14
DO	10-11
ELSEIF	10-9
END	10-15

ENDIF	10
general	1
logical IF	1
PAUSE	10
STOP	10
unconditional GOTO	1
\$ control statements	1

- D -

DATA statements	3-10, 1
Data types	
basic	4
character	4
integer	4
logical	4
real	4
Dimension declarators	6
DIMENSION statement	6
Direct files	6
DO control statement	11-1
DO loop	10-1
DO variable	10-1
DO variable, example	10-1
DO variable expressions	A-

- E -

Edit descriptors	12-
ELSEIF control statement	10-1
END control statement	10-1
ENDFILE statement	11-2
ENDIF control statement	10-11
END statement	3-9, 3-10, 3-12
EOF intrinsic function	A-8
EQUIVALENCE statement	6-12, 13-12

E (real editing) 12-13

Error messages

- compile time A-13
- general A-13
- run-time A-26

Executing a program 2-5

Expression, computed GOTO A-5

Expressions

- arithmetic 8-3
- arithmetic, different operands 8-5
- arithmetic, integer division 8-5
- arithmetic operators 8-3
- character 8-6
- evaluation of 8-10
- logical 8-8
- logical operators 8-9
- logical operators, precedences 8-9
- operator precedence 8-10
- relational 8-6
- relational operators 8-7

Expressions in I/O list A-5

\$EXT compiler directive 14-12

Extensions to standard

- backslash edit control A-7
- EOF intrinsic function A-8
- general A-6
- lowercase input A-8

EXTERNAL statement 6-10

\$EXT statement 14-4

- F -

Features, unsupported A-3

Format, program units 13-3

Format specifications

- edit descriptors 12-7
- general 12-3

- input/output list 12-5
- interactions 12-5

Format specifier ('f') 11-17

FORMAT statements 12-3

Formatted I/O 12-3

Formatted/unformatted files 11-7

FORTRAN

- character set 3-3
- compilation 14-3
- 77 Compiler 2-3
- error messages A-13
- I/O System 11-3, 11-10
- name or identifier 5-3
- program 3-3
- sample program A-9
- scope of names 5-3
- undeclared names 5-5

FORTRAN 77 1-3

F (real editing) 12-12

Full-language features

- computed GOTO expression A-5
- expressions in I/O list A-5
- general A-4
- generalized I/O A-5
- subscript expressions A-4
- unit I/O number A-5

Functions

- external 13-7
- general 13-6
- intrinsic 13-9
- parameters 13-12
- RETURN statement 13-11
- statement functions 13-10

FUNCTION statement 3-10

- H -

Hollerith editing 12-8

- I -

Identifier, FORTRAN 5-3
 I (integer editing) 12-12
 IMPLICIT statement 3-11, 6-3
 Implied DO lists 11-19
 Initial lines 3-9
 Input entities 11-18
 Input/output list 11-17, 12-5
 Input program form 2-5
 Internal files 11-8
 INTRINSIC statement 6-11
 I/O statements 11-20
 BACKSPACE 11-26
 CLOSE 11-22
 elements 11-16
 ENDFILE 11-26
 format specifier ('f') 11-17
 general 11-15
 implied DO lists 11-19
 input entities 11-18
 input/output list 11-17
 OPEN 11-20
 output entities 11-18
 READ 11-23
 restriction 11-27
 REWIND 11-27
 unit specifier ('u') 11-16
 WRITE 11-25
 I/O System
 BACKSPACE 11-14
 common file operations 11-11
 concepts/limitations 11-10

direct files 11-14
 file name 11-6
 file position 11-6
 file properties 11-5
 files 11-5
 formatted/unformatted files 11-6
 function side effects 11-15
 general 11-3
 internal files 11-8
 internal files, properties 11-8
 I/O statements 11-15, 11-20
 limitations 11-14
 other file operations 11-12
 overview 11-3
 records 11-4
 sequential/direct access 11-7
 units 11-9

- L -

Label assignment statement 9-4
 Labels 3-8
 Lines, continuation 3-9
 Linking Pascal/FORTRAN 14-7
 L (logical editing) 12-14
 Logical expressions 8-8
 Logical IF statement 10-6
 Logical operators 8-9
 Lowercase input A-8

- M -

Main program 13-3
 MARK 14-7

Index

- N -

Nonrepeatable edit descriptors	
apostrophe editing	12-7
H (Hollerith editing)	12-8
/ (slash editing)	12-8
X (positional editing)	12-8
Notational conventions	1-4
Numeric editing	12-11

- O -

OPEN statement	11-20
Output entities	11-18
Overview, Manual	1-3

- P -

Parameters	13-12
Partial compilation	14-3
PAUSE control statement	10-15
Program columns	3-4
Program, sample	A-9
PROGRAM statement	13-3
Program statement	3-10
Program units	3-10

- R -

READ statement	11-23, 12-3
Records	11-4
Relational expressions	8-6
Relational operators	8-7
RELEASE	14-7
Repeatable edit descriptors	

A (character editing)	12-15
E (real editing)	12-13
F (real editing)	12-12
I (integer editing)	12-12
L (logical editing)	12-14
numeric editing	12-11
RETURN statement	13-3
REWIND statement	11-27
RTUNIT.CODE	2-4
Runtime support	2-4

- S -

Sample program	A-9
SAVE statement	6-12
Screen Oriented Editor	2-5
Segments	14-3
Separate compilation	14-6
Sequential/direct access	11-7
Slash editing	12-8
Source lines	3-4
Source lines, allowed	4-6
Specification statements	
CHARACTER	6-8
COMMON	6-9
DIMENSION	6-5
EQUIVALENCE	6-12
EQUIVALENCE, restrictions	6-13
EXTERNAL	6-10
IMPLICIT	3-11, 6-3
INTRINSIC	6-11
SAVE	6-12
Statement ordering	
within program unit	3-10
Statements	3-8
DATA	3-10, 7-3
END	3-9, 3-10, 3-12

Index

\$EXT. 14-4
 FUNCTION. 3-10
 general. 3-9
 INTRINSIC. 13-9
 ordering, general. 3-10
 program units. 3-10
 SUBROUTINE. 3-10
 \$USES. 14-4, 14-6
 \$USES A. 14-6
 \$USES B. 14-6
 \$USES F. 14-6
 \$USES UX statement. 14-4
 Statements, specification. 6-3
 STOP control statement. 10-14
 Subroutines
 CALL statement. 13-5
 general. 13-4
 parameters. 13-12
 SUBROUTINE statement. 13-4
 SUBROUTINE statement. 3-10, 13-4
 Subscript expressions. A-4

- T -

Type statements 6-7

- U -

Unconditional GOTO statement. 10-3
 Unit I/O number. A-5
 Units. 11-9, 14-3
 Unit specifier ('u'). 11-16
 \$USES A statement. 14-6
 \$USES B statement. 14-6
 \$USES compiler directive. 14-5
 \$USES F statement. 14-6

\$USES statement. 14-4, 14-6
 \$USES UX statement. 14-4

- W -

WRITE statement. 11-25, 12-3

- X -

X (positional editing). 12-8