DataGeneral

Customer Documentation

FORTRAN 77 Reference Manual

FORTRAN 77
Reference Manual

093-000162-03

For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085-series) supplied with the software.

Ordering No. 093-000162

Printed in the United States of America
Revision 03, July 1985

NOTICE

FORTRAN 77
Reference Manual
093-000162

Revision History:

Original Release - December 1980

First Revision - March 1981

Second Revision - October 1983

Third Revision - July 1985

Effective with:

AOS, AOS/VS, MP/AOS, MP/AOS-SU, DG/UX, FORTRAN 77 and F7716 Rev. 3.00

A vertical bar or an asterisk in the margin of a page indicates substantive change or deletion, respectively from the previous revision. Appendixes F and G contain no change bars; they are new with this revision.

Preface

FORTRAN originated in 1954. Ever since, it has been the workhorse language for number manipulation: straightforward, unpretentious, and reliable. With the advent of FORTRAN 77 — as named in the American National Standards Institute's (ANSI) 1978 standard — the language that is renowned for numbers handles characters as well.

This manual describes Data General's FORTRAN 77, which includes the *full* ANSI FORTRAN standard and has numerous extensions. FORTRAN 77 (called F77 in this book) differs from earlier FORTRANs in several important ways, including the character data type, block IF statement, and enhanced I/O capabilities.

This manual assumes that you have some experience with FORTRAN. It is primarily a reference manual, but gives some introductory and conceptual information in Chapters 1–7.

For easy reference *we have repeated material* wherever it pertains. Thus, you may find the same information in several different places.

The manual is organized as follows:

Chapter 1	Covers some points in the ANSI standard and lists the extensions that DG's F77 offers to this standard; gives the F77 character set; covers the basic design of an F77 program; and includes a sample program.
Chapter 2	Explains F77 components and entities: data types, symbolic names, variables and arrays, and expressions.
Chapter 3	Describes assignment of data types and values: arithmetic, logical, and character.
Chapter 4	Covers flow of control within a program unit: changing and directing it with GO TO, IF, DO, and pausing and ending the program.
Chapter 5	Describes input and output: units, files, records, and format identifiers; control and I/O lists; data transfers via READ, WRITE, PRINT statements; the I/O statements OPEN, INQUIRE, CLOSE; and file positioning.
Chapter 6	Details format specifications: edit-directed formatting via FMT= and FORMAT; all F77 edit descriptors; and F77's new list-directed formatting.
Chapter 7	Explores subprograms: intrinsic functions supplied with F77, statement functions, function subprograms, and subroutine subprograms — with interprogram communication via arguments and the COMMON statement.
Chapter 8	Is the quick reference chapter. It lists all F77 statements alphabetically, summarizing what they do, and includes examples.
Chapter 9	Tells you how to build F77 programs — compile, link, and execute them.
Appendix A	Is an ASCII-octal-decimal-EBCDIC translation table.
Appendix B	Shows the syntax of all F77 statements and directives in the form of railroad charts.
Appendix C	Describes how DG's F77 differs from its FORTRAN 5; this will interest you if you know FORTRAN 5 and want to convert FORTRAN 5 programs to F77 programs.
Appendix D	Describes the F77PRINT postprocessor, which allows you to print certain types of F77-created files easily.
Appendix E	Summarizes the differences between AOS/VS F77, AOS F77, AOS F77 under AOS/VS (F7716), MP/AOS F77, and MP/AOS-SU F77. Read this if you are accustomed to F77 on one operating system and want to use it on another system.

Appendix F	describes the differences between DG/UX™ F77 and the other F77s. DG/UX F77 programmers should read this appendix first.
Appendix G	lists the differences between VAX-11 FORTRAN and AOS/VS F77. This appendix also explains the conversion of VAX-11 FORTRAN software to AOS/VS F77 software.

You need not read this whole manual; for conceptual or introductory material on an item, you can examine the item in one of Chapters 1–7. For pure reference, find the statement in Chapter 8. To *build* your F77 source program units into executable programs, read Chapter 9.

MP/AOS F77 and MP/AOS-SU F77 are now supported products. In this manual, most references to AOS F77 also apply to MP/AOS F77 and MP/AOS-SU F77. Any exceptions are explicitly noted.

What Other Books Do You Need?

This — the F77 Reference Manual — gives all the information you need to write, and the general information you need to build, F77 programs.

There are two other books you might want to examine. The first is the *SWAT Debugger* (high-level) *User's Manual*, 093-000258. The other is the F77 Environment Manual that applies to your operating system. AOS/VS F77 users should obtain the *FORTRAN 77 Environment Manual (AOS/VS)*, 093-000288. AOS and F7716 users should obtain the *FORTRAN 77 Environment Manual (AOS)*, 093-000273. MP/AOS and MP/AOS-SU F77 programmers should check their F77 Release and Update Notices for the announcement of their Environment Manual. The current Release or Update Notice for your operating system's FORTRAN 77 contains the latest list of manuals that relate to the software.

In addition, Data General *strongly* recommends that you have the Software Release Notices and Update Notices for FORTRAN 77 and related software. These Notices may contain corrections to this manual and additional information beyond the scope of this manual. For example, the documentation for F77's handling of overflow and underflow appeared in Release Notices before this revised manual was created. And, the Notices may contain suggestions for corrections or adjustments to current software problems.

Reader, Please Note:

We use shading in the text to indicate formal DG extensions to the ANSI standard; for example, feature. The shading convention applies throughout *except for program examples.*

The shading exists *only* to help you write standard conforming source programs. Ignore the shading if standard programs are not essential. Only DG features that inherently produce a nonconforming program are shaded; for example, the INTEGER*2 type statement. If a DG feature is not mentioned in the standard but the feature will not, by itself, produce a nonstandard program, the feature is not shaded. For example, compiler switches in Chapter 9 are not shaded because they don't affect the syntax of the source program.

We use these conventions for command formats in this manual:

COMMAND required *[optional]* ...

Where	Means
COMMAND	You must enter the command (or its accepted abbreviation) as shown.
required	You must enter some argument (such as a filename). Sometimes, we use: $\left\{ \begin{matrix} \text{required}_1 \\ \text{required}_2 \end{matrix} \right\}$ which means you must enter *one* of the arguments. Don't enter the braces; they only set off the choice.
[optional]	You have the option of entering this argument. Don't enter the brackets; they only set off what's optional.
...	You may repeat the preceding entry or entries. The explanation will tell you exactly what you may repeat.

Additionally, we use certain symbols in special ways:

Symbol	Means
↵	Press the NEW LINE or carriage return (CR) key on your terminal's keyboard.
□	Be sure to put a space here. (We use this only when we must; normally, you can see where to put spaces.)

All numbers are decimal unless we indicate otherwise; e.g., 35_8.

Finally, in examples we use

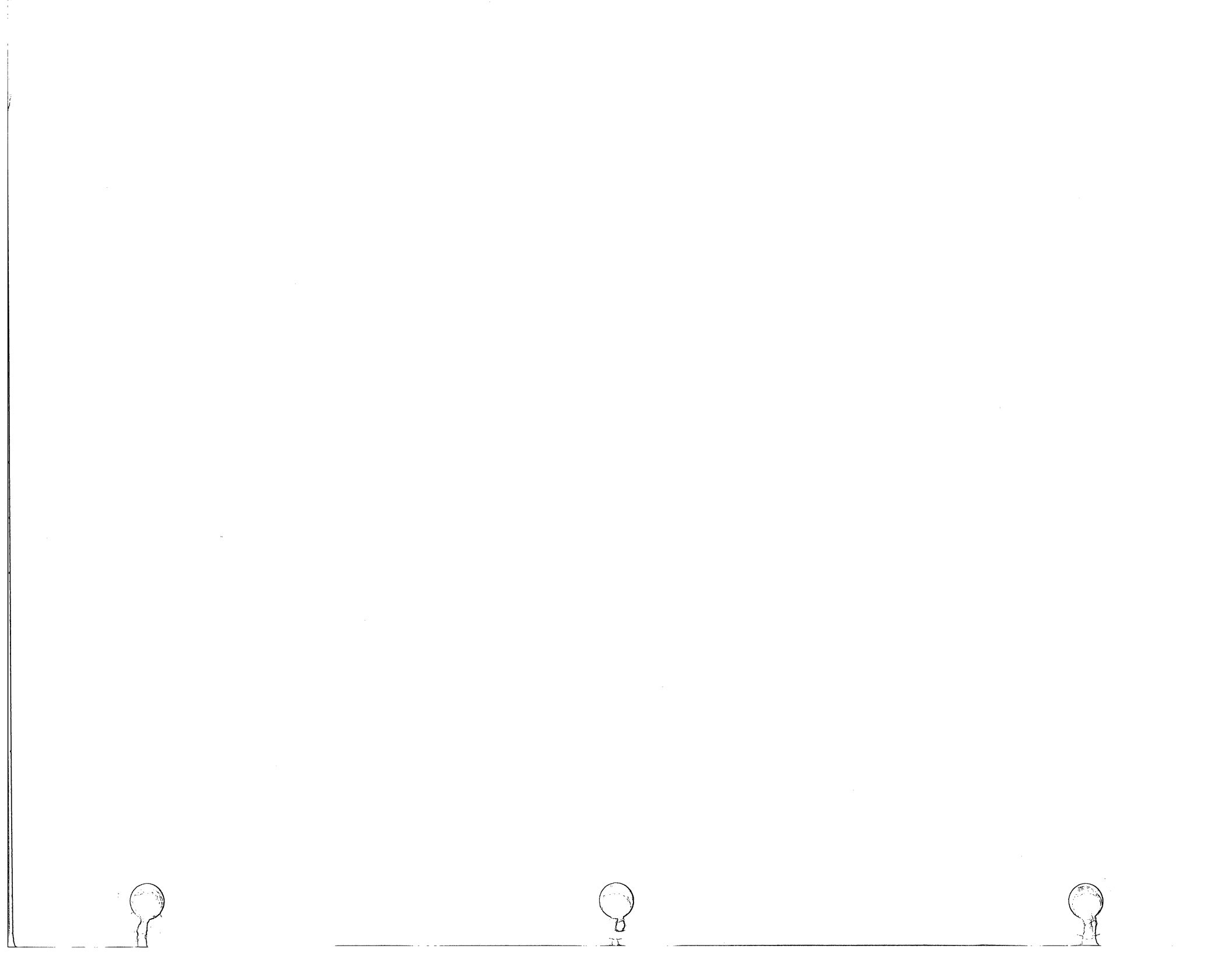
THIS TYPEFACE TO SHOW YOUR ENTRY)
THIS TYPEFACE FOR SYSTEM QUERIES AND RESPONSES.

) is the CLI prompt.

Contacting Data General

- If you have comments on this manual, please use the prepaid Remarks Form that appears after the Index. We want to know what you like and dislike about this manual.
- If you need additional manuals, please use the enclosed TIPS order form (USA only) or contact your Data General sales representative. *

End of Preface



Contents

Chapter 1 - About FORTRAN 77

Chapter 2 - FORTRAN Components

Chapter 3 - Defining Data Types, Values, and Constants

Chapter 4 - Directing Flow within a Program Unit

Chapter 5 - Input and Output

Chapter 6 - Format Specification (FMT= and FORMAT)

Chapter 7 - Functions and Subroutines

Chapter 8 - Statements and Directives, Alphabetically

Chapter 9 - Building F77 Programs

Appendix A - ASCII Character Set

Appendix B - Syntax Charts of F77 Statements and Directives

Appendix C - How DG's FORTRAN 77 Differs from FORTRAN 5

Appendix D - Running the F77PRINT Postprocessor

Appendix E - Differences Between AOS/VS (32-bit) F77, and AOS, F7716, MP/AOS, and MP/AOS-SU (16-bit) F77s

Appendix F - DG/UX F77 Differences

Appendix G - DEC VAX-11 FORTRAN and DG AOS/VS F77 Compatibility

Tables

Table

Illustrations

Figure

Chapter 1
About FORTRAN 77

Data General's FORTRAN 77 (called F77 in this manual) conforms to the *full* ANSI standard, as documented in ANSI publication X3.9-1978, often called FORTRAN 77.

This chapter defines some points in ANSI's standard and describes DG's extensions to this standard. Then it introduces F77 itself. The main sections proceed:

- Some Terms from the ANSI Standard
- DG's Extensions to ANSI Standard F77
- Program Units
- Parts of a Program Unit
- General Order of Statements in a Program Unit
- F77 Character Set
- Compiler Features
- %INCLUDE and %LIST Directives
- On-Line HELP Files
- Building F77 Programs
- F77 Program Example

Some Terms from the ANSI Standard

The ANSI F77 standard formalizes extensions that many manufacturers have made to FORTRAN since ANSI's last (1966) FORTRAN standard. The standard's goal is to make FORTRAN sources transportable between computer systems — a very attractive prospect which promises to make F77 *the* FORTRAN for at least the first half of the 80s.

Certain terms in the ANSI standard and general definitions are important to this book. These terms are:

conformance — The standard tries to avoid defining a processor; instead, it emphasizes *program* conformity. "An executable program conforms to this standard if it uses only those forms and relationships described herein [within the standard] and if the executable program has an interpretation according to this standard."

Programs written in DG's F77 — without DG's extensions to the standard — and built and executed on an appropriate DG system *are* standard conforming. Such programs will compile and run on any other ANSI standard conforming processor.

processor — The standard defines a processor as "The combination of a data processing system and the mechanism by which programs are transformed for use on that data processing system..." This book assumes an appropriate ECLIPSE® computer and operating system as the "data processing system"; it refers to these as "the system." The "...mechanism by which programs are transformed for use..." is primarily DG's F77 compiler but can also be system programs called the Command Line Interpreter (CLI) and the Link utility. This manual refers to this software respectively as "the F77 compiler" or "the compiler" or "F77", "the CLI", and "Link". In other words, DG's "processor" has components that we mention by name.

storage unit — The "standard" numeric storage unit for integer, real, and logical data types is 32 bits (4 bytes); it is the same for all three types, as specified by the standard. For the character data type, it is 1 byte per character. As an extension, DG's F77 allows short (2-byte) integer and logical types, 1-byte logical types (AOS/VS), 8-byte real types, and 16-byte complex types. You can specify length by type statement and/or — for integer and some logical types — by compiler switches.

DG's Extensions to ANSI Standard F77

This section summarizes the extensions that DG's F77 offers to the ANSI standard, chapter by chapter in this book. Numbers in parentheses, such as (3.2.1), refer to a section in the ANSI standard.

Chapter 1 — About FORTRAN 77

1. Comments (3.2.1). You can place a comment on the same line as a statement by following the statement with an exclamation point (!); e.g.,

   ```
   DO 50 I = 1, 10          ! Start DO loop.
   ```

   A comment can occupy any or all character positions between the exclamation point and the end of the line. This book often uses such comments to clarify examples. You can also use an exclamation point in column 1 or in any column beyond 6 to start a comment line.

2. Statements, Comments, and Lines (2.3). F77 allows tabs in source lines, treating tabs and spaces interchangeably, except for the first tab.

3. Character Set (3.1). Quotation marks (") that delimit character strings are treated the same way as apostrophes ('); for example,

   `PRINT *, "Hello"` and `PRINT *, 'Hello'`

   are equally acceptable to the compiler.

   The F77 compiler is case insensitive. It accepts lowercase keywords and symbol names; for example,

   `go to 50` is identical to `GO TO 50`

   and

   `rarray(10)` is identical to `RARRAY(10)`

   in terms of the compiler's response.

4. An %INCLUDE directive or INCLUDE statement lets you instruct the F77 compiler to include and compile other files as part of the current source program. A %LIST directive allows you to suppress and restore compiler output to the listing file.

Chapter 2 — FORTRAN Components

1. You can represent an octal constant as nK, where n is a string of octal digits. An example is 377K (=255). You can also represent an octal constant as O'n', where n is a string of octal digits. An example is O'357'. However, O"357" is illegal. Similarly, you can write a hexadecimal constant as Z'n', where n is a string of hexadecimal digits. An example is Z'1A' (=26).

2. Integer, real, and logical data types have the same storage unit size, as specified by the standard. This size is 32 bits (4 bytes). However, you can specify 16-bit (2-byte) integer or logical types in either (or both) of two ways:

   a. Globally, when you issue the compile command, via a switch; for example,

      F77/INTEGER=2 MYPROG ↵

      The switch sets the default length, which individual type statements in the program(s) can override. For AOS, MP/AOS, and MP/AOS-SU, the compilation macro supplied with F77 (named F77.CLI) includes /INTEGER=2 and /LOGICAL=2 switches. For F7716, its compilation macro (named F7716.CLI) also includes /INTEGER=2 and /LOGICAL=2 switches.

   b. Selectively, by appending *2 to the type declaration in a program unit; e.g.,

      ```
      INTEGER*2 IARRAY(20)
      ```

      F77 also allows a type*4 declaration in a program unit to override an /INTEGER=2 global switch setting for a compilation.

   You can specify an 8-bit (1-byte) logical data type in AOS/VS F77 programs. This data type also holds small integers and single characters.

3. F77 will give an integer constant in a source program 4 bytes of storage if
   - it is too large to fit in 2 bytes (such as 45863), *or*
   - it contains more than 5 digits (even 000000), *or*
   - the compiler /INTEGER=2 switch is omitted.

4. You can include control codes within angle brackets in character strings; e.g., '<BEL>' sounds the terminal beeper.

5. Symbolic Names (2.2). F77 allows symbolic names of up to 32 letters, numbers, the underscore character (_), or the question mark character (?). Each symbolic name must begin with a letter or a question mark.

Chapter 3 — Defining Data Types, Values, and Constants

1. DG's F77 allows data initialization in type statements; e.g.,

   ```
   INTEGER FOO /445/
   ```

2. You can also specify byte length in type statements, as described in Chapter 2; e.g.,

   ```
   INTEGER*2 SUM_3D
   LOGICAL*1 COUNT(5)     ! same as BYTE COUNT(5)
                          !    under AOS/VS
   ```

3. You *can* concatenate a character entity whose length you declare with an asterisk in parentheses just as you can concatenate any other character entity. Such a character entity must be in a subprogram and in the subprogram's argument list. For example:

```
SUBROUTINE CAT_STRING (GOOD_MORN)
CHARACTER*(*) GOOD_MORN
PRINT *, GOOD_MORN // 'ING'
RETURN
END
```

4. You can specify an 8-bit (1-byte) logical data type under AOS/VS. This data type contains logical data, numeric (integer) data, and character data. For example, the following statements are legal.

```
LOGICAL*1 L1A, L1B, L1C
L1A = .TRUE.      ! L1A contains 11111111
L1B = 26          ! L1B contains 00011010
L1C = 'E'         ! L1C contains 01000101
```

Chapter 4 — Directing Flow within a Program Unit

1. DG's F77 treats an assigned GO TO statement as an unconditional GO TO statement; i.e., it does not check the statement label list for validity.
2. DG's F77 allows DO WHILE, END DO, and unlabelled DO statements.

Chapter 5 — Input and Output

1. The control information list (cilist) for data transfer and the OPEN and INQUIRE specifier lists offer the following extensions, all of which you can default:

Extension	Brief Comment
BLOCKSIZE	Disk block size.
CARRIAGECONTROL	Output carriage control for printing.
DELIMITER	Input data-sensitive delimiter control.
ERRORLOG	Specify an error log file.
EXCLUSIVE	Exclusive access to file.
FORCE	Avoid F77 and system I/O buffering.
IOINTENT	Open unit for input, output, or both.
MAXRECL	Set maximum record length.
MODE	Select text or binary transfer.
PAD	Pad records to a fixed size.
POSITION	Set initial file position.
RECFM	Operating system record format.
RETURNRECL	Return length of record written/read.
SCREENEDIT	Allow screen editing control characters.
TYPE	Operating system file type.

MP/AOS and MP/AOS-SU systems don't accept BLOCKSIZE and MODE.

Chapter 6 — Format Specification (FMT= and FORMAT)

1. Edit descriptors. You can use the edit descriptors Ow*[.m]* for octal, Zw*[.m]* for hexadecimal, and Bw*[.m]* for binary; or $ (depending on carriage control) to suppress output of a NEW LINE character. Descriptors O, Z, and B have the same form as Iw*[.m]*.
2. FORMAT expressions in AOS/VS. You can use expressions with certain edit descriptors. For example, the edit descriptor I<J+2> is valid. If J is 3 at runtime, this edit descriptor has the same effect as I5.
3. FORMAT/entity mismatch. If you've given the /FMTMM switch to the Link macro (F77LINK.CLI), then F77 outputs certain variables and arrays whose types do not match their edit descriptors. For example, trying to output the real*4 number 63.8 with the edit descriptor I6 results, by default, in a runtime error. If instead you give the /FMTMM switch to F77LINK, then the result is the string of six characters □□□□63. (Here, F77 converts 63.8 to an integer*4 number and outputs the resulting number.)

Chapter 7 — Functions and Subroutines

1. F77 offers a group of word and bit manipulation routines, including BTEST, IAND, IBCLR, IBITS, IBSET, IEOR, IOR, ISHFT, ISHFTC, IXOR, MVBITS, and NOT. MVBITS is a subroutine instead of a function.
2. Within any program unit, you may use a DATA statement to initialize entities in blank common.
3. F77 saves named common blocks — as well as blank common blocks — upon return from subprograms. It also saves the current values of variables and arrays initialized via DATA or type statements upon return.
4. F77 allows both character and noncharacter data within the same common block.

Chapter 8 — Statements and Directives, Alphabetically

1. This chapter includes all statements in DG's F77. Because the extensions have been summarized by chapter and category above, we don't repeat them here.

Chapter 9 — Building F77 Programs

1. The material in this chapter describes operating procedures that are outside the scope of the standard. The concept of "extensions to the standard" doesn't apply to this chapter.

Program Units

An F77 program contains one or more program units. A *program unit* is a sequence of statements with optional comment lines; it is either a main program or a subprogram. You can write and compile program units as separate source files or as parts of one source file; after compilation, you link the desired unit(s) into an executable program.

A main program unit does not contain a FUNCTION, SUBROUTINE, or BLOCK DATA statement; it may start with a PROGRAM statement, but it need not. An F77 program can have only one main program unit.

A subprogram unit contains a FUNCTION, SUBROUTINE, or BLOCK DATA statement. These statements define a user function subprogram, user subroutine subprogram, or user block data subprogram respectively.

Parts of a Program Unit

A program unit contains statements and optional comment lines and is terminated by an END statement. An END statement is a line that consists solely of the letters E N D in the statement field.

Statements and Lines

A statement is either executable or nonexecutable. Executable statements specify actions that occur at runtime. Nonexecutable statements describe the characteristics, arrangement, and initial values of variables; contain editing information; specify statement functions; and classify program units.

A *statement* must start at character position 7 or beyond, and it can extend up to character position 136. You can press the tab key once to tab to position 9, or you can press the space bar 6 or more times. After typing a statement label, you can tab to position 9.

If the record is in card format, the statement must terminate at or before character position 72. You can put sequence numbers (or any information you like) in columns 73 through 80. You must use *spaces* (not tabs) to space from the end of the statement to column 73.

You write statements in *lines*. The first line of a statement is the initial line, and you must put either a 0 or a blank in character position 6 to indicate an initial line. If a statement is too long to fit on one line, follow the initial line with continuation lines, for which you must put a character other than 0 or blank in character position 6. F77 executes all executable lines sequentially, from the beginning of the program unit to the end, unless a statement like GO TO redirects control.

You may identify any executable statement with a statement label so that other statements can refer to it. You can label a nonexecutable statement, but no other statement can refer to the label.

Blanks (Spaces) and Tabs

Blanks and tabs are not significant to F77 except in

- columns 1–6;
- character constants; or
- apostrophe, quotation mark, or H-format edit descriptors.

Outside of these places, you can use tabs and blanks at will for clarity and readability.

Comment Lines (C, *, !)

A line that has a C, c, asterisk (*) or exclamation point (!) in character position 1 is a *comment line*. The comment text can start anywhere after the comment indicator.

A comment line is not a statement and has no effect on the execution of the program. It provides documentation that makes a program easier to read and understand. For example:

```
      ...
      ...
C     Call subroutine to compute taxes.

      CALL MARRIED(TAXABLE, MARITAL_STATUS, TAX)

C     Tax is in TAX.
      ...
      ...
```

Comments on a Statement Line (!)

By default, a NEW LINE character <NL> ends each statement line. (In card format, column 72 ends each statement line.) But DG's F77 also allows an exclamation point (!) to end a line. This provides a handy way to comment each statement. The comment can start at column 1 or from character position 7 onward. For example,

```
DO 50   I = 1, 10    ! Start Loop.
```

Exclamation points within character strings (within apostrophes or quotation marks) and within Hollerith constants are maintained as part of the lines that contain them. An exclamation point in column 6 indicates a continuation line. An exclamation point that appears in columns 2 through 5 will cause an "Illegal label" error.

Blank Lines

Blank lines are allowed within a program unit by the standard. Like comment lines, they do not affect program execution.

Statement Labels

If character position 1 does not contain a C, c, *, !, or a % to start a compiler directive, F77 reserves character positions 1 through 5 for a statement label. (Only a % in character position 1 starts a compiler directive.) The label must be an unsigned positive integer of 1 to 5 digits. You can place the label anywhere in character positions 1 through 5. Leading 0s are not significant; e.g., 12 and 00012 are treated as the same label. Embedded blanks are ignored; e.g., 2□□0 is interpreted as 20.

F77 ignores statement labels on specification statements and on statements to which you do not refer. If more than one statement has the same label, or if a continuation line is labelled, the compiler signals an error. Some valid and invalid statement labels are

Valid	Invalid	Because
0099	0	Zero not allowed.
98990	2344510	Too many digits.
4	A26	An alpha character.

Statement and Comment Examples

Some examples of executable and nonexecutable statements, continuation lines, and labelled statements are

```
C   The following 4 statements are nonexecutable.

      SUBROUTINE   IQ             ! Subroutine.
      dimension   INDEX (200)     ! Dimension.
      CHARACTER*20   TX(10)       ! Type and
                                  !  dimension
                                  !  declaration.
      common  RARY(100), MON      ! Common and
                                  !  dimension
                                  !  declaration.

C   These are executable:

      PRINT *, 'Hello.'           ! I/O.
      if (I .GT. J) go to 88      ! If.
      IF ( I .GT. J   .OR.        ! If.
     +     J .LE. 7 )
     +                      SUM_3  = 0.0
      I = J + 2                   ! Assignment.
      do   40    I = 1, N         ! DO.
```

General Order of Statements in a Program Unit

F77 requires certain kinds of nonexecutable statements to precede executable statements within a program. Statements that identify a program unit — as PROGRAM and SUBROUTINE do — must come first, and these must be followed by any *specification statements* the program requires. The specification statements are IMPLICIT, DIMENSION, type statements, PARAMETER, EQUIVALENCE, INTRINSIC, EXTERNAL, SAVE, and COMMON. Figure 1-1 shows the general order required for statements in an F77 program unit.

Within Figure 1-1, vertical lines indicate kinds of statements that you can mix and horizontal lines indicate the order of statements and the kinds of statements you cannot mix. For example, you can mix comments with any kind of statement. If the program requires a SUBROUTINE statement, you must place it before all statements except comments. You can put FORMAT and ENTRY anywhere between PROGRAM, etc., and END. You can mix PARAMETER with all specification statements and place DATA anywhere between specification statements and END.

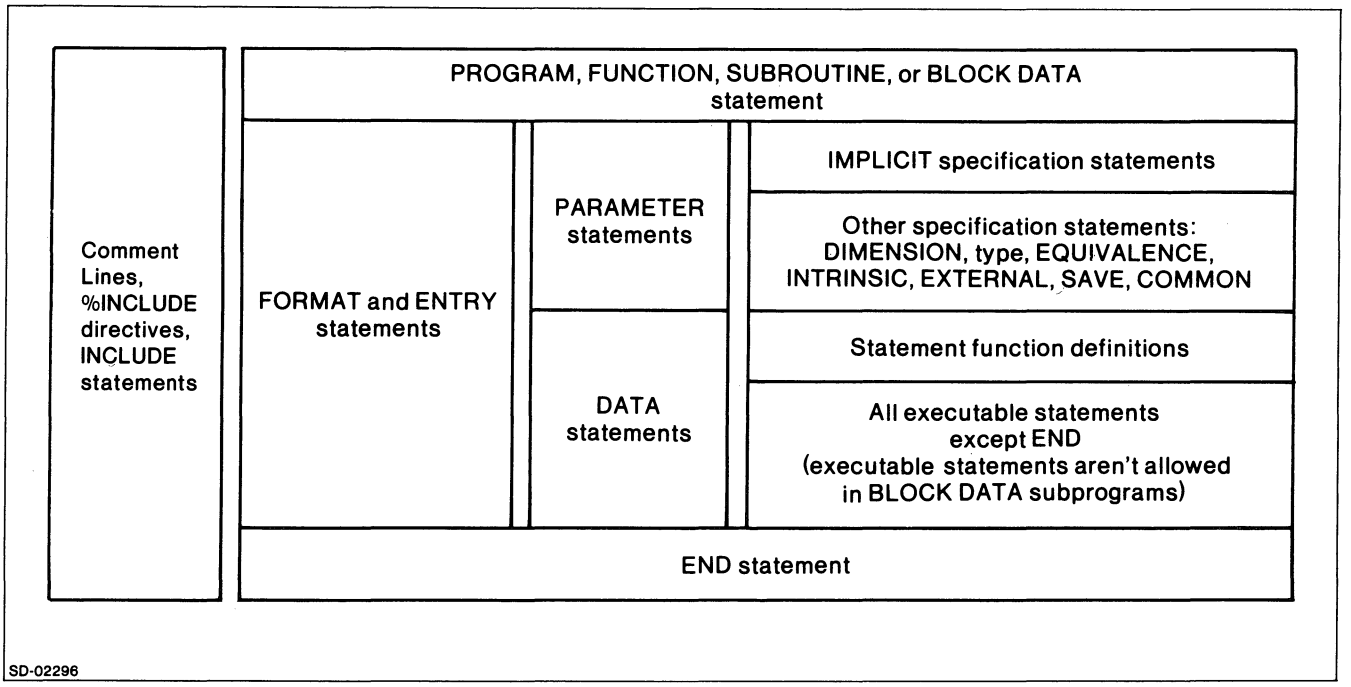


Figure 1-1. General Order of Statements in a Program Unit

F77 Character Set

The F77 character set includes the characters shown in Table 1-1. Characters that are extensions to the ANSI standard are ?, !, %, ", and lowercase letters.

Table 1-1 shows the *legal* F77 characters. *Any* printable ASCII character (except delimiters like NEW LINE) may appear in a comment. String constants may contain any printable or nonprintable character; you specify a

Table 1-1. F77 Character Set and Collating Sequence

Character	Name	ASCII 7-Bit Octal Code	Character	Name	ASCII 7-Bit Octal Code
	Blank, space	40	,	Comma	54
!	Exclamation point	41	-	Minus	55
"	Quote (same function as apostrophe)	42	.	Decimal point	56
$	Dollar sign	44	/	Slash	57
%	Per cent	45	0–9	Digits	60–71
'	Apostrophe	47	:	Colon	73
(	Left parenthesis	50	=	Equals	75
)	Right parenthesis	51	?	Question Mark	77
*	Asterisk	52	A-Z	Alphabet, uppercase.	101–132
+	Plus	53	a-z	Alphabet, lowercase.	141–172

nonprintable character by enclosing its ASCII octal equivalent in angle brackets. For example,

```
CHARACTER*4 STRING_CON
...
STRING_CON = 'AB<207>D'
```

places <101><102><207><104> in the first four bytes of character variable STRING_CON.

Collating Sequence

The order in which the characters appear in Table 1-1 is the F77 collating sequence; it is simply the ASCII rank of the characters. An ASCII translation table appears in Appendix A.

Compiler Optimization

The F77 compiler allows you to select from three levels of optimization, ranging from level 1 (least optimization) to level 3 (greatest optimization). You can select each level with the compiler switch /OPTIMIZE; for example,

```
F77/OPTIMIZE=3 MYPROG
```

If you omit this switch, the compiler does not optimize. The /OPTIMIZE switch and its effects are detailed in Chapter 9, in the section "Compiler Switches."

%INCLUDE Directive, %LIST Directive, INCLUDE Statement

As extensions, DG's F77 offers two directives to control compiler action. The first, %INCLUDE, allows you to insert an F77 source file in the current F77 program during compilation. The second, %LIST(OFF) and %LIST(ON), allows you to control program listing.

The INCLUDE statement provides the same functionality as the %INCLUDE directive.

%INCLUDE or INCLUDE

Use %INCLUDE or INCLUDE to include one or more F77 source files in the current program during compilation. This is useful for numeric error definitions and definitions of other parameters. %INCLUDE or INCLUDE can occur anywhere a statement can appear, but cannot be continued to a second line.

When the compiler encounters an %INCLUDE directive or INCLUDE statement, it obtains and compiles the statements of the included file, then resumes compilation with the statement following %INCLUDE or INCLUDE.

The source file to be included can in turn contain an %INCLUDE or INCLUDE directive to a file that can in *its* turn contain %INCLUDE or INCLUDE, and so on, up to a nesting limit of seven %INCLUDEs or INCLUDES. The form of %INCLUDE or INCLUDE is

```
%INCLUDE 'pathname'
%INCLUDE "pathname"
    INCLUDE 'pathname'
    INCLUDE "pathname"
```

where either apostrophes or quotation marks can delimit pathname, and pathname indicates a path to an F77 source file that you want included. If this source file is in the working (current) directory, its pathname is simply its filename. Typically, but not necessarily, pathname ends with the characters .F77.

The %INCLUDE directive *must* begin at column 1. The INCLUDE statement *must* begin at column 7 or beyond. 3 pairs of directives/statements with equivalent effects are:

```
C 000000011111111112 ...
C 345678901234567890 ...

%INCLUDE ":UTIL:DATA_STANDARDS.F77"
        INCLUDE ":UTIL:DATA_STANDARDS.F77"

%INCLUDE 'ERR.F77.IN'
        INCLUDE 'ERR.F77.IN'

%INCLUDE 'F77DIR:TABLEDEFS.F77'
        INCLUDE 'F77DIR:TABLEDEFS.F77'
```

%LIST (OFF) and %LIST (ON)

You may specify a listing file in the compilation command (via the /L switch, described in Chapter 9). Then, the compiler will generate a program listing with line numbers and error notes; the listing can also include other information.

To stop compiler output to the listing file, insert the %LIST(OFF) directive in your source; to resume listing, use %LIST(ON). The effect of each directive will persist until the next LIST directive or end of file, whichever comes first. An %INCLUDEd or INCLUDEd file (see above) can set its own listing options to override those of the original source; the original source list setting will return after the included file. The forms of LIST are

%LIST(OFF)

%LIST*[(ON)]*

Like %INCLUDE, a %LIST directive *must* start at column 1, and it cannot continue onto a second line.

An example is

```
        PROGRAM MYPROG
        ...
C       Omit INCLUDE file definitions from
C          this program's listing file.

%LIST (OFF)

C       Include two files here:

%INCLUDE 'ERR.F77.IN'
%INCLUDE 'MYDEFS.QQ'

C       Resume listing the program.

%LIST(ON)

        ...
        ...
```

The %LIST directives appear in the listing file. However, as the example stands, its %LIST(OFF) statement prevents the names of the %INCLUDEd files from appearing in the listing file. You can simply replace

```
%LIST(OFF)
```

by

```
            ! Don't list %INCLUDEd files
%LIST(OFF)  !    'ERR.F77.IN' and 'MYDEFS.QQ'.
```

On-Line HELP Files

Almost all Data General installations include a directory whose pathname is :HELP. This directory contains help about various DG software topics. If your system manager has followed the directions in the latest F77 Software Release Notice, then you have quick access to help about many F77 topics.

To find the names of these topics, give the CLI command HELP *F77. To obtain information about a topic whose name appears as a result of the HELP *F77 command, give the CLI command F77HELP <name-of-topic>.

For example, the CLI command

```
HELP *F77 ⤸
```

results in the display of the topic CARDFORMAT in the category COMPILATION SWITCHES. The CLI command

```
F77HELP CARDFORMAT
```

results in the following output on your terminal.

CARDFORMAT (F77 COMMAND SWITCH)

(may be specified in its minimally unique form)
Interprets only the first 72 characters of each source file record. Short records are padded with blanks, and all 72 characters are significant. If this switch is not specified the entire data-sensitive input record is scanned, but trailing blanks are discarded.

If no help file exists for a specified topic, you see the error message

UNKNOWN F77 HELP TOPIC

You can usually give a minimally unique abbreviation for the topic. Thus, F77HELP STATISTICS and F77HELP STA have the same effect. F77HELP ST results in an error message (*NOT MINIMALLY UNIQUE*) because the topics STATISTICS and STRINGS have the same first two letters.

Building F77 Programs

Here we give a quick sketch of how to use the system to write and build F77 programs. Planning usually precedes the steps described below; debugging and integrating programs for an application follow the steps. Details on building programs appear in Chapter 9.

1. Log on the operating system with your password; the system Command Line Interpreter (CLI) prompt — a right parenthesis and a space — appears on your terminal.
2. Use one of the system's interactive text editors (SED or SPEED or SLATE) to create one or more source program files. The resulting files will have data-sensitive records, as F77 requires. You can put more than one program unit (for example, the main program and subprograms it will use) in one source file. Or you can put each program unit in its own source file. A source file can have any valid system filename: 1 to 32 (15 for MP/AOS and MP/AOS-SU) alphanumeric characters, period (.), question mark (?), dollar sign ($) or underscore (_). By convention, F77 source files end in .F77, but this is not required; an example is MYPROG.F77.

3. Compile your source program with the CLI macro command:

   F77*[switches]* filename

   You don't have to give any .F77 suffix that is part of filename. For example, if the source file is MYPROG.F77, then the command to compile it is

   F77 MYPROG ⤸

   You may compile several source files with one command and parentheses. For example, to compile source files MAIN1.F77, SUB_1.F77, and SUB_2.F77, give the command

   F77 (MAIN1 SUB_1 SUB_2) ⤸

   Switches modify compiler action and are further described in Chapter 9. Correct compilation errors (if any) using error message text as a guide. Compilation produces an object file (filename.OB) from each source program file.

4. Use another CLI macro to invoke the Link utility and make desired program object files into an executable program file:

   F77LINK*[switches]* filename_1 *[filename] [...]*

   Link produces *one* executable program file named filename_1.PR. For example, the command

   F77LINK MYPROG ⤸

   creates MYPROG.PR from MYPROG.OB. For another example, the command

   F77LINK MAIN1 SUB_1 SUB_2 ⤸

   creates MAIN1.PR from MAIN1.OB, SUB_1.OB, and SUB_2.OB.

5. Execute the program file:

   XEQ filename

   If the program runs properly, you're done; if not, diagnose problem(s) with runtime results, runtime error messages, and/or the SWAT™ high-level debugger or assembly language debugger. Fix the program source(s); then run through the compiling, linking, and executing steps again.

F77 Program Example

Program DEMO_1.F77 in Figure 1-2 calculates the mean and the standard deviation of values typed in at a terminal. To try the program, follow the steps given earlier under "Building F77 Programs".

When you run the program, it will display the message:

How many values do you want to average (range 2-99)?

on the terminal. After you respond with a valid integer and NEW LINE character, for example:

3 ⤸

it will prompt for real values, up to the number of values you gave originally.

Then it will ask whether you want to confirm the values and, if you want to confirm, it will display each value for confirmation or replacement. Finally it will calculate the mean and the standard deviation of all the values typed, and stop. You must end each line of input with a NEW LINE or other data-sensitive delimiter.

Below is typical dialog as a user compiles, links, and executes DEMO_1:

```
) F77 DEMO_1 ⤸
Source file: DEMO_1.F77
Compiled on 7-Jun-84 at 16:57:23 by AOS/VS F77 Rev ...
Options: F77
) F77LINK DEMO_1 ⤸
LINK REVISION ...
=DEMO_1.PR CREATED
) X DEMO_1 ⤸
How many values do you want to average (range 2-99)? 3 ⤸
Enter value for array element  1: 13 ⤸
Enter value for array element  2: 15 ⤸
Enter value for array element  3: 17 ⤸
To verify the values, type Y; to compute, type other char. N ⤸
Average value: 15. Standard Deviation: 1.63299
*** END OF PROGRAM ***
STOP
```

```
      PROGRAM DEMO_1

C  This program calculates the mean and the standard deviation of up to 99
C    single-precision values.  It prompts for values from the terminal, asks
C    for verification of the values typed, then does its calculations.

      REAL           ARRAY(99)       ! Array to receive typed values.
      CHARACTER*32   PROMPT          ! Character variable for prompt.
      CHARACTER*1    ANSWER          ! Character variable for answer.

C     Accept and validate the number of values to average;
C        then assign characters to the prompt variable.

    5 PRINT *                        ! Blank line.
      PRINT *, 'How many values do you want to average (range 2-99)? '
      READ (*,*) NUMBER              ! Accept the count.

      IF ((NUMBER .GT. 99) .OR. (NUMBER .LT. 2)) GO TO 5      ! Correct range.

      PROMPT = 'Enter value for array element '

C     Now get the values and put them in ARRAY.

      DO 40 I = 1, NUMBER
      PRINT 32, PROMPT, I            ! Prompt for each element.
   32 FORMAT (1X, A, I2, ':  ')

      READ (*, *)  ARRAY (I)         ! Accept a value.
   40 CONTINUE

C     Offer to play them back for verification. If the user wants to
C     verify, play them back; otherwise go and compute.

      PRINT *, 'To verify the values, type Y; to compute, type other char. '
      READ (*, '(A)') ANSWER         ! Accept answer.
      IF (ANSWER .NE. 'Y' ) THEN     ! Block IF checks and ...
            GO TO 100
      ELSE                           !    ... plays back on request.
            PRINT *, 'To accept a value, type Y or any nonblank value;'
            PRINT *, 'to change it,  type C.'
            PRINT *

            DO 60 I = 1, NUMBER            ! For each element...
            PRINT 52, I, ARRAY (I)         ! Display element and value.
   52       FORMAT (1X, 'Old value of element ',I2, ' is ', G16.6)
            READ (*, '(A)')  ANSWER
            IF (ANSWER .EQ. 'C') THEN      ! On 'C', accept new value.
                  PRINT *, 'Type new value for element ',I
                  READ (*,*)  ARRAY (I)
            END IF                         ! Terminate inner block.
   60       CONTINUE                       ! Display next element.
      END IF                               ! Terminate outer block.
```

Figure 1-2. F77 Program Example DEMO_1.F77 (continues)

```
C       Calculate and display the average and standard deviation.

  100   SUM = 0.0
        SUMSQ = 0.0

        DO 190  I = 1, NUMBER
                SUM = SUM + ARRAY (I)          ! Sum the values.
                SUMSQ  = SUMSQ + ARRAY(I)**2   ! Sum their squares.
  190   CONTINUE

        AVG = SUM / NUMBER                    ! Calculate the average...
        STD_DEV = SQRT(SUMSQ/NUMBER - AVG**2) ! ... and standard deviation.

C       Print the results.

        PRINT *, 'Average value:   ', AVG,'  Standard Deviation:   ', STD_DEV
        PRINT *, '<NL>*** END OF PROGRAM ***<NL>'

        STOP
        END
```

Figure 1-2. F77 Program Example DEMO_1.F77 (concluded)

Below is typical dialog as a user compiles, links, and executes DEMO_1:

```
) F77 DEMO_1 ↓
Source file: DEMO_1.F77
Compiled on 7-Jun-84 at 16:57:23 by AOS/VS F77 Rev
  ...
Options: F77
) F77LINK DEMO_1 ↓
LINK REVISION ...
=DEMO_1.PR CREATED
) X DEMO_1 ↓
How many values do you want to average (range 2-99)?
  3 ↓
Enter value for array element     1: 13 ↓
Enter value for array element     2: 15 ↓
Enter value for array element     3: 17 ↓
To verify the values, type Y; to compute, type other char.
  N ↓
Average value: 15. Standard Deviation: 1.63299
*** END OF PROGRAM ***
STOP
```

End of Chapter

Chapter 2
FORTRAN Components

This chapter explains the building blocks of FORTRAN:

- Data Types
- Constants
- Symbolic Names
- Variables
- Arrays
- Expressions
- Arithmetic Expressions
- Relational and Logical Expressions
- Character Expressions
- Function References

Data types, numbers, names, and expressions form "entities" — the variables, arrays, and functions that are the basis of FORTRAN programming. The word "entity" occurs frequently in the rest of this manual.

Data Types

Each F77 entity has a data type. There are six categories of data types:

- Integer (includes types integer*4 and integer*2)
- Real (includes types real*4 and real*8)
- Double precision (real*8)
- Complex (includes types complex*8 and complex*16)
- Logical (includes types logical*4, logical*2, and — for AOS/VS — logical*1)
- Character

The data type of an entity is implied by the name rule (described later under "Symbolic Names"), or it is inherent in the makeup of the entity, or you can establish it directly by an IMPLICIT or type statement.

The data type you specify for an entity depends on what you want to do with it. The data type determines the entity's storage needs, range, and accuracy, as shown in Table 2-1.

Constants

A *constant* is a fixed value that does not change during program execution. The value can be arithmetic, logical, or a character string. To assign a symbolic name to a constant, use a PARAMETER statement.

Aside from the character data type, the rules for constants of a given type also apply to variables and array elements of that type.

Integer Constants

An *integer constant* is a whole number, with or without a sign. It may have a positive, negative, or zero value. If you omit a sign, the value is positive.

An integer constant may contain only the digits 0 through 9 and the sign. Leading zeros and interspersed blanks have no effect on its value.

The system stores each integer in twos-complement form. It uses 4 bytes for a 4-byte integer and 2 bytes for a short (2-byte) integer. The bounds of an integer depend on the operating system, as shown in Table 2-2.

If an integer constant is too large to fit in the amount of storage reserved for it, F77 signals an error.

To conform to the ANSI standard, integer, real, and logical entities must be 4 bytes.

The compiler allots 4 bytes of storage to an integer constant *unless all* of the following conditions apply:

- The compiler switch /INTEGER=2 is in effect. For AOS, F7716, MP/AOS, and MP/AOS-SU, the compiler macro supplied with F77 includes this switch, which you may override with /INTEGER=4.
- The decimal constant is between -32768 and 32767, or the octal constant (described later) is less than or equal to O'177777', or the hexadecimal constant (described later) is less than or equal to Z'FFFF'.
- The decimal constant contains 5 or fewer digits, or the octal constant contains 6 or fewer digits, or hexadecimal constant contains 4 or fewer digits.

Table 2-1. FORTRAN 77 Data Types

Data Type	Description	Amount of Storage Required (bytes)
Integer	An integer of the default length. The default length is 4 bytes unless set to 2 bytes by a compiler switch.	4 or 2
Integer*4	A 4-byte integer has at least the range -2,147,483,647 to 2,147,483,647.	4
Integer*2	A short (2-byte) integer, stored in half the space of a standard integer. Integer*2 range is -32,768 to 32,767.	2
Integer*1	Does *not* exist, but the logical*1 data type has many expected properties of an integer*1 data type.	—
Real	A decimal number that can be a whole number, a decimal fraction, or both. A real number can include an exponent field and can range from 5.4*10**-79 to 7.2*10**75. It has significance to about 6.7 decimal digits.	4
Real*4	Same as standard real.	4
Real*8	A real number with 8 bytes of storage; a double-precision real number as described next.	8
Double Precision	A double-precision real number; has the same range of a real number but has significance to about 16.4 decimal digits.	8
Complex	An ordered pair of real values. The first value is the real part of the complex number; the second value is its imaginary part.	8
Complex*8	Same as standard complex.	
Complex*16	An ordered pair of values, each stored as a double-precision type. The first value is the real part of the complex number; the second is its imaginary part.	16
Logical	A logical entity of the default length. The default length is 4 bytes unless set to 2 bytes by a compiler switch. A logical entity can have at least the value .TRUE. or .FALSE.	4, 2, or 1
Logical*4	A 4-byte logical entity.	4
Logical*2	A 2-byte logical entity.	2
Logical*1	A 1-byte logical entity (also with numeric and character properties). Another word for "logical*1" is "byte". Only AOS/VS F77 has the logical*1/byte data type.	1
Character	A character type is a string of characters, each occupying 1 byte of storage. The string can include any legal F77 characters and valid control characters like <BEL>. You delimit a character string with apostrophes or quotation marks.	n

Table 2-2. Storage of Integers

Integer Length / Operating System / Bound	Two-Byte (short)	Four-Byte	
	AOS/VS, AOS, MP/AOS, MP/AOS-SU	AOS, MP/AOS, MP/AOS-SU	AOS/VS
Upper	2**15 - 1 = 32,767	2**31 - 1 = 2,147,483,647	
Lower	-2**15 = -32768[1]	-(2**31) + 1 = -2,147,483,647	-(2**31) = -2,147,483,648

[1]See the explanation of the /NOFNS compiler switch in Chapter 9.

NOTE 1-byte integer constants do *not* exist. However, the logical*1 data type (in AOS/VS F77) can contain integers between -128 and +127 in one byte.

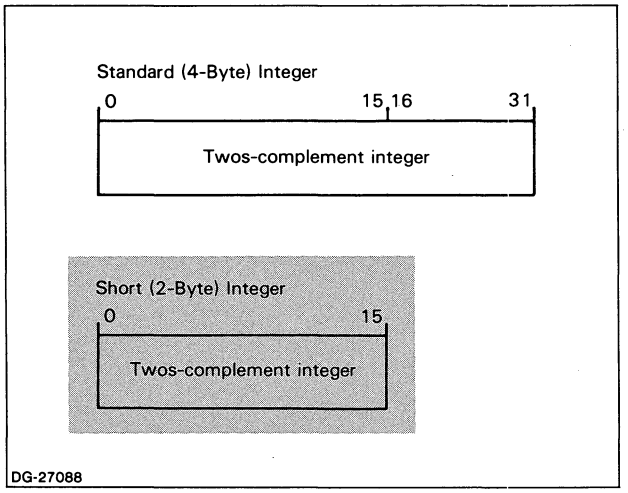


Figure 2-1. Internal Representation of an Integer

You can force a decimal integer constant to occupy 32 bits merely by writing it with six or more digits, such as 016384 and -007007.

An integer's internal representation is shown in Figure 2-1.

Examples of integer constants are

Valid	Invalid	Because
125	12.5	Has decimal point.
0	66,391	Has comma.
4351	2200000000	Too large.
-588	+33333	Too large for 2 bytes, gets 4.
17899	2E4	Exponential

Octal Constants

An *octal constant* (an alternate way to represent an integer) is one or more digits followed by the letter K or k. The digits must be in the range 0–7.

For 4-byte integers, the range is 0 through 37777777777K; for 2-byte integers, the range is 0 through 177777K. You can use an octal constant just as you would a decimal constant, except as a statement label, within a FORMAT specification, or as a digit string in a PAUSE or END statement. An octal constant cannot be signed.

An octal constant will be given 4 bytes of storage unless all conditions described for "Integer Constants" apply. Furthermore, you can force an octal constant to occupy 32 bits merely by writing it with seven or more digits, such as 0000000K and 0177777K.

Examples:

Valid	Invalid	Because
377K	26,371K	Has comma.
4301k	4389K	Has 8s or 9s.
144770K	+23K	Has sign.

Another way to specify an octal constant is with the letters O or o followed by the digits 0 through 7 delimited by apostrophes (').

Examples:

Valid	Invalid	Because
o'377'	o'26,371'	Has comma.
O'4301'	O"4301"	Has " marks.
O'144770'	O4389	Has 8s or 9s.
O'23'	O'+23'	Has sign.
O'34'	O34	No apostrophes.

These alternately specified octal constants obey the same rules as the octal constants specified with a trailing K or k. For example, O'23' cannot be a statement label.

Be careful with relatively large octal constants because of twos-complement storage of integer data. For example, the statement IVAR=O'177777' will set all 16 bits of integer*2 variable IVAR on; this is the bit pattern for -1. The binary number 111...1 (16 1s) may have a decimal value of 65535, but FORTRAN treats the bit pattern of 16 1s in an integer*2 variable as -1.

Hexadecimal Constants

A *hexadecimal constant* (an alternate way to represent an integer) consists of, from left to right:

- The letter Z or z
- An apostrophe (')
- 1 through 8 digits. Each digit is a hexadecimal digit (0–9, A–F, a–f).
- An apostrophe

For 4-byte integers, the range is Z'00000000' through Z'FFFFFFFF'; for 2-byte integers, the range is Z'0000' through Z'FFFF'. You can use a hexadecimal constant just as you would a decimal constant, except as a statement label, within a FORMAT specification, or as a digit string in a PAUSE or END statement. A hexadecimal constant cannot be signed.

The compiler allots 4 bytes of storage to a hexadecimal constant *unless all* of the following conditions apply:

- The compiler switch /INTEGER=2 is in effect. For AOS, F7716, MP/AOS, and MP/AOS-SU, the compiler macro supplied with F77 includes this switch.
- The number is between Z'0' and Z'FFFF', inclusive.
- The number contains less than 5 digits. If it contains 5 or more digits (even z'00000') it will be given 4 bytes.

You can force a hexadecimal constant to occupy 32 bits merely by writing it with five or more digits, such as Z'00001' and z'0000bf'.

Examples:

Valid	Invalid	Because
Z'3A7'	Z'1,234'	Has comma.
z'4f'	z'4g'	Has illegal character.
Z'20'	z'+23'	Has sign.
Z'30C'	Z"30C"	Has quotation marks.
Z'A1B'	ZA1B	No apostrophes.
Z'00005'	Z'000000009'	Too many digits.

Be careful with relatively large hexadecimal constants because of twos-complement storage of integer data. For example, the statement IVAR=Z'FFFF' will set all 16 bits of integer*2 variable IVAR on; this is the bit pattern for -1. The binary number 111...1 (16 1s) may have a decimal value of 65535, but FORTRAN treats the bit pattern of 16 1s in an integer*2 variable as -1.

Real Constants

A *real constant* consists of an optional sign, a string of decimal digits with a decimal point, and an optional exponent. You can place the decimal point before, after, or anywhere in the string. A real constant can be positive, negative, or zero.

For the exponent, use the letter E or e followed by an optionally signed integer constant to denote the power of 10. For example, 9E2 means 9 times 10^2 (=9*10**2 in FORTRAN notation). You need not include a decimal point in the number if you include an exponent; e.g., 9.E2 and 9E2 are both valid and mean the same thing. A real number must precede the E, even if your desired basic real value is 1; for example, 1E-2. The integer constant that follows the E can explicitly be 0, but cannot be blank.

The system stores a single-precision real number in 4 bytes, with sign and exponent preceding the mantissa. The high-order portions of the mantissa precede the low-order portions. A single-precision real number must be in the range 5.4*10**-79 to 7.2*10**+75. It has the significance of 6 hexadecimal digits, approximately 6.7 decimal digits. Its internal representation is shown in Figure 2-2.

Real*8 (double-precision) constants are described in the next section.

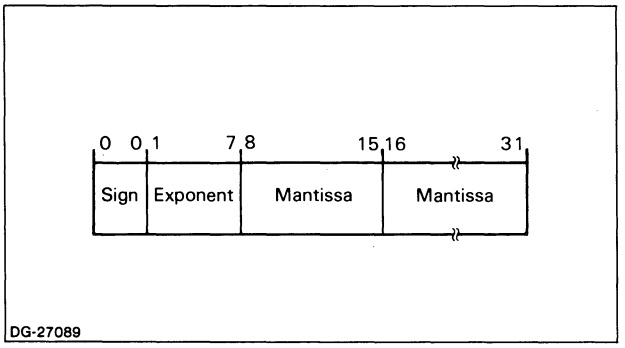


Figure 2-2. Internal Representation of a Real Number

Bits 1 through 7 in Figure 2-2 contain the exponent (in standard "excess 64" notation). This is the power to which 16 must be raised to give the correct value to the number. Bits 8 through 31 contain the mantissa, left-justified.

For example, consider the decimal number 7.5. You may rewrite it as +(.78*16) where 7 is a hexadecimal digit, 8 is a hexadecimal digit, and the implied exponent of 16 is 1. To confirm, note that

$.78*16 = 7*16^{0} + 8*16^{-1} = 7*1 + 8*1/16 = 7.5$

From 7.5 = +(.78*16) follows:

- Bits 0 – 0 contain 0 for the sign.
- Bits 1 – 7 contain 1000001 = 1+64 ("excess 64").
- Bits 8 – 31 contain the hexadecimal digits 780000.

Some examples of real constants are

Valid	Invalid	Because
+.00056	$45.62	Has dollar sign.
15.	15	Lacks decimal point.
15.E04	15.E4.2	Exponent isn't integer.
-.005E2	-.005E	Lacks exponent.
3200.E1	3,200.	Has comma.
4.5E-2	$2.00	Has dollar sign.
4E-10,4.E-10, 4.0E-10	4E-10.0	Exponent isn't integer.

Double-Precision Constants

A *double-precision* or real*8 constant resembles a single-precision real constant, but *must* include the letter D before the exponent.

To specify a double-precision constant, use the letter D or d followed by an optionally signed integer constant to denote the power of 10. For example, 9D3 means 9 times 10^{3} (=9*10**3 in FORTRAN notation). You need not include a decimal point. A real number must precede the D even if your desired basic value is 1; for example, 1D+03. The integer constant that follows the D can be explicitly 0 but cannot be blank.

The system stores a double precision (real*8) number in 8 bytes, and stores the sign and exponent in the same manner as real numbers. A double-precision constant has the significance of 14 hexadecimal digits, approximately 16.4 decimal digits. Its internal representation is shown in Figure 2-3.

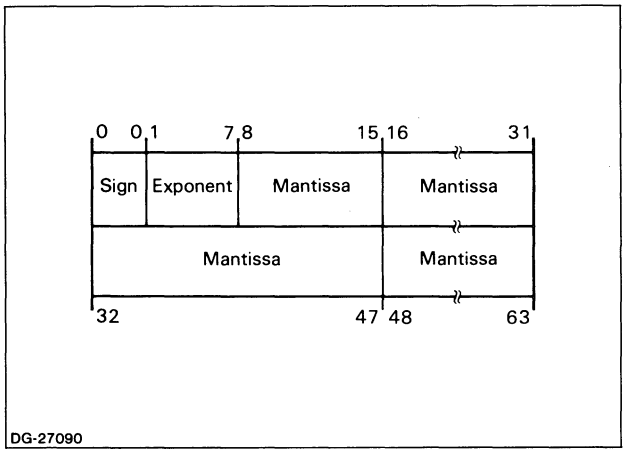


Figure 2-3. Internal Representation of a Double-Precision Number

Examples of double-precision constants are

Valid	Invalid	Because
2.5D-1	2.5D	No integer for exponent.
2.5D3	2.5D3.1	Exponent isn't integer.
5.0D-3		
.213D+11		
2D-14		
2.D-14		
2d14		

Complex Constants

A *complex constant* is a pair of integer or real constants. The first part is called the real part, the second the imaginary part. The constants are enclosed in parentheses and separated by a comma; for example, (25,10.5). The mathematical complex number $9+3.2i$ appears as the FORTRAN complex constant (9,3.2).

A standard complex (complex*8) constant is stored as two contiguous standard real constants, in 8 bytes (4 for the real part and 4 for the imaginary part). The internal representation of *each* part is standard real, shown in Figure 2-2.

A complex*16 constant is stored as two contiguous double-precision constants, in 16 bytes (8 for the real constant and 8 for the imaginary constant). F77 will allot 16 bytes for a complex constant if *either* part is double precision (contains a D). The internal representation for *each* part of a complex*16 constant is shown in Figure 2-3.

Examples of complex constants are

Valid	Invalid	Because
(2, 2)	(2 2)	Lacks comma.
(2, 2.E2)	2, 2.E2	Lacks parentheses.
(2, 2e2)	(2, 2.E2.2)	Exponent isn't integer.
(2.0, 2.0)	2 + 2*SQRT(-1)	Mathematical notation doesn't translate directly; FORTRAN doesn't accept square roots of negative numbers; lacks comma and parentheses.
(2.1, 0.0)		
(31D0, 45D03)		
(-9.3D-1,+.006D9)		
(346.012D-5,.0045D+3)		

Logical Constants

A *logical constant* has either a true value or a false value, represented as either .TRUE. or .FALSE.. You must include the delimiting periods as part of the constant.

A 4-byte logical entity is stored in 4 bytes; a 2-byte entity in 2 bytes. You can specify the length (2- or 4-byte) either globally, at compilation time via a command such as

```
F77/LOGICAL=2 MYPROG↓
```

or locally, in a program unit via a type statement such as

```
LOGICAL*2 TESTIMONY
```

The F77 compiler macro supplied with AOS, F7716, MP/AOS, and MP/AOS-SU systems includes the /LOGICAL=2 switch. For either 4- or 2-byte logical entities, F77 stores .FALSE. as all 0s and .TRUE. as all 1s.

The logical*1/byte data type (AOS/VS) stores .FALSE. as all 0s and .TRUE. as all 1s. This datatype, in contrast to logical*4 and logical*2 datatypes, can also accept numeric and character data. The numbers are integers between -128 and +127; the characters are any ASCII characters.

To conform to the ANSI standard, logical, integer, and real entities must be 4 bytes.

Examples of logical constants are

Valid	Invalid	Because
.TRUE.	TRUE	Lacks delimiting periods.
.true.	25	Invalid logical value.
	.T.	Abbreviations aren't allowed.

Character Constants

A *character constant* is a string of characters, including those from the F77 character set (Chapter 1); it may also include any ASCII characters. A character constant can be 1 through 255 characters long. (A character variable can be 1 through 32,767 characters long.) Either apostrophes (') or quotation marks (") must delimit the string, but these delimiters are not stored as part of the constant. You must use the same delimiter to start and end a constant; for example, 'ABC" is invalid.

To insert a delimiter character *within* the constant, use it twice or use the other delimiter; for example:

```
'Item''s current cost is:'
```

or

```
"Item's current cost is:"
```

F77 stores each character in 1 byte. You should reserve enough space for each character in the character type statement or IMPLICIT statement that sets up the entity.

You may use character constants in arithmetic and logical expressions, usually in an INTEGER environment. However, you may not use them as actual arguments to statement functions, or to intrinsic functions that do not accept arguments of type character.

The /HOLLERITH compiler switch controls how F77 stores character constants used in expressions and as arguments to subprograms. See Chapter 9. Unless otherwise stated between here and Chapter 9, all explanations of character constants used in expressions and as arguments to subprograms make an assumption: the programs are compiled either with no /HOLLERITH= switch or else with the /HOLLERITH=ANSI switch.

Logical*1/byte entities can contain one character.

Using Angle Brackets

DG's F77 allows you to make special characters, like bell, part of a character string. Simply insert the octal value or mnemonic of the character — with angle brackets — just as you'd insert any character; for example,

```
"Data follows:□<7>"
```

F77 includes mnemonics for characters as shown in Table 2-3.

Table 2-3. Special F77 Character Mnemonics

Name	Mnemonic	Octal Value
Null	<NUL>	<0>
Bell	<BEL>	<7>
Tab	<HT>,<TAB>	<11>
Line feed	<LF>	<12>
New Line	<NL>	<12>
Form feed	<FF>	<14>
Carriage Return	<CR>	<15>
Less than (<)	<LT>	<74>
Greater than (>)	<GT>	<76>

You can insert the ASCII value for *any* character symbol by enclosing its octal equivalent in angle brackets. Valid character codes range from 0 through 377 octal.

You cannot use the PARAMETER statement to define additional symbolic names for ASCII octal codes. The mnemonics shown in Table 2-2 are the only legal names for these characters and they cannot contain embedded blanks or tabs. If the value within the brackets is an invalid octal number or mnemonic (such as <NULL> or <183>), the compiler will signal an error.

Note that if you want the compiler to accept angle brackets literally (not as control characters), include the compiler switch /STRINGS=ANSI in the F77 compiler command; for example,

```
F77/STRINGS=ANSI MYPROG ↵
```

Unless you include this switch, the compiler interprets angle brackets and their contents as control characters.

Examples of character constants are

Specifier	Character Count	Output
"DON'T"	5	DON'T
"Don't"	5	Don't
"Hi□<BEL>"	4	Hi□(beep on terminal)
'<LT>0<GT>'	3	<0>
'<GT>0'	2	>0
'<NL>Newline'	8	↵ Newline
'Not□again'	9	Not□again
'"'	1	"
""""	1	"
"<042>"	1	"

As another example, suppose a program contains the character constant '<GT>0' from line 5 of the previous example. Compiling the program with the /STRINGS=ANSI switch results in output of the five characters *<GT>0*.

Hollerith Constants

The Hollerith constant, as described in ANSI's 1966 standard, was deleted from ANSI's FORTRAN 77 standard in favor of the character type. DG's F77 retains Hollerith constants in several ways.

Hollerith constants are strings of ASCII characters that can be used in limited ways. They can be initialized to integer, real, or logical values with DATA statements or used in argument lists of subroutine or function references. They, like character constants, may also be used in arithmetic and logical expressions, usually in an integer context. Similarly, they may not be used as actual arguments to statement functions or to intrinsic functions of any nature.

A Hollerith constant has the form:

$nHh_1 [... h_n]$

where n gives the number of characters in the constant, and h is a character. n must be an unsigned, positive integer. A character can be any character that is not a data-sensitive delimiter like NEW LINE. F77 does not interpret characters in Hollerith constants but passes them literally; e.g., 5H<BEL> is <BEL>.

F77 stores 1 character of each Hollerith constant in a byte; so n must equal the number of h characters.

The /HOLLERITH compiler switch controls how F77 stores Hollerith constants used in expressions and as arguments to subprograms. See Chapter 9. Unless otherwise stated between here and Chapter 9, all explanations of Hollerith constants used in expressions and as arguments to subprograms make an assumption: the programs are compiled either with no /HOLLERITH= switch or else with the /HOLLERITH=ANSI switch.

Examples of Hollerith constants are

Valid	Invalid	Because
3HCo.	3H^LCo.	Wrong length.
5HHello	0H0	Zero length.
4H<LT>	1H<LT>	Wrong length.
1H"	1H""	Wrong length.

Symbolic Names

Symbolic names — names you create and use — identify all F77 entities. A symbolic name can include from 1 to 32 letters, digits, and the special characters underscore (_) and question mark (?); it must begin with a letter or a question mark.

F77 ignores blanks or tabs within symbolic names. Examples are

Valid	Invalid	Because
M_ARGIN	_MARGIN	Starts with illegal character.
SUB_602	602_SUB	Starts with illegal character.
A	1A	Starts with illegal character.
Do_Inventory		
DO_INVENTORY		
?NAME		

Integer and real data types are the most common in FORTRAN, and the easiest way to type these is to follow the name rule. (The name rule doesn't apply to complex, logical, or character data types.) The name rule asserts that if a symbolic name begins with ?, I, J, K, L, M, or N, the entity is type integer. If the type name begins with any other letter, the entity is type real. For example:

Integer Type	Real Type
I	RVALUE
KFOO	ADATA
NUMBER	X
?XYZ	x?yz

You can establish your own variation of the name rule by inserting an appropriate IMPLICIT statement near the beginning of the program. For example, the statement

```
IMPLICIT REAL (?, I-N)
```

overrides the integer portion of the name rule and implies type real for every (otherwise untyped) entity in the program. IMPLICIT allows you to imply the type of *any* entity; for example,

```
IMPLICIT CHARACTER (C)
```

establishes all C- names as type character — unless a C- name is specifically typed otherwise. Chapter 3 explains the IMPLICIT statement fully, including its IMPLICIT NONE variation.

The other way to specify data type — for a variable, array, or function subprogram — is with a *type statement*. A type statement declares the symbolic name as either INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, or CHARACTER. A type declaration overrides the name rule (and IMPLICIT statement, if any), if there is a conflict. For example:

```
IMPLICIT COMPLEX (B)  ! Imply Complex for B-.

INTEGER BETA          ! BETA is integer despite
                      !    the IMPLICIT statement.

REAL N                ! N is real despite the
                      !    I-through-N name rule.
```

The data type of each array element is the type of the array.

In a function subprogram, you specify the function's data type in the FUNCTION statement or a type statement within the function; if you omit both, the name rule determines the function's data type.

The data types of intrinsic functions supplied with F77 generally conform to the name rule; in any case, you need concern yourself only with the data types of arguments to such functions.

To associate a symbolic name with a *constant* or *constant expression*, use the PARAMETER statement. The data type of the constant or expression must match the data type of the name.

The symbolic name that identifies a main program, subroutine subprogram, or common block has no data type.

Variables

A *variable* is an entity with a symbolic name and data type. You can define and refer to a variable. A variable has the value currently stored in its storage location; you can change this by assigning a new value to the variable. A variable may be undefined, in which case it has no value.

You can assign or reassign a value to a variable with an assignment or READ statement; and you can assign an initial value with a DATA or type statement.

You can associate two or more variables by putting their values in the same storage location. You accomplish this via a COMMON statement, an EQUIVALENCE statement, or actual and dummy arguments in subprogram references.

Data Types of Variables

A variable can be any of the F77 data types: integer (*4 or *2), real (*4 or *8), double precision, complex (*8 or *16), logical (*4, *2, or *1), or character. Logical*1 variables and byte variables are exactly the same.

The symbolic name of a variable implies its data type. The name rule or IMPLICIT statement implies the data type, or you can establish the data type directly with a type statement. A variable has the same range and storage requirements as a constant of its data type — except that a character variable can contain up to 32,767 characters while a character constant can contain up to 255 characters.

NOTE: Hollerith variables do *not* exist; only Hollerith constants exist.

Examples of variables with value assignments are

```
C       Integer and real variables:

        I = 3           ! Integer I is 3.
        J = 8           ! Integer J is 8.
        R = 1.2         ! Real R is 1.2.
        R = 1.2*I       ! Real R is 3.6.
        J = J+I         ! Integer J is 11.

        K = 'B'         ! Variable K contains
C         or            !    <102><040><040><040>
C       K = 1HB         !    = 10210020040K
                        !    = 1109401632.
                        !    If K were INTEGER*2,
                        !    then its value would
                        !    be <102><040>
                        !    = 41040K = 16928

        S = 'B'         ! Variable S contains
C         or            !    <102><040><040><040>
C       S = 1HB         !    = 0.3212549E+02

        BELL = '<7>'    ! Variable BELL contains
                        !    <007><040><040><040>
        NO_BELL = 3H<7> ! Variable NO_BELL
                        !    contains
                        !    <074><067><076><040>

C       Complex and character variables:

        COMPLEX PLX     ! Declare cmplx variable.
        PLX = (2, 2.E2) !    Assign it a value.

        CHARACTER*10 CH ! Declare character
                        !    variable.
        CH = 'abcde'    ! Assign it a value.
```

Arrays

An *array* is a sequence of data elements and has one or more dimensions. The name of an array or an IMPLICIT statement implies the array's data type; or, you can establish the data type directly with a type statement. An array element has the same range and storage requirements as a variable of its data type. You specify each array element with the array name and a qualifier called a *subscript*.

An array may have up to seven dimensions. A column of figures is an example of a one-dimensional (linear) array. If there are two columns of figures, you have a two-dimensional (rectangular or matrix) array; to point to a specific element of this array you must specify both its row number and its column number. In most cases, one- and two-dimensional arrays will suffice, but, if need be, you can use as many as seven. For example, if columns of numbers continue over several pages, you could use a three-dimensional array. A page number, row number, and column number would point to each element.

You can use a DIMENSION statement, a COMMON statement, or any type statement to establish an array. Using one of these statements, you define the name of the array, the number of dimensions in the array, the number of elements in each dimension, and, optionally, the range of the elements in each dimension. Five examples follow.

```
DIMENSION IARRAY (10)
```

IARRAY is a one-dimensional array with 10 elements. Unless an IMPLICIT statement is specified otherwise, IARRAY and all its elements are type integer by the name rule.

```
DIMENSION A (10,10)
```

A is a two-dimensional array of 100 (=10x10) elements. According to the name rule, A and all its elements are type real.

```
COMMON B (2,5,5)
```

B is a three-dimensional array of 50 (=2x5x5) elements. Again, the name rule implies type real.

```
INTEGER C(5,2,2,2)
```

C is a four-dimensional array of 40 (=5x2x2x2) elements. The INTEGER type statement makes C and its elements type integer, overriding the name rule and any conflicting IMPLICIT statement.

```
CHARACTER*10 CARRAY (2)
```

CARRAY is a one-dimensional character array with 2 elements, CARRAY(1) and CARRAY(2). Each element has storage space for up to 10 characters.

Subscripts

A *subscript* is a qualifier that you append to an array name. It determines which element of the array you are referencing. The form of an array reference with subscript is

array name (expr *[,expr] ...*)

where expr is a subscript expression. To refer to an array element, you must specify one subscript expression for each dimension defined for the array. For example, if array IARRAY is a one-dimensional array, a valid array reference is

```
IARRAY(2)
```

If array XYZ is a two-dimensional array, you must specify the two dimensions in the subscript; e.g.:

```
XYZ (3,5)
```

A subscript expression can be any valid arithmetic expression that resolves to an integer; e.g., IFOO*2. But if the expression is not type integer, F77 will convert the value to type integer before using it, according to the rules for arithmetic conversion in Chapter 3.

When you establish an array, via either DIMENSION, a type statement, or COMMON statement, you specify the upper and lower bounds of each dimension. If you omit a lower bound, it is 1, as in the examples above. If you need a lower bound that is not 1, you must specify both bounds, with a colon (:) between them, when you establish the array. For example:

```
DIMENSION JARRAY (0:4)
```

JARRAY is a one-dimensional array with 5 elements, JARRAY(0) through JARRAY(4).

```
REAL RARRAY (-1:3, 5:10)
```

RARRAY is a two-dimensional array with 30 elements. Elements in the first dimension are numbered -1 through 3; elements in the second dimension are 5 through 10. To refer to the first element in the array, you'd use RARRAY(-1,5).

During program execution, the subscript of an array must not assume a value that is less than the lower bound or greater than the upper bound for that dimension of the array. At compilation time, you can use the /SUB switch to generate code that will detect out-of-range subscripts at runtime. For example, you could give the command

```
F77/SUB MYPROG ↵
```

This switch is further described in Chapter 9. If you do not use the /SUB switch, and an array subscript is out of range, results are undefined and may be fatal.

Some examples with subscripts are

```
DIMENSION IARY (3)
IARY(1) = 5
IARY(2) = 10
IARY(3) = 1789
```

IARY, an integer array, has 3 elements. Element IARY(1) contains 5, IARY(2) contains 10, and IARY(3) contains 1789.

```
DIMENSION Y(2,4)
Y(1,1) = 3.
Y(1,2) = 5.
Y(2,1) = 4.6E2
```

Y, a real array, has 8 elements. Y(1,1) contains 3.0, Y(1,2) contains 5.0, and Y(2,1) contains 4.6*10**2; elements Y(1,3), Y(1,4), Y(2,2), Y(2,3), and Y(2,4) are undefined.

Array Storage

F77 stores elements of an array sequentially in memory. It stores an array with only one dimension so that the first array element occupies the array's first storage location and the last array element occupies the last storage location. Given a multidimensional array, F77 stores the array elements such that the leftmost subscript varies most rapidly. We call this the "order of subscript progression." When a nonsubscripted array name appears in a statement that assigns values to that array, F77 assigns the values in the order of subscript progression. You can use the order of subscript progression to improve execution time in nested DO operations, as described in Chapter 4 under "DO."

Array storage sequence is shown in the following three examples:

1. Elements of one-dimensional array C(15) are stored as:

   C(1), C(2), ..., C(15)

2. Elements of two-dimensional array A(10,10) are stored as:

   A(1,1), A(2,1), ..., A(9,1), A(10,1), A(1,2), A(2,2), ..., A(9,10), A(10,10)

3. Elements of three-dimensional array B(2,3,4) are stored as:

   B(1,1,1), B(2,1,1), B(1,2,1), B(2,2,1), B(1,3,1), B(2,3,1), B(1,1,2), ..., B(1,3,4), B(2,3,4)

Expressions

An *expression* is a group of one or more operands and optional operators and evaluates to a single value. The operators specify the computations to perform on the operands.

F77 has four kinds of expressions: arithmetic, relational, logical, and character. An arithmetic expression yields a numeric value; both relational and logical expressions yield logical values (.TRUE. or .FALSE.); character expressions yield character strings.

Arithmetic Expressions

Arithmetic operators and arithmetic operands form an *arithmetic expression.* An arithmetic operand may be a variable, an array element, a function reference, or a constant (numeric, character, or Hollerith). An arithmetic operator selects the kind of computation to perform on the operands and produces a numeric value. The arithmetic operators are

Operator	Function	Example
**	Exponentiation	A ** 3
*	Multiplication	A * 3.
/	Division	A / 3.
(+)	Plus (unary)	A / (+3.)
(-)	Minus (unary)	A / (-3.)
+	Plus	A + 3.
-	Minus	A - 3.

Most operators are called binary operators because they operate on *two* operands. Unary + and - operators are used to confirm or change the sign of an arithmetic operand. Furthermore, applying a unary operator to a logical*1/byte variable or array element results in a compiler-generated temporary variable. This temporary variable is 2 or 4 bytes long, depending on the /INTEGER= compiler switch (F77 switch).

The symbol ** means *raising to power.* For example, A**K represents A raised to the Kth power.

An arithmetic operand can have one of four data types: integer, real, double precision and complex. As shown near the beginning of this chapter, you can make nonstandard choices for the length of integer, real, and complex data types.

When a character or Hollerith constant appears in arithmetic expressions, F77 determines the data type of the constant according to these rules:

- When used with a binary operator, the constant takes the type of the other operand. For example, if I = 2, the equivalent expressions

  `I + 'AB'` and `I + 2HAB`

  have the identical value <000><002> + <101><102> = <101><104> if I is INTEGER*2. These expressions have the identical value <000><000><000><002> + <101><102><040><040> = <101><102><040><042> if I is INTEGER*4.
- When the syntax requires a particular data type (most often INTEGER*4) the constant takes the type required. For example, 'A' used as a subscript expression would be stored as <101><040><040><040> = 10110020040K = 1,092,624,416.
- In any other case, the constants are INTEGER*4, except that the /HOLLERITH compiler switch affects constants used as arguments to subprograms. See Chapter 9 for an explanation of this switch.

Integers in Division and Exponentiation

An arithmetic expression that involves two integers always produces an integer value. If the expression specifies *division*, F77 truncates (discards) any remainder from the quotient. For example,

```
9 / 5
```

yields the value 1 — whereas the real expression 9./5. (or 9./5 or 9/5.) yields the true value 1.8. Because integer division produces only an integer value, you should use it only where you want the integer result.

Wherever possible, you should use integer, as opposed to real, values for exponentiation. Real exponents need extensive logarithmic processing, which takes time, and sometimes produces less accurate results than integer exponents. For example,

```
A**3
```

can be computed much more efficiently (and perhaps slightly more accurately) than

```
A**3.0
```

Mixed Data Types in Arithmetic Expressions

If both operands have the same data type and the operator is (+), (-), +, -, *, or /, the result will have the same data type as the operands. Whenever possible, it is sound programming practice to use operands of the same type. If you need to convert one type to another before using it in an expression, you can use the appropriate F77 intrinsic function (Chapter 7) to do so.

Nonetheless, an expression may contain operands of differing data types. For unary operators (+) or (-), F77 converts other operands in the expression to the data type of the unary operand; for example, the result of

```
3 / (+1.)
```

is real. If the operator between the differing data types is +, -, *, or /, F77 converts the operand with the lower ranking data type to the higher ranking data type. The result of the evaluated expression has the higher ranking data type. For example, if you combine a noncomplex operand with a complex operand, F77 promotes the noncomplex operand to a complex number with an imaginary part of zero. For another example, the result of (+3)/1.0 is real. The exception to this rule is an expression with a double-precision operand and a standard (8-byte) complex operand; here, the double-precision operand becomes a `complex*16` number as does the resulting evaluated expression.

When a logical*1/byte entity (AOS/VS) appears on the right of the = sign with a binary operation, F77 converts the entity to the type of the other operand before performing the operation. However, if the other operand is logical*1/byte or a Hollerith constant, then F77 converts both to the default integer length before performing the operation.

The order in which F77 ranks data types is

Type	Rank
Complex*16	Highest
Complex	|
Double Precision / Real*8	|
Real	|
Integer*4	|
Integer*2	|
Logical*4	|
Logical*2	|
Logical*1 (AOS/VS)	Lowest

In an exponential expression (** operator), base and exponent operands with the same data type yield a value of the same data type. For example:

I**3	Yields an integer value.
R**3.	Yields a real value.

Base and exponent operands with differing data types yield a value of the higher ranking data type.

Examples of Arithmetic Expressions

Expression	Comments
5	Integer value of 5.
5.	Real value of 5.
5.E2	Real value of 500.
5.D2	Double-precision value of 500.
(5.,5)	Complex value of 5.0 (real part) and 5**i* (imaginary part); stored in 8 bytes with both halves containing the real*4 number 5.0.
-3.2	Real value of -3.2.
5**2	Integer 5 squared.
5**.3333	Integer 5 to a power approx. 1/3; approximates cube root.
J**2	J (integer by name rule) squared.
R**2	R (real by name rule) squared.
R**J	R to the power J.
8	Integer value of 8 (four bytes containing <000><000><000><010>)
'8' or 1H8 or "8"	Four bytes containing <070><040><040><040> = 7010020040K = 941629472 (= the bit configuration for the real*4 number 0.2921796E-10)
'85' or 2H85	Four bytes containing <070><065><040><040> = 7015220040K = 943005728 (= the bit configuration for the real*4 number 0.4831735E-10)
IB(3)	Element 3 of array IB.
IB(N-1)	Element N-1 of array IB.
J(3,4)	Element in the 3rd row and 4th column of array J.
INT (3.2)	F77 function yields integer 3.
INT (R)	F77 function yields integer of R.
REAL (3)	F77 function yields real 3.
REAL (I)	F77 function yields real of I.

Table 2-4. Evaluating Mixed Data Types for Addition, Subtraction, Multiplication, Division, and Exponentiation

DATA TYPE		Operand A					
		Integer*2	Integer*4	Real	Double Precision	Complex*8	Complex*16
Operand B	Integer*2	Integer*2	Integer*4	Real	Double Precision	Complex*8	Complex*16
	Integer*4	Integer*4	Integer*4	Real	Double Precision	Complex*8	Complex*16
	Real	Real	Real	Real	Double Precision	Complex*8	Complex*16
	Double Precision	Double Precision	Double Precision	Double Precision	Double Precision	Complex*16	Complex*16
	Complex*8	Complex*8	Complex*8	Complex*8	Complex*16	Complex*8	Complex*16
	Complex*16	Complex*16	Complex*16	Complex*16	Complex*16	Complex*16	Complex*16

Table 2-4 shows the rules for evaluating mixed data types. The resulting data type is in the intersecting box.

Operator Precedence

F77 evaluates arithmetic expressions in order, by the precedence of each operator, as follows:

Operator	Precedence
**	Highest (evaluated first)
* /	|
+ - (Unary)	|
+ -	Lowest (evaluated last)

When operators of equal precedence appear in an expression, the result is as if F77 evaluates them from left to right for (+) or (-), +, -, *, and /, and right to left for **. The compiler, because it might optimize, may *set up* operations to execute in differing orders. But the restrictions built into the compiler are that

- it must preserve the order established by explicit parentheses;
- it must ultimately obtain results as if it evaluated operators as described (left to right for +, -, *, and /, and right to left for ** and =).

In other words, the resulting value of an expression is predictable (subject to range and precision limitations), but the means the compiler will employ to get that result is not.

Another facet of compiler optimization is that F77 may not necessarily perform an evaluation that is not needed. For example, given the expression

```
.TRUE.  .OR.  (FUNC(X) .EQ. 0)
```

F77 need not evaluate the element (FUNC(X).EQ.0) because the ultimate value of this expression is .TRUE. regardless of the value of FUNC(X).

Where appropriate, F77 may revise the order of evaluating certain combinations of elements; it does this by applying the commutative and associative laws of arithmetic (subject to parentheses, described next). For example, given the expression A+B, the compiler may evaluate it as A+B or B+A — an example of the commutative law. For another example, given the expression A+B+C, the compiler may initially evaluate A+B or B+C — an example of the associative law.

If you use a function name in an expression, the function must not alter the value of any other element within the expression. For example, if you specify the expression X+FOO(X), where the function FOO modifies its argument, the value of the expression is undefined. F77 may add the function's value to either the original or modified value of X.

Using Parentheses

If you want the compiler to treat an expression as an entity, enclose that expression in parentheses. F77 preserves the integrity of parentheses when it evaluates operands and performs operations. If you nest parenthesized expressions, F77 evaluates the innermost first.

If you put more than one expression within parentheses, F77 evaluates them by order of operator precedence. For example,

Expression	Steps ([ ]) Involved	Result
8 + 4 * 9 - 6 / 2	[4*9=36], [6/2=3], [8+36-3] =	41
(8 + 4) * 9 - 6 / 2	[8+4=12], [12*9=108], [6/2=3], [108-3] =	105
(8 + 4 * 9 - 6) / 2	[4*9=36], [8+36=44], [44-6=38], [38/2] =	19
(8 + 4 * (9 - 6)) / 2	[9-6=3], [4*3=12], [8+12=20], [20/2] =	10

If you have any doubt about the order in which an expression will be evaluated, use parentheses to specify exactly what you want.

Relational and Logical Expressions

Relational and logical expressions allow you to test and compare entities against one another or against constants.

Relational Expressions

A *relational expression* is two arithmetic or character expressions separated by a relational operator. The value of a relational expression is either true (value .TRUE.) or false (value .FALSE.) depending on whether or not a stated relationship exists. The relational operators are

Relational Operator	Relationship	Example
.LT.	Less than	I .LT. 3
.LE.	Less than or equal to	I .LE. 3
.EQ.	Equal to	I .EQ. 3
.NE.	Not equal to	I .NE. 3
.GT.	Greater than	I .GT. 3
.GE.	Greater than or equal to	I .GE. 3

You must include the delimiting periods as part of each relational operator.

Other examples of relational expressions are

Expression	Result
1 .GT. 2	.FALSE.
I. GT. J	Depends on variable values.
'c' .GT. 'C'	.TRUE. because 'c' has a higher ASCII value (99) than 'C' (67).

Logical Expressions

A *logical expression* is a combination of logical operands and logical operators. A logical operand is a logical entity: variable, array element, or relational expression. A logical expression yields a single logical value, true (.TRUE.) or false (.FALSE.). The logical operators are

Operator	Example	Meaning
.AND.	L .AND. M	Logical conjunction. The expression yields true if both operands are true. It yields false if one or both operands are false.
.OR.	L .OR. M	Logical disjunction. The expression yields true if either operand, or both, are true. It yields false if both operands are false.
.XOR.	L .XOR. M	Logical exclusive disjunction. The expression yields true only if exactly one of the operands is true.
.NOT.	.NOT. L	Logical negation. The expression yields true if the operand is false.
.EQV.	L .EQV. M	Logical equivalence. The expression yields true if both operands are true or false. It yields false if one operand is true and the other false.
.NEQV.	L .NEQV. M	Logical inequivalence. The expression yields true only if exactly one of the operands is true.

The .AND., .OR., .XOR., .EQV., and .NEQV. operators are binary operators; .NOT. is a unary operator. Table 2-5 is a logical truth table that shows differing yields from logical expressions, given logical operands L and M.

Note that applying one of these logical operators to an illegal or undefined logical value will yield undefined results. The following statements illustrate this point.

```
LOGICAL LOG_VAR
INTEGER INT_VAR
EQUIVALENCE (LOG_VAR,INT_VAR)
INT_VAR = 10
IF (LOG_VAR .EQV. .TRUE. ) PAUSE 001
IF (LOG_VAR .EQV. .FALSE.) PAUSE 000
! NEITHER PAUSE STATEMENT EXECUTES.
...
```

However, assigning -1 or 0 to INT_VAR will result in the respective execution of the first or second PAUSE statement. Keep in mind that writing programs that depend on the bit representation of logical values results in nonstandard and nonportable programs.

Rules for Evaluating Logical and Relational Expressions

You may combine logical and relational operators within an expression. For example,

```
(A .LE. B)  .AND. (D .GT. F)
```

The arithmetic rules for operator precedence apply to logical and relational expressions. F77 evaluates parenthesized expressions as entities, and then evaluates operators according to precedence, evaluating operators of equal precedence left to right.

Table 2-5. Logical Truth Table

If L Operand Is	If M Operand Is	.NOT.L Is	L.AND.M Is	L.OR.M Is	L.XOR.M Is	L.EQV.M Is	L.NEQV.M Is
.FALSE.	.FALSE.	.TRUE.	.FALSE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.
.FALSE	.TRUE.	.TRUE.	.FALSE.	.TRUE.	.TRUE.	.FALSE.	.TRUE.
.TRUE.	.FALSE.	.FALSE.	.FALSE.	.TRUE.	.TRUE.	.FALSE.	.TRUE.
.TRUE.	.TRUE.	.FALSE.	.TRUE.	.TRUE.	.FALSE.	.TRUE.	.FALSE.

If an expression includes mixed operations, F77 evaluates arithmetic expressions first by the arithmetic rules of precedence. Then it does relational operations, then logical operations. The precedence of all F77 operators is

Operator	Precedence
**	Highest
* /	|
+ - (Unary)	|
+ - (Binary)	|
.GT. .GE. .EQ. .NE. .LT. .LE.	|
.NOT.	|
.AND.	|
.OR. .XOR.	|
.EQV. .NEQV.	Lowest

When any two of the F77 operators (except **) appear in the same expression and are of equal precedence, F77 evaluates the operands from left to right. It evaluates ** operators from right to left.

For example, assume that you give variables L, I, J, and K the respective values .TRUE., 2, 4, and 6 as follows:

```
LOGICAL L       ! Make L a logical variable.
L = .TRUE.      ! Make L .TRUE.
I = 2           ! Assign 2 to I.
J = 4           ! Assign 4 to J.
K = 6           ! Assign 6 to K.
```

When a logical*1/byte entity (AOS/VS) appears in a relational expression (.LT., .LE., .EQ., .NE., .GT., .GE.), F77 converts the entity to the type of the other operand. However, if the other operand is logical*1/byte or a Hollerith constant, F77 converts both to the default integer length before the comparison.

When a logical*1/byte entity (AOS/VS) appears in a logical expression (.AND., .OR., .XOR., .EQV., .NEQV.), F77 converts the entity to the type of the other operand. If the other operand is logical*1/byte, the result will have the type logical*1/byte. If the other operand is a logical constant, F77 interprets it to be logical*1/byte. The result of .NOT.(logical*1/byte entity) is a logical*1/byte value.

The following examples show how F77 would evaluate some relational and logical expressions using these values. The parentheses as shown *do not* affect the order of evaluation or result; we include them to clarify the expressions. The case of the letters, as is true for all F77 keywords, is irrelevant.

Expression	Interpretation	Result
(I .LE. J)	(2 .LE. 4) =	.True.
(I .LE. J) .AND. (I .LT. K)	(2 .LE. 4) = .True. ; (2 .LT. 6) = .True. ; (.True.) .AND. (.True.) =	.True.
(I .NE. J) .AND. (.NOT. L)	(2 .NE. 4) = .True. ; (.NOT. .True.) = .False. ; (.True.) .AND. (.False.) =	.False.
(.NOT. L) .OR. (J .EQ. K)	(.NOT. .True) = .False. ; (4 .EQ. 6) = .False. ; (.False.) .AND. (.False.) =	.False.

Character Expressions

A *character expression* yields a string of type character. The string can be from 1 to 32,767 characters long.

A character expression can include a character constant, character variable, character array element, character function reference, or character *substring*. One form of expression involves *concatenation*, which you can do with any character entity via the concatenation operator. This operator is

//

You can concatenate character constants directly with this operator; for example:

```
CHARACTER*20 H_MELVILLE, NAME

NAME = 'Ishmael'
H_MELVILLE = 'Call me ' // NAME
```

Another example is

```
CHARACTER*16 CA,CZ
CHARACTER*32 CSTRING

CA = 'Part number is: '
CZ = '93-000162-03'
CSTRING = CA // CZ      ! Concatenate.
```

Concatenation yields a character string that is the combination of both operand strings. The "Part" example yields a string, CSTRING, 32 characters long, that looks like this:

```
Part□number□is:□93-000162-03□□□□
```

Consider the following example.

```
CHARACTER*6 FIRST_6
FIRST_6 = 'ABCD'           ! PARTIAL
FIRST_6 = FIRST_6 // 'EF'  ! COMPLETE

PRINT *, 'FIRST_6 EQUALS ', FIRST_6

STOP
END
```

Execution of the program displays

FIRST_6 EQUALS ABCD□□

instead of

FIRST_6 EQUALS ABCDEF

as you might expect. Why? Because the first assignment statement places ABCD□□ in FIRST_6; FIRST_6 can't contain any more letters, such as EF. To generalize from this example, you can't have the same characters specified on both sides of the equal sign.

Character Substrings

A *substring* is a portion (or all) of a character string. You can extract substrings and assign them at will to character entities. To refer to a substring, use this form:

v (*[expr$_1$]* : *[expr$_2$]*)

where:

v	is a character variable or array element name.
expr$_1$	is an integer expression that evaluates to the leftmost character position you want. You can omit *expr$_1$* to specify character position 1; for example, C_ARR(1:5) and C_ARR(:5) extract the same substring from C_ARR.
:	is a colon. You must include a colon to specify a substring.
expr$_2$	is an integer expression that evaluates to the rightmost character position you want. You can omit *expr$_2$* to specify a range through the last character of entity v.

The *expr$_1$* must not be smaller than 1, and *expr$_2$* must not be greater than the length of the character entity. Examples, derived from the concatenation example above, are

Expression	Resulting String
CA(1:6)	Part n
CA(:6)	Part n
CA(7:)	umber□is:□
CA(1:2) // CA(4:4)	Pat
CA(1:2) // CA(4:4) // CA(12:14)	Pat is
I=4 CA(I:I) // CA(3*I:3*I+2) // CZ(1:1)	t□is9

NOTE: In a character assignment statement, the same character positions of one string must not be specified on both sides of the equal sign; e.g., the following statement is illegal.

```
CA(1:3) = CA(2:4)
```

The following statements also illustrate illegal concatenation.

```
CHARACTER*10 CM, CN
CM = "Str 1"
CN = "Str 2"
CM = CM // CN
CM = CM(1:5) // CN
CM = CN // CM
```

The following statements illustrate character array elements and their substrings.

```
CHARACTER*4 CA(3)    ! CHARACTER ARRAY
CHARACTER*4 SUB_S    ! SUB_STRING

CA(1) = 'ABCD'
CA(2) = 'MNOP'
CA(3) = 'WXYZ'

SUB_S = CA(2)        ! SUB_S = 'MNOP'
SUB_S = CA(2)(1:4)   ! SUB_S = 'MNOP'
SUB_S = CA(1)(2:4)   ! SUB_S = 'BCD '
! SUB_S = CA(3)(2:5) ! ILLEGAL
```

Character Relational Expressions

You can use the relational operators (.GT., etc.) with character strings or substrings. If the strings have different lengths, F77 temporarily pads the shorter string with blanks to produce equal lengths. With operators other than .EQ. or .NE., F77 bases the operation on the ASCII value of the corresponding character in each string. The ASCII value is the collating order, shown in Chapter 1 and Appendix A. Here is the order, high to low, of some commonly used characters:

Lowercase letters, a-z;

Uppercase letters, A-Z;

Digits 0-9;

Blank or space.

For example, using the previous entities CA (containing Part□number□is:□) and CZ (containing 93-000162-02□□□□), the following expressions have the following results:

Expression	Result
CA .GT. CZ	.TRUE. because P has a greater value than 9.
CA(5:5) .GT. CZ(5:5)	.FALSE. because □ does not have a greater value than 0.

Occasionally, you may want to index or obtain the lexical value of characters within a string. F77 functions like INDEX and ICHAR, described in Chapter 7, can help with such operations. For example, you may need to know if "ME□" is in the string "VT□NH□ME□MA□CT□RI□". The following statements show one way to find out.

```
PROGRAM TEST_INDEX

CHARACTER*18 NE_STATES /'VT NH ME MA CT RI '/
CHARACTER*3  MAINE     /'ME '/

IFOUND = INDEX(NE_STATES,MAINE)
! IFOUND IS 7 -- BUT IT WOULD BE 0 IF
!    'ME ' WASN'T IN NE_STATES.

PRINT *, 'IFOUND = ', IFOUND

STOP
END
```

Function References

A *function* is either a statement or a separate program unit that returns a value. F77 contains some functions of its own, called intrinsic functions; you can write other functions at will. All functions have data types.

A *function reference* is the name of a function followed by a list of actual arguments. After the function executes, it returns its value to the statement that referred to it. Functions and function references are covered in Chapter 7.

End of Chapter

Chapter 3
Defining Data Types, Values, and Constants

The type of each entity should match the kind of data you will put in the entity. This chapter describes defining data types, values, and constants. The sections proceed:

- Defining Data Types
- Using the IMPLICIT and IMPLICIT NONE Statements
- Using the DIMENSION Statement
- Using Type Statements
- Assigning Values with Assignment Statements
- Assigning Initial Values with the DATA Statement
- Defining Constants with PARAMETER
- Using EQUIVALENCE
- Other Ways to Assign Values

Defining Data Types

For integer and real entities, the name rule (I-N and ? integer, every other name real) can imply the data type. But you must specify the types of other entities either implicitly via an IMPLICIT statement or directly with a type statement such as CHARACTER.

Because arrays need more than one storage location, you must declare each array and its size — including integer and real types — with a DIMENSION statement, type statement, or COMMON statement.

Using the IMPLICIT and IMPLICIT NONE Statements

IMPLICIT allows you to set up your own symbolic naming conventions for data types within each program unit. IMPLICIT is a specification statement and must *precede* other specification statements (except PARAMETER) in a program unit. (However, if an IMPLICIT statement follows a PARAMETER specification statement, it must not affect the PARAMETER statement.) Its form is

IMPLICIT typ *[*len]* (a *[,a] [...]*)
*[,typ [*len] (a [,a] [...])] [...]*

where:

typ gives the data type and is INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, BYTE or CHARACTER.

len is an optional byte length specifier for the data type: for INTEGER, it can be 2 or 4 (default size); for REAL it can be 8 or 4 (default size); for COMPLEX, it can be 16 or 8 (default size); for LOGICAL, it can be 1, 2 or 4 (default size); or for CHARACTER it can be a positive unsigned integer from 1 (default) through 32767. The logical*1/byte data type exists only in AOS/VS F77. If you omit *len*, all data types will have an implied standard length; you can override this in a type statement or, for INTEGER or LOGICAL types, in a compilation command such as

```
F77/INTEGER=2 MYPROG ↵
```

Exception: The /INTEGER=1 and /LOGICAL=1 switches are illegal.

a is a single letter, question mark, or forward range of letters separated by a minus (-); e.g., A-H. Symbolic names that begin with the specified letter(s) will have the type (and length, if given) that precedes this a, unless you override this in a type statement. *

IMPLICIT can change or confirm the data type of all variables, arrays, or functions, except F77 intrinsic functions. IMPLICIT applies only to the symbolic names within its program unit, and it must precede such names within the program unit. If you wish, you can use IMPLICIT in conjunction with the name rule to establish default types for all data types you plan to use. Then follow your rules as you create symbolic names.

The default name rule is

```
IMPLICIT REAL(A-H,O-Z), INTEGER(?,I-N)
```

Any IMPLICIT statements you use override this, for the letters and types specified, within this program unit. And, a program unit may contain at most one IMPLICIT statement.

The form of the IMPLICIT NONE statement is simply

IMPLICIT NONE

IMPLICIT NONE is useful if you want to explicitly declare all variables and arrays according to their type. This statement has the compiler flag any variable or array whose type is not declared. For example:

```
IMPLICIT NONE      ! All variables
                   !   must be declared.

REAL VAR1

VAR1 = 2.0 + 3.0   ! No compiler error

ISUM = 2 + 3       ! Compiler displays an
                   !    error for this
                   !    statement because
                   !    ISUM wasn't
                   !    declared.

PRINT *, VAR1
PRINT *, ISUM

END
```

A program unit may contain at most one IMPLICIT NONE statement.

IMPLICIT Example

```
C       Make C-names character*16, make L-names
C          logical, make X-, Y-, Z names complex.

        IMPLICIT CHARACTER*16 (C), LOGICAL (L),
     +     COMPLEX (X-Z)
        INTEGER YEAR            ! Override the
                                !    previous
                                !    statement.

        CLIST = 'ascii chars'   ! C- is
                                !    character*16.
        LSTAT = .TRUE.          ! L- is logical.
        ZIMAG = (10., 2.3E2)    ! Z- is complex.
        RVAR = 10.3             ! R- names are
                                !    real by name
                                !    rule.
        IVAR = 4321             ! I- names are
                                !    integer by
                                !    name rule.
        YEAR = 1983             ! YEAR is integer
                                !    by the type
                                !    statement.
```

The IMPLICIT specifications could all be written on one line. You may have only one IMPLICIT or IMPLICIT NONE statement in a program unit.

Using the DIMENSION Statement

DIMENSION names and dimensions arrays of the data types given by IMPLICIT or a preceding type statement. DIMENSION statements are nonexecutable. You can label a DIMENSION statement, but no other statement can refer to the label.

DIMENSION is a specification statement; as such, it must precede statement function definitions and executable statements in a program unit. It must *follow* pertinent IMPLICIT or type statements.

The form of DIMENSION is

DIMENSION a (d) *[/ clist /] [, a (d) [/ clist /]] [...]*

where:

a — is the symbolic name of an array. If a preceding IMPLICIT or type statement has not specified a data type for this name, the type is integer or real according to the name rule. The array data type determines the size of each element.

d — is a dimension specifier: usually one or more integers or integer expressions that establishes the dimensions of the array.

clist — is a list of values ("constant list") that you want F77 to insert in the elements of the array.

You can establish an array's dimensions only once per program unit, in a DIMENSION, COMMON, or type statement. COMMON is described in Chapter 7.

Dimension Specifiers

Instead of using integers or integer expressions to establish the dimensions, you can use dummy expressions for adjustable dimensions, if the array name is a dummy argument name (described in Chapter 7).

If you use only one expression per dimension (e.g., IFOO(3)), it is the upper bound; 1 is the lower bound. To specify a lower bound other than 1, insert expressions for both lower and upper bounds, separated by a colon; e.g., DIMENSION IFOO(-1:9).

If the array will have more than one dimension, separate the dimension expressions with commas; e.g., DIMENSION RARRAY(5,10). F77 calculates the number of storage locations for an array by taking the product of the dimensions. For example, array AMY(4,6,6) gets 4x6x6=144 elements, and array FRED(0:2,-1:4) gets 3x6=18 elements.

When you use a variable as a dimension specifier and the array is not a dummy argument, then be sure to define the variable via a PARAMETER statement. This statement effectively assigns a name to a constant. (PARAMETER statements are fully explained later in this chapter.) For example:

```
PARAMETER (IARR_LEN = 8)
DIMENSION ARRAY(IARR_LEN)     ! OK
DIMENSION ARRAY(IARR_LEN+3)   ! OK

DIMENSION ARRAY(KONSTANT)     ! Wrong

LEN_FOO = 15
DIMENSION FOO(LEN_FOO)        ! Wrong
```

Initial Values

The initial values you give in *clist* are stored when the program is built; they cannot be reassigned via a second

DIMENSION / *clist* /

statement. Between the slashes (/.../), you can specify initial single values for sequential elements, separated by commas (e.g., DIMENSION IARY(4)/7,9,-3,12/); or, for the same initial value for multiple elements, with the number of elements, an asterisk, and the value; e.g., DIMENSION ORAY(4)/4*0.0/ initializes each element to 0.0. You must have as many clist values as there are elements in the array. Thus, the statement DIMENSION ORAY(5)/3*0.0,1.2/ is unacceptable because the clist does not assign ORAY(5) a value.

Here are more examples of initial values that use Hollerith and character constants:

```
      DIMENSION ICE_CREAM_FLAVORS(5)
     1 / 'VANL', 'CHOC', 'STRB', 'MAPL', 'BANA' /
```

initializes the 5 elements of ICE_CREAM_FLAVORS to VANL, CHOC, STRB, MAPL, and BANA respectively (as long as ICE_CREAM_FLAVORS has 4-byte integer elements.) More realistically, use the statement pair

```
      DOUBLE PRECISION ICE_CREAM_FLAVORS
      DIMENSION ICE_CREAM_FLAVORS(5)
     1 / 'VANILLA', 'CHOCLATE', 8HSTRAWB'Y,
     2   'MAPLE'  , 'BANANA'                /
```

The seventh byte of ICE_CREAM_FLAVORS(1) contains <101> from the second A of VANILLA and the eighth byte contains <040> for the trailing and padded space. This example also shows that you may mix character and Hollerith constants in initialization statements. Alternate ways to initialize ICE_CREAM_FLAVORS(3) are the eleven characters 'STRAWB"Y' or the ten characters "STRAW'BY".

Before you assign a value to a variable or array element, it may contain any value; it is not always initialized to zero or blank(s) before you use it. So for some applications you may want to zero or blank the variable or array element. In a subprogram, the current values of elements initialized via *clist* are automatically saved on return from the subprogram.

DIMENSION Examples

The following statements declare arrays via the name rule without an IMPLICIT statement.

```
DIMENSION ARY(30)        ! ARY is type real with
                         !    30 elements.

DIMENSION IRY(30)        ! IRY is type integer with
                         !    30 elements.

DIMENSION ?RY(30)        ! ?RY is type integer with
                         !    30 elements.

DIMENSION ZRY(10,4)      ! ZRY is 2-dimensional,
                         !   real, and has
                         !   40 elements.

DIMENSION RY(0:8,10)     ! RY is 2-dimensional,
                         !   real, and has
                         !   90 elements.

INTEGER*2 CODES          ! CODES is short integer;
DIMENSION CODES(5)       !   it is a 1-dimensional
                         !   array with 5 elements.

CHARACTER*20 CHR
DIMENSION CHR(10)        ! CHR has space for 20
                         !   characters in each
                         !   of 10 elements.
```

The following statements declare arrays with an IMPLICIT statement.

```
IMPLICIT COMPLEX (Z)     ! All Z- names are
                         !   complex.

DIMENSION IRY(40)        ! Integer array according
                         !   to the name rule.

DIMENSION ZPG(200)       ! Complex array because
                         !   of the IMPLICIT
                         !   statement.
```

Using Type Statements

Type statements specify the data type and optionally the byte length of an entity; they can also dimension arrays. A type statement overrides the name rule and an IMPLICIT statement if there is a conflict. Type statements are nonexecutable. You can label a type statement, but no other statement can refer to the label.

All type statements are specification statements and as such must precede statement function definitions and executable statements in a program unit. The form of a type statement is

INTEGER
REAL
DOUBLE PRECISION
COMPLEX
LOGICAL
BYTE
CHARACTER

*[*len] v [(d)] [*len] [/ clist /] [, v [(d)] [*len] [/ clist /] [...]*

where:

len is an optional byte length specifier for the variable or array elements. As shown in Chapter 2, it can be: for INTEGER, 2 or 4 (default); for REAL, 8 or 4 (default); for COMPLEX, 16 or 8 (default); for LOGICAL, 1 or 2 or 4 (default); or for CHARACTER, *n* (default character length is 1 byte; up to 32,767 bytes are allowed).

LOGICAL*1 and BYTE statements belong only in AOS/VS F77 programs.

For CHARACTER only, the *len* specifier may be an asterisk in parentheses, e.g., CHARACTER CA*(*), in two cases. The first is in any program unit if a value will be assigned the entity with PARAMETER. The second is in a subprogram's CHARACTER declaration of an entity received as a dummy argument. Both are further explained under "CHARACTER" in Chapter 8.

For integer or logical data types (except logical*1), you can also specify byte length globally at compilation time (e.g., F77/INTEGER=2 MYPROG). If you will specify 2-byte integers at compilation time, and need a 4-byte integer here, specify it as *4; the compiler will then allot it 4 bytes even though you specified 2-byte integers globally in the compilation command. For AOS, F7716, MP/AOS, and MP/AOS-SU, the F77 compiler macro includes /INTEGER=2 and /LOGICAL=2 switches.

If the type statement begins with DOUBLE PRECISION or BYTE, it may not contain the *len* specifier.

v is the name you want for the variable, array, function, or constant. This can be any valid symbolic name; the name rule and IMPLICIT statement (if any) do not apply.

d applies only to an array and declares a dimension. As in the DIMENSION statement, *d* can be an integer constant, or, in a subprogram, a dummy argument.

clist is a list of constants that includes one or more values for the entity. As with DIMENSION, the processor will initialize the entity to these values when it builds the program. For character data types, enclose the characters to be inserted in apostrophes (') or quotation marks ("); e.g.,

```
CHARACTER*15 CF /'balance□is'/
```

If you initialize an array via *clist*, the *clist* must include precisely enough values to fill the entire array. In a subprogram, the current values of entities initialized via *clist* are automatically saved on return from the subprogram.

You can establish the data type of a variable or array, and dimension an array, only once in a program unit.

Type Statement Examples

The following statements illustrate the use of type statements. Note how *clist* entities are numeric, character, and Hollerith constants. And, the LOGICAL*1/BYTE statements belong only in AOS/VS F77 programs.

```
INTEGER A              ! A is an integer variable.
INTEGER*2 A1           ! A1 is a short (2-byte
                       !    length) integer
                       !    variable.
INTEGER*2 A5, A6*4     ! A5 is a 2-byte variable,
                       ! A6 a 4-byte variable.
INTEGER I(20)          ! I is a 20-element integer
                       !    array.
integer A2(20)         ! A2 is a 20-element integer
                       !    array.
INTEGER A3(3) /1,2,4/  ! Array A3 elements 1-3
                       !    contain 1, 2, and 4.
INTEGER*2 A4(100)      ! A4 is a 100-element
                       !    integer array with
                       !    short integers.
INTEGER*2 LAST /'ZZ'/  ! LAST is a short integer
                       !    that contains
                       !    'ZZ' = <132><132>.
```

```
INTEGER*4 N_1 /4HPhil/ ! N_1 is a long integer
                       !    that contains 'Phil' =
                       !    <120><150><151><154>.
REAL K                 ! K is a real variable.
REAL*8 K1              ! K1 is a real*8 variable.
REAL K2 (20)           ! K2 is a 20-element real
                       !    array.
REAL R3 (20)           ! R3 is a 20-element real
                       !    array.
REAL N(2) /'Jack','Tom'/ ! N(1) contains
                         !    <112><141><143><153>,
                         ! N(2) contains
                         !    <124><157><155><040>.
REAL QUARTER /0.25/    ! QUARTER is a real variable
                       !    that contains 0.25.
DOUBLE PRECISION X     ! X is a double precision
                       !    variable.
DOUBLE PRECISION Y(9)  ! Y is a 9-element double
                       !    precision array.
COMPLEX Z              ! Z is a complex variable.
COMPLEX Z2(20,20)      ! Z2 is a complex array with
                       !    400 elements.
LOGICAL L              ! L is a logical variable.
LOGICAL L1 /.TRUE./    ! L1 is a logical variable
                       !    that contains .TRUE.
LOGICAL*2 L2           ! L2 is a short (2-byte)
                       !    logical variable.
LOGICAL L5(20)         ! L5 is a logical array of
                       !    20 elements.
LOGICAL*1 L6A /.TRUE./ ! L6A contains .TRUE. .
BYTE      L6B /.TRUE./ ! L6B contains .TRUE. .
LOGICAL*1 L7A / 100 /  ! L7A contains 100.
BYTE      L7B / 100 /  ! L7B contains 100.
LOGICAL*1 L8A / 'M' /  ! L8A contains 'M'.
BYTE      L8B / 'M' /  ! L8B contains 'M'.
LOGICAL*1 L8C / 1HM /  ! L8C contains 'M'.
BYTE      L8D / 1HM /  ! L8D contains 'M'.
LOGICAL*1 L9A(2) /7,9/ ! L9A(1) contains 7,
                       ! L9A(2) contains 9.
BYTE      L9B(2) /7,9/ ! L9B(1) contains 7,
                       ! L9B(2) contains 9.
CHARACTER C / 'N' /    ! C is a 1-byte character
                       !    variable that
                       !    contains 'N' = <116>.
CHARACTER*20 C1        ! C1 is a 20-byte character
                       !    variable.
CHARACTER*20 NAMES(5)  ! NAMES is a 5-element
                       !    character array; each
                       !    element has 20 bytes.
```

Other character examples are

```
      CHARACTER*10 C1, C2     ! Each has 10
                              !    characters.
      CHARACTER*10 C3, C4*15  ! 10 and 15
                              !    characters.

C     Two 20-element arrays, each element with
C        10 characters.
      CHARACTER*10 CA1(20), CA2(20)

C     One 20-element array with 10-character
C        elements, and one 10-element array
C        with 15-character elements.
      CHARACTER*10 CA3(20), CA4(10)*15

C     One 5-element array, each element with 10
C        characters and the elements' values
C        initialized with data.
      CHARACTER*10 ICE_CREAM_FLAVORS(5)
     1    / 'VANILLA   ' , 'CHOCOLATE ' ,
     2      'STRAWBERRY' , 'MAPLE     ' ,
     3      'BANANA    '                  /
```

The VANILLA entry does not have to have three trailing spaces before the second apostrophe, but placing them eases reading. The compiler would have stored 'VANILLA' with three trailing spaces if we hadn't deliberately created them.

Note that this example shows the best way to handle the situation presented in the examples with array ICE_CREAM_FLAVORS in the previous section, titled "Initial Values".

Assigning Values with Assignment Statements

The DIMENSION and type statements establish data types and can set initial values, but in the program body you will want to assign other values. The primary way to do this is with assignment statements, which use the equals sign (=). (There is an ASSIGN *statement*, but it assigns a *statement label,* not a value, to a variable; it is described in Chapter 4.)

The equals sign directs the computer to evaluate the expression on the right side of the sign and then place the value in the entity specified on the left side. For example,

```
A = B + 5.0
```

means "replace A's value with the sum of B's current value and the real number 5.0."

```
KOUNT = KOUNT + 1
```

is, as a mathematical equation, wrong. However, F77 responds to this statement by working from right to left. That is, F77 adds KOUNT and 1; then it places this sum in KOUNT, replacing whatever value was in KOUNT.

The operand to the left of the equals sign is always a variable or array element name; it cannot be a constant, subprogram name, array name, statement label, or predefined F77 function name. A variable or array element must be defined before using the variable or element in an expression.

The variable or array element that receives an arithmetic value can be type logical*1/byte (AOS/VS), integer, real, double precision, or complex; the variable or array element that receives a logical value must be type logical; the variable, element, or substring that receives a character value must be type character or logical*1/byte.

Arithmetic Values

The form of an arithmetic assignment statement is

```
v = expr
```

where:

v is a variable or array element.

expr is an arithmetic expression.

The entity to the left of the equals sign may or may not already have been assigned a value; but each item in expr must be defined before this assignment statement executes.

If the data types on each side of the equals sign are the same, F77 assigns the value directly. If the data types differ, F77 converts the value of the expression to the data type of the entity on the left, then assigns the value. (You may want to use mixed data types as infrequently as possible.) Table 3-1 shows how F77 converts the value of the expression on the right before assigning it to the expression on the left.

A statement function or intrinsic F77 function name cannot appear to the left of the equals sign.

Arithmetic Assignment Examples

```
AVG = (A+B+C)          ! AVG is a type real variable
                       !    by the name rule.
B = C(I)               ! Variable B gets the value
                       !    of array C's element I.
B = C(I)+(SIN(Y)+6)    ! Variable B gets the value
                       !    C(I) + (Sine-of-Y + 6).
Z = 4.11D-10           ! Variable Z receives the
                       !    double precision value.
B(I) = A*2.1           ! Element I of array B
                       !    gets the value A*2.1.

B(I,J) = A((K-1) * 10 + K) ! Element I,J of array
                           !    B gets the value
                           !    of the A element
                           !    number (K-1)*10+K.
```

Assigning Character Constants and Hollerith Constants to Noncharacter Variables

The following examples show how the assignment statement works in several situations involving character constants and Hollerith constants. I2 is an INTEGER*2 variable and I4 is an INTEGER*4 variable.

In the examples, a - (hyphen) to the right of an entry in the "Value Stored" column indicates that the compiler has truncated the expression to the right of the equals sign and it will report a warning message. A + in this location indicates that the compiler has padded the entry with blanks and that it will report a warning message.

Statement	Value Stored in I2
I2 = 12	<000><014>
I2 = 12K	<000><012>
I2 = '12'	<061><062>
I2 = '<12>'	<012><040> +
I2 = '<NL>'	<012><040> +
I2 = "<NL>"	<012><040> +
I2 = <NL>	None - Illegal statement
I2 = 4H<NL>	<074><116> -
I2 = 'A'	<101><040> +
I2 = 1HA	<101><040> +
I2 = 'A '	<101><040>
I2 = 'AB'	<101><102>
I2 = 'ab' ! (lower case)	<141><142>
I2 = 2HAB	<101><102>
I2 = 'A '	<101><040> -
I2 = 'ABCD'	<101><102> -
I2 = '!<BEL>'	<041><007>
I2 = 'A<12>'	<101><012>
I2 = ' A'	<040><101>
I2 = '<0>A'	<000><101>
I2 = ' A'	<040><040> -
I2 = '<67><177>'	<067><177>
I2 = '<67><377>'	<067><377>
I2 = '<67>'	<067><040> +

Statement	Value Stored in I4
I4 = 12	<000><000><000><014>
I4 = 12K	<000><000><000><012>
I4 = '12'	<061><062><040><040> +
I4 = '<12>'	<012><040><040><040> +
I4 = '<NL>'	<012><040><040><040> +
I4 = "<NL>"	<012><040><040><040> +
I4 = <NL>	None - Illegal statement
I4 = 4H<NL>	<074><116><114><076>
I4 = 'A'	<101><040><040><040> +
I4 = 1HA	<101><040><040><040> +
I4 = 'A '	<101><040><040><040> +
I4 = 'AB'	<101><102><040><040> +
I4 = 2HAB	<101><102><040><040> +
I4 = 'A '	<101><040><040><040> +
I4 = 'AB '	<101><102><040><040> +
I4 = 'ABC'	<101><102><103><040> +
I4 = 'ABCD'	<101><102><103><104>
I4 = 'abcd' !(lower case)	<141><142><143><144>
I4 = 'ABCDE'	<101><102><103><104> -
I4 = 'GO!<BEL>'	<107><117><041><007>
I4 = 'A<12>B'	<101><012><102><040> +
I4 = ' A'	<040><040><040><101>
I4 = '<0>A'	<000><101><040><040> +
I4 = '<0><0><0>A'	<000><000><000><101>
I4 = ' A'	<040><040><101><040> +
I4 = '<67><177>'	<067><177><040><040> +
I4 = '<67><177><0>'	<067><177><000><040> +
I4 = '<67><377>'	<067><377><040><040> +

If the expression is a character constant or a Hollerith constant then F77 places those characters left-justified into the variable. If necessary, F77 either pads the unused bytes of the variable with blanks or else ignores the right bytes of the constant, but in either case the compiler gives a warning message about the padding or truncation. For example (where the variables are REAL*4):

```
REAL_ 4_A = 'June'       ! REAL_ 4_A contains
                         !    <112><165><156><145>
REAL_ 4_A_1 = 4HJune     ! REAL_ 4_A_1 also contains
                         !    <112><165><156><145>
REAL_ 4_B = 'May'        ! REAL_ 4_B contains
                         !    <115><141><171><040>
                         !    and the compiler
                         !    gives a warning
                         !    message about
                         !    the padding.
REAL_ 4_C = 'August'     ! REAL_ 4_C contains
                         !    <101><165><150><165>
                         !    and nothing happens
                         !    to "st"; the compiler
                         !    gives a warning
                         !    message about the
                         !    truncation.
```

Table 3-1. How F77 Converts Arithmetic Expressions with Mixed Data Types

Data type of variable or element on left of = sign (v)	Data Type of Expression on Right of Equals Sign (expr)				
	INTEGER*2 INTEGER*4	**REAL**	**DOUBLE PRECISION**	**COMPLEX*8 (standard)**	**COMPLEX*16 (extension)**
INTEGER*2 INTEGER*4 (If expr is out of v's range, an error occurs.)	Assign expr to v.	Fix expr to integer and assign to v.	Fix expr to integer and assign to v.	Fix the value of the real part of expr and assign to v; imaginary part is dropped.	Fix the value of the real part of expr and assign to v; imaginary part is dropped.
REAL	Float expr and assign to v.	Assign expr to v.	Assign MS portion of expr to v; LS portion of expr is dropped.	Assign the real part of expr to v; imaginary part is dropped.	Assign MS portion of the real part of expr to v; LS portion of the real part of expr is dropped as is imaginary part.
DOUBLE PRECISION	Float expr and assign to MS portion of v; LS portion of v is zero.	Assign expr to MS portion of v; LS portion of v is zero.	Assign expr to v.	Assign real part of expr to MS portion of v; LS portion of v is zero; imaginary part is dropped.	Assign the real part of expr to v.
COMPLEX*8 (standard)	Float expr and assign to MS portion of v; imaginary part of v is zero.	Append an imaginary part of zero to expr and assign to v.	Assign MS portion of expr to real part of v; LS portion of expr is dropped; imaginary part of v is zero.	Assign expr to v.	Assign MS portion of both real and imaginary parts of expr to v; LS portions of expr are dropped.
COMPLEX*16 (extension)	Float expr and assign to MS portion of v; LS portion and imaginary part of v are zero.	Assign expr to MS portion of v; LS portion and imaginary part of v are zero.	Assign expr to v; imaginary part of v is zero.	Assign expr to MS portions of real and imaginary parts of v; LS portions of real & imaginary parts of v are zero.	Assign expr to v.

MS = Most Significant (high-order)
LS = Least Significant (low order)

Logical Values

The variable or array element to which you assign a logical value must be type logical. The expression whose value is assigned must evaluate to true or false whenever the variable or array element is logical*2 or logical*4.

The expression whose value is assigned should have the following values whenever the variable or array element is logical*1/byte (AOS/VS only):

Expression	Values
logical	.TRUE. or .FALSE.
integer	-128 to +127
character	'c' or "c" or 1HC, where c is a single character

See the next two sections for the results when the expression is not logical*1/byte.

The form of the logical assignment statement is

v = expr

where:

v is a logical variable or logical array element.

expr is a logical or relational expression that is .TRUE. or .FALSE., .true. or .false., or evaluates to true or false.

All items within expr must have been assigned values. The values in expr can be arithmetic, logical, or character. The expression must yield a logical value: true or false.

Logical*1 Assignments: Receiving (AOS/VS Only)

When the receiving variable or array element v is logical*1/byte and the expression expr is not, F77 performs the following conversions as it executes v=expr.

Expression	Result
integer*4, integer*2	F77 places the rightmost 8 bits into v.
other numeric	F77 converts the value to an integer*4 temporary variable according to Table 3-1, and then places the rightmost 8 bits of the temporary variable into v.
logical*4, logical*2	F77 evaluates the rightmost bit, and sets v to 1 or 0 according to the respective values .TRUE. or .FALSE. of expr.
character	F77 places the leftmost 8 bits into v.

Logical*1 Assignments: Sending (AOS/VS Only)

When the sending expression expr is logical*1/byte and the variable or array element v is not, F77 performs the following conversions as it executes v=expr.

Variable or Array Element	Result
numeric	F77 sign extends the value of the expression to an integer*2 temporary variable, and then converts the temporary variable to the appropriate data type according to Table 3-1.
character	F77 copies the expression into the first 8 bits of the character entity.
logical*4, logical*2	F77 sign extends the value of the expression to 32 or 16 bits.

Sign extending an 8-bit logical*1/byte entity to an integer*2, logical*2, or logical*4 target entity occurs as follows. The rightmost 7 bits of the logical*1/byte sending entity go to the rightmost 7 bits of the target; and the leftmost bit is propagated through the remaining bits of the target.

Logical Assignment Examples

```
LOGICAL L, LG(10), Y     ! Assign
LOGICAL*2 L2             !   logical
CHARACTER*4 C4           !   type.
LOGICAL*1 LM1, LM2, LM3 ! AOS/VS only

LM1 = .TRUE.             ! Logical*1/byte can
LM2 = -84                !   receive logical,
LM3 = 'X'                !   numeric, and
LM3 = 1Ht                !   character data.
Y = .FALSE.              ! All 0s.
LM3 = Y                  ! LM3 receives the right-
                         !   most 8 bits of Y,
                         !   which are 00000000=0.
LM3 = 'DGC'              ! LM3 receives the left-
                         !   most 8 bits, which
                         !   are 01000100 = 'D'.
LM3 = 4HCorp             ! LM3 receives the left-
                         !   most 8 bits, which
                         !   are 01000011 = 'C'.
LM1 = .TRUE.             ! Reset to
LM2 = -84                !   initial
LM3 = 'X'                !   values.
IVAR = LM2               ! IVAR contains -84 in
                         !   either 16 or 32 bits.
RVAR = LM2               ! RVAR contains -84.0
                         !   in 32 bits.
C4 = 'ABCD'              ! Character assignment.
C4 = LM3                 ! C4 contains 'XBCD'.
L2 = LM1                 ! The two bytes of L2 are
                         !   <11111111> <11111111>.

L = I .GT. 4             ! L receives .TRUE. or
                         !   .FALSE., depending
                         !   on I.
LG(1) = (X.GT.5.) .OR. (Y.LT.Z) ! Element LG(1)
                                 !   gets .TRUE. if
                                 !   X > 5 or if
                                 !   Y < Z.
Y = .TRUE.               ! Variable Y receives
                         !   .TRUE.
Y = -1                   ! Illegal statement.
```

Character Values

Each variable or array element to which you assign characters must be type character. The number of characters you assign must be from 1 to 32,767.

A character variable is an entity that you declared without a number of dimensions; e.g., if you declare CHARACTER*20 CH, then CH is a character variable. A character array element is one element of an array declared as type character; e.g., after you declare CHARACTER*20 CHA(5), then CHA(5) is a character array element.

You can use character variables anywhere you use text characters in the program: to clarify output, interact with the terminal, and for subroutine or function calls.

The form of the character assignment statement is

v = expr *[// expr] [// expr] [...]*

where:

v — is a character variable name, character array element, or character substring specifier.

expr — is a character variable or function name, character array element, character substring specifier, or string of ASCII characters delimited with apostrophes (') or quotation marks ("). You can include any ASCII character or certain mnemonics within angle brackets. To concatenate (join) character expressions, insert two slashes (//) between the expressions you want to join. Angle brackets, substrings, and concatenation are covered in Chapter 2.

An IMPLICIT-directed symbolic name or character type statement sets up a character entity. Then, the name or statement reserves a byte of storage for each of the number of characters specified.

The assignment statement replaces the reserved bytes with the expr characters, starting with the leftmost character position. If the number of assigned characters is smaller than the entity, F77 inserts blanks after the characters to fill the entity. If the number of assigned characters won't fit, F77 uses the leftmost characters and drops the remaining ones. For example, after the statements

```
CHARACTER*10 CH, CH1     ! 10 characters each
CH = '1234567890abc'     ! 13 characters
CH1 = '123456'           ! 6 characters
```

CH will contain only '1234567890' and CH1 will contain '123456□□□□'.

To start replacing characters at other than the first character position, specify a substring portion of the v on the left of the equals sign. For example, the statement

```
CH(3:10) = 'abcdefg'
```

replaces the characters in positions 3–10 of CH with 'abcdefg'; it does not replace the characters in positions 1–2 of CH.

Character variables and array elements always retain their original size (this is true for *all* data types, but it can be more conspicuous for characters because they appear literally on the terminal and page). This can affect the way you handle concatenation; for example,

```
C       Declare two 10-character variables and two
C          20-character variables.

        CHARACTER*10 C1, C2*10, C3*20, C4*20
        C1 = 'Hi'             ! Assign 'Hi' to C1
        C2 = 'there'          !   and 'there' to C2
        C3 = C1 // C2         ! Concatenate all
                              !   of C1 and C2.
        C4 = C1(1:3) // C2(1:5)  ! Concatenate
                                 !   substrings.
```

After these statements execute, C3 will contain 'Hi□□□□□□□□there□□□□□' and C4 will contain 'Hi□there□□□□□□□□□□□□□'. The latter form might be more suitable for your needs.

Note that you cannot use the same entire variable, element, or substring specifier on both sides of the equals sign: none of the character positions being defined in v may be referred to in expr. However, you can use a *different* substring of one v on each side. For example, using variable C2 above:

```
C2 = C2                 ! Is illegal.
C2(1:3) = C2(2:4)       ! Is illegal.
C2(1:3) = C2(4:6)       ! Is acceptable.
```

Using the same substring on both sides of the equals sign will not produce an error but *will* yield strange results.

You can use character substrings and relational operators; for example,

```
IF ( C2(1:1) .GT. C3(1:1) ) C4 = C2
```

If you use .GT., etc., and the collating sequence on your system is not ASCII, your program may not function identically on other systems. Instead of .GT., etc., use the F77 lexical intrinsic functions LLE, LGT, etc., to provide portability. These functions are described in Chapter 7.

Character Assignment Examples

An example with bell is

```
CHARACTER*12 CBELL       ! 12-character variable.
CBELL="Attention!<BEL>"  ! CBELL will print
                         !    (type) as
                         !    Attention!(beep)□.
```

A character example that uses names is

```
        CHARACTER CVAR           ! 1-character
                                 !    variable
        CHARACTER*20 NAMES(5)    ! 5-element array
        CHARACTER*20 SCRATCH(5)  ! 5-element
                                 !   scratch array
        NAMES(1) = 'Heitz, T.'   ! Assign
                                 !   characters to
                                 !   NAMES elements
        NAMES(2) = 'McDonald, M.'
        NAMES(3) = 'Pearl, D.'
        NAMES(4) = 'Perry, D.'
        NAMES(5) = 'Phong, D.'

C       Try a simple sort...

        IF ( NAMES(1)(1:3) .GT. (NAMES(2)(1:3) )
     +     SCRATCH(1) = NAMES(1)
        ...
```

The third example shows how to eliminate a character — in this case a blank — from a character entity.

```
        CHARACTER*20 C5      ! Original variable
        CHARACTER*20 CNEW    ! Destination variable
        CHARACTER XX / ' '/  ! Variable with blank
        ...                  ! Program assigns
        ...                  !   some value to C5.
        J = 0                ! Count of nonblanks
        DO 5 I = 1, 20       ! Do for all C5
                             !   characters
C               If ' ', get the next character.
                IF ( C5(I:I) .EQ. XX ) GOTO 5
                J = J + 1    ! Increment the count.
                CNEW (J:J)=C5(I:I) ! Put nonblank
                                   !    character
                                   !    in CNEW.
    5   CONTINUE
        ...
```

Assigning Initial Values with the DATA Statement

The DATA statement initializes variables and array elements to the values you specify. It directs the F77 processor to insert values before program execution. Having initialized an entity with DATA, you cannot reinitialize that entity with DATA.

The DIMENSION and type statements can also assign values to entities, but DATA has a little more flexibility with arrays. Also, DATA can assign initial values to both named and blank COMMON blocks in any program unit.

DATA is functionally *different* from the assignment statement, which can assign and reassign values dynamically during program execution.

DATA is a nonexecutable statement. If you label it, you cannot use that label in another statement.

The form of the DATA statement is

DATA vlist / clist / *[,vlist / clist /] [...]*

where:

vlist — is a list of one or more variable, array element, character substring, and/or array names. It can also include implied DO lists, described in Chapter 5.

clist — is a list of constants that will be assigned to the vlist items. A constant can be a numeric, character, logical, or Hollerith constant; or numerical, character, or logical variable/array element name; and/or it can be a PARAMETER, described in the next section. The clist can also include a repeat count, described below.

The number of names in the vlist must match the number of constants in clist; if not, the compiler signals an error. For example, if IVAR is a variable name, the statement

```
DATA IVAR / 3, 7 /
```

would evoke an error message. Similarly, the statements

```
DIMENSION IARY(4)
DATA IARY / -16, 23, 17 /
```

result in a compiler error message.

To initialize array elements, you can insert the element names one by one, then their values; e.g.,

```
      DIMENSION ARY(4)
      DATA ARY(1), ARY(2), ARY(3), ARY(4)
     +     / 1.0, 3., -5.2, 13 /
```

Or, to insert the same values in multiple elements, you can insert elements and a repeat count. A repeat count has the form:

n*constant

where n is the number of times you want the constant inserted. For example,

```
DIMENSION ARY(3)
DATA ARY(1), ARY(2), ARY(3) / 7.9, 2*4.0 /
```

You need not list the element names one by one. The two previous examples are simplified by the statements

```
DIMENSION ARY_1(4), ARY_2(3)
DATA ARY_1 / 1.0, 3., -5.2, 13 /
DATA ARY_2 / 7.9, 2*4.0 /
```

If the repeat count specifies too many elements for the array, the compiler signals an error. You can specify a *portion* of an array in a repeat count to initialize part of the array — as long as the clist contains exactly as many constants as there are array elements. For example,

```
      DIMENSION IX(8)
      DATA IX / 15, 3*-11,
     +          0,  38,    2*0 /
```

is correct.

The order and grouping of names and value lists is unimportant as long as you maintain the correct name/value association. For example, the following three statements have identical effects:

```
DATA B, N(1,1), L, CHAR / 2*9, .TRUE., 'Price:' /
DATA B, N(1,1) /2*9 /, L /.TRUE./, CHAR /'Price:'/
DATA L, B, CHAR /.TRUE., 9, 'Price:'/, N(1,1) /9/
```

Each DATA statement initializes variable B and element N(1,1) to 9, logical variable L to .TRUE., and character variable CHAR to 'Price:'.

If the data types of a variable and its corresponding constant do not match, F77 converts the constant to the variable's type according to the rules for assignment given earlier in this chapter.

To initialize a variable or element of type complex, be sure to enclose the complex value in parentheses; otherwise, the compiler will convert the first value, assign it, then encounter the second and, assuming too many values, will signal an error. A correct example is

```
COMPLEX X
DATA X / (2.0, 4.5) /
```

DATA Examples

```
REAL A
DIMENSION R(8)
DATA A, R / 7.1, 8*3.3 /
```

This initializes A to 7.1 and the 8 elements of R to 3.3.

```
REAL R(10)
CHARACTER*10 C, C1
DATA R(3), C, C1 / 7E-4, 'Size is', 'Size was' /
```

This initializes element R(3) to 7E-4 (=.0007), variable C to 'Size□is□□□', and variable C1 to 'Size□was□□'. R(1), R(2), and R(4)-R(10) are undefined.

```
DIMENSION IY(20), JY(20)
DATA ( IY(I), JY(I), I = 1,20 )   / 20*1, 20*9 /
```

This example uses an implied DO (Chapter 5) to initialize elements 1–10 of arrays IY and JY to 1. It also initializes elements 11–20 of these arrays to 9.

```
      COMPLEX*16 MENU_CHOICES(10)
C     "PR" = PRINT, "MID-TM" = "M-T" = MID-TERM
      DATA MENU_CHOICES     /
     1  'PR MID-TM ROSTRS',
     2  'ENTER M-T GRADES',
     3  'PR MID-TM GRADES',
     4  'PR FINAL ROSTERS',
     5  'ENTER FIN GRADES',
     6  'PR FINAL GRADES ',
     7 4*'                '   /   ! Reserve for
C         future expansion of jobs for the
C         registrar's office.
```

This example stores character constants in a numeric array.

```
LOGICAL*1 COUNT(5), MIDDLE_INIT(10)
LOGICAL*1 L1(4),    HOLL_QUOTE(4)
DATA COUNT          / 5*0 /
DATA MIDDLE_INIT    / 10*' ' /
DATA L1             / 4*.TRUE. /
DATA HOLL_QUOTE     / 4*1H' /
```

This example places numbers, characters, and logical constants into logical*1/byte arrays. Only AOS/VS F77 programs have the logical*1/byte data type.

Defining Constants with PARAMETER

PARAMETER associates a symbolic name with a constant value. Like DATA, it is a nonexecutable statement and, once it has defined a name's value, cannot redefine that name's value within the program unit. PARAMETER can make a program easier to understand and maintain by defining an often-used constant once. Thereafter, the program can refer to the constant by name. If, in the future, you want to change the constant, you need change it only once, in the PARAMETER statement.

PARAMETER is a specification statement and as such must precede statement function definitions and executable statements in a program unit. The form of PARAMETER is

PARAMETER (p = expr *[,p = expr] [...]*)

where:

p is a symbolic name.

expr is a numeric, logical, or character expression whose value is determined by the compiler. All operators are allowed, but a numeric value may be raised only to an integer power; e.g., 3**2.1 is illegal.

The constant value you assign in expr must match the general type of the symbolic name, p. If the name is implied (by name rule) or declared as a numeric data type, expr must be numeric. If the name is type character, expr must be type character and enclosed in apostrophes or quotation marks. If the name is type logical, expr must be .TRUE. or .FALSE. or be a constant expression that evaluates to .TRUE. or .FALSE.

You cannot reassign the constant value in this program unit; the name will retain its PARAMETER value throughout.

You can use the symbolic name in a subsequent DATA statement or to form part of another PARAMETER constant. You can use it to define an entire format specification, but you cannot use it for parts of a format specification (such as edit descriptors). For example,

```
CHARACTER*10 FOO

PARAMETER (FOO = '(1X, F9.2)' )
...
WRITE (*, FOO) A
```

is legal.

You can define a character entity's length as an asterisk if later you use PARAMETER to define that character name as a constant. For example,

```
CHARACTER*(*) CPARAM

PARAMETER (CPARAM = 'A named character constant')
...
```

This can save time for often-used character strings that may change with the passage of time, as shown in the last example below.

You cannot refer to a substring of a character constant named with PARAMETER.

PARAMETER Examples

3 Specific Examples:

```
DOUBLE PRECISION PI              ! Real*8
PARAMETER (PI = 3.14159265361)   !   parameter PI.
...
AREA = PI*R**2                   ! Use PI.

PARAMETER (QZ = 0.1731D-7)       ! Parameter QZ.
...
RZ = QZ / K                      ! Use QZ.

PARAMETER (M=2, N=4, L9=M*N)     ! Parameters
                                 !   M, N, L9.
DIMENSION JA(N,L9)               ! Use N, L9.
```

A contextual example:

```
      PARAMETER (IP=10000)       ! Param IP, number
                                 !   of elements in
                                 !   PARTS file.
      ...
      DIMENSION PARTS(IP)        ! Use IP...
      DO 50 I = 1, IP            !    ...and again.
      ...
   50 CONTINUE
```

Using an asterisk for CHARACTER length:

```
CHARACTER*(*) PART               ! Use * for length
CHARACTER*10 P_NUM

PARAMETER (PART='093-162-')      ! Define ...
...
P_NUM = PART  // '03'            !    ... and use
                                 !    the constant
```

P_NUM contains the value 093-162-03.

A logical*1/byte example (AOS/VS only):

```
LOGICAL*1 TINY_CON          ! TINY_CON contains
PARAMETER (TINY_CON = 'B') !   'B' = 01000010
```

Using EQUIVALENCE

Sometimes — with either a single program unit or multiple units — it is desirable to have two or more names refer to the same storage area. The EQUIVALENCE statement allows you to do this.

EQUIVALENCE is a specification statement. As such, it must precede statement function definitions and executable statements in a program unit. The form of EQUIVALENCE is

EQUIVALENCE (vlist) *[, (vlist)] [...]*

where:

vlist is a list of *two or more* variables, arrays, and/or array elements that have integer constants for subscripts. Each vlist must contain at least two names. In a subprogram, names of dummy arguments must not appear in a vlist. If a variable name is also a function subprogram name, this name must not appear in a vlist.

For each vlist, the compiler allots storage to the largest entity, then associates other items in this vlist with this storage area. You can include in an EQUIVALENCE statement:

- any noncharacter data type with any other noncharacter data type
- any character type with another character type
- (with one exception) any character type with any noncharacter type

The following statements are legal.

```
REAL*4      VAR_A
INTEGER*2   INT_2
REAL*8      DVAR_A
REAL*8      DVAR_B(3)
EQUIVALENCE (INT_2, VAR_A)  ! INT_2 and
!   the leftmost 16 bits of VAR_A share.
EQUIVALENCE (DVAR_A, DVAR_B(1))  ! DVAR_A
!   and DVARB(1) share.
```

The following statements are also legal.

```
REAL*4      RVAR4
CHARACTER*4 CVAR4
REAL*8      RVAR8
CHARACTER*8 CVAR8
EQUIVALENCE (RVAR4, CVAR4)
EQUIVALENCE (RVAR8, CVAR8)
...
CVAR4 = 'ABCD'
WRITE (10, '(F10.7, 2X, A4)') CVAR4, RVAR4
```

The output from the WRITE statement is

ABCD 4.1414223

The bit pattern for 'ABCD' is <101><102><103><104>, which is the same as the bit pattern of the real*4 number 4.1414223.

The following statements show legal EQUIVALENCE relationships among logical*1/byte, character, and integer*2 data.

```
BYTE          BYTE_1
CHARACTER*1   CHAR_1
INTEGER*2     INT_2
EQUIVALENCE   (BYTE_1, CHAR_1)
EQUIVALENCE   (BYTE_1, INT_2 )  ! BYTE_1
!   and the high-order bits of INT_2 share.
```

Here is the exception to including character and noncharacter data types in an EQUIVALENCE statement: all nonbyte-aligned entities must not be on an odd byte boundary. The following statements violate this restriction.

```
CHARACTER*1 L1_ARRAY(4)
INTEGER*2 INT_2
EQUIVALENCE (L1_ARRAY(2), INT_2)
! INT_2 doesn't begin on a word boundary.
```

You must have dimensioned an array, via DIMENSION, type statement, or COMMON statement, before you can use it or any of its elements in an EQUIVALENCE statement. An EQUIVALENCE statement cannot contain dummy argument or function names.

You can refer to the first array element in an EQUIVALENCE expression with either:

- A Complete Subscript. As always, the number of subscripts equals the number of dimensions in the array. For example, A(1,1,1) indicates the first element of three-dimensional array A (assuming all three lower bounds are 1).
- No Subscript. This (e.g., A) indicates the first element of the array.

Using an element of one array and an element of another array in an EQUIVALENCE statement implicitly places all other corresponding elements of the arrays into the memory-sharing relationship. For example, the statements

```
DIMENSION A(5,4), B(20)
EQUIVALENCE (A(1,1), B(1))
```

not only indicate that array elements A(1,1) and B(1) share the same storage locations, but that A(2,1) and B(2), A(3,1) and B(3), etc., also share the same storage locations.

```
DIMENSION C(20), D(20)
EQUIVALENCE (C(20), D(1))
```

indicate that C(20) and D(1) share the same storage location. This is the only storage location that C and D share.

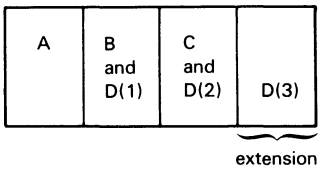
EQUIVALENCE in Common Blocks

If an entity is part of a common block, you cannot use it in an EQUIVALENCE statement with another entity in that block or with a component in any other common block. (The COMMON statement is described in Chapter 7.)

You can use an array element in the program unit in an EQUIVALENCE statement with an entity of a common block — as long as doing so does not add locations to the beginning of the block. For example, the statements

```
COMMON A,B,C
DIMENSION D(3)
EQUIVALENCE (B,D(1))
```

extend the common block in the following manner:



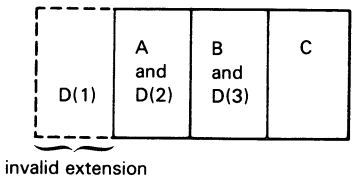
DG-27091

As described above, array D and some common block entities share memory because of an EQUIVALENCE statement. But, since locations are added to the *end* of the common block, this is allowed.

You cannot place an array element and a common block entity in an EQUIVALENCE statement if — via the insertion of the whole array — it would extend the block by placing array elements *before* the block's beginning. The statements

```
COMMON A,B,C
DIMENSION D(3)
EQUIVALENCE (B,D(3))
```

attempt to extend the common block invalidly:



DG-27092

This kind of mistake evokes a compiler error message.

EQUIVALENCE Examples

```
DIMENSION A(100), B(0:99)
EQUIVALENCE (A(1), B(0))
```

Array B, with its 100 elements numbered 0–99, and array A, with its 100 elements numbered 1–100, share the same storage locations. A(1) and B(0) will contain the same value, as will A(2) and B(1), A(3) and B(2), and so on.

```
DIMENSION    VALUES(10), SUBTOTALS(9)
EQUIVALENCE (VALUES,SUBTOTALS), (VALUES(10),TOTAL)
```

This specifies that elements 1 through 9 of VALUES and SUBTOTALS share the same locations and that VALUES(10) and variable TOTAL share the same location.

For another example, assume that in the first part of your program, you will need to use a large array, X. Then you will need to use arrays Y and Z, but will no longer need X. So you will use X's storage for Y and Z:

```
DIMENSION   X(40,40), Y(20,20), Z(4,5)
EQUIVALENCE (X(1,1), Y(1,1)),    (X(1,11), Z(1,1))
```

Another way to write these statements is

```
DIMENSION   X(40,40), Y(20,20),   Z(4,5)
EQUIVALENCE (X, Y),               (X(1,11), Z)
```

This EQUIVALENCE statement has arrays Y and Z use some of the same storage locations as array X. Y will use the first 400 (20*20) locations of X, so you call for Z to start at the 401st element of X — because you don't want Y and Z to overlap. The 401st element of X is X(1,11) because of the sequential storage of X in blocks (think of them here as columns) of 40:

```
X(1,1)-X(40,1),
X(1,2)-X(40,2),
X(1,3)-X(40,3),
      ...,
X(1,10)-X(40,10),
X(1,11), X(2,11),
      ...,
X(40,11),
X(1,12)-X(40,12),
      ...,
X(1,40)-X(40,40)
```

NOTE: For a multidimensional array, F77 by default varies the leftmost subscript most rapidly when it sequentially processes the array.

See Figure 3-1.

Other Ways to Assign Values

The READ statement obtains values from the terminal or another file and assigns them to specified entities. Within any I/O statement, you can use an implied DO list to assign values. READ and the implied DO are described in Chapter 5.

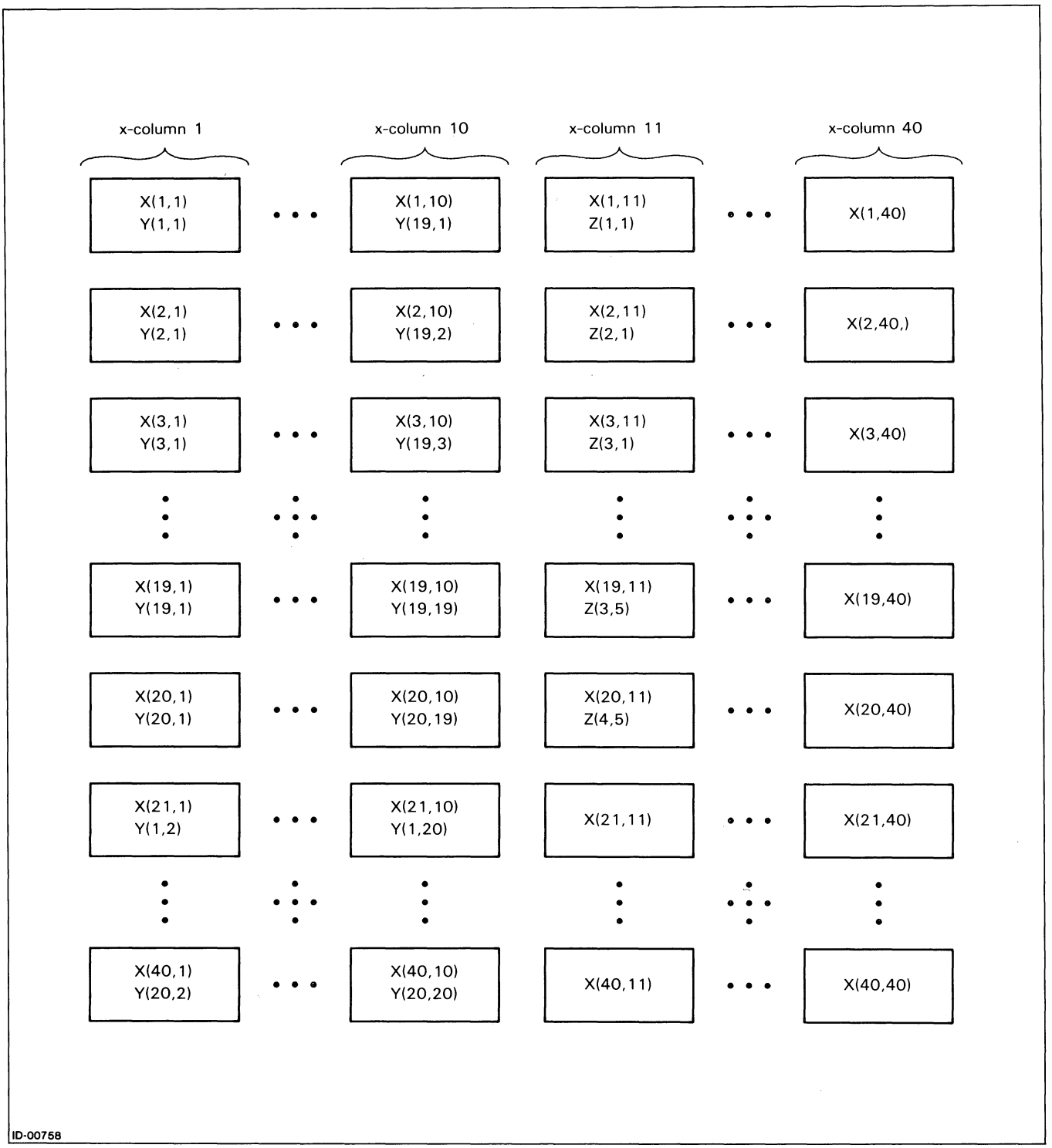


Figure 3-1. An Example of EQUIVALENCE Storage

End of Chapter

Chapter 4
Directing Flow within a Program Unit

F77 executes statements sequentially, from the beginning of a program unit to its end — unless you redirect control with a control statement. This chapter describes using the statements that allow you to redirect control within a program unit. The sections proceed:

- Using GO TO
- Using IF Conditionals
- Using DO Loops
- Using Unlabelled DO Loops
- Using DO WHILE Loops
- Using CONTINUE
- Pausing, Stopping, and Ending the Program

To change the flow of a program, use GO TO (unconditional, computed, or assigned) or IF (arithmetic, logical, or block). To loop, use DO. As a placeholder before the next statement, use CONTINUE. To suspend or terminate execution, use PAUSE or STOP respectively; and to mark the end of a program unit use END.

Altering the flow *between* program units, via CALL and RETURN, is described in Chapter 7.

Using GO TO

A GO TO statement transfers control to a labeled statement. The three types of GO TO statements are

- Unconditional GO TO
- Computed GO TO
- Assigned GO TO

Unconditional GO TO

The unconditional GO TO simply transfers control to the statement specified. Its form is

GO TO s

where:

s is the label of an executable statement within the current program unit.

If the label s refers to a nonexistent label or nonexecutable statement, the compiler signals an error. You may use the symbols GO TO or GOTO.

You should label the first executable statement that follows an unconditional GO TO statement, or the compiler will signal a warning. For example, the compiler gives a warning message as a result of the three statements

```
      GO TO 30
      I = 5
30    PRINT *, 'I IS ', I
```

because the second statement will never execute.

Many unconditional GO TOs can make a program difficult to understand and maintain (so-called "spaghetti logic"), so you may want to use as few as possible. See the description of the block IF statement.

Unconditional GO TO Example

```
      GO TO 25    ! The unconditional GO TO.
10    R = 2.0*A   ! Label is recommended.

      ...

25    I = I + 1   ! GO TO directs control here.
```

GO TO transfers control unconditionally to the statement labelled 25. Unless this program unit directs control to the statement labelled 10, it and the statements that immediately follow it will never be executed.

Computed GO TO

A computed GO TO transfers control to one of several statements depending on the value of an integer expression. You can use the expression to select the desired statement.

The form of a computed GO TO is

GO TO (s_1 *[*, s_2 , ..., s_n*]*) *[,]* iexpr

where:

s is the label of an executable statement within this program unit.

iexpr is an integer (including logical*1/byte) expression.

F77 transfers control to the statement *whose position* in the s list corresponds to iexpr. If iexpr is 1, control goes to statement label s_1; if iexpr is 2, control goes to statement label s_2; if iexpr is *n*, control goes to statement label s_n. If iexpr is out of range (less than 1 or more than *n*), control passes to the next executable statement. You can use the same statement label in more than one s item. However, a computed GOTO statement cannot have more than n statement labels. For AOS/VS, n is 254; for AOS, F7716, MP/AOS, and MP/AOS-SU, n is 63.

A statement label must refer to an executable statement within the same program unit, or the compiler will signal an error.

Computed GO TO Example

```
      GO TO (10, 100, 50, 10, 9) K_VAR
      PRINT *, 'K_VAR HAS AN OUT-OF-RANGE ',
     +         'VALUE OF ', K_VAR
```

The variable K_VAR must be 1, 2, 3, 4, or 5 for a branch to occur. If K_VAR is 1, control passes to the statement labelled 10; if K_VAR is 2, control passes to the statement labelled 100, and so on. If K_VAR is less than 1 or greater than 5, the next statement — PRINT *, 'KVAR HAS ... — executes.

Assigned GO TO

An assigned GO TO assigns a statement label to a symbolic name. You can then refer to the label by the symbolic name.

You must assign a statement number to an integer variable before using this form of GO TO. To assign a statement number, use the ASSIGN statement.

ASSIGN Statement

ASSIGN associates a statement label with an integer variable name. It allows you to use this name in subsequent assigned GO TO statements. ASSIGN's form is

ASSIGN s TO v

where:

- s is the label of an executable or FORMAT statement in the current program unit.
- v is an integer variable name. It must be integer*4 under AOS/VS but can be any length under AOS, F7716, MP/AOS, and MP/AOS-SU.

ASSIGN places the statement label s in variable v. For example,

```
ASSIGN 20 TO ICOMP
```

places the *label* 20 in ICOMP. The = operator cannot do this; for example, ICOMP=20 assigns the *value* 20 — not the label 20 — to ICOMP. Then, you may place the statement GOTO ICOMP in your program; it has the same effect as GOTO 20 (as long as ICOMP doesn't change).

You cannot use the variable in arithmetic expressions *while it is defined with the statement label.*

The Assigned GO TO Statement

An assigned GO TO transfers control to a previously assigned statement label. Its form is

GO TO v *[[,] (s_1, s_2... s_n)]*

where:

- v is an integer variable name: integer*4 in AOS/VS, either length in AOS, F7716, MP/AOS, and MP/AOS-SU.
- *s* is the label of an executable statement within the current program unit.

An assigned GO TO transfers control to the statement whose label you assigned to the integer variable v.

The comma between v and s_1 is optional, as are all *s* statement labels. F77 will always pass control to the statement label assigned to v, regardless of the other labels in the *s* list. For example, the statements

```
GOTO LABEL_1
```

and

```
GOTO LABEL_1, (10, 27, 20, 40)
```

produce the same results (as long as LABEL_1 previously received a statement label).

Be very careful with assigned GO TOs. v must be defined with the value of an executable statement label; if not, the assigned GO TO will cause a runtime error that will terminate your program. The compiler does not check the label for validity.

You should label the first executable statement that follows an assigned GO TO statement. If you don't, the compiler will signal a warning.

Assigned GO TO Example

```
      REAL NEW_AMT
      ...
      IF (LOG_EXPR) THEN
            ASSIGN 600 TO JAIL
            GO TO JAIL      ! Go directly to
                            !    jail.
      END IF
230   NEW_AMT = OLD_AMT + 200 ! Collect $200.00
                            !    (statement
                            !    label is
                            !    recommended).
      ...
600   NEW_AMT = OLD_AMT       ! Do not collect
                            !    $200.00.
      ...
```

Control transfers to the statement labelled 600.

Using IF Statements

IF statements are central to program control. There are three kinds of IF statements:

- Arithmetic IF
- Logical IF
- Block IF

The Arithmetic IF

An arithmetic IF transfers control conditionally to one of three statements, based on the value of an expression. Its form is

IF (expr) s_1, s_2, s_3

where:

expr is an arithmetic expression of any type except complex.

s is the label of an executable statement within the current program unit.

F77 evaluates the arithmetic expression. If the value is less than 0, F77 passes control to statement s_1; if the result is 0, it passes control to statement s_2; if it is greater than 0, it passes control to s_3.

You must specify all three statement labels. They need not refer to different statements, but they must refer to executable statements within the current program unit. If not, the compiler signals an error.

An arithmetic IF statement tests for exact 0. If expr is type real and involves arithmetic, it may not produce true 0. This happens because the results of real computations — including those with an apparent value of 0 — often are approximate. Thus a real arithmetic expression that might appear to produce exact 0 will probably not produce an exact 0. You can deal with this problem by testing for a "small" difference with the ABS (absolute value) function. See the fifth and sixth examples next. They begin with respective comments IF A EQUALS B and IF A IS CLOSE ENOUGH TO B.

Arithmetic IF Examples

```
      IF (INDEX) 100,200,300

      IF (K(I,J) - L) 10,4,30

      IF (IQ * IR) 5,5,2

      IF (JTEST1**4 - JTEST2/2) 10,15,15

C     IF A EQUALS B, BRANCH TO STATEMENT 40;
C          OTHERWISE, BRANCH TO STATEMENT 50.
      A = 1.0
      B = 3.0*(1.0/3.0)
      IF ( A - B ) 50, 40, 50  ! Statement 50
                               !    is next.

C     IF A IS CLOSE ENOUGH TO B, BRANCH TO
C          STATEMENT 40; OTHERWISE, BRANCH TO
C          STATEMENT 50.
      A = 1.0
      B = 3.0*(1.0/3.0)
      EPSILON = 1.0E-6     ! Allowable difference
                           !    between A and B
      IF ( ABS(A - B) - EPSILON ) 40, 40, 50
           ! Statement 40 is next.

C     NAME is INTEGER*4 and read under A1 format
      IF ( NAME(1) - 'J' ) 80, 90, 80
C     IF ( NAME(1) - 1HJ ) 80, 90, 80  IS
C          EQUIVALENT TO THE PREVIOUS STATEMENT.
   80 CONTINUE  ! NAME(1) is not 'J' =
                !    <112><040><040><040>
      ...
   90 CONTINUE  ! by processing Jensen, Jones,
                !    Judd, etc.
      ...
```

The Logical IF

A logical IF executes an accompanying statement if an expression evaluates to true; otherwise it passes control to the next executable statement.

Logical IF allows you to test a value of any data type —arithmetic, logical, or character, using relational or logical operators — against another value and act if the result is true.

The form is

IF (expr) st

where:

expr is a logical expression.

st is any executable statement except a block IF, DO, DO WHILE, ELSE, ELSE IF, END DO, END IF, END, or another logical IF statement.

F77 evaluates the logical expression. If the expression evaluates to .TRUE., F77 executes statement st. Control then passes to the first executable statement following statement st, unless st directs control elsewhere. If the expression evaluates to .FALSE., then F77 passes control to the next executable statement and bypasses st.

The parentheses around expr are required.

When F77 tests a logical expression, it examines *only* the rightmost bit. A value of 1 results in true; a value of 0 results in false. For example, consider the following statements.

```
LOGICAL   L_D        ! default length
LOGICAL*1 L_1        ! AOS/VS only

L_D = .TRUE.         ! L_D contains all 1s
L_1 = 5              ! L_1 contains 00000101

IF ( L_D )           PRINT *, 'TRUE A'
IF ( L_1 )           PRINT *, 'TRUE B'
IF ( L_D .EQV. L_1 ) PRINT *, 'TRUE C'
```

At runtime, all three PRINT statements execute.

If you want to test for real values in a logical IF expression, remember that real values are often approximate. For example, the logical expression A.EQ.0.0 of the statement IF(A.EQ.0.0)ISWITCH=0 may not be true when A apparently is zero, because real values are often approximate. In this example, a preceding statement such as A=9.0*(1.0/9.0)-1.0 results in A having a value slightly different from zero. In such situations, you can do one of two things: either use operators .GT., .GE., .LT., and/or .LE. instead of .EQ., or test values for equality within a reasonable tolerance, using the ABS function. For example, instead of IF(A.EQ.B), you could use

```
IF ( ABS(A - B) .LE. 1.0E-6 ) ...
```

which will be true whenever A and B are equal within a reasonable tolerance, .000001. (However, the floating-point bit pattern for .000001 isn't exactly equal to .000001.)

Logical IF Examples

```
      IF ( X .GT. 0.0 ) GO TO 25

      if ( INDEX .GT. 1 ) INDEX = 1

      IF ( A .LE. 0.0 ) A = 0.

      IF ( (A .GT. 3.0 ) .AND. (J .EQ. 1) )
     +    GO TO 50

      IF ( ARY(I,J) .LT. ARY(I+1,J) )
     +    CALL SWAP (ARY, I, J)

      IF ( (T .GT. 3.0) .OR. (TH .GT. 1.0) )
     +    F = SIN (R)

      IF ( .NOT. (I .GT. 5  .AND.  I .LE. 20) )
     +    NEW_SUM = 0

C     IF A EQUALS B, BRANCH TO STATEMENT 40;
C          OTHERWISE, BRANCH TO STATEMENT 50.
      A = 1.0
      B = 3.0*(1.0/3.0)
      IF ( A .EQ. B ) GO TO 40    ! Statement 50
                                  !    is next.
   50 NEXT_TOTAL = 0

C     IF A IS CLOSE ENOUGH TO B, BRANCH TO
C          STATEMENT 40; OTHERWISE, BRANCH TO
C          STATEMENT 50.
      A = 1.0
      B = 3.0*(1.0/3.0)
      EPSILON = 1.0E-6    ! Allowable difference
                          !    between A and B
      IF ( ABS(A - B) .LE. EPSILON ) GO TO 40
           ! Statement 40 is next.

C     A Character example:
      CHARACTER*10 TX / 'abcd' /
      CHARACTER*10 TY / 'defg' /
      CHARACTER*10 SCR
      ...
      IF (TX(1:1) .GT. TY(1:1)) SCR(1:1)=TX(1:1)
      ...

C     NAME is INTEGER*4 and read under A1 format
      IF ( NAME(1) .EQ. 'J' ) GO TO 90
C     IF ( NAME(1) .EQ. 1HJ ) GO TO 90    IS
C          EQUIVALENT TO THE PREVIOUS STATEMENT.
C     NAME(1) is not 'J' = <112><040><040><040>
      ...
   90 CONTINUE  ! by processing Jensen, Jones,
                !    Judd, etc.
      ...
```

The Block IF

A block IF starts with a block IF statement and ends with an END IF statement. Its block form assists you in constructing structured programs.

A block IF can also contain multiple ELSE IF...THEN statements and one ELSE statement (described below). The original IF...THEN, each ELSE IF...THEN, and the ELSE can have its own block of FORTRAN statements. The ELSE statement must follow any ELSE IF...THEN statements.

The form of a block IF is

```
IF (expr) THEN
   [if block of statements]
[ELSE IF (expr) THEN
   else if block of statements]
[ELSE
   else block of statements]
END IF
```

where:

expr is a logical expression.

Only IF(expr)THEN and END IF are mandatory; ELSE IF(expr)THEN and ELSE are optional. These statements all relate to the block IF, so we describe them together.

Term	Description
IF block	The statements between IF(expr)THEN and the next corresponding ELSE IF, ELSE, or END IF statement.
ELSE IF block	The statements between ELSE IF and the next corresponding ELSE IF, ELSE, or END IF statement.
ELSE block	The statements between ELSE and the next corresponding END IF statement.

```
IF ( J .EQ. K ) THEN
END IF
```

is valid but a waste of time because it would do nothing.

Recall from the previous section (The Logical IF) that when F77 tests a logical expression, it examines *only* the rightmost bit. This examination also occurs in block IF and in ELSE IF statements.

Block IF Execution

F77 evaluates the logical expression in the IF(expr)THEN. If the expression is true (.TRUE.), F77 executes the IF block statements sequentially. Control then passes to the END IF, which passes control to the next executable statement (unless an IF block statement directs control elsewhere). For example,

```
IF (2 .EQ. 2) THEN      ! The expression is true.
                        ! IF block next.
    I = J               ! Execute this
    X = Y               !   and this.
                        ! Exit to END IF.
ELSE
                        ! ELSE block next.
    I = K               ! Not executed.
    X = Z               ! Not executed.
END IF                  ! Control goes on.
```

If the expression in the IF(expr)THEN statement does not evaluate to true (is .FALSE.), control passes to the next ELSE IF or ELSE statement; if there are no ELSE IF or ELSE statements, or their blocks are empty, control passes to END IF and to the next executable statement.

If an ELSE IF statement gets control, F77 evaluates the expression as for the original IF; if true, the ELSE IF block is executed. Control then goes to the next executable statement after the END IF (unless directed elsewhere in the ELSE IF block). If the ELSE IF expression is false, control passes to the next ELSE IF, ELSE, or END IF statement. For example,

```
IF (2 .EQ. 4) THEN          ! False.
    I = J                   ! Block 1:
    X = Y                   !   skip.
ELSE IF (2 .EQ. 1) THEN     ! Check;
    I = K                   ! Block 2:
    X = Z                   !   skip.
ELSE IF (2 .EQ. 2) THEN     ! Check;
    I = L                   ! Block 3:
    X = V                   !   execute.
END IF                      ! Proceed.
```

Read the following example and convince yourself of these facts: Block 1 executes if A is true and B is true; block 2 executes if A is true and B is false; block 3 executes if A is false.

```
IF (A) THEN
       IF (B) THEN
              ! Block 1
       ELSE
              ! Block 2
       END IF
ELSE
       ! Block 3
END IF
```

If all preceding expressions evaluate to false and there is an ELSE statement, F77 executes all statements in the ELSE block. Control then goes to END IF and the next executable statement (unless an ELSE block statement directs it elsewhere). For example,

```
IF (2 .EQ. 4) THEN      ! The expression is false.
    I = J               ! Skip this
    X = Y               !   and this.
ELSE
                        ! Start here.
    I = K               ! Execute this
    X = Z               !   and this.
END IF                  ! Control goes on.
```

We have aligned the ELSE and ELSE IF statements, and indented the block statements for clarity; we suggest that you also do this. F77 disregards these spaces, or extra tabs, but you can use them to identify quickly the decision structure of the block IF.

To sum it up, you can use virtually any number of ELSE IFs in a block IF, before an ELSE. F77 will process them sequentially and will execute the block of the first IF or ELSE IF whose expression evaluates to true; then it will proceed to END IF. If no ELSE IF expression evaluates to true, F77 will execute the ELSE block statements (if present) and proceed to END IF. For example:

```
IF...THEN
    block
ELSE IF...THEN
    block
ELSE IF...THEN
    block
ELSE IF...THEN
    block
ELSE
    block
END IF
```

ELSE and ELSE IF statements aren't allowed in an ELSE block.

You can transfer control (via GO TO) *to* an IF...THEN statement, but you cannot transfer control into the block IF from outside it. You *can* transfer out of a block IF with GO TO.

A practical example of a block IF is

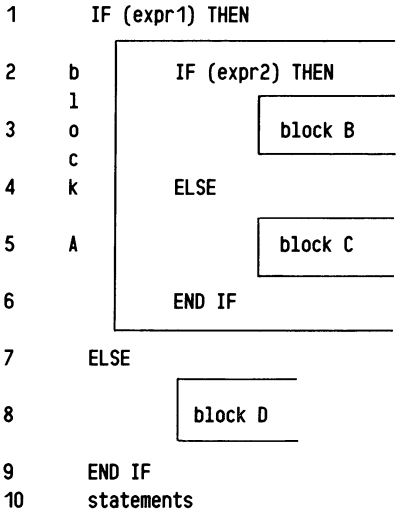
```
! This example computes regular pay and overtime
!    for an hourly employee. The program has
!    already obtained values for HOURS AND RATE.

...
IF ( HOURS .GT. 40.0 ) THEN
       REGPAY = RATE * 40.0
       OVTPAY = 1.5 * RATE * (HOURS - 40.0)
ELSE
       REGPAY = RATE * HOURS
       OVTPAY = 0.0
END IF
PAY = REGPAY + OVTPAY
...
```

If the hours worked are greater than 40, the IF block statements are executed, the ELSE statements are skipped, and control passes through the END IF to the PAY assignment statement. If the hours worked are not greater than 40, the IF block statements are skipped, the ELSE statements executed, and control passes on through the END IF to the PAY assignment statement.

Nested Block IFs

You can nest block IFs (enter a block IF while in a block IF), but make sure that you close each block IF with an END IF. If the number of END IFs doesn't equal the number of IF...THENs, the compiler signals an error.

Each IF block has its own level, and F77 executes statements within that level before proceeding to another block. For example, consider the following nested block IFs:



First, F77 executes statement 1; if expr1 is true, it executes statement 2 and then either 3, 6, 9, and 10 (expr2 is true) or 4, 5, 6, 9, and 10 (expr2 is false).

If 1 is not true, F77 bypasses statements 2–6 and executes 7, 8, 9, and 10.

Using an extra tab stop for each nested block IF statement, and indenting the block of statements a few spaces, can help you keep track of the levels.

By establishing a single entry point and exit point for what can be a very elaborate procedure, nested block IFs can help you structure your programs and prevent bugs.

Block IF Example

Here is an example that tests and solves quadratic equations. It gets values for A, B, and C from the terminal via the statement READ * and writes to the terminal via the statement PRINT *.

```
! First, obtain the discriminant if possible.

  READ *, A, B, C
  IF ( ABS(A) .GT. 0.0 ) THEN
        DISCRIM = B**2 - 4.0*A*C
        IF ( DISCRIM .GE. 0.0 ) THEN
            ROOT1 = (-B + SQRT(DISCRIM)) / (2*A)
            ROOT2 = (-B - SQRT(DISCRIM)) / (2*A)
            PRINT *, 'Root1 is ', ROOT1
            PRINT *, 'Root2 is ', ROOT2
        ELSE
            PRINT *, 'Roots are complex.'
        END IF
  ELSE
        PRINT *, 'Equation is not quadratic.'
  END IF
```

Here, the outer block IF tests for range and the inner block IF does the work.

Using DO Loops

DO loops execute a group of statements zero or more times. DO loops are among the most useful structures in FORTRAN. They, like IFs, allow you to specify iteration. Furthermore, they can be implied in input/output lists (Chapter 5) to further simplify things.

The form of a DO loop is

DO s *[,]* i = e_1, e_2 *[*, e_3*]*

sts

s terminal statement

where:

s is the label of an executable statement within the current program unit after the DO statement. This is the *terminal statement*. It can be any statement except: another DO, a DO WHILE, an unconditional or arithmetic GO TO, IF...THEN, ELSE IF, ELSE, END DO, END IF, RETURN, STOP, or END. CONTINUE is a traditional choice for a terminal statement. If the terminal statement is a logical IF, it may contain any executable statement except DO, DO WHILE,IF...THEN, ELSE IF, ELSE, END DO, END IF, END, or other logical IF.

i is an integer (including logical*1/ byte) or real variable name, called the *DO-variable.*

e_1 is an integer (including logical*1/byte) or real expression that is the initial value for the DO-variable, called the *initial parameter.*

e_2 is an integer (including logical*1/byte) or real expression that is the terminal value for the DO-variable, called the *terminal parameter.*

e_3 is an integer (including logical*1/byte) or real expression that gives the number by which the DO-variable will be incremented in each pass through the loop; it is called the *incrementation parameter.* If you omit this parameter, the DO-variable will be incremented by the default value, 1, in each pass. It can have any value valid for its data type, *except 0.* An increment of 0 would run the DO loop forever.

sts are any statements, except those that redefine i or make it undefined.

The DO statement sets up a loop that begins at the DO and ends at the terminal statement, labeled s. This set of statements is *the range of the DO.* By default, the range of the DO loop can execute zero or more times; but you can override this at compilation time to ensure that the loop executes at least once via the command

```
F77/DOTRIP=1 MYPROG
```

Consequently, you may not modify e_2 or e_3 within the range of the DO loop.

NOTE: We sometimes refer to this DO statement as the "iterative DO statement" to distinguish it from the DO WHILE and unlabelled DO statements (described later in this chapter). The iterative DO statement is the traditional —that is, pre-F77 — FORTRAN DO statement.

DO Execution

When it encounters a DO, F77 takes the following steps:

1. It examines the data types of i, e_1, e_2, and e_3 (if present), evaluates expressions in any of these, then converts the data types of all es to the data type of i (if needed).
2. It assigns initial parameter e_1 to the DO-variable, i.
3. It effectively establishes the *trip count* (number of passes through the range of the DO). The trip count is the value of the expression:

   MAX(INT((e_2 - e_1 + e_3) / e_3), 0)

   The trip count is 0 whenever:

   $e_1 > e_2$ and $e_3 > 0$, or
   $e_1 < e_2$ and $e_3 < 0$.
4. If the trip count is not 0, F77 executes the range of the DO. Unless a statement transfers control out of the loop, F77 executes the terminal statement labelled s and increments the DO-variable by the value of the incrementation parameter e_3.
5. F77 then tests the DO-variable against the terminal parameter e_2. If the DO-variable, in positive or negative steps according to the increment, has not passed the terminal parameter, F77 repeats the loop. If the DO-variable has passed the terminal parameter, F77 exits from the loop and passes control to the first executable statement *after* the terminal statement.

The DO-variable i and e parameters can be any integer or real data type: real, real*8, or double precision. They can have any valid value except 0 for e_3. It is good practice to make all these arguments one data type even though F77 will convert them to match the type of i. Note that the trip count may not be precise for real arguments because the representation of fractions is often only a close approximation. Thus, generally, you will want to use real or double-precision parameters only when the precise trip count is unimportant.

After a DO loop terminates, control goes to the first executable statement after the terminating statement s. The e arguments retain the values they had when the loop began. If you try to change the i value, F77 will report an error. You *can* change the value of e parameters but this will not affect the trip count; it may have other effects.

Nesting DO Loops

You can nest DO loops to any depth. The range of an inner DO loop cannot extend beyond the range of the outer DO loop, but they may share the same terminal statement. The following example is incorrect:

```
      DO 10 J = 1,9          ! Outer loop.
      ...
            DO 20 K=1,6      ! Inner loop.
            ...
10    CONTINUE               ! Error: outer
                             !   terminator
                             !   precedes the
                             !   inner one.
20          CONTINUE         ! Inner terminator.
      ...
```

One way you can correctly nest these statements is

```
      DO 10 J = 1,9          ! Outer loop.
      ...
            DO 20 K = 1,6    ! Inner loop.
            ...
20          CONTINUE         ! Correct: inner
                             !   terminator
                             !   precedes the
                             !   outer one.
10    CONTINUE               ! Outer terminator.
      ...
```

As with block IFs, indentation can help clarify nesting structures.

Two loops may use the same terminal statement if they execute in normal sequence. If, however, a statement in the outer loop transfers control to the terminal statement, the results will be undefined.

In a nested DO, the DO-variable of the innermost loop varies most rapidly and the DO-variable of the outermost loop varies least rapidly. In an array, the first subscript bound varies most rapidly, and the last subscript bound varies least rapidly. You can take advantage of this when using nested DOs with multidimensional arrays. By matching the innermost DO-variable to the first subscript bound of the array, the next DO-variable to the second subscript bound, etc., you specify sequential access to the array elements. This can reduce execution time significantly. For example, assume that you want to assign values to the elements of array A(10,10,10). You would probably want to choose the more efficient way, as follows:

More Efficient

```
DO 5 K = 1,10
DO 5 J = 1,10
DO 5 I = 1,10
A(I,J,K)=A(I,J,K)*I/J
5 CONTINUE
```

Less Efficient

```
DO 5 I = 1,10
DO 5 J = 1,10
DO 5 K = 1,10
A(I,J,K)=A(I,J,K)*I/J
5 CONTINUE
```

DO Loop Restrictions

- You may not transfer control into the range of a DO from elsewhere in the program unit. You may transfer control out of the range of a DO.
- You may not terminate the range of a DO with another DO or any statement described in the DO form description. CONTINUE is a traditionally used terminating statement.
- If you use a real number for the increment i, the trip count may not be precisely what you expect because the internal representation of real numbers is not exact.
- You may not redefine the increment i within the range of a DO loop. You can redefine the other parameters but this will not affect the trip count.
- If you place a DO loop in a block IF, you must place the entire range of the loop within the IF block, ELSE IF block, or ELSE block.
- If you place a block IF in a DO loop, you must close the block IF (with END IF) within the loop.

DO Loop Examples

```
      JMIN = 100
      JMAX = 1000
      DO 23 I = JMIN, JMAX
      ...
      ...
23    CONTINUE

      LMAX = 100
      LMIN = -1000
      DO 26 I = LMAX, LMIN, -1
      ...
      ...
26    CONTINUE

      ISPY = 007
      DO 31 INDEX = 1, ISPY, 2
      ...
      ...
31    CONTINUE
```

A practical example of a DO loop is

```
C     Compute the sum of all elements in an
C        array.  Assume array ARY(24) exists
C        and has real values.

      ...
      TOTAL = 0.0
      DO 37 I = 1, 24
            TOTAL = TOTAL + ARY (I)
 37   CONTINUE
      ...
```

Here, the array *values* remain real although all control parameters in the DO statement are integer.

The following example shows that you can transfer out of a DO loop.

```
C     Find the location of the first occurrence
C        of 16 in array IVAR.
      DIMENSION IVAR(100)
      ...
      DO 10 I = 1, 100
            IF ( IVAR(I) .EQ. 16 ) THEN
C                 I've found 16.
                  GO TO 20  ! Transfer out
                            !   of DO loop.
            END IF
 10   CONTINUE

 20   IF ( I .EQ. 101 ) THEN
C           The DO loop didn't find 16.
            PRINT *, '16 isn't in array IVAR'
      ELSE
            PRINT *, '16 is in array IVAR ',
     +              'at location ', I
      END IF
      ...
```

Observe that statement 20 illustrates the use of a DO loop's terminal value.

Using Unlabelled DO Loops

The unlabelled DO statement is an alternate form of the iterative DO loop. Its format is

DO i = e_1, e_2 *[, e_3]*

where

i — is any noncomplex arithmetic variable, known as the *DO-variable*.

e_1, e_2, e_3 — are each expressions of any noncomplex arithmetic data types.

The terminating statement must be an END DO statement.

The rules about iterative DO statements apply to the unlabelled DO statements.

An example using an unlabelled DO statement follows:

```
INTEGER ODD(11)
DO I = 1, 11, 2
      ODD(I) = 3*I
      ! ODD() CONTAINS 3, ?, 9, ?, ..., 33
END DO
```

Using DO WHILE Loops

The form of a DO WHILE loop is

DO *[s [,]]* WHILE (log_expr)

where:

s	is the statement label of an END DO statement. This END DO statement appears within the current program unit, after the DO WHILE statement.
log_expr	is a logical expression.

The DO WHILE statement functions similarly to the plain iterative DO statement. However, a logical expression, not a numeric variable, controls the loop.

The terminating statement of a DO WHILE loop must be an END DO statement. If *s* appears in the DO WHILE statement, then *s* must be the label of the corresponding END DO statement. Like iterative DO loops, DO WHILE loops may nest within each other. Similarly, you can't transfer into the range of a DO WHILE loop from outside the loop.

When the DO WHILE statement executes, it evaluates log_expr. If log_expr is true, execution continues with the first statement in the range of the DO loop. When this execution arrives at the corresponding END DO statement, control returns to the DO WHILE statement, which retests the logical expression. On the other hand, if the value of log_expr is false, the DO loop becomes inactive. Then execution continues with the next statement after the corresponding END DO statement.

DO WHILE Loop Restrictions

- You may not transfer control into the range of a DO WHILE from elsewhere in the program unit. You may transfer control out of the range of a DO WHILE.
- You may terminate the range of a DO WHILE statement only with an END DO statement, not with CONTINUE.
- If you place a DO WHILE loop in a block IF, you must place the entire range of the loop within the IF block, ELSE IF block, or ELSE block.
- If you place a block IF in a DO WHILE loop, you must close the block IF (with END IF) within the loop.

DO WHILE Loop Examples

```
      INTEGER ARRAY(5) ! TO CONTAIN 99, 98, 97,
                       !    96, AND 95.
      I = 1
      DO WHILE (I .LE. 5)
             ARRAY(I) = 100 - I
             I = I + 1
      END DO
      STOP
      END
```

```
      INTEGER ARRAY(5) ! TO CONTAIN 99, 98, 97,
                       !    96, AND 95.
      I = 1
      DO 10 WHILE (I .LE. 5)
             ARRAY(I) = 100 - I
             I = I + 1
10    END DO
      STOP
      END
```

```
      CHARACTER*80 LINE
      OPEN (2, FILE="MY_DATA_LINES")
      IER = 0
      DO WHILE (IER .EQ. 0)
             READ (2, 10, IOSTAT=IER) LINE
10           FORMAT (A)
C            Process LINE.
      END DO
      ...
```

Using CONTINUE

CONTINUE is an executable statement but has no effect. It is designed for, and generally used as, the terminating statement of an iterative DO loop. It is a place holder that makes the program clearer and easier to understand.

You must label a CONTINUE statement when using it as a DO terminator. Otherwise, you need not label CONTINUE, and it can appear anywhere in your program. Its form is

CONTINUE

You have already seen examples using CONTINUE in programs with iterative DO loops. In the next example of CONTINUE, I_SWITCH_A is a "switch" variable whose value is either 0 or 1. Note how control always arrives at statement number 40, regardless of I_SWITCH_A's value.

```
      ...
      IF ( LSWITCHLA .EQ. 1 ) GO TO 30
C     LSWITCHLA IS 0.  PROCEED WITH THE
C       NECESSARY CALCULATIONS.
      ...
      GO TO 40

   30 CONTINUE
C     LSWITCHLA IS 1.  PROCEED WITH THE
C       NECESSARY CALCULATIONS.
      ...

   40 CONTINUE   ! THE PROGRAM'S WORK.
      ...
```

Thus, CONTINUE can serve as a convenient label; it effectively is a "no-operation" instruction because it only passes control to another statement.

Pausing, Stopping, and Ending

Occasionally you will want your program to wait for operator intervention; for example, to mount a tape or put special forms in the line printer. Often, at several points, you may want it to stop, and always you must mark the end of each program unit.

This section describes the statements that do these things: PAUSE, STOP, and END.

Pausing (PAUSE)

PAUSE temporarily suspends program execution and waits for user or operator intervention. Its form is

PAUSE *[n]*

where:

n is a string of digits or a character constant.

F77 displays PAUSE and the text message *n* (if any) on a terminal; it may display other messages as described next.

There are two runtime reactions to the PAUSE statement, depending on whether the runtime operating system is either AOS/VS or AOS, or else MP/AOS or MP/AOS-SU.

AOS/VS and AOS Runtime PAUSE Result

If the program was originally executed in interactive mode (from a terminal via XEQ or PROCESS commands), the F77 processor suspends the program and attempts to send the PAUSE message to the terminal from which the program was executed. Its format is

**** PAUSE *** Message is:*
<string of digits or characters>
Type NEW-LINE to continue.

At this point, you or anyone at the terminal can

- continue the program by pressing any data-sensitive delimiter, like NEW LINE (⤶); or
- terminate the program by typing CTRL-C, CTRL-B.

The program remains suspended until it receives one of these sequences.

If the F77 processor cannot send the PAUSE message to the terminal that received the XEQ program-name command, it attempts to walk up the system process tree from son to father process. (This would happen if the program had been executed in batch mode, and thus *didn't have* a terminal.)

If a father process has a terminal connected, F77 sends the message *FORTRAN PAUSE,* then the message *n* if any, then instructions. The person at this terminal can either UNBLOCK or TERMINATE the program per instructions; the program will wait indefinitely. If a father process has a terminal connected, but messages to this terminal are disabled, F77 issues a nonfatal error message to the process's @OUTPUT file and continues up the process tree. If a father process has *no* terminal connected, F77 simply continues up the process tree.

If F77 reaches the operator process (OP, PID 2), it checks if the system operator is off duty. If the system operator is off duty, F77 displays *FORTRAN PAUSE* and message *n* (if any); then it displays *program continuing* and continues program execution. If the system operator is on duty, or if F77 can't tell whether an operator is on duty, it displays *FORTRAN PAUSE*, message *n* (if any), and instructions. It will wait indefinitely to be unblocked or terminated if the operator is not off duty.

MP/AOS and MP/AOS-SU Runtime PAUSE Result

Here, PAUSE functions differently. PAUSE attempts to open the files connected to channels ?INCH (process input) and ?OUCH (process output) when the program began to execute. If both these files can be opened and are terminal files (@TTIx, @TTOy), then the PAUSE message goes to ?OUCH and the program waits for a response from ?INCH before continuing. One of the following messages goes to ?OUCH:

**** Fortran Pause ****
Type NEW-LINE to continue.

**** Fortran Pause *** Message is:*
<string of digits or character constant in PAUSE n statement>
Type NEW-LINE to continue.

If F77 cannot open both channels or if both files are not terminal files, then an appropriate error message is sent to all ERRORLOG files and execution continues. The error message includes any string of digits or character constant in the PAUSE statement.

PAUSE Examples

Assume that an AOS/VS or AOS program named MYPROG has the statement

```
PAUSE 'Need tape 1'
```

The first of the following examples shows the dialog that would occur if MYPROG had been executed in interactive mode; the second shows dialog that might occur if MYPROG had been executed in batch mode. In both examples, the person involved wants to continue the program.

```
) XEQ MYPROG ↵
```

****PAUSE*** Message is:*
Need tape 1
Type NEW-LINE to continue.
(User mounts the tape.)
↵
(Program continues.)

The AOS/VS or AOS batch example is

```
) QBATCH XEQ MYPROG ↵     (on user terminal)
```

(On OP terminal)

*From Pid 39: *** FORTRAN PAUSE *** Message is:*
From Pid 39: Need tape 1
From Pid 39: Use UNBLOCK 039 to continue or
TERMINATE 039 to stop.

(Operator mounts the tape.)
) UNBLOCK 039 ↵
(Program continues.)

Stopping (STOP)

If your program does not branch, you don't absolutely need the STOP statement; but if it does branch, there may be more than one place where you want it to terminate. In the second case, use STOP. Used with informative messages, STOP can also help you debug complex programs. When STOP executes, it halts the entire process — not just the current program unit.

If the program was executed in interactive mode (from a terminal), F77 displays the STOP message on the terminal. If you executed the program with the CLI /S switch (XEQ/S MYPROG ↵), the message is returned to the CLI variable STRING. If the program was executed in batch mode (QBATCH command), F77 writes the message into the batch output listing.

The form of STOP is

STOP *[n]*

where:

n is a string of digits or a character constant.

F77 writes STOP, a blank, and the text message *n* (if any) on the appropriate terminal or file.

STOP Examples

```
STOP

STOP 'LABEL 70'

STOP "Error on write to output file"
```

Ending (END)

The END statement marks the end of the program unit; it must end every program unit. Its form is

END

The letters E, N, and D must be on the same line, in that order, and must start on or after character position 7.

In the main program, END returns to the initial program environment, which stops the program *without* a termination message (as opposed to STOP).

In a function or subroutine subprogram, END returns control to the calling program. RETURN in a subprogram has the same effect.

END Examples

```
C       Main program.
        PROGRAM MONITOR
        ...
        A_RESULT = ALPHA(A1, A2)
        ...
        END


C       Function subprogram.
        CHARACTER FUNCTION ALPHA (D1, D2)
        ...
        ...
        END   ! Return to program MONITOR
```

End of Chapter

Chapter 5
Input and Output

Input and output (I/O) statements allow F77 programs to transfer data between internal storage and files.

The statements that actually transfer the data are READ, WRITE, and PRINT. Statements that prepare for and relate to transfer are the *auxiliary I/O* statements OPEN, INQUIRE, and CLOSE. Statements that relate to file positioning are BACKSPACE, REWIND, and ENDFILE.

Before describing these statements, this chapter provides background material on files and units, records, formats, and lists that relate to statement execution. The topics within the chapter proceed:

- I/O Overview
- Files and Unit Numbers
- Records
- Format Specifier for Data Transfer
- Control Information List (cilist) for Data Transfer
- Input/Output List (iolist) for Data Transfer
- Using READ
- Using WRITE
- Using PRINT
- Auxiliary I/O
- Using OPEN
- Using INQUIRE
- Using CLOSE
- File Positioning with BACKSPACE, ENDFILE, REWIND
- Terminal I/O
- Summary of Rules for I/O

The file, record, and list sections provide a basis for statement use, so you may want to read them first. Or, you might choose only the "Using READ", "Using WRITE" and "Using OPEN" sections, which include all essential material with examples.

For details on record and record formatting — sketched in this chapter — you also need to examine Chapter 6.

I/O Overview

Chronologically, you generally follow this sequence for I/O with files like disk files:

- Open the file(s) with the properties you want. If a disk file doesn't exist, OPEN can create it.
- Read, write, and/or print records in the file(s), using other FORTRAN statements to process the data in the records. You can use INQUIRE to check file properties at will.
- Position the file(s) as needed with BACKSPACE and REWIND to reread or rewrite records; establish new ends of files as desired with ENDFILE.
- Close the file(s). This is not absolutely required because program termination causes closing of all files.

For other files, such as F77 preconnected files (such as @INPUT) or character entities (called internal files), you simply read, write, and/or print records in the file(s). Other FORTRAN statements process the data in the records.

Because you need not use the auxiliary or positioning statements (OPEN, INQUIRE, BACKSPACE, etc.) for many applications, these statements are described *after* the data transfer READ and WRITE statements — which actually do the I/O.

Files and Units

A *file* is either a collection of data or a device that reads or writes the data. A file on a hardware device — disk with a disk filename, magnetic tape, line printer, or the terminal — is an *external file*. A file that is a FORTRAN entity, like an array or variable, is an *internal file*.

To gain access to an external file (disk, tape, printer), you connect a valid unit number to it, then use the unit number to refer to the file. A unit is either connected or not connected; when connected, it refers to the file. The statement that makes the unit/file connection is OPEN.

Preconnections allow you to use files such as @CONSOLE without explicitly opening them. F77 has the following preconnections:

Unit	Filename and Description
5	@INPUT. For interactive use, this is the terminal keyboard. On READ statements, you can use either the unit number 5 or an asterisk (*) to specify this file.
6	@OUTPUT. For interactive use, this is the terminal screen or terminal printer. On WRITE statements, you can use either the unit number 6 or an asterisk (*) to specify this file.
9	@DATA. This is a file whose name you can set with the system CLI DATAFILE command (except under MP/AOS and MP/AOS-SU).
10	@OUTPUT. It is the same file as 6, but the preconnection has different carriage control properties (for placement of output) and you cannot use an asterisk to specify it.
11	@INPUT. It is the same file as 5, but you cannot use an asterisk to specify it.
12	@LIST. This is a file whose name you can set with the system LISTFILE command (except under MP/AOS and MP/AOS-SU). If you give the commands LISTFILE @LPT XEQ MYPROG LISTFILE @LISTFILE and MYPROG.F77 contains statements such as WRITE(12,...), then the line printer is an output file.

MP/AOS and MP/AOS-SU F77 do not have preconnected units 9 and 12. By default, their preconnected units 5 and 11 use channel ?INCH (process input file), while their preconnected units 6 and 10 use channel ?OUCH (process output file).

Programs that use these preconnections are not directly transferrable to (non-DG) processors that have different preconnections.

Having opened a file, you can open it again on the same unit number to change some of the OPEN specifications. This is called a reopen. A file can be open on more than one unit at a time.

If you open a second file on a unit number, F77 closes the original file and opens the second file with this unit number.

For example,

Standard Conforming

```
OPEN (15, FILE='MYFILE')
READ (15, ...) VAR1
OPEN (15, FILE='MYFILE')
WRITE (15, ...) VAR1

OPEN (18, FILE='PAPER_1')
OPEN (18, FILE='PAPER_2')
!   PAPER_1 is closed,
!     PAPER_2 is open.
```

Nonstandard Conforming (But Permitted)

```
OPEN (19, FILE='MYFILE')
...
OPEN (20, FILE='MYFILE')
```

A CLOSE statement severs a unit/file connection — but is not strictly necessary because F77 closes all open files when the program terminates.

You can specify a unit number as an integer expression. This expression evaluates to an integer from 0 through 255 for AOS/VS or from 0 through 63 for AOS, F7716, MP/AOS, and MP/AOS-SU. The following example shows two ways to represent a unit number:

```
OPEN (13, FILE='MYFILE')
READ (13, ..., ) X, Y, Z
CLOSE (13)

J = 13
OPEN (J, FILE='MYFILE')
READ (J, ..., ) X, Y, Z
CLOSE (J)
```

External Files

An external file can be any named file in the system: a disk file (pathname :UDD:JOE:FOO, filename FOO) or a peripheral device (@MTA0:1). You must always use a unit number (or, for units 5 or 6, an asterisk) to refer to an external file.

Internal Files

An internal file is simply a character variable, array, or substring into which you read or from which you write records. Internal file unit identifiers are symbolic names and — like any F77 array or variable — you don't open or close them. The following example reads a character variable from a file, extracts seven numbers from the string of characters, and then places the extracted (and rearranged) numbers in another file.

```
C   Convert characters to integers and reals.

    CHARACTER*80 REC       ! REC is the
                           !    internal file.
    DIMENSION NUMBERS(4)   ! Array to
                           !    receive integers.
    DIMENSION REALS(3)     ! Array to
                           !    receive reals.

    OPEN (2, FILE='SOURCE' ,...)     ! Input file.
    OPEN (49, FILE='NEWSOURCE',...) ! Output file.

4   READ (2, 90, ...) REC ! is a record of characters
90  FORMAT (A80)

C   Extract the numbers from the characters in
C       internal file REC.
    READ (REC, 91) NUMBERS, REALS
91  FORMAT (4I8, 1X, 3F8.2)
C   Process the numbers in arrays NUMBERS
C      and REALS.
    ...
    WRITE (REC, 92) NUMBERS, REALS  ! Place the
C                                       numbers
C                                       back into
C                                       REC ...
92  FORMAT (4I9, 5X, 3F9.2)
    WRITE (49, 90) REC              ! ... and then
C                                       into the
C                                       output
C                                       file.
    GO TO 4
    ...
```

This example illustrates that READ can function the same as other FORTRANs' DECODE statement, and that WRITE can function the same as their ENCODE statement. Items in an internal file are like any other character entities.

Records

The information transferred during each I/O operation is subdivided into *records*, which match the way the information is stored in the file. The length of records varies, depending on how you write them. There are three kinds of records:

- formatted
- unformatted
- endfile

Records written as *formatted* records contain characters. Records written without a format specification are *unformatted* and contain bytes that represent up to 256 possible binary values. For consistent results, you should not mix record types in a file; the file should have either all formatted or all unformatted records. (F77 cannot enforce this, however, because it maintains no information external to the current unit connection.)

An endfile record is written with an ENDFILE statement. It simply ends the file.

The OPEN statement establishes the connection property — formatted or unformatted — for a unit, and all READs and WRITEs to the unit must conform to this property. Records in an internal file must be formatted.

For each data transfer statement, F77 reads or writes at least one whole record — consisting of characters if formatted or bytes if unformatted. F77 can transfer more than one formatted record in a transfer statement — described in Chapter 6 under "Multiple Record Formatting".

F77 maintains a record position within each connected external file and internal file. The *start position* is just before the first record. The *end position* is just after the last record. The *current position* is just before or in the current record. The record that the current data transfer statement will process or is processing is the *current record*.

Formatted Records

A formatted record is composed of zero or more characters (letters, numbers, and special symbols); you write it with a *format specification*, defined later in this chapter and in the next chapter. F77 uses the editing directives in the format specification when it reads characters from or writes them to the record.

A formatted record is literally a sequence of characters. For example, assume that variables J1, J2, and J3 have the following values:

J1 = 123

J2 = 456

J3 = 789

Also assume that unit 49 is opened and the following I/O statement and format specification:

```
        WRITE (49, 100) J1, J2, J3
100     FORMAT (I4, I4, I4)
```

F77 writes an entire record using the values of J1, J2, and J3. Variables J1, J2, and J3 are *input/output list* entities, detailed later. I4 is a symbol, called an *edit directive* or *format*, that specifies a data item in a record. F77 writes the values of these entities sequentially into the record, using the format I4 for each value. The record will consist of the three formatted values and look like this in the file on unit 49:

□123□456□789

The I4, I4, I4 format specified four character positions for each of three respective values. Since the value in J1, J2, and J3 requires only three character positions, F77 padded each value (to its left) with a blank.

Formatted records are sometimes necessary, but their transfer is often slower than unformatted records. For example, if variable J1 is INTEGER*2, it requires 2 bytes (= 16 bits) of primary storage — but 4 bytes of secondary storage if it is written with format I4. The arithmetic to convert 123 from its internal bit pattern

00000000.01111011

to its external bit pattern

00100000.00110001.00110010.00110011 = "□123"

can require considerable time. In contrast, 123 would exist in both internal storage and in an unformatted record as the bit pattern

00000000.01111011

— thus, no conversion to or from ASCII characters is necessary. (The decimal points in the bit patterns of this example serve only to separate the bits into bytes.)

The sequence of characters is critical and affects all future I/O on the record. If, after repositioning before the record, the same format were used to read the record back, the values read would be the same as those written; for example,

```
      READ (49, 100) K1, K2, K3
100   FORMAT (I4, I4, I4)
```

F77 reads the entire record; then, using the format, it assigns values to the input/output list entities: K1, K2, and K3. K1, K2, and K3 get values that match the values of J1, J2, and J3 at the time the J values were written. On the other hand, suppose a different format — say I3 — were used for the READ statement:

```
      READ (49, 110) K1, K2, K3
110   FORMAT (I3, I3, I3)
```

Recall that the record contains the ASCII characters □123□456□789. These values would be read into K1, K2, and K3 sequentially from the record, using I3 format:

```
K1 = □12      (interpreted as 12)
K2 = 3□4      (interpreted as 34)
K3 = 56□      (interpreted as 56)
```

NOTE: The previous values for K1, K2, and K3 are correct if the BLANK specifier at the time of the file's opening has a value of 'NULL' (default). If this specifier were 'ZERO', the respective values for K1, K2, and K3 would be 12, 304, and 560.

But for FORMAT statement 100, using I4 format, the values were

```
J1 = □123     (interpreted as 123)
J2 = □456     (interpreted as 456)
J3 = □789     (interpreted as 789)
```

As you can see, the FORMAT statement actually controls the values.

The values written or read by a formatted I/O statement can be numeric, logical, or alphanumeric; but they are sequences of characters with *precisely the length given by the format.*

A formatted, data-transfer I/O statement that includes an input/output list always transfers at least one whole record. On input, it assigns values from the record to the list entities, using the format specification. On output, it writes the values of the list entities, using the format specification, to the record.

If you wish, before you write formatted records, you can instruct F77 to use the first character in each record for vertical carriage control. This is described under the OPEN statement later in this chapter. It may relate to any file whose records you plan to print, either on the terminal or line printer.

Unformatted Records

An unformatted record is a sequence of 0 or more bytes, and it is written by an unformatted WRITE statement — or even by a language other than FORTRAN, like PL/I or COBOL.

An unformatted record neither requires nor allows editing. Unformatted records are generally unrecognizable. To change this, you must read them unformatted and write them formatted. Unformatted records are usually more efficient for storage and always faster for data transfer.

The length of an unformatted record is the number of bytes written; it is constant for fixed-length records. If a record that is not fixed length was written by a FORTRAN WRITE statement, its length depends on the number and type of values in the WRITE's input/output list. For example,

```
B1 = 12345.67
B2 = 3.55
WRITE (18)   B1, B2    ! Unformatted.
```

Assuming standard (4-byte) real numbers, the values in the record on unit 18 would look like this in octal:

```
| 104140346541 101161463151 |
```

4 bytes for B1 — 4 bytes for B2

B1 and B2 could be read back via any two standard real entities; for example,

```
READ (18)   C, C1    ! Unformatted.
```

In an unformatted read or write, F77 reads or writes only one record.

Finally, unformatted records contain data with exactly the same bit patterns as the data in primary storage. For example, you have just seen that the REAL*4 number 3.55 exists in both an unformatted record and in primary storage as the 32 bits

01.000.001.001.110.001.100.110.011.001.101

The time required to convert these 4 bytes to 4 bytes of a formatted record that contain

00110011.00101110.00110101.00110101 = "3.55"

is considerable. (The decimal points in this paragraph exist only to separate the bits into convenient groups.)

Endfile Records

An endfile record — written with an ENDFILE statement — truncates and physically ends the file. If you omit ENDFILE, the operating system physically ends the file after the last record written. Endfile records apply only to files opened for sequential (as opposed to direct) access.

An endfile condition exists when a READ statement encounters an endfile record. Then, the file position is *after* the endfile record. An endfile condition also exists when a READ statement attempts to read a record beyond the end of an internal file.

Once a file position is after the endfile record, you may not issue a data transfer I/O statement. Doing so results in a runtime error. However, you may issue a BACKSPACE or REWIND statement to change the file position.

More specifically:

- After an error-free read of a nonendfile record, specifier IOSTAT (explained as part of the OPEN, READ, and WRITE statements) contains 0.
- After a read *of* an endfile record, specifier IOSTAT contains a negative number and F77 branches according to the END= directive.
- After a read *after* an endfile record, specifier IOSTAT contains the value of F77ERIOUEN. F77ERIOUEN is an error parameter in file ERR.F77.IN.

Access to Records

You can open a unit for either sequential or direct access to its records. Sequential or direct access is allowed with both formatted and unformatted records.

In a file connected for sequential access, F77 reads or writes records sequentially; F77 cannot process the records randomly. You cannot read past the end of the file (or endfile record), but you can write over the endfile record to append to the file. Sequential access to formatted records is the most common I/O combination, and it is mandatory for internal files.

For direct access, you must open the unit for direct access, and its records must have all same length — which you specify in the OPEN statement. Each direct-access record is given a record number. The first record is number 1. You specify the number when you create the record; later, you use the number to read or rewrite the record contents. You can never change the record numbers, but you *can* write over records or send the records in different orders to another file. A record number is not part of the record; it simply identifies the record. There is no end of file or endfile record in a direct-access file. If, on a read, the specified record has not been written, all items in the input list become undefined.

Operating System File Organizations

The operating system, under F77 control, can create a file with any of four record formats, called "organizations" here to avoid ambiguity with the word "formats". F77 allows you to select an organization when an OPEN statement creates a file (but in many cases you will want to accept the default organization). The organizations are data sensitive, variable length, fixed length, and dynamic. The maximum lengths of each are described under OPEN.

Strictly speaking, it's somewhat misleading to talk about "FORTRAN data files" and "COBOL data files." All DG languages actually use the operating system as they create and use files in one of these four organizations. That is, at runtime a language I/O command (such as an F77 READ statement) results in the *operating system's* transfer of data to and from primary storage. Thus, for example, you can create a COBOL program to read a file that a FORTRAN 77 program has created.

The default record organization for most files, if F77 creates the file, is data sensitive. A data-sensitive record contains a series of characters terminated by a data-sensitive delimiter. The standard delimiter characters are NEW LINE (<NL>=<012>), null (<NUL>=<000>), carriage return (<CR>=<013>), and form feed (<FF>=<014>). You can insert NEW LINE directly but need not do so because each WRITE statement to such a file automatically inserts a NEW LINE (<NL>) at the end of the record (with CARRIAGECONTROL='LIST'). Data-sensitive files can be printed directly on the terminal or line printer without postprocessing. Other types may require postprocessing with the F77 postprocessor program, F77PRINT, before they will print perfectly.

In a variable-record file, each record has a header that specifies the number of bytes in it. This 4-byte header gives the byte count of the entire record. For example, 'GEOFFREY' as a variable record contains the 12 bytes 0012GEOFFREY (=<060><060><061><062><107>...<131>).

If the records in the file will have a fixed length (implied if you open a new file for direct access), F77 creates a file with fixed-length record organization. F77 expects all files connected for direct access to have fixed-length records and fixed-length record organization. The length of each record is the number of bytes you specify when you open the file.

A dynamic record has no inherent size definition. To get dynamic organization for a new file, you must specify 'DYNAMIC' as the value of the RECFM OPEN specifier. To gain access to dynamic records, you must open the unit with the OPEN specifiers FORM='UNFORMATTED' and ACCESS='SEQUENTIAL'.

Format Specifier/Identifier for Data Transfer

For formatted records — the most common — you must *include* a format specifier in each data transfer statement. (For unformatted records, *omit* the format specifier.) The format specifier is either a character expression within the transfer statement *or* a separate FORMAT statement. The format *identifier* — either character expression or *label* of a FORMAT statement — is called f. You can place the f identifier within a READ, WRITE, and PRINT statement as follows:

READ (unit, *[FMT=]* f *[...other-arguments]*)
WRITE (unit, *[FMT=]* f *[...other-arguments]*)

or

READ f, *[...other-arguments]*
PRINT f, *[...other-arguments]*

READ has either a general form, shown in the first set of statements, or a short form, shown in the second set of statements. The FMT= is optional with WRITE and the general form of READ; it is not allowed with the short form of READ or with PRINT.

The format identifier f is *one* of the following:

1. f can be a character expression that contains the format specification, placing the whole specification within the I/O statement. The expression can be (or include) a character constant or the name of a character variable, array element, or substring that contains the format specification. Examples of character format identifiers are

```
C       Character constant in statement.
        WRITE (6, FMT='(1X, I8)') J
        PRINT '(1X, I8)', J ! Same as WRITE above.

C       Character variable and character array
C       element name.

        CHARACTER*10 FT / '(1X, I8)' /
        WRITE (6, FMT=FT) J
        PRINT FT, J ! Same as WRITE above.

        CHARACTER*10 CY(3)
        CY (1) = '(1X, F9.2)'
        CY (2) = '(1X, I8)'
        CY (3) = '(1X, E10.4)'
        WRITE (6, FMT=CY(2) ) J
        PRINT CY(2), J ! Same as WRITE above.
```

   Alternatively, f can be the name of an array whose elements, read sequentially, provide a format. The elements can contain either character or Hollerith values. Examples of both are

```
C       Chararacter array.
        CHARACTER*6 CARY(2)
        ...
        CARY(1) = '(1X,'
        CARY(2) = 'I4,I3)'
        WRITE (*, CARY) J, K

C       Array of Hollerith data.
        INTEGER*4 IFMT(3) / 4H(1X, , 4HI4,I , 2H3) /
        ...
        WRITE (*, IFMT) J, K
```

The character expression used as a format specification is new with FORTRAN 77; the array of Hollerith constants is traditional. (The edit descriptors used, such as 1X and I8, are detailed in Chapter 6.)

2. Secondly and traditionally, f can be the label of a FORMAT statement that has the format specification. This FORMAT statement must be in the current program unit. For example,

```
        WRITE (6, FMT=100) J   ! f is label 100.
        PRINT 100, J           ! f is label 100.
100     FORMAT (1X, I8)        ! FORMAT here.
```

   As with any executable statement label, the label of the FORMAT statement can be assigned to an integer variable. For example,

```
        ASSIGN 100 TO K
        WRITE (6, FMT=K) J     ! f is K.
        PRINT K, J             ! f is K.
100     FORMAT (1X, I8)        ! FORMAT here.
```

3. Instead of a character expression or label of a FORMAT statement, f can be an asterisk to specify *list-directed formatting*. List-directed formatting is new with FORTRAN 77. List-directed formatting bases the format on the data type of the input/output entity and is well suited for I/O with the terminal. For example,

```
READ (5, FMT=*) J      ! f is list directed.
READ *, J              ! f is list directed.
WRITE (6, FMT=*) L     ! f is list directed.
PRINT *, L             ! f is list directed.
```

   The asterisk specifies list-directed formatting.

The first two f specifications — character expression and FORMAT statement — give an explicit format via edit-directed formatting. The records processed with edit-directed formats always have precisely the format given. Edit-directed formatting is the most common; it is needed for the precise appearance of output, and often when output will be input to an F77 program. Edit-directed formatting is detailed in Chapter 6.

The third f — an * for list-directed formatting — uses only the input/output list data types to format values. List-directed formatting will suffice for almost all terminal I/O. It is also useful for units other than the terminal and is detailed toward the end of Chapter 6. If you are familiar with DG's FORTRAN 5, list-directed formatting resembles FORTRAN 5's READ FREE and WRITE FREE statements.

Within the WRITE examples above, unit 6 could be replaced by an asterisk; within the READ examples, unit 5 could be replaced by an asterisk. (The short forms of PRINT and READ assume units 6 and 5 respectively and don't accept a unit specifier.) We have not used an asterisk as a unit specifier above because an asterisk also specifies list-directed formatting and would confuse the issue. For example, in the statements

```
WRITE (*, *)  J
READ (*, *)   J
```

the first asterisk specifies unit 6/5 and the second asterisk specifies list-directed formatting.

Control Information List (cilist) for Data Transfer

Each WRITE statement must include a *control information list* in parentheses, or *cilist* that describes the unit number and other control items. Each READ statement for a unit other than 5 must also have a cilist. A PRINT statement, which implies unit 6, doesn't accept a cilist.

The forms of READ, WRITE, and PRINT are

Statement	Description
READ (cilist) iolist	Read from the unit given in the cilist, using (for formatted records) the format identified in the cilist. For terminal I/O, the unit specifier can be * to specify unit 5 and the format specifier can be * for list-directed formatting; e.g., READ (*,*) iolist. The iolist is an input/output list, described in the next section.
READ f, iolist	READ without a cilist implies unit 5, preconnected to file @INPUT. The format identifier f can be *, but the comma between * and the iolist is required.

WRITE (cilist) iolist	Write to the unit given in the cilist using (for formatted records) the format identified in the cilist. As for READ, the unit and format specifiers can each be * to select unit 6 and list-directed formatting; e.g., WRITE(*,*) iolist.
PRINT f, iolist	Write to unit 6, preconnected as @OUTPUT, using the given format f. The f can be *, but the comma between * and the iolist is required.

A cilist must contain at least a unit specifier, and, for formatted transfer statements, a format identifier/specifier. It can contain other specifiers. The forms of cilist specifiers are

Specifier	What It Specifies
[UNIT=] iu	Unit. You can omit UNIT= if this is the first argument in the cilist.
[FMT=] f	Format. You can omit FMT= if you omitted UNIT= from the unit specifier and if this is the second argument in the cilist.
IOSTAT= ios	Integer variable to return status.
ERR= s	Statement to execute on error.
END= s	Statement to execute on end of file.
REC= irn	Integer record number for direct access.
RETURNRECL=iret	Integer variable to return record length.

Details on these specifiers appear in the sections on the READ, WRITE, PRINT, and file-positioning statements.

Input/Output List (iolist) for Data Transfer

For each I/O statement, F77 reads at least one entire record from the unit or writes at least one entire record to the unit. The *input/output list* or *iolist* gives the entities into which F77 will read values from the record or from which it will write values to the record.

An iolist entity can be

- a variable name, like J;
- an array element name, like B(I);
- an array name;
- a character substring name, like CHAR(1:4);
- an implied DO list, described in the next section; or
- for *output only,* an entity that includes an expression, e.g., I+J+2 or 'ABC'.

If you use an array name without a subscript in an iolist, F77 processes all its elements in the traditional column/row order, described in Chapter 2.

In the previous examples, for format identifiers/specifiers and control lists, variables J, K, and L are iolist entities. Here are some other iolist examples:

```
      DIMENSION B(100)
      ...
      WRITE (*, FMT='(1X, I8)') J  ! J is iolist.
      WRITE (*, *)   J, K          ! J, K are iolist.
      WRITE (10, 90) X, Y, Z    ! X, Y, Z are iolist.
90    FORMAT (3F10.2)
      READ (3, '(3E10.4)') Q,R,S  ! Q,R,S are iolist.
      WRITE (4, '(1X, 2F9.2)') B(6) ! B(6) is iolist.
      WRITE (4, '(1X, 2F9.2)') B ! B array is iolist.
      WRITE (4, '(1X, 2F9.2)') B(J+K)   ! B(J+K) is
                                        !   iolist.
```

For formatted I/O, the data type of the iolist entity must match the format used for data from or to the record. If not, F77 signals a runtime error. For example, you cannot use an integer (I) format to process an iolist entity that contains character data. This is detailed under the edit descriptors in Chapter 6.

In a subprogram, you cannot use an assumed-size array (dimensioned with an asterisk as its last dimension) in an iolist.

Implied DO List

An implied DO list — including one or more iolist entities — is simply a shorthand DO loop. The form of an implied DO list is

(dlist, i = $expr_1$, $expr_2$ *[, $expr_3$]*)

where:

dlist	is an iolist.
i	is an integer or real variable name, called the DO-variable.
$expr_1$	is the initial parameter for the loop.
$expr_2$	is the terminating parameter for the loop.
$expr_3$	is the incrementation parameter for the loop; if you omit this, the value is 1.

As with the standard DO (Chapter 4), F77 starts the loop with the value of $expr_1$ and increments it by the

expr$_3$ value until it exceeds expr$_2$; then F77 exits the loop. Implied DOs, like standard DOs, can be executed zero or more times.

An implied DO list example is

```
DIMENSION RARY(10)
READ (*, *)  (RARY(J), J = 10, 1, -1)
WRITE (14, '(1X, F9.1)')  (RARY(K), K = 1, 10)
```

This gets 10 values from the terminal and assigns them to RARY elements in descending order: RARY(10), RARY(9), ..., RARY(1). Then it writes each value as a record to unit 14. To assign values in *ascending* order, no implied DO would be needed; you'd simply use the array name as the iolist entity and F77 would access its elements in order: READ(*,*) RARY.

You can nest implied DO lists, just like standard DO loops; for example,

```
DIMENSION IRY(10,9)
...
! Program assigns values to IRY.
...
WRITE (*,'(1X, I8)')  ((IRY(L,K), K=1,9), L=1,10)
```

This prints the values of IRY(1,1), IRY(1,2), IRY(1,3), ..., IRY(1,9), IRY(2,1), ..., IRY(10,9) on the terminal. The conventional order, if IRY had been the iolist, would have been IRY(1,1), IRY(2,1), IRY(3,1), ..., IRY(10,1), IRY(1,2), ..., IRY(10,9).

For another example of implied DO loops, when a program containing the statements

```
   INTEGER A_ARRAY(2,3), B_ARRAY(3,4)

   PRINT *, 'GIVE ME A_ARRAY (2,3), ',
+          'AND THEN B_ARRAY (3,4):<NL>'
   READ (*,*) ((A_ARRAY(I,J), J=1,3), I=1,2),
+             ((B_ARRAY(K,L), K=1,3), L=1,4)

   PRINT *, 'A_ARRAY (2,3) FOLLOWS:'
   PRINT *, A_ARRAY
   PRINT *
   PRINT *, 'B_ARRAY (3,4) FOLLOWS:'
   PRINT *, B_ARRAY
```

executes, the terminal dialog could be

GIVE ME A_ARRAY (2,3), AND THEN B_ARRAY (3,4):

```
10,11,12, 20,21,22 ↓
30,31,32, 40,41,42, 50,51,52, 60,61,62 ↓
```

A_ARRAY (2,3) FOLLOWS:
10 20 11 21 12 22

B_ARRAY (3,4) FOLLOWS:
30 31 32 40 41 42 50 51 52 60 61 62

The elements of A_ARRAY contain A_ARRAY(1,1)=10, A_ARRAY(1,2)=11, A_ARRAY(1,3)=12, A_ARRAY(2,1)=20, A_ARRAY(2,2)=21, and A_ARRAY(2,3)=22. The effect of the statement

```
PRINT *, A_ARRAY
```

is to print the elements in *column* order: 10, 20, 11, 21, 12, 22.

The effect of the statement

```
PRINT *, ((A_ARRAY(I,J), J=1,3), I=1,2)
```

would be to print the elements in *row* order: 10, 11, 12, 20, 21, 22.

Using READ

READ is *the* F77 input statement. READ always reads at least one record and assigns values from it to the iolist variables. If you include a format specification, F77 uses the format to read characters from the record into the iolist variables. The forms of READ are

READ f *[,iolist]*

READ (cilist) *[iolist]*

where:

f — in the first form, with no cilist, is a format identifier: label of a FORMAT statement, an integer variable assigned the label of a FORMAT statement, a character array name, a character expression (entity name or constant like '(1X,I8)'), or an asterisk to specify list-directed formatting. Because this form of READ implies unit 5, you may want to use list-directed formatting often with it.

cilist — is a control information list. It must include a unit specifier and, for formatted records, a format identifier/specifier. Other specifiers are optional. READ cilist specifiers are named next and detailed below:

[UNIT=] iu
[FMT=] f
IOSTAT = ios
ERR = s
END = s
REC = irn
RETURNRECL = iret

iolist — is an I/O list of entities that will receive the values read. We described iolist earlier under "Input/Output List".

The first form of READ reads from unit 5, preconnected to file @INPUT. This is a formatted, sequential READ.

The second form of READ can read from any unit. For a formatted read, include *FMT=*f; the data types of each iolist entity must match the data types of characters read from the record. For an unformatted read, omit *FMT=*f; F77 will read values into the iolist entities regardless of their data types.

Each READ statement for an internal file is sequential, and the statement must include an edit-directed format identifier. Unformatted READs or READs that specify list-directed formatting (FMT=*) are not allowed for internal files.

F77 always attempts to read characters from the record into all the iolist entities. If the iolist includes an unsubscripted array name, F77 will attempt to read values from the record into all array elements. If the record does not contain enough characters to satisfy the iolist entities, F77 will generally report an end-of-record runtime error. (This depends on the OPEN PAD= property.) F77 will read in a new record if:

1. It encounters a slash in the format specification; or
2. The format specification has too few edit descriptors, causing format reversion.

Both of these conditions are described in Chapter 6, "Multiple Record Formatting."

After a READ executes successfully, F77 has read at least one record and the iolist entities are defined with values read. The file is positioned at the beginning of the next record.

If an error occurs, the iolist entities and file position are undefined.

READ cilist Specifiers

[UNIT=] iu
is the unit specifier. For an external file, iu must be an integer expression that evaluates to a unit number from 0 through 255 for AOS/VS or from 0 through 63 for AOS, F7716, MP/AOS, and MP/AOS-SU. (But you can use * to specify unit 5, preconnected as the terminal keyboard.) For I/O with an external file that is not preconnected, iu must evaluate to the unit number on which you opened the file. For an internal file, iu is the name of the character variable, array, or substring that is the internal file. You can omit UNIT= if the unit specifier is the first argument in the cilist.

[FMT =] f
f is the format identifier/specifier, described earlier. It can be a character expression, the label of a FORMAT statement, or * to specify list-directed formatting. If you omitted UNIT= from the unit specifier, and the format identifier will be the second argument in the cilist, you can omit FMT=. For example, the following statements are equivalent:

```
READ (UNIT=14, FMT ='(I8)')   J, K
READ (14, '(I8)')   J, K
```

IOSTAT= ios
will return a status indicator. *ios* is an integer variable or integer array element that returns a status indicator:

- 0 if the READ executed normally
- a negative integer on end of file
- a positive integer that is either a F77 or operating system code if an error occurred.

If you include IOSTAT= and omit ERR= and END=, program execution continues at the next executable statement on an error or on an end of file.

ERR= s
s is the label of an executable statement that will receive control if an error occurs. Statement *s* must be in this program unit. When *s* gets control, IOSTAT's variable will contain the error code. On an error, a READ statement terminates, the iolist entities are undefined, and file position becomes indeterminate; if you omitted both IOSTAT= and ERR=, the program terminates.

END= s
s is the label of an executable statement that will receive control if READ encounters an endfile record or a CTRL-D sequence from @INPUT. Label *s* must be in this program unit. When *s* gets control, IOSTAT's variable will contain a negative value. Upon encountering an endfile record, a READ statement terminates. If you omitted both IOSTAT= and END=, the program terminates on an end of file.

REC= irn
gives the record number for *direct access only*. *irn* is a positive integer expression that specifies the number of the record you want accessed for a direct-access READ. The unit must have been opened for direct access. If you specify a nonexistent record number, the iolist entities will be undefined. If you include REC=, you cannot also include END= in the cilist. For an internal file, REC= is not allowed.

RETURNRECL= iret
This returns the byte length of the record that was just read. *iret* is an integer variable or array element that will return the actual byte length of the record on normal completion of the statement.

READ Examples

```
C       The following list-directed READ
C         statements each read values from unit 5.

        READ (UNIT=*, FMT=*)   B,C,D
        READ (*, *)   B,C,D
        READ *, B,C,D

C       An implied DO.
        READ *, (IARRY(I), I = 3, 1,-1 )

C       List-directed with cilist.
        READ (8, *, IOSTAT=IOS)  X,Y,Z

C       Edit directed, FORMAT statement.
        READ (8, 90)  X,Y,Z
        READ (3,90,IOSTAT=IOS) A,B,C
90      FORMAT (F9.2, F9.2, F9.2)

C       A formatted, direct-access READ.
        READ (2, 95, REC=J, ERR=99)   B1, B2
95      FORMAT (2 E11.4)

C       An unformatted READ.
        READ (3, IOSTAT=IOS)   B3, B4
```

Another example, which reads formatted real records from a file connected for sequential access, is

```
        ...
        DIMENSION RARY2(100)
        ...
        OPEN (2, FILE='PFILE')

        READ (2, 50, END=150)  RARY2
50      FORMAT (F13.0)

150     WRITE (*, 200)  (J, RARY2(J), J=1,100)
200     FORMAT (1X, 'Value ', I4, ' is ', G11.4)

        CLOSE (2)
        ...
```

The following example reads a two-record file four times.

```
        OPEN (3, FILE='MLFILE')

        READ (3, 60, IOSTAT=IOS) VAR1 ! IOS is 0
  60    FORMAT (F6.2)
        READ (3, 60, IOSTAT=IOS) VAR2 ! IOS is 0

        READ (3, 60, IOSTAT=IOS) VAR3 ! IOS
!               contains a negative number
!               because the endfile record was
!               just read.  The file is
!               positioned just AFTER the
!               endfile record.

        READ (3, 60, IOSTAT=IOS) VAR4 ! IOS
!               contains F77ERIOUEN.

        CLOSE (3)
        STOP
        END
```

There are other READ examples under OPEN.

Using WRITE

A record must exist before it can be read — either by people or by F77 programs. WRITE is the primary statement used to create and assign values to records (although PRINT can transfer them to unit 6).

WRITE transfers all iolist values to a unit as one record (unless, for formatted records, you specified more than one, described in Chapter 6, "Multiple Record Formatting".) If you include a format, F77 formats the iolist values before writing them to the record. The form of WRITE is

WRITE (cilist) *[iolist]*

where:

cilist — is, as for READ, a control information list. It must include a unit specifier and, for formatted records, a format identifier. Other specifiers are optional. All cilist specifiers are named next and detailed later in this section.

[UNIT=] iu
[FMT=] f
REC = irn
IOSTAT = ios
ERR = s
RETURNRECL = iret

iolist is an I/O list of entities whose values will be written to the unit, described earlier under "Input/Output List". You can include expressions; e.g., 41 or K+5 or 'ABC'.

Unless you specified unformatted records and/or direct access on the OPEN, F77 expects a formatted, sequential-access WRITE.

For a formatted WRITE, include *[FMT=]*f; the values in each iolist entity must match the format used. The first character in each record may be used for carriage control as described under OPEN, CARRIAGECONTROL= property.

For an unformatted WRITE, omit *[FMT=]*f; F77 will write values from the iolist entities as they are stored.

Each write to an internal file is sequential and the statement must include an *edit-directed* format identifier. Unformatted WRITEs or WRITEs that specify list-directed formatting (FMT=*) are not allowed for internal files.

For any WRITE statement, F77 will write characters from all the iolist entities to the current record. It will start a new record in this write only under the circumstances described in Chapter 6, "Multiple Record Formatting" and "List-Directed Formatting."

If the iolist includes an unsubscripted array name, F77 will write until it has written values from all array elements in the traditional column/row order described in Chapter 2.

When a sequential WRITE statement executes, the end of file is automatically defined after the record just written. If the file you are writing is open on another unit or by another process, the results of the WRITE are undefined, and the system may return an error from it. If other processes may use a file to which you will want to write, you may want to open it with the EXCLUSIVE access specifier (described under OPEN).

To end the file, you can either stop writing records to the unit or, if the file is connected for sequential access, write the endfile record with ENDFILE. To append to the file, you can either open the file with the POSITION='END' OPEN specifier or read until you encounter the end of file (END=); then you can overwrite the last record — which is the endfile record — via a BACKSPACE and WRITE sequence.

After a WRITE statement executes successfully, F77 has written at least one record that consists of values from all iolist entities. The file is positioned at the beginning of the next record. If an error occurs, file position is undefined.

Fixed-Length Records

Internal files, files you write via direct access, and files opened with RECFM='FIXED' (described under OPEN) all have fixed-length records. When you write to a fixed-length record, and the iolist values require *fewer* bytes than the record length, F77 pads the values to the the record length. The pad character used is blank for a formatted record or null for an unformatted record. If the iolist entities require *more* bytes than exist in the record, F77 signals an error and the record is undefined.

When you write to an internal file, the record length is the number of bytes declared for the character variable, array, array element, or substring that is the internal file.

WRITE cilist Specifiers

[UNIT=] iu
is the unit specifier. For an external file, iu must be an integer expression that evaluates to a unit number from 0 through 255 for AOS/VS or from 0 through 63 for AOS, F7716, MP/AOS, and MP/AOS-SU. (But you can use * to specify unit 6, preconnected to file @OUTPUT). For I/O with an external file that is not preconnected, iu must evaluate to the unit number on which you opened the file. For an internal file, iu is the name of the character variable, array, array element, or substring that is the internal file. You can omit UNIT= if the unit specifier is the first argument in the cilist.

[FMT=] f
f is the format identifier/specifier, described earlier. It can be a character expression, the label of a FORMAT statement, or * to specify list-directed formatting. If you omitted UNIT= from the unit specifier, and the format identifier will be the second argument in the cilist, you can omit FMT=. For example, the following statements are equivalent:

```
WRITE (UNIT=14, FMT ='(I9)')   J, K
WRITE (     14,      '(I9)')   J, K
```

And, the following list-directed I/O statements are equivalent:

```
WRITE (UNIT= *, FMT= *)   L, M
WRITE (      *,      *)   L, M
```

IOSTAT= ios
will return a status indicator. *ios* is an integer variable or integer array element that returns

- 0 if the WRITE executed normally, or
- a positive integer that is either a F77 or operating system code if an error occurred.

If you include IOSTAT= and omit ERR=, program execution continues at the next executable statement on an error.

ERR= s
s is the label of an executable statement that will receive control if an error occurs. Statement *s* must be in this program unit. When *s* gets control, IOSTAT's variable will contain the error code. On an error, a READ or WRITE statement terminates and file position becomes indeterminate; if you omitted both IOSTAT= and ERR=, the program terminates.

REC= irn
gives the record number for direct access only. *irn* is a positive integer expression that specifies the number of the record you want to write. In direct access, preceding record numbers need not exist; for example you can write record 3 whether or not records 1 and 2 exist.

You cannot use REC= unless the unit was connected for direct access. Your OPEN statement must have the specifier ACCESS='DIRECT'. If an OPEN does not include ACCESS='DIRECT', F77 connects the unit for sequential access.

RETURNRECL= iret
iret is an integer variable or array element that will return the number of bytes written to the record.

WRITE Examples

```
C     Assume real values for X, Y and Z and integer
C        values for J, K, and L. Assume units
C        2 and 1 are OPEN.

C     Formatted, sequential, list-directed WRITEs.
         WRITE (UNIT=*, FMT=*)   X,Y,Z
         WRITE (*,*)    Q,R,S
         WRITE (1, *, IOSTAT=IOS)   X,Y,Z

C     Formatted, sequential, edit-directed WRITEs.
         CHARACTER*10 CX / '(1X,  3I8)' /
         WRITE (8, CX)   J, K, L

         WRITE (1, 85)    X,Y,Z
85       FORMAT (1X, F9.2, F9.2, F9.2)

         WRITE (*,90)   J,K,L
90       FORMAT (1X, I8, I8, I8)

C     A formatted, direct-access WRITE.
         WRITE (1,25,IOSTAT=IOS,REC=J)   X,Y
25       FORMAT (1X, 8F10.2)

C     An unformatted WRITE.
         WRITE (2, IOSTAT=IOS) X,Y,Z
```

The next example writes formatted records to a file connected for sequential access.

```
         ...
         dimension RY(100)
         ...
C        Here, program assigns real values to RY.
         ...
         open (3, file='PFILE')
         write (3, '(1X, F14.2)' )    RY
         close (3)
         ...
```

File PFILE can now be queued for printing on the line printer. The OPEN examples, later, include other WRITE statements.

Using PRINT

PRINT is a shorthand WRITE. Like WRITE, it transfers iolist values, but always to unit 6, which is preconnected to file @OUTPUT (for interactive use, the terminal screen or printer). PRINT cannot write to any unit other than 6. Its form is

PRINT f *[,iolist]*

where:

f — is a format identifier/specifier. It can be a character expression, the label of a FORMAT statement, or asterisk (*) for list-directed formatting. Because PRINT implies file @OUTPUT, you may want to use list-directed formatting often with it.

iolist — is an I/O list of entities whose values will be written to unit 6, as described earlier under "Input/Output List". You can include expressions; e.g., 41 or K+5 or 'ABC'.

PRINT is preconnected to unit number 6 — file @OUTPUT which, for interactive use, is the terminal screen/printer.

In terms of formats and iolist entities (arrays, etc.), PRINT functions just as WRITE does.

After PRINT executes, the file is positioned after the last record printed.

PRINT Examples

```
      PRINT *, 'IFOO is}', IFOO   ! List-directed.

      PRINT *, B,C,D              ! List-directed.

      PRINT '(1X, 3F9.2)', B,C,D  ! Edit directed.

C     Edit-directed via FORMAT statement:
      PRINT 96,   B,C,D
96    FORMAT (' Values are ' , 3F9.2)

C     Edit-directed via character entity:
      CHARACTER*10 CZ / '(1X,F9.2)' /
      PRINT CZ, B,C,D

C     List-directed with implied DO:
      PRINT *, (RARY(J), J = 50, 100)

C     Edit-directed with implied DO:
      PRINT 99, (RARY(K), K = 10, 20)
99    FORMAT (1X, 'Values ', F12.2)
```

Other PRINT examples appear under OPEN.

Auxiliary I/O

The auxiliary I/O statements and their functions are

OPEN	establishes a unit/file connection;
INQUIRE	returns information about units and/or files;
CLOSE	severs a unit/file connection.

OPEN, CLOSE, and INQUIRE don't transfer records, and thus don't use formats or iolists; instead, each has its own set of specifiers.

These statements apply to external files only; you cannot use them for internal files.

Using OPEN

OPEN connects a unit to an external file. It establishes the connection properties that you either specify or, by omitting, select by default. The most important of these properties are

- a method of *access*, sequential or direct. The default is sequential.
- a *form*, formatted or unformatted. The default is formatted for sequential access.
- a *maximum record length*, for sequential access or *record length*, for direct access.
- a *blank handling* property for formatted records. This determines whether blanks in numeric fields will have no value or will be treated as zeros; it is irrelevant on output. The default is that blanks have no value.
- a *carriage-control* handling property for formatted records. This determines whether the first character of each output record will be interpreted for print control. The default property does not interpret the first character for carriage control.

When you open a file, you must specify a unit number and may specify a filename. The unit number must be an integer expression that evaluates to a positive integer from 0 through 255 for AOS/VS or from 0 through 63 for AOS, F7716, MP/AOS, and MP/AOS-SU.

To change the properties of a connection, you can reopen the connection simply by opening the same unit/file combination with the new properties you want.

If a unit number is in use, and you specify a filename that differs from the filename connected to this unit, F77 closes the original connection and opens the file specified on this number. Units 5, 6, 9 (except MP/AOS and MP/AOS-SU), 10, 11, and 12 (except MP/AOS and MP/AOS-SU) are preconnected.

Each unit/file connection is global while it exists; it applies to *all* program units. Any program unit can issue an OPEN, which may be a reOPEN if any unit has already opened the connection.

The form of OPEN is

OPEN (open-property-specifiers)

where the open-property-specifers must include a unit number *(UNIT=*iu*)* and a legal combination of zero or more of the following property specifiers (sketched in Table 5-1, detailed in the next section). Most specifiers are optional. Most *arguments* to specifiers are type character; those of type integer begin with *i*. All default values, if you omit the specifier, are noted. The general specifiers — FILE, IOSTAT, ERR and STATUS — are first; the others follow alphabetically.

AOS/VS programmers may use a logical*1/byte entity as an OPEN specifier whenever character or integer entities are allowed. An exception is IOSTAT, since all I/O error codes exceed 127.

Table 5-1. OPEN Property Specifiers

Property Specifier	Description
FILE= fin	Specifies filename; use *fin* of filename or pathname to connect.
IOSTAT= ios	Returns status; use an integer variable for *ios*.
ERR= s	Error statement; for *s*, use label of statement to execute on error.
STATUS= sta	File status; for *sta*, use 'NEW', 'FRESH', 'OLD', 'SCRATCH', or 'UNKNOWN' (default).
ACCESS= acc	For *acc*, use 'DIRECT' or 'SEQUENTIAL' (default).
BLANK= blnk	For *blnk*, use 'ZERO' or 'NULL' (default).
BLOCKSIZE= iblks	Disk block size; selecting 0 defaults this to standard (typically 512 bytes). MP/AOS and MP/AOS-SU do not support BLOCKSIZE; their compilers issue a warning message.
CARRIAGECONTROL=cc	For *cc*, use 'FORTRAN', 'NONE', or 'LIST' (default).
DELIMITER= del	For *del*, use 'INCLUDE' or 'EXCLUDE' (default).
ERRORLOG= erl	For *erl*, use 'YES' or 'NO' (default).
EXCLUSIVE= exc	For *exc*, use 'YES' for exclusive access or 'NO' (default).
FORCE= for	For *for*, use 'YES' to minimize buffering or 'NO' (default).
FORM= fm	For *fm*, use 'UNFORMATTED' or 'FORMATTED' (default).
IOINTENT= oio	I/O intent; for *oio*, use 'OUTPUT', 'INPUT', or 'BOTH' (default).
MAXRECL= imax	Maximum record length; for *imax*, use integer expression with maximum length or default.
MODE= mod	Data transfer mode; for *mod*, use 'BINARY' or 'TEXT' (default). MP/AOS and MP/AOS-SU do not support MODE; their compilers issue a warning message.
PAD= pad	Use *pad* of 'YES' to pad to MAXRECL on input or 'NO' (default).
POSITION= pos	Use *pos* of 'END', 'START,' or 'CURRENT' (default).
RECFM= rfm	Use *rfm* of 'DATASENSITIVE' or 'DS' (default), 'FIXED', 'DYNAMIC', or 'VARIABLE'.
RECL= irl	Direct access record length; use integer expression for *irl*.
SCREENEDIT= sed	Use *sed* of 'YES' to enable SCREENEDIT Control Character keys during READ statements issued to terminals; use 'NO' (default) to disable these keys. Only AOS/VS F77 allows this specifier.
TYPE= ityp	For *ityp*, use integer system file type.

You can include a filename (which can be a pathname including the filename) in FILE=; if you omit FILE=, F77 may use a predetermined disk file as described in STATUS=. The IOSTAT= and ERR= specifiers are error-handling options; if you omit both of them, the program will terminate if an error occurs on the OPEN.

The properties whose values you can change via a successive OPEN (reopen) are ACCESS, BLANK, CARRIAGECONTROL, DELIMITER, ERRORLOG, FORCE, FORM, MAXRECL, MODE, PAD, RECFM, RECL, and SCREENEDIT. For any properties you omit on a reopen, F77 uses the existing properties of the connection. If you specify unneeded properties on *any* OPEN, F77 discards them; but if you specify conflicting properties, F77 will report a compile or runtime error. Conflicting properties are described later.

OPEN Property Specifiers

UNIT= iu
iu identifies the unit in all auxiliary I/O statements except an INQUIRE by file. You can omit UNIT= if you place the number iu first in the list; if you put unit specifier anywhere else, you must include UNIT=. Number iu must be an integer expression that evaluates to an integer from 0 through 255 for AOS/VS or from 0 through 63 for AOS, F7716, MP/AOS, and MP/AOS-SU.

FILE= fin
fin is a character expression that gives the filename or the pathname to the file you want connected. Valid system filenames include alphabetic A–Z, upper- and/or lower-case, numbers 0–9, $, period (.), question mark (?), and underscore (_); they can include up to 32 characters (15 for MP/AOS and MP/AOS-SU). If the file will not be in the directory where you will execute the program, you can specify a pathname through the directory tree to it; e.g., ':UDD:MYDIR:MYFILE'. Or, you can specify the file's directory in a search list. Both filenames and search lists are further described in your system *CLI User's Manual*.

If you omit FILE=, F77 will use a predetermined file as described under STATUS=.

IOSTAT= ios
ios must be an integer variable or array element to receive a status indicator. After the OPEN, the value of *ios* will be 0 if the OPEN executed normally. If an error occurred, *ios* will contain an F77 or system error code. These codes are positive integer values. Those that pertain directly to F77 are described in an F77 disk file named ERR.F77.IN.

If you include IOSTAT= and omit ERR=, program execution will continue at the next statement if an error occurs — which means that you should use the next statement to check *ios*. If you omit both IOSTAT= and ERR=, program execution will terminate on an OPEN error.

ERR= s
ERR= directs control to the statement labelled *s* if an error occurs; *s* must be an executable statement in the current program unit. IOSTAT's variable, if you included IOSTAT=, will contain an error code.

STATUS= sta
STATUS allows you to specify how the system will establish a unit/file connection. You can use it to open new files or scratch files, but can omit it for successive opens. *sta* is a character expression that, after F77 has ignored any trailing blanks, is 'NEW', 'FRESH', 'OLD', 'SCRATCH,' or 'UNKNOWN.'

- If you want to connect to a new file (i.e., a file whose name doesn't exist in the pertinent directory), use STATUS='NEW' and specify the filename in FILE=. F77 will attempt to create the file and will return an error if this file already exists.
- To connect to a fresh version of a file, whether it exists or not, use STATUS='FRESH'. If you specify a filename in FILE=, F77 will delete the file if it exists, then create a new, empty version with the given filename. If you omit FILE=, F77 will use the filename ?pid.F77.UNIT.unit (?pid.F77S.unit under MP/AOS and MP/AOS-SU), where pid is your three-digit integer process ID (such as 016) and unit is the integer unit number.
- To connect to a file only if it already exists, use STATUS='OLD' and specify the filename in FILE=. F77 will attempt to OPEN the file. If the file does not exist, F77 will return an error.
- To connect to a scratch (temporary) file, use STATUS='SCRATCH' and *omit* FILE=. F77 will create (or delete and recreate, if it exists) a scratch disk file in the working directory. The filename, inaccessible from F77 programs, will be ?pid.F77.SCRATCH.unit.TMP (?pid.F77S.unit under MP/AOS and MP/AOS-SU), where pid is your three-digit integer process ID number and unit is the integer unit number. When you close this unit, or the program ends (closing all files), F77 will have deleted the scratch file. From the standpoint of F77, a scratch file has no name.

- To connect to a file whether or not it exists, use 'UNKNOWN' (the default). If you include FILE=, F77 will create and open (or just open, if it exists) the file. If you omit FILE=, F77 will create and open (or just open) the file with the filename ?pid.F77.UNIT.unit (?pid.F77U.unit under MP/AOS and MP/AOS-SU), where pid is your three-digit integer process ID and unit is the integer unit number. File position will be at the beginning of the file unless you specify otherwise with POSITION=. ('UNKNOWN' differs from 'FRESH' in that it does not delete the file if it exists.)

On an initial OPEN statement, a status of 'UNKNOWN' connects to any file; 'UNKNOWN' is the default. If you specify 'OLD' and the file does not exist, or if you specify 'NEW' and the file already exists (perhaps created by a previous F77 OPEN), F77 will return an error.

On a reopen, you can always omit STATUS= — but, if you include STATUS= on a reopen, it must match the existing property of the file: 'OLD' or 'SCRATCH'.

ACCESS= acc
This specifies the type of access. *acc* is a character expression that, when F77 has ignored any trailing blanks, is 'DIRECT' or 'SEQUENTIAL'. If you specify ACCESS='DIRECT', you must include a length in RECL=, later in this list.

If you omit the ACCESS= specifier, F77 connects the file for sequential access.

BLANK=blnk
This controls the way F77 will handle blanks when it reads numeric values. *blnk* is a character expression that, after F77 has ignored any trailing blanks, is 'NULL' or 'ZERO'.

When it reads a numeric value, F77 always ignores leading blanks. By default, it treats embedded blanks as nulls; e.g., it interprets 2□□1 as 21 (but it always interprets a value of all blanks as 0). If you specify BLANK='ZERO', F77 will treat each blank in numeric values as 0; e.g., it will interpret 2□□1 as 2001. You can also control this in FORMAT specifications with the BZ/BN edit descriptors, described in Chapter 6. If there is a conflict between the value of BLANK= and a subsequent BZ/BN edit descriptor in a FORMAT statement, then the BZ/BN value prevails.

BLOCKSIZE= iblks
This allows you to specify a nonstandard device block size; the standard size is 512 bytes. In most cases, you will want to use the standard size, so omit this item; the system will then use the standard size. If you specify a different size, *iblks* must be a positive integer expression that evaluates to the desired block size in bytes; e.g., 1024. An *iblks* value of 0 selects the standard block size.

MP/AOS and MP/AOS-SU F77 do not support BLOCKSIZE. If you specify it, the compilers display a warning message, generate no code from BLOCKSIZE, and continue.

CARRIAGECONTROL= cc
This lets you select carriage control for output of formatted records. F77 ignores it on input and with unformatted records. *cc* is a character expression that, after F77 has ignored any trailing blanks, is 'LIST', 'NONE', or 'FORTRAN'. LIST is the default.

CARRIAGECONTROL is most meaningful for files that you will print. "Print" in this description of CARRIAGECONTROL=cc includes the output of data to a terminal as well as to a disk file and printer. Therefore — in most cases — you will use it with data-sensitive files. Data-sensitive files are the default.

If you omit the CARRIAGECONTROL= specifier, F77 will use LIST carriage control, the default. For data-sensitive files with LIST carriage control, F77 inserts a NEW LINE (<NL>=<012>) character after each record. This prints a record on the current line and ensures that the next record appears on the next line. If you specify CARRIAGECONTROL='NONE' for a data-sensitive file, F77 will not insert the NEW LINE delimiter (as it would normally do for LIST) after each record.

If you specify CARRIAGECONTROL='FORTRAN' for a data-sensitive file, F77 will use the first character in each record for traditional carriage control as follows:

First Character in Record	Result when Record Is Printed
□	Prints record on next line (NEW LINE, record).
0	Prints record on line after next (NEW LINE, NEW LINE, record).
1	Prints record on top of next page (form feed, record).
+	Prints record on same line, starting with column 1 (overprints).
#	Prints record starting at current column position (useful to print multiple records on one line).
other character	Prints record on next line (NEW LINE).

The first character in each record does *not* print; its sole purpose is to control the placement of the remaining characters in the record.

On output to a data-sensitive file opened with FORTRAN carriage control, F77 does *not* terminate the file with a NEW LINE character. Such files will print correctly, but you cannot edit the last line successfully with a line-oriented text editor (the SPEED editor works well). One solution is to simply write a NEW LINE character to this file just before closing it. This line doesn't have to contain any other characters. For example,

```
      WRITE (2, 999)
999   FORMAT (1H )   ! or  FORMAT (' ')
```

will do the job. The Hollerith character □ results in the output of a NEW LINE character.

For organizations other than data sensitive, F77 will insert characters that will produce the same *effect* as with data-sensitive records. But files with organizations other than data sensitive will usually not print directly. To get the desired result after specifying CARRIAGECONTROL on a nondata-sensitive file, you must run the F77PRINT utility on the file. The F77PRINT utility can produce a printable version of *any* F77 output file. F77PRINT is described in Appendix D.

To get an organization other than data sensitive, usually you must use the RECFM= specifier, described later in this section. If an OPEN statement creates a new file, it is created with the default organization or the organization given in RECFM=. CARRIAGECONTROL and RECFM determine the type of the new file, unless you specify a different TYPE=. (Knowing the type is useful because you can use the system CLI command FILESTATUS with the /TYPE= switch to identify files of a given type.)

For unformatted records (with FORM='UNFORMATTED'), the type is always UDF. For formatted records, the types are as follows:

If cc Is	Data-Sensitive RECFM	Variable or Fixed RECFM
'LIST'	new file type is TXT.	new file type is LCC.
'NONE'	new file type is TXT.	new file type is NCC.
'FORTRAN'	new file type is OCC.	new file type is FCC.

Dynamic RECFM does not appear here because dynamic records are always unformatted.

DELIMITER= del
This lets you specify the way the delimiters for data-sensitive records will be handled on input. DELIMITER has no effect on output or with other file organizations (e.g., variable). *del* is a character expression that, after F77 has ignored any trailing blanks, is 'INCLUDE' or 'EXCLUDE'. 'EXCLUDE' means that on input, the record delimiter character (usually NEW LINE) is stripped. DELIMITER='INCLUDE' *retains* the input delimiter as part of each record; it means that you can manipulate the delimiter just like any other character in the record. 'EXCLUDE' is the default.

ERRORLOG= erl
erl is a character expression whose value (after F77 has ignored any trailing blanks and has translated any lowercase letters to uppercase) is 'YES' or 'NO'.

If *erl* is 'YES', then the unit is added to the program's set of error log units. This means that F77, in response to any runtime error, will write a report about the error to this set of units/files.

If *erl* is 'NO', the unit never receives an error message when a runtime error occurs. The default value of *erl* is 'NO'.

The default preconnection for unit 6, @OUTPUT, includes ERRORLOG='YES'.

F77 sends runtime error messages to all units OPENed with ERRORLOG='YES'. If there is no such unit, then F77 attempts to send any such message to @OUTPUT.

The current value of *erl* may change if a program unit reopens the unit.

EXCLUSIVE= exc
This specifies whether or not you want the file connected for exclusive access. If you specify exclusive access, neither your program nor any other program can make another connection to this file. *exc* is a character expression that, after F77 has ignored any trailing blanks, is 'YES' or 'NO'. Use EXCLUSIVE='YES' for exclusive access. If you specify 'YES' and the file is already connected, you'll receive an error when the OPEN statement executes. The default is 'NO'.

FORCE= for
FORCE allows you to minimize F77 buffering and system buffering. *for* is a character expression that, after F77 has ignored any trailing blanks, is 'YES' or 'NO'.

When you specify FORCE='NO' (the default), normal F77 and system buffering collect the file's records in memory. The records move from buffers to the output file, or from the input file to the buffers. The movement occurs when the buffers are exhausted (full for output, empty for input). Buffering increases disk I/O speed.

When you specify FORCE='YES', all output goes directly to the file, bypassing F77 and system buffers. Any previously buffered records awaiting output move to the file. On input, F77 does not place the file's data into a buffer although the system may do so. More specifically, AOS and AOS/VS process a ?READ system call (resulting from an F77 READ statement) by returning data from a system buffer. MP/AOS and MP/AOS-SU react to an F77 READ statement by reading the file directly.

The OPEN statement's FORCE setting remains in effect for all subsequent I/O operations on the specified unit. You may modify the FORCE setting during a reopen.

Preconnected units 5, 6, 10, and 11 are opened with FORCE='YES'; units 9 and 12 are opened with FORCE='NO'. Recall that MP/AOS and MP/AOS-SU F77 don't have preconnected units 9 and 12.

FORM= fm
This allows you to specify formatted or unformatted records. *fm* is a character expression that, after F77 has ignored any trailing blanks, is 'FORMATTED' or 'UNFORMATTED'. If you said ACCESS='SEQUENTIAL' or defaulted ACCESS=, the default FORM is formatted. If you said ACCESS= 'DIRECT', the default is unformatted. FORM allows you to specify combinations like sequential access to unformatted records or direct access to formatted records.

IOINTENT= oio
This allows you to specify output, input, or both for a unit. *oio* is a character expression that, after F77 has removed any blanks, is 'OUTPUT', 'INPUT', or 'BOTH'. BOTH is the default. If you specify 'OUTPUT', you will not be able to read the file; if you specify 'INPUT', you will not be able to write to it.

MAXRECL= imax
This allows you to specify a maximum length in bytes for formatted records in a file connected for sequential access (but is ignored for dynamic records). Use RECL=, not MAXRECL=, with a file connected for direct access. *imax* is an integer expression that gives the maximum number of bytes you want allowed for any record in this unit.

If any record exceeds the maximum length on a write, F77 will generate a runtime error. If, on a read, a data-sensitive record exceeds *imax,* F77 reports no error but reads the record up to *imax*; the rest of the record will be available on the next read. If, on a read, a variable-length record exceeds *imax,* F77 also will generate a runtime error.

For AOS/VS variable records, the default maximum record length is 9,995 bytes; it is 136 bytes for other organizations. The AOS/VS system limit for variable records is 9,995 bytes. For data-sensitive records the limit is 32,766 bytes. For fixed-length records it is 32,767 bytes (formatted) or 65,534 bytes (unformatted). For dynamic records, there is no inherent limit to length.

For AOS, F7716, MP/AOS, and MP/AOS-SU, the default maximum record length is 136 bytes for all organizations. Their system maximum for variable-length records is 9,995 bytes; for other organizations the limit is 65,534 bytes.

The system file organization — variable length, data sensitive, fixed, or dynamic — is given or selected by default as described under RECFM=, below.

MAXRECL specifies the exact length of all the records when a file with RECFM='FIXED' is created.

In contrast to the other operating systems, MP/AOS and MP/AOS-SU do *not* keep track of a file's record length. You need to include MAXRECL with every OPEN statement in which you specify RECFM='FIXED' (MP/AOS, MP/AOS-SU programs only). The specified value of MAXRECL becomes the length of all records in the file.

MODE= mod
mod is a character expression whose value after F77 has ignored any trailing blanks is 'TEXT' or 'BINARY'. MODE='BINARY' instructs F77 and the operating system to transfer bytes on reads or writes without parity checks or interpretation. This allows F77 to read character sequences like CTRL-C CTRL-B or others from the terminal without interrupting program execution. BINARY mode lasts only while this connection is open. The default mode is 'TEXT', in which control character sequences *are* interpreted.

MP/AOS and MP/AOS-SU F77 do not support MODE. If you specify it, the compilers display a warning message, generate no code from MODE, and continue.

PAD= pad
PAD can instruct F77 to pad records to MAXRECL on input. *pad* is a character expression that, after F77 has ignored any trailing blanks, is 'YES' or 'NO'. If PAD='YES', F77 will pad each input record to the maximum record length if needed; the pad character(s) will be blanks for formatted records and nulls for unformatted records. READ's RETURNRECL variable will *not* include these pad characters. Padding will prevent "Attempt to read past end of record" errors when the input format specifies more characters than remain in the record. But F77 will report such an error if the input format specifies more characters than the MAXRECL.

PAD is meaningful only for variable or data-sensitive file organizations; F77 automatically pads records in fixed files to the length you give with the RECL= OPEN specifier. If PAD='NO', F77 will not pad the records on

input. The default value is 'NO'. On list-directed reads, *pad* is ignored and F77 treats the read as if PAD='NO'.

POSITION= pos
POSITION allows you to position the file to its start, end, or current position. *pos* is a character expression that, after F77 has ignored any trailing blanks, is 'START', 'CURRENT', or 'END'. On the first OPEN of a file, the current position is the start position. On subsequent OPEN statements (reopens) of this file, the current position set during the previous OPEN will be carried over. For example, if you open a file for sequential access, read five records, then reopen with POSITION='CURRENT', the current position will be after record five. The only legal value on a reopen is 'CURRENT'.

On the initial OPEN, POSITION='END' positions at the end of the file before the endfile record; this is useful when you want to write records at the end of the file (append to it). The default position is 'CURRENT' — which, on the initial OPEN, is the same as 'START'. POSITION has no meaning for files that cannot be positioned, like a terminal.

RECFM= rfm
RECFM allows you to specify the operating system record organization when an OPEN statement creates the file. The organization attribute is stored with the file and cannot be changed while the file exists. *rfm* is a character expression that, after F77 has ignored any trailing blanks, is 'DATASENSITIVE' (or 'DS'), 'VARIABLE', 'FIXED', or 'DYNAMIC'. The default value of RECFM is 'DATASENSITIVE' (or 'DS').

If the OPEN statement creates the file, it will have the organization you specify in RECFM=. If you omit RECFM= and ACCESS='SEQUENTIAL', F77 will create the file with data-sensitive organization. If ACCESS='DIRECT', F77 will create the file with fixed organization. If FORM='UNFORMATTED', the default value of RECFM is 'VARIABLE'.

If the file exists, you omit RECFM=, and other specifiers do not imply a RECFM, F77 will use the RECFM used when the file was created.

Figure 5-1, later in this chapter, clarifies these conditionals.

If the file exists and you specify a RECFM that differs from the file's organization, F77 will gain access to its records via the new RECFM. But using records with one kind of organization via another kind is tricky and you should do it with extreme care, if at all.

In contrast to the other operating systems, MP/AOS and MP/AOS-SU do *not* keep track of a file's record format. Unless the value of the RECFM specifier is the default, you need to include it with every OPEN statement in your MP/AOS and MP/AOS-SU F77 programs.

RECL= irl
RECL specifies the record byte length *for direct access*. *irl* is an integer expression that evaluates to a positive integer value which is the byte length of each record. The records can be formatted or unformatted for direct access, but they must have the same length. When an OPEN that includes RECL= creates a new file, the length given in *irl* becomes the record length for that file. However, you can specify a different length in subsequent OPEN statements.

To connect a file for direct access, you must include ACCESS='DIRECT' and RECL= in the OPEN statement; to connect it for sequential access, you can omit ACCESS= and must omit RECL=.

In contrast to the other operating systems, MP/AOS and MP/AOS-SU do *not* keep track of a file's record length. You need to include RECL with every OPEN statement in which you specify ACCESS='DIRECT' (MP/AOS, MP/AOS-SU programs only).

SCREENEDIT= sed
sed is a character expression that, after F77 has ignored any trailing blanks, is 'YES' or 'NO'. If SCREENEDIT='YES', F77 will allow an on-line user to use SCREENEDIT control characters (such as the ERASE EOL key and the CTRL-A two-key sequence) while preparing a string of ASCII characters as input to a READ statement. If SCREENEDIT='NO' (default), then an on-line user cannot take advantage of these SCREENEDIT control characters while entering data.

SCREENEDIT control characters are usable only on video display terminals. If you explicitly specify SCREENEDIT='YES' for an OPEN statement to a terminal with the /HARDCOPY characteristic, a "File read" error will be reported on the first read to that unit. If you implicitly open a terminal with the /HARDCOPY characteristic by preconnection to units 5 and 11 (and thus the "*" unit specifier), the F77 I/O system will not enable the use of SCREENEDIT control characters, and "File read" errors will not occur.

The combination of specifiers, SCREENEDIT='YES' and MODE='BINARY', will produce a F77 runtime error.

When used in conjunction with a nonterminal file or device, such as a disk or tape file, the operating system ignores SCREENEDIT reads. Hence, the value of the SCREENEDIT specifier on the OPEN statement for such a file or device is immaterial.

The value of the SCREENEDIT specifier may be changed on a reopen.

For example, suppose a program contains these related statements:

```
      CHARACTER*20 ADDRESS
      ...
      READ (5, 30) ADDRESS
30    FORMAT (A)
      ...
```

The following 31 keystrokes (which the user makes and sees on one line) will result in the correct entry of "2 Clearview Drive":

22□Clw< CTRL-U >2□Cleav< DEL >rviw< ← >
< CTRL-E >e< CTRL-E >< → >< → >Drive ↓

TYPE= ityp
This allows you to specify an operating system file type if this OPEN statement creates the file. *ityp* is an integer expression that gives the file type, described in your operating system reference manual.

Table 5-2 shows the illegal combinations and values for the OPEN property specifiers.

Table 5-2. Illegal OPEN Property Specifiers

Property	Other Property, Value, Description
UNIT= <0	always an error.
UNIT= >255 (AOS/VS)	always an error.
UNIT= >63 (AOS, F7716, MP/AOS, MP/AOS-SU)	always an error.
ACCESS='SEQUENTIAL' (or omitted)	with RECL=, an error.
ACCESS='SEQUENTIAL' (or omitted) and RECFM='FIXED.'	without MAXRECL=, an error (unless the fixed-length file already exists).
ACCESS='DIRECT'	without RECL=, an error; with RECFM= anything but 'FIXED', an error.
CARRIAGECONTROL=*cc* or DELIMITER=*del*	with FORM='UNFORMATTED', specifier is ignored, no error.
FORM='FORMATTED'	with RECFM='DYNAMIC', an error.
MAXRECL= <1 or MAXRECL= >65534	always an error.
MAXRECL= >9995	with variable records, an error.
POSITION='START' or POSITION='END'	on reopen, an error.
RECFM='DYNAMIC'	with FORM='FORMATTED', an error.
RECFM='DATASENSITIVE'	with FORM='UNFORMATTED', an error.
RECL=*irl*	with ACCESS='SEQUENTIAL', an error.
RECL=*irl* (omitted)	with ACCESS='DIRECT', an error.
STATUS='SCRATCH'	with FILE=, an error.
STATUS='NEW' or STATUS='OLD'	without FILE=, or on reopen, an error.
STATUS='UNKNOWN' or STATUS='SCRATCH'	On reopen, an error.

Figure 5-1 shows the relationships among some OPEN specifiers. The figure applies only to initial opens, *not reopens*. Three related facts that are not in the figure are:

- If the file exists and has a record format, then RECFM= the file's original record format.
- If the file exists and the record length was specified at the file's creation, then RECL= the file's original record length.
- If the file exists and the maximum record length was specified at the file's creation, then MAXRECL= the file's original maximum record length.

These three facts don't apply to MP/AOS and MP/AOS-SU F77. MP/AOS and MP/AOS-SU don't keep track of as much information about files as the other operating systems do. You've already seen that MP/AOS and MP/AOS-SU OPEN statements require the RECFM and MAXRECL/RECL specifiers (unless the record format is the default, data sensitive).

F77 Preconnections

F77 opens its preconnected units only when needed during program execution; this is an *implied* or *default* OPEN. Table 5-3 shows the unit numbers, their preconnected files, and OPEN properties.

MP/AOS and MP/AOS-SU F77 do not have preconnected units 9 and 12. By default, their preconnected units 5 and 11 use channel ?INCH (process input file), while their preconnected units 6 and 10 use channel ?OUCH (process output file).

Programs that use these preconnections are not directly transportable to (non-DG) processors that have different preconnections.

F77 does not create the files referred to by @DATA and @LIST; you must create and set them to your datafile and listfile via the CLI. For best results, create the listfile with data-sensitive organization. For example:

```
) CREATE MY.DATAFILE ⤸
) DATAFILE MY.DATAFILE ⤸
) CREATE/DATASENS MY.LISTFILE ⤸
) LISTFILE MY.LISTFILE ⤸
)
```

Now you can run the program that uses preconnected units 9 (@DATA) and 12 (@LIST); these units will be preconnected to MY.DATAFILE and MY.LISTFILE respectively. If a program attempts I/O to preconnected unit 9 or 12 and the pertinent file (e.g., MY.DATAFILE) doesn't exist, F77 returns a "File does not exist" error.

For another example, suppose that program INVENTORY_100.F77 uses file MASTER for input and file MASTER_EXTRACT_100 for output. Suppose also that INVENTORY_100.F77 uses preconnected units 9 and 12, with no OPEN statement that associates unit 9/MASTER and unit 12/MASTER_EXTRACT_100. Macro XEQ_INVENTORY_100.CLI might contain the following:

```
PUSH
DATAFILE MASTER
DELETE/2=IGNORE MASTER_EXTRACT_100
LISTFILE MASTER_EXTRACT_100
XEQ INVENTORY_100
COMMENT RESTORE THE DATAFILE AND LISTFILE.
POP
```

Note that if you close a preconnected unit without opening it, or if you open another file on a preconnected unit number, you *sever the preconnection* and cannot re-establish the preconnection while the program runs. The preconnections will be re-established for the next program you run.

These two examples are impossible under MP/AOS and MP/AOS-SU.

You may delete, change, or add to the units in Table 5-3. See the explanation of the /PRECONNECTIONS switch under the description of macro F77LINK's switches in Chapter 9.

Using the Line Printer

You can print any data-sensitive file via the CLI QPRINT command.

To print records and files from F77 programs, you can open the line printer by using its queue name, @LPT for example; then write to it just as to any other file. When you open @LPT, the operating system creates a data-sensitive disk file with a unique name and, for every write, sends the records involved to the disk file. When you close the unit on which you opened @LPT (or when the program terminates, closing all files), the system queues the disk file for printing. Then, after the system prints this file, it deletes the disk file. If you reopen the unit number with FILE='@LPT' before closing it, the system closes the existing disk file, queues it for printing, and creates another disk file. Thus there will be a separate disk file for each reopening with FILE='@LPT'; so, if you want output to this unit printed as one file, don't reopen this unit with FILE='@LPT'. (Omit FILE= if you reopen the unit.)

The way the records print depends on the file organization and carriage control used in the source file. As described earlier under CARRIAGECONTROL=, only data-sensitive files (with or without FORTRAN carriage control) will print properly. For other file organizations, run the F77PRINT postprocessor on the file before trying to print it — either from an F77 program or from the CLI.

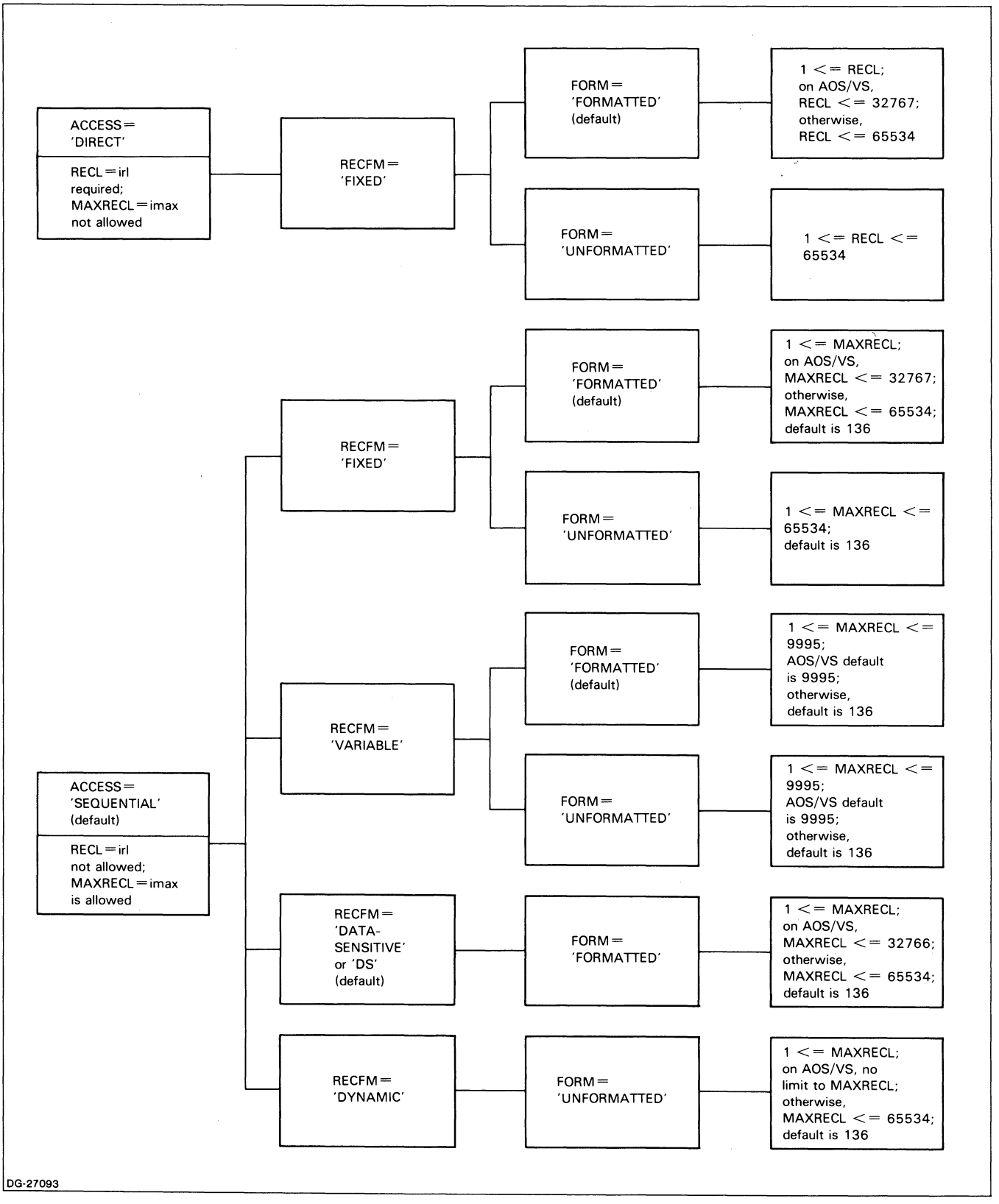


Figure 5-1. Relationships among OPEN Specifiers ACCESS, RECFM, FORM, RECL, and MAXRECL

Table 5-3. F77 Unit/File Preconnections

Unit	Filename	OPEN Properties	Comments
5	@INPUT	Only those under ALL below are relevant, except that SCREENEDIT='YES' for F77s other than MP/AOS and MP/AOS-SU. FORCE='YES'	For interactive use, @INPUT is the terminal keyboard. You can use an asterisk (*) instead of number 5 to specify it.
6	@OUTPUT	CARRIAGECONTROL='FORTRAN' POSITION='END' FORCE='YES'	For interactive use, @OUTPUT is the terminal screen or terminal printer. For batch use, @OUTPUT is the batch output file. POSITION relates only to batch and appends to the output file. You can use an asterisk instead of number 6 to specify this file.
9	@DATA	CARRIAGECONTROL='LIST' POSITION='START' FORCE='NO'	@DATA refers to your own datafile, a disk file you can create and set as shown below. MP/AOS and MP/AOS-SU F77 do not have this preconnection.
10	@OUTPUT	CARRIAGECONTROL='LIST' POSITION='END' FORCE='YES'	Same file as unit 6, but does not allow asterisk as unit specifier and has LIST carriage control.
11	@INPUT	Only those under ALL below are relevant, except that SCREENEDIT='YES' for F77s other than MP/AOS and MP/AOS-SU. FORCE='YES'	Same file as unit 5, but does not allow asterisk unit specifier.
12	@LIST	CARRIAGECONTROL='FORTRAN' POSITION='END' FORCE='NO'	@LIST refers to your listfile, which you can create and set as shown below. MP/AOS and MP/AOS-SU F77 do not have this preconnection.
ALL	ALL	ACCESS='SEQUENTIAL' BLANK='NULL' DELIMITER='EXCLUDE' EXCLUSIVE='NO' FORM='FORMATTED' IOINTENT='BOTH' MAXRECL=136 MODE='TEXT' PAD='YES' RECFM='DS' SCREENEDIT='NO' STATUS='UNKNOWN'	All preconnections have these properties, except MODE='TEXT' under MP/AOS and MP/AOS-SU.

OPEN Examples

Some examples of OPEN statements are

```
      OPEN (1)     ! Uses UNKNOWN status and file.

      OPEN (15, ERR=99, FILE=':UDD:AL:IDIR:IDATA')  ! Uses UNKNOWN status.

      open (0, status='SCRATCH')   ! SCRATCH status.

      open (2, recfm='DS', status='FRESH', carriagecontrol='FORTRAN',
     +         file= 'M_DATA')

      OPEN (3, FILE='@LPT')     ! Line printer queuename.

      open (4, access='DIRECT', recl= 80, iostat=IOS, file='D_RECS',
     +         status='NEW')
```

The next example copies records to a destination file.

```
      ...
      character*136 REC     ! Variable for current record.
      open (4, file='IN_FILE', status='OLD', pad='YES')
      ...
      open (22, file='OUT_FILE', status='FRESH')
      ...
      ICNT = 0
600   read (4, '(A)', returnrecl=ILEN, end=700)   REC

C     Write only those characters read to the file.

      write (22, '(A)')  REC (1:ILEN)
      ICNT = ICNT + 1
      go to 600
700   print *, ICNT, ' Records copied to OUT_FILE.'
      ...
```

The next example copies all records that begin with "B" to a new file.

```
      PARAMETER (INPUT= 15)     ! Unit 15 for input (source) file.
      CHARACTER*80 KEY          !  For current record.
      ...
      OPEN (INPUT, PAD='YES', FILE='NAMES', STATUS='OLD')

      OPEN (INPUT+1, FILE='BNAMES', STATUS='FRESH')

      LCNT =  0
10    READ (1, '(A80)', END=110)    KEY
      ...
      IF  (KEY(1:1) .EQ. 'B')   THEN   ! Here, could process the record...
                                       ! ... but simply write it.
            WRITE (2, '(A80)')    KEY
            LCNT = LCNT + 1
      END IF
      GO TO 10
110   PRINT *, LCNT, ' records beginning with B written to file BNAMES.'
      ...
```

The next example creates an error log file for program INVENTORY_100.

```
      OPEN (1, FILE='INVENTORY_100.ERR', ERRORLOG='YES',
     +         STATUS='FRESH')
```

Using INQUIRE

INQUIRE allows you to check the properties of files and units; you can then take appropriate action. The file or unit need not be connected or even exist.

There are two forms of INQUIRE: INQUIRE by unit or INQUIRE by file. The two are mutually exclusive. The forms of INQUIRE are

INQUIRE (*[UNIT=]* iu, other-inquire-properties)
INQUIRE (FILE= fin, other-inquire-properties)

where:

UNIT= iu — is a unit specifier, mandatory for an INQUIRE by unit.

FILE= fin — is a character expression that gives the filename, mandatory for an INQUIRE by file.

All INQUIRE property items are sketched in Table 5-4 and detailed in the next section. Most of them are optional. For most of them, you must reserve a character variable, array element, or substring to receive information; for others you must use a variable or array element of logical or integer type (as noted).

AOS/VS programmers may use a logical*1/byte entity as an INQUIRE property whenever character or integer entities are allowed. An exception is IOSTAT, since all I/O error codes exceed 127.

If a property/list item doesn't pertain (perhaps because there is no unit/file connection), the item's return variable becomes undefined.

When you inquire by unit, you may want to include OPENED= because other items may return undefined if the unit isn't open.

When you inquire by file, you may want to include EXIST= and/or OPENED= because some items may return undefined if the file doesn't exist or isn't open.

Character Entities for INQUIRE Items

You can use any character variable, array element, or substring to receive INQUIRE information. If you will want to pass one of these entities directly to another statement (e.g., READ), be sure to make the entity long enough to receive the whole value. If the entity is *longer* than the return value, F77 will left justify the value and will pad positions on the right with blanks, allowing you to pass the value directly to another statement.

But if the character entity is *shorter* than the return value, F77 assigns only the leftmost characters. This allows you to check characters from the left but not to pass the value to another statement. For example:

```
CHARACTER*10 ACC
CHARACTER*5 ACC1
...
OPEN (2, FILE='FOO', RECL=80)
...
INQUIRE (2, ACCESS= ACC)
INQUIRE (2, ACCESS= ACC1)
```

After the INQUIRE statements execute, ACC will contain DIRECT□□□□□ and ACC1 will contain DIREC.

INQUIRE Property Items

UNIT= iu
iu identifies the unit for an INQUIRE by unit. You can omit UNIT= if you place the number iu first in the list; if you put a unit specifier anywhere else, you must include UNIT=. Number iu must be an integer expression that evaluates to an integer from 0 through 255 for AOS/VS or from 0 through 63 for AOS, F7716, MP/AOS, and MP/AOS-SU.

FILE= fin
fin is a character expression that gives the filename or the pathname of the file. If the file will not be in the directory where you will execute the program, you can specify a pathname through the directory tree to it; e.g., ':UDD:MYDIR:MYFILE'. Or, you can specify the file's directory in a search list, described in your system *CLI User's Manual*.

IOSTAT= ios
ios is an integer variable or array element. After the INQUIRE statement, it contains 0 if the INQUIRE executed normally or an F77 or system error code if INQUIRE encountered an error. These codes are positive integer values. Those that pertain directly to F77 are described in F77 disk file ERR.F77.IN.

ERR= s
ERR= directs control to the statement labelled *s* if an error occurs; *s* must be an executable statement in the current program unit. IOSTAT's variable, if you included IOSTAT=, will contain an error code.

ACCESS= acc
acc is a character variable, array element, or substring that will receive a value. The value will be SEQUENTIAL if the unit or file is connected for sequential access, DIRECT if it is connected for direct access. If the unit/file is not connected, *acc* will be undefined.

BLANK=blnk
blnk is a character variable, array element, or substring that will receive a value. The value will be NULL if the BLANK property is null for this connection, ZERO if the property is zero.

Table 5-4. INQUIRE Property Items

Property Item	Description
UNIT= iu	For INQUIRE by unit, use unit specifier for *iu*.
FILE= fin	For INQUIRE by file, use file or pathname for *fin*.
IOSTAT= ios	Integer variable *ios* returns status indicator.
ERR= s	Statement label *s* executes on error.
ACCESS= acc	*acc* returns string DIRECT or SEQUENTIAL.
BLANK= blnk	*blnk* returns string ZERO or NULL.
BLOCKSIZE= iblks	Integer variable *iblks* returns device block size or 0 for default. MP/AOS and MP/AOS-SU F77 do not support BLOCKSIZE.
CARRIAGECONTROL= cc	*cc* returns string NONE, FORTRAN, or LIST.
DELIMITER= del	*del* returns string INCLUDE or EXCLUDE.
DIRECT= dir	*dir* returns string YES, NO, or UNKNOWN.
ERRORLOG= erl	*erl* returns string YES or NO.
EXCLUSIVE= exc	*exc* returns string YES or NO.
EXIST= ex	Logical variable *ex* returns .TRUE. or .FALSE..
FORCE= for	*for* returns string YES or NO.
FORM= fm	*fm* returns string UNFORMATTED or FORMATTED.
FORMATTED= fmt	*fmt* returns string NO, YES, or UNKNOWN.
IOINTENT= oio	*oio* returns string OUTPUT, INPUT, or BOTH.
MAXRECL= imax	Integer variable *imax* returns maximum record length.
MODE= mod	*mod* returns string BINARY or TEXT. MP/AOS and MP/AOS-SU F77 do not support MODE.
NAME= nam	*nam* returns file's full pathname from root directory.
NAMED= nmd	Logical variable *nmd* returns .TRUE. or .FALSE..
NEXTREC= inr	Integer variable *inr* returns next record number.
NUMBER= inum	Integer variable *inum* returns unit number.
OPENED= od	Logical variable *od* returns .TRUE. or .FALSE..
PAD= pad	*pad* returns string YES or NO.
RECFM= rfm	*rfm* returns string DATASENSITIVE, FIXED, DYNAMIC, VARIABLE, or UNKNOWN.
RECL= irl	Integer variable *irl* returns fixed record length.
SCREENEDIT= sed	*sed* returns string NO, YES, or UNKNOWN.
SEQUENTIAL= seq	*seq* returns string NO, YES, or UNKNOWN.
TYPE= ityp	Integer variable *ityp* returns system file type.
UNFORMATTED= unf	*unf* returns string YES, NO, or UNKNOWN.

BLOCKSIZE= iblks
iblks is an integer variable or array element that will receive the integer value of the block size if the BLOCKSIZE specifier set it on the OPEN statement. If you selected the default block size, *iblks* will return 0; if there is no unit or file connection, *iblks* will be undefined.

MP/AOS and MP/AOS-SU F77 do not support BLOCKSIZE. If you specify it, the compilers display a warning message, generate no code from BLOCKSIZE, and continue.

CARRIAGECONTROL= cc
cc is a character variable, array element, or substring that will receive the carriage-control connection property: it will be LIST if the unit/file is connected for LIST carriage control (default), NONE if connected for NONE, and FORTRAN if connected for FORTRAN carriage control. This INQUIRE statement works only for formatted files connected for sequential access; otherwise *cc* will be undefined.

DELIMITER= del
del is a character variable, array element, or substring that will receive the value of the DELIMITER property: it will be EXCLUDE if the delimiter is excluded, or INCLUDE if otherwise. DELIMITER is relevant only for data-sensitive files that are open; otherwise *del* will be undefined.

DIRECT= dir
dir is a character variable, array element, or substring that will receive the value YES if direct access to the unit is allowed, NO if direct access isn't allowed, or UNKNOWN if F77 can't tell if it's allowed. If the unit is not connected or the file doesn't exist, *dir* will be undefined.

ERRORLOG= erl
erl is a character variable, array element, or substring that will receive the value YES if the unit/file will receive runtime error messages of any kind and NO if it will not. If the unit/file is not connected, *erl* will be undefined.

EXIST= ex
ex is a logical variable or array element that will receive a value. This value will be .TRUE. if inquire is by unit and the unit number exists, or if inquire is by file and the file exists. Otherwise *ex* will return .FALSE.. EXIST can help you decide what kind of OPEN statement to use.

EXCLUSIVE= exc
exc is a character variable, array element, or substring that will receive the value YES if the unit/file is connected for exclusive access and NO if it is not. If the unit/file is not connected, *exc* will be undefined.

FORCE= for
for is a character variable, array element, or substring that will receive the value YES if buffering is minimized or NO if buffering always occurs (to improve file I/O speed). If the unit is not connected or the file does not exist, *for* is undefined.

FORM= fm
fm is a character variable, array element, or substring that will receive the value FORMATTED if the unit/file is connected for formatted I/O or the value UNFORMATTED if the unit/file is connected for unformatted I/O. If the unit/file is not connected, *fm* will be undefined.

FORMATTED= fmt
fmt is a character variable, array element, or substring that returns the value YES if formatted I/O to the unit is allowed, NO if formatted I/O isn't allowed, and UNKNOWN if F77 can't tell if it's allowed. If the unit isn't connected or the file doesn't exist, *fmt* is undefined.

IOINTENT= oio
oio is a character variable, array element, or substring that will receive OUTPUT if the unit is connected for output only, INPUT if it is connected for input only, or BOTH if it is connected for both input and output. If the unit/file is not connected, *oio* is undefined.

MAXRECL= imax
imax is an integer variable or array element that will receive the integer maximum record length, as defaulted or set by MAXRECL on an OPEN statement. If the file does not exist or the unit/file is not connected, *imax* is undefined.

MP/AOS and MP/AOS-SU do not keep track of as much information about a file as the other operating systems do. If an MP/AOS or MP/AOS-SU F77 program inquires into an existing but unconnected file, then the returned value of *imax* is 0.

MODE= mod
mod is a character variable, array element, or substring that will receive the value BINARY if the unit or file is connected with MODE='BINARY' or TEXT if the unit or file is connected with MODE='TEXT'. If the unit/file is not connected, *mod* is undefined.

MP/AOS and MP/AOS-SU F77 do not support MODE. If you specify it, the compilers display a warning message, generate no code from MODE, and continue.

NAME= fn
fn is a character variable, array element, or substring that will receive a value. The value will be the full pathname (which includes the filename) of the file connected to this unit. (*fn* will also return the pathname if you inquire by file.)

A typical pathname might include the directory names for the root directory, UDD directory, and user control

point directory; e.g., :UDD:SALLY:filename. The pathname will include subordinate directory names if the file is in a subordinate directory. If the unit is not connected, is connected to a scratch file, or if the file doesn't exist, *fn* will be undefined.

NAMED= nmd
nmd is a logical variable or array element that will receive the value .TRUE. if the file has a name or .FALSE. if it does not have a name. The only files that do not have a name are scratch files and files that the program has opened and has deleted while open. So, if *nmd* is .FALSE. for a nonscratch file, you can assume a) that you have opened it and b) that a program has deleted it and that it will be physically deleted from the disk after you close it or after the program terminates. If the unit/file is not connected, *nmd* is undefined.

NEXTREC= inr
This specifier applies to direct-access connections only. *inr* is an integer variable or array element that will receive the number of the next record in the file. If the file is connected but no records have been read or written since the OPEN statement, *inr* will return the value 1. If there is no connection or the connection is sequential, *inr* will be undefined.

NUMBER= inum
inum is an integer variable or integer array element that returns the unit number connected to the file. You can then use *inum* for other I/O to the file. If the file/unit is not connected, *inum* is undefined.

OPENED= od
od is a logical variable or array element that will receive the value .TRUE. if the file is connected to a unit or if the unit is connected to a file. If the file/unit is not connected, *od* returns the value .FALSE..

PAD= pad
pad is a character variable, array element, or substring that returns YES if padding was specified on the OPEN statement, NO if it was not specified. If the records are not data sensitive or variable length, or if the file/unit is not connected, *pad* is undefined.

RECFM= rfm
rfm is a character variable, array element, or substring that will return a value. The value will be DATASENSITIVE, VARIABLE, FIXED, DYNAMIC, or UNKNOWN, depending on the record organization specified in RECFM= on the OPEN statement. This property may differ from the organization given when the file was created. If the unit isn't connected or the file doesn't exist, *rfm* will be undefined.

If you inquire by file, and the file exists but is not open, *rfm* will return the organization used when the file was created.

MP/AOS and MP/AOS-SU do not keep track of as much information about a file as the other operating systems do. If an MP/AOS or MP/AOS-SU F77 program inquires into an existing but unconnected file, then the returned value of *rfm* is UNKNOWN.

RECL= irl
Direct access connections only. *irl* is an integer variable or array element that will receive the byte length of each record in the file as given on the OPEN statement. If the unit is connected for sequential access, or the file does not exist, *irl* will be undefined.

MP/AOS and MP/AOS-SU do not keep track of as much information about a file as the other operating systems do. If an MP/AOS or MP/AOS-SU F77 program inquires into an existing but unconnected file, then the returned value of *irl* is 0.

SCREENEDIT=sed
sed is a character variable, array element, or substring that will receive the value YES if an on-line user may use SCREENEDIT control characters while preparing a string of ASCII characters as input to a READ statement, NO if these control characters aren't allowed, and UNKNOWN if F77 can't tell if it is allowed. If the unit/file isn't connected, *sed* will be undefined.

SEQUENTIAL= seq
seq is a character variable, array element, or substring that will receive the value YES if sequential access to the unit is allowed, NO if sequential access isn't allowed, and UNKNOWN if F77 can't tell if it is allowed. If the unit/file isn't connected, *seq* will be undefined.

MP/AOS and MP/AOS-SU do not keep track of as much information about a file as the other operating systems do. If an MP/AOS or MP/AOS-SU F77 program inquires into an existing but unconnected file, then the returned value of *seq* is UNKNOWN.

TYPE= ityp
ityp is an integer variable or array element that will receive the operating system file type, given or selected by default on the OPEN statement, as described in your operating system reference manual.

UNFORMATTED= unf
unf is a character variable, array element, or substring that will receive the value YES if unformatted I/O to the unit is allowed, NO if unformatted I/O isn't allowed, and UNKNOWN if F77 can't tell if it is allowed. If the unit/file isn't connected, *unf* will be undefined.

INQUIRE Examples

Some examples of INQUIRE statements are

```
C   Example 1.
        LOGICAL OD, EX
        ...
        INQUIRE (FILE='FOO', OPENED= OD, EXIST= EX, NUMBER= IU)

C   Example 2.
        CHARACTER*30 FN
        CHARACTER*3 EXC
        LOGICAL OD
        ...
        INQUIRE (3, NAME=FN, EXCLUSIVE=EXC, OPENED=OD)
        ...
```

A contextual example with INQUIRE by file is

```
C   Check RECFM and other properties, then report. For variable record
C       files, create printable output file.

        character*136 C_REC   ! character record
        character*15 RFM
        logical OD
        ...
        IU = 1
        ...
        inquire (file='MYFILE', opened=OD, recfm=RFM)
        if ( .not. OD ) go to 90
        if ( RFM  .eq. 'VARIABLE' ) then
            open (IU+1, file='P_FILE', recfm = 'DATASENSITIVE', pad= 'YES')
20          read (IU, '(A)', end= 65) C_REC
            write (IU+1, '(A)')       C_REC
            go to 20
65          print *, 'MYFILE has variable records, printable output'
            print *, '   file named P_FILE created from it.'
        else if ( RFM .eq. 'DATASENSITIVE') then
            print *, 'MYFILE has data-sensitive records.'
        else
            print *, 'MYFILE has FIXED or DYNAMIC records.'
        end if
        go to 95
   90   print *, 'File MYFILE not open, take appropriate action.'
        ...
   95     continue
        ...
```

An example with an INQUIRE by unit is

```
C   See if unit that will report errors is OPEN.
        LOGICAL OD
        ...
        INQUIRE (IU, OPENED=OD)
        IF (OD) THEN
            WRITE (IU, '..text of error message...')
          ELSE
            PRINT *, 'Error unit not OPEN.'
        END IF
        ...
```

Using CLOSE

CLOSE severs the connection between a unit and file, regardless of the number of times you've reopened this unit and file. The form of CLOSE is

CLOSE (*[UNIT=]* iu *[,IOSTAT=ios] [,ERR=s] [,STATUS=sta]*)

where these specifiers can occur in any order. They are

UNIT= iu	iu is an expression that evaluates to the integer number of the unit you want to close. You can omit UNIT= if the expression is the first argument.
IOSTAT= ios	*ios* is an integer variable that returns a status indicator: 0 for normal, an error code if an error occurred. If you include IOSTAT= without ERR=, program execution continues at the next statement on an error.
ERR= s	*s* is a statement label to which control will go on an error.
STATUS= sta	*sta* is a character expression that, after F77 has ignored any trailing blanks, is 'KEEP' or 'DELETE'. If 'KEEP' (the default), the file will continue to exist after a CLOSE. (But specifying 'KEEP' for a file you opened with STATUS='SCRATCH' will cause an error.) 'KEEP' will not create a file that does not exist. 'DELETE' will delete the file if it exists.

You can close a unit from any program unit. F77 closes all units automatically when the main program ends. Before F77 closes a unit, it flushes any buffers that contain data for a READ statement or data from an output statement. Thus, an output file receives all the data directed to it by WRITE or PRINT statements. Unless you specify STATUS='DELETE', F77 keeps all files except scratch and unnamed files when a CLOSE statement executes. If you close a preconnected unit without opening it, *the preconnection will be severed while the program runs*; for the next program you run, the preconnection will be re-established. For example, you may have a program with many WRITE(6,...) statements where unit 6 is the printer with traditional carriage control. Place the following among the first executable statements.

```
CLOSE(6)
OPEN (6, FILE='@LPT', CARRIAGECONTROL='FORTRAN')
```

CLOSE Examples

```
CLOSE (14)

CLOSE (22, IOSTAT=IER, STATUS = 'DELETE')
```

Another example, that uses REWIND (described in the next section) is

```
    ...
    logical OD, NMD
    character*80 CHZ      ! Buffer.
    character*15 RFM
    ...
C   Program assigns some unit value to IU.
    ...
    inquire (IU, named= NMD, opened= OD,
   +            recfm= RFM)
    if ( (OD) .and. (.not. NMD) ) then
      open (IU+1, file= 'SAVE_IT', recfm= RFM,
   +             status='FRESH')
      rewind IU
      do while (.true.)   ! forever ...
        read (IU, fmt='(A80)', end= 50)  CHZ
        write (IU+1, fmt='(A80)')        CHZ
      end do
50    close (IU+1)

    else if (OD) then
      close (IU)
    end if
    ...
```

In this example, the program wants to retain the file open on unit IU. It checks the status of the file on IU; if this file is open (OD) and not named (.not.NMD), it is either a scratch file or another program has deleted it; in either case it would be deleted from the disk on the CLOSE. So the program opens a new file, SAVE_IT, and copies the file open on unit IU to it. If the file on IU is named, it will not be deleted on the CLOSE so the example simply closes it.

File Positioning with BACKSPACE, ENDFILE, and REWIND

The statements described in this section allow you to alter position in *external* files connected for sequential access. These statements are illegal with internal files and files connected for direct access. They are meaningless for terminals. They are

BACKSPACE	Backspaces one record.
ENDFILE	Writes an endfile record.
REWIND	Positions before a file's first record.

These statements are useful in situations where you want to reread the current record (BACKSPACE), reread or rewrite the file (REWIND), or truncate a file (ENDFILE).

Using BACKSPACE

BACKSPACE positions a unit's file before the current record, allowing you to reread or rewrite the record. The record can be formatted, unformatted, or an endfile record.

After you read an endfile record, attempting to read or write further will cause an error. But you *can* backspace over the endfile record and write to append to a file. Another way to append is to open the file with POSITION='END', then write.

The forms of BACKSPACE are

BACKSPACE iu

BACKSPACE (*[UNIT=]* iu *[,IOSTAT=ios] [,ERR= s]*)

where the specifiers in the second form can appear in any order. They are

UNIT= iu — iu is an expression that evaluates to the integer number of the unit you want to backspace. You can omit UNIT= if the unit specifier is the first argument.

IOSTAT= ios — *ios* is an integer variable or array element that returns a status indicator: 0 for normal, or an error code if an error occurred. If you include IOSTAT= without ERR=, execution will continue at the next statement on an error.

ERR= s — *s* is a statement label to which control will go on an error. This must be an executable statement within the current program unit. If you omit both IOSTAT= and ERR=, the program will terminate on an error.

BACKSPACE works only for files connected for sequential access. It can backspace over the endfile record of any file opened for sequential access.

BACKSPACE works on intermediate records (between the start and end positions) with any organization except dynamic. However, backspacing a data-sensitive file takes a lot of time. So, if you want to use BACKSPACE, and care about speed, use it with a variable or fixed file. Also, for BACKSPACE to work on a data-sensitive file, the file must have been opened with LIST carriage control.

Backspacing from the first (start position) record or on a device that cannot position a file has no effect.

BACKSPACE Example

```
      character*40 REC
      open (1, file='MYDATA',    status='OLD',
     +        pad='YES')
      open (2, file='MORE_DATA', status='OLD',
     +        pad='YES')
      ...

5     read (1, '(A)', END= 6)  REC
C     Process record here, display results.
      ...
      go to 5

6     backspace 1            ! Back over endfile.
7     read (2, '(A)', END= 9) REC ! From 2nd file.
C     Process record.
      write (1, '(A)')   REC ! Append to 1st file.
      go to 7
9     continue
C     display end-of-program messages.

      stop
      end
```

The example above uses END= with a READ statement to transfer control to BACKSPACE on an end of file. The next WRITE statement then overwrites the endfile record with a record from another file.

Using ENDFILE

ENDFILE writes an end-of-file (endfile) record as the file's next record and positions the file *after* the end of file. ENDFILE is not required to establish an end of file, but does allow you to truncate an existing file. The form is

ENDFILE iu

ENDFILE (*[UNIT=]* iu *[,IOSTAT=ios] [,ERR=s]*)

where the specifiers in the second form can appear in any order. They are

UNIT= iu — iu is an expression that evaluates to the integer number of the unit to which you want to write an endfile record. You can omit UNIT= if the unit specifier is the first argument.

IOSTAT= ios — *ios* is an integer variable or array element that returns a status indicator: 0 for normal, or an error code if an error occurred. If you include IOSTAT= without ERR=, execution will continue at the next statement on an error.

ERR= s — *s* is a statement label to which control will go on an error. This must be an executable statement in the current program unit. If you omit both IOSTAT= and ERR=, the program will terminate on an error.

As you write records to a file, the end of file is implicitly just after the last record written. After ENDFILE writes an endfile record to the file, you cannot read or write the file's records unless you use the BACKSPACE or REWIND statement.

The operating system supplies an endfile record when you close a disk or tape file. But ENDFILE writes an endfile record while the file is open.

ENDFILE is designed for files connected for sequential access; it will return an error on a file connected for direct access.

You cannot write an ENDFILE record if the file is open on another unit *or* if another process has the file open. If either condition exists, F77 will return an error from the ENDFILE statement.

Furthermore:

- When ENDFILE executes, any records past the record that ENDFILE writes are truncated. Technically, the system call ?TRUNCATE executes at runtime in response to an ENDFILE statement. This system call actually shortens the file so that ENDFILE's effect persists beyond the life of the file's connection during the program's execution.
- When ENDFILE executes, normally your program must execute BACKSPACE or REWIND statements to read or write the file's records. The exception to this rule occurs with a terminal (a file with type CON). This exception allows simple conversational I/O between you and your program to continue.

ENDFILE Example

This example opens a file and attempts to read 4000 records. If the file has more than 4000 records, it truncates the file by writing an endfile record after the 4000th record.

```
      OPEN (2, 'FILE=XDAT', STATUS='OLD', PAD='YES')
      CHARACTER*80 RECORD
      ...
      DO 87 I = 1, 4000
         READ (2, '(A)', END = 90) RECORD
87    CONTINUE
      ENDFILE 2      ! Write endfile record.
90    CONTINUE       ! Proceed.
```

Using REWIND

REWIND positions a connected unit to the file's start point, before its first record. It applies to all files that are connected for sequential access and that allow positioning. This includes disk files but not devices like the terminal. The forms of REWIND are

REWIND iu

REWIND (*[UNIT=]* iu *[,IOSTAT=ios] [,ERR=s]*)

where the specifiers in the second form can appear in any order. They are

UNIT= iu — iu is an expression that evaluates to the integer number of the unit you want to REWIND. You can omit UNIT= if the unit specifier is the first argument.

IOSTAT= ios — *ios* is an integer variable or array element that returns a status indicator: 0 for normal, or an error code if an error occurred. If you include IOSTAT= without ERR=, execution will continue at the next statement on an error.

ERR= s — *s* is a statement label to which control will go on an error. This must be an executable statement within the current program unit. If you omit both IOSTAT= and ERR=, the program will terminate on an error.

REWIND positions a connected or preconnected unit at the file's start point; it works only with files connected for sequential access. If the file is already positioned at its start point, REWIND has no effect. If the unit is not connected, F77 signals an error.

REWIND Example

```
    ...
    OPEN (IUNIT, FILE='DFILE', STATUS ='OLD')
    READ (IUNIT,..., END = 98)   X, Y, Z
    ...
    ...
98  REWIND (IUNIT, IOSTAT=IER)
    ...
```

Terminal I/O

As shown in Table 5-3 earlier, units 5 and 11 are preconnected to file @INPUT. Units 6 and 10 are preconnected to file @OUTPUT.

Instead of unit 5, you can use an asterisk specifier on reads; instead of unit 6, you can use an asterisk specifier on writes.

List-directed formatting (detailed later in Chapter 6) is well suited to terminal I/O. It frees you from concern with format details like edit descriptors, field width, and decimal point position.

Preconnections to units 5 and 6 and list-directed formatting allow you to do all terminal I/O with these simple forms:

```
READ *, iolist
    or
READ (*,*) iolist
```

```
WRITE (*, *) iolist
    or
PRINT *, iolist
```

Units 5, 11, 6, and 10 are opened as data-sensitive files. Thus, on READs from unit 5 (or *) or 11, you can enter values easily and terminate input with a ↵ (NEW LINE).

Unit 6 (*) has FORTRAN carriage control, which means that F77 uses the first character in each record for carriage control. However, with list-directed formatting you can ignore this, because F77 inserts a blank at the beginning of each record before outputting it — and this blank serves for the carriage-control character.

Unit 10 has LIST carriage control. This means F77 will not use the first character for carriage control and will output a NEW LINE after each record.

For an example of terminal I/O, assume the program statements:

```
C   Get integer values for 3 entities from the
C      terminal, average them, and display
C      the average:

    PRINT *, 'Type 3 integer values to average. '
    READ *, J, K, L
    IAVG = (J+K+L)/3
    PRINT *, 'Average is ', IAVG
    END
```

When this executes, the interaction at the terminal could be

(blank line)
Type 3 integer values to average. 3, 4, 5 ↵
(blank line)
Average is 4

As you can see, the preconnections to units 5 and 6, in conjunction with list-directed formatting, allow you to type values on the same line as the text that prompts for them.

The other preconnections, to units 10 and 11, have LIST carriage control. Using these units, the program code above would be

```
C  Get integer values for 3 entities from the
C     terminal, average them, and display
C     the average:

   WRITE(10,*) 'Type 3 integer values to average.'
   READ (11,*) J, K, L
   IAVG = (J+K+L)/3
   WRITE (10, *) 'Average is ', IAVG
   END
```

During execution, the interaction at the terminal could be

Type 3 integer values to average.
3,4,5 ↵
Average is 4

In summary, I/O to the terminal is quite simple — because you need no OPEN statement or cilist and you can use list-directed formatting.

Summary of Rules for I/O

The rules for I/O, as described in this chapter, are

1. If a file is preconnected (units 5, 6, 9, 10, 11, 12), you need not open it. If it is an internal file, you *must not* open it. To gain access to any other file, you must open the file on a valid unit number, then use the unit number for other I/O to and from the file. Valid unit numbers are 0 through 255 for AOS/VS or from 0 through 63 for AOS, F7716, MP/AOS, and MP/AOS-SU. MP/AOS and MP/AOS-SU F77 programs do not have preconnected units 9 and 12.
2. Records in a file can be formatted or unformatted. For formatted records, use formatted WRITE and READ statements. For unformatted records, use WRITE and READ without a format statement.
3. F77 always writes or reads at least one record to or from a unit, using the input/output list (iolist) entities. On READ statements, it reads values from the record into the iolist entities; on WRITE statements, it writes values from the iolist entities to the record. If the read or write is formatted, F77 reads or writes entities into or from the iolist entities via the format specified.
4. If you specify a format, it directly affects the value read into or written from the iolist entities. The data types of characters transferred from the record must match the format and the iolist data type.

   You specify a format via a format identifier in the I/O statement. The format identifier can specify the format via a character expression; or it can be the label of a FORMAT statement that specifies the format; it can be an asterisk (*) to specify list-directed formatting. List-directed formatting is useful for terminal I/O.
5. An input/output list (iolist) contains the entities that receive values on input and deliver values on output. An iolist entity can be a variable, array element, or character substring; it can also be an array name, to specify all array elements in traditional column/row order; or it can be an implied DO list. On output, it can also be an expression.
6. A READ or WRITE statement that deals with any unit other than 5 or 6 must include a control information list (cilist). The cilist must always give the unit number or internal filename. For formatted I/O, the cilist must also include a format identifier. The cilist may contain other specifiers, like REC= for direct access and IOSTAT= to return status.
7. The OPEN statement establishes the unit/file connection and certain connection properties, including the filename and file status, sequential or direct access, formatted or unformatted access, blank interpretation, and carriage-control interpretation. You can reopen a unit to change some of its connection properties.

   You can print any file created and/or opened with data-sensitive organization (default for formatted records). For FORTRAN vertical carriage control, you can specify CARRIAGECONTROL='FORTRAN'; or you can default carriage control to LIST.

   If a file is not data sensitive and was written with LIST or FORTRAN carriage control, the F77PRINT postprocessor can copy it into a printable output file.
8. The INQUIRE statement returns information about a file or unit's properties.
9. File position changes as you write or read records in the file. The BACKSPACE, ENDFILE, and REWIND statements allow you to reread/rewrite records, truncate files, or rewrite/reread files.
10. If your program terminates with an unexpected "Attempt to read past end of record" error for formatted records in a file connected for sequential access, try adding the specifier "pad='YES'" to the OPEN statement.
11. The end-of-file sequence from @INPUT is CTRL-D.
12. Lastly, to thoroughly understand I/O, you need to understand format specification — the way F77 really reads and writes formatted records. This is described in the next chapter.

Restrictions on I/O

1. The ANSI standard says that an I/O statement must not contain a function reference if such a reference causes execution of an I/O statement. For example, the standard does not allow the PRINT statement in the following *main* program:

```
PROGRAM FOO          ! Main program
...
JJ = 40
XX = FUNCT(JJ)       ! Function reference
PRINT *, 'According to FUNCT: ', FUNCT(JJ)
...
END

FUNCTION FUNCT(JX)   ! Function
PRINT *, 'Value of JX is ', JX
...
END
```

   As an extension, DG's AOS/VS F77 (but not AOS, F7716, MP/AOS, or MP/AOS-SU F77) *does allow* an I/O statement with a function reference to cause execution of another I/O statement — *if* the second I/O statement does not refer to the same unit as the first statement. For example, if the output statement in FUNCT(JX) above were

```
WRITE (10, *) 'Value of JX is ', JX
```

   the function reference would be acceptable and would execute without error.

2. If a unit does not have the properties needed for execution of an I/O statement, the I/O statement must not refer to the unit; for example, a READ statement issued to the line printer is illegal.

3. Within a subprogram (Chapter 7), an assumed-size array cannot appear in an iolist. For example, the following is illegal:

```
SUBROUTINE FOO (ARRAY2)
DIMENSION ARRAY2(*)
PRINT *, ARRAY2
```

4. On your first write to a unit, or when you write to a unit after reading from it, the pertinent file cannot be open on any other unit or by another process. Thus, if you plan to write to a file, you should not open it on a second unit, and, if another process might open the file, you should open it with EXCLUSIVE='YES' for writes. These restrictions also apply to ENDFILE writes.

End of Chapter

Chapter 6
Format Specification (FMT= and FORMAT)

A format specification describes the way F77 will encounter data in a record during a READ statement or will place it in a record during a WRITE statement. It can specify the kinds of characters, number of characters, position of decimal point, record position, and/or other things. In other words, "format" means "layout" or "organization"; a FORMAT or FMT= statement specifies the layout or organization of a record coming from or going to a file.

This chapter builds on the I/O information given in Chapter 5 and details each aspect and *edit descriptor* that can be part of a format specification. The major sections proceed:

- Edit-Directed Formatting Overview
- Edit-Directed Format Specifications
- Number-Oriented Editing (I,O,Z,B,F,E,D,G,P,BZ,BN,S)
- Logical Editing (L)
- Character Editing (A, ', ", H)
- Positional Editing (X, T, $)
- Colon Editing (:)
- Carriage Control
- Descriptor Separators
- Repeat Count and Nested Format Specifications
- Multiple Record Formatting (/)
- Format Expressions
- Format/Entity Mismatch
- List-Directed Formatting
- Summary of Rules for Formatted I/O

As described in Chapter 5, there are three kinds of format identifier f that you can use in an I/O statement. They are

1. A character expression that specifies the format. This can be a character constant as in

```
WRITE ( 58, '(1X, F9.2, 2X, F9.2)' ) A, B
```

   or the name of a character variable or array element that contains the format.

2. The label of a FORMAT statement that contains the format; e.g.,

```
    WRITE (2, 100)  B1, B2
100 FORMAT (1X, F9.2, 2X, F9.2)
```

   or an integer variable assigned the label of a FORMAT statement; e.g.,

```
ASSIGN 100 TO J
```

   followed by

```
WRITE (2, J) B1, B2
```

3. An asterisk (*) to specify list-directed formatting; e.g.,

```
WRITE (2, FMT= *) B1, B2
```

   or simply

```
WRITE (2, *) B1, B2
```

You can use any of these specifications to input (READ) or output (WRITE, PRINT) records.

The first kind of specification — character expression or entity name — yields traditional format handling but its syntax is new with FORTRAN 77. The second kind, the FORMAT statement, is traditional. Each specifies *edit-directed* formatting.

List-directed formatting — the third kind — is new to FORTRAN 77. An I/O statement that uses a list-directed format writes and reads records according to iolist data types. This kind of format is well suited to terminal I/O but has some limitations. List-directed formatting is described near the end of this chapter because it involves some of the concepts of edit-directed formatting. (It corresponds to DG FORTRAN 5's READ FREE and WRITE FREE.)

Edit-Directed Formatting Overview

An edit-directed format, applied with either a character expression or FORMAT statement, determines the length and kind of conversion for values that will be read into or written from the iolist entities in a READ, WRITE, or PRINT statement.

Conversion means, for a formatted READ statement, the translation of a string of ASCII characters to the internal representation of an iolist entity. Conversion means, for a formatted WRITE or PRINT statement, the translation of the internal representation of an iolist entity to a string of ASCII characters. For example, a formatted READ statement can translate the string of characters "083" to the integer*4 number

00000000000000000000000001010011

(in binary notation). For another example, a formatted WRITE or PRINT statement can translate the integer*2 number

0000000000101010

to the string (among others) of ASCII characters "□□□42". Such conversions, while necessary when people must use the data, require a fair amount of arithmetic. Thus, try to use unformatted I/O statements whenever possible; for instance, when transferring data from one disk file to another and people don't have to use the data immediately.

Edit descriptors within the format control data transfer. The ANSI standard divides edit descriptors into two broad categories: repeatable and nonrepeatable. A *repeatable* descriptor is one that actually transfers data to and from the record. The repeatable descriptors are I, O, Z, B, F, E, D, G, L, and A. *Nonrepeatable* edit descriptors control interpretation of blanks, change position, output character strings, and start new records. The nonrepeatable descriptors are P, S, SP, SS, BZ, BN, apostrophe/quotation mark, H, X, T, TL, TR, $, colon, and slash.

When F77 executes an I/O statement with an edit-directed format, it scans both the format specification and iolist. Then it assigns values to or writes values from each iolist entity according to its corresponding repeatable edit descriptor. Unless the iolist entity is type complex, F77 uses one repeatable descriptor for each entity.

Input

When F77 executes a formatted READ statement, it reads the *entire current record*. Then it takes the characters in the record, groups them according to the repeatable edit descriptors, and extracts the values of the iolist entities from the groups of chararacters. The READ acts as an assignment statement — assigning values to the iolist entities.

Format control terminates for this record, and F77 positions the file at the next record, when any of the following conditions occurs

- all iolist entities have had values assigned to them and no *nonrepeatable* descriptors remain in the format. (F77 processes certain nonrepeatable descriptors even though it has assigned values to all iolist entities.)
- F77 encounters the rightmost parenthesis of the format specification.
- F77 encounters a slash (described later) in the format specification.

For a comprehensive example of all this, take the repeatable descriptors I (Integer editing), F (Floating-point editing), and A (Alphanumeric editing) and assume two things:

- that the current record consists of the eight characters 12345678 (all formatted records consist of ASCII characters).
- that independent READ statements read the record using the following edit descriptors.

Characters Read	Format	Iolist Entities	Effect of Statement
1. 1	I1	J	J = 1
2. 123456	I6	J	J = 123456
3. 1234567	I2, I3, I2	J, J1, J2	J = 12, J1 = 345, J2 = 67
4. 1234567	I2, I3, I2	J, J1	J = 12, J1 = 345
5. 12345678	F5.2, I3	BD, J	BD = 123.45, J = 678
6. 1234567	F7.2	BD	BD = 12345.67
7. 12345	A5	CH (*5)	CH = '12345'

In example 1, F77 reads the whole record, formats 1 character according to I1, and assigns the value to J. Because the format I implies type integer, the variable J must be type integer as would normally be true by the name rule. If J were a different type, like character, F77 would signal an error.

In example 2, F77 reads the whole record, formats the first 6 characters according to I6 and assigns them to J.

In example 3, F77 reads the entire record, formats the first 2 characters according to I2 and assigns them to J, formats the second 3 according to I3 and assigns them to J1, and formats the next 2 according to I2 and assigns them to J2.

In example 4, F77 reads the record and assigns values to J and J1 as in 3, but since J1 is the last iolist entity, it stops formatting and the I/O statement terminates. The last edit descriptor — I2 — goes unused.

In example 5, F77 reads the entire record, formats the first 5 characters according to F5.2, producing 123.45, and assigns this value to BD. BD is a double-precision entity. The decimal point in the format doesn't count as a character. (If a decimal point were *in the record*, it *would* count as a character.) Then F77 formats the next 3 characters according to I3 and assigns them to J.

In example 6, the format F7.2 specifies the first seven characters in the record. F77 formats the characters and assigns them to BD.

In example 7, F77 formats 5 characters according to A editing rules and assigns them to CH. The apostrophes are *not* inserted in CH; we include them to distinguish data of type character. The example assumes that CH was declared type character with 5 characters (CHARACTER*5 CH). Input list items that are processed with the A edit descriptor may be of any data type. This is one way that noncharacter entities can be defined with character data. Characters assigned with the A descriptor cannot be used directly in computations.

Note that although F77 need not process extra repeatable edit descriptors, it *must* process all entities in the iolist. If the number of iolist entities exceeds the number of repeatable edit descriptors, F77 will read the next record and reuse the format. This is detailed later under "Multiple Record Formatting".

Formatting records on input can be tricky. For example, if the numeric record read is too short (if its terminator appears before the format given is filled), F77 will signal an "Attempt to read past end of record" error unless you opened the unit with PAD='YES'. In the examples above, any format that specified more than 8 characters (e.g., I9 or F9.2) would cause this error unless the unit had been opened with PAD='YES'.

If the record read is *too long* for descriptors in the specification, F77 will read in characters from the left until it fills the iolist entities; remaining values in the record will be unused.

Blank Characters in Input Values

F77 always ignores *leading* blanks in input values. By default, it ignores blanks *within* values, treating them as nulls. For example, the value:

2□□1 is treated as □□21

But if you opened the file with BLANK='ZERO', F77 will treat embedded blanks as 0s. For example:

2□□1 is treated as 2001

You can also control interpretation of blanks with the BZ/BN edit descriptors, explained later.

Output

When it executes an output statement, F77 uses the values of the iolist entities to create character strings according to the edit descriptors, and writes the character strings *as one record* to the specified unit. As with input, F77 *must* process all iolist entities. Format control terminates, and F77 positions the file to its next record, when no more iolist entities remain to be formatted *and* one of the following conditions occurs:

- F77 encounters a colon or repeatable edit descriptor in the format specification; or
- F77 encounters the rightmost parenthesis in the format specification.

When you print a file, the first character in each record may be used for printer carriage control, not printed, as described in the next section.

For a comprehensive output example, assume a WRITE statement with LIST carriage control and the following formats and iolist entity values:

Example/ Format	Value(s) in Iolist Entity(ies)	Output Record Is
1. 1X, I1	12345678	□* *(format too small)*
2. 1X, I9	12345678	□□12345678
3. 1X, I8, I8	12345678, 12345678	□1234567812345678
4. 1X,F8.2,I8	1234567.8, 12345678	□********12345678
5. 1X, F10.1	1234567.8	□□1234567.8
6. 1X, I8, I8	12345678	□12345678
7. 1X, A5	'ABCDE'	□ABCDE

In example 1, the 1X produces the leading blank. Then F77 outputs 1 asterisk to fill the format *field* I1, because the field is too small to hold the iolist entity's value.

In example 2, 1X provides a leading blank and F77 pads the value with a blank to fill the field I9.

In example 3, 1X provides a leading blank, then F77 outputs the two values, using I8 for each value.

In example 4, 1X provides a leading blank. The first value has 9 characters (including the decimal point) but format F8.2 provides only 8 spaces, so F77 outputs 8 asterisks. The second format, I8, is large enough for the second value.

In example 5, the 1X provides the first leading blank. The second blank appears because the value has 9 characters and the format F10.1 specifies 10 characters.

In example 6, 1X produces the leading blank. There is only one value, so F77 uses only one I8 descriptor; it ignores the other descriptor.

In example 7, 1X produces the leading blank, then F77 outputs the five characters ABCDE. The apostrophes are not part of the output record because they are not actually part of the value.

On output, as on input, F77 can ignore extra repeatable edit descriptors but must process each iolist entity. If there aren't enough repeatable descriptors, F77 outputs a new record and continues writing, as described later under "Multiple Record Formatting".

Printed Records

To understand printed output, visualize a record as a blank line, beginning with column 1 and continuing sequentially through the last column. The last column can be numbered 80 or so for a screen or up to 132 or so for a line printer.

When you direct F77 to place data into a data-sensitive file that was opened with CARRIAGECONTROL='FORTRAN', F77 does not print the first character of each record. Instead, it uses this first character for vertical carriage control. With FORTRAN carriage control, F77 treats the first character of each record as follows (where <NL> is the NEW LINE character, <012>, <FF> is the form feed character, <014>, and <CR> is the carriage return character, <015>):

First Character in Record	Result when Record Is Printed
□	Prints record on next line (single spacing); <NL>record.
0	Prints record on line after next (double spacing); <NL><NL>record.
1	Prints record on top of next page; <FF>record.
+	Prints record starting at column 1 of the line (overprints); <CR>>record.
#	Prints record starting at the column position used for the last record; allows you to print multiple records on one line.
other character	Prints record on next line (single spacing); <NL>record.

You can insert any of these characters in the left column literally as the first record character, or use the X or a T-series (tab) descriptor to produce one or more blanks at the beginning of each record. Examples of both are

```
WRITE (2, '(" ", I8)') B
WRITE (2, '(1X , I8)') B
```

When you direct F77 to place data into a data-sensitive file that was opened with CARRIAGECONTROL='LIST' (the default carriage control), F77 does not interpret the first character; instead by default it places a NEW LINE character (<012>) after each record.

Preconnected units 6 and 10, @OUTPUT, each have RECFM='DATASENSITIVE'; unit 6 has FORTRAN carriage control; and unit 10 has LIST carriage control. This information is expanded, with examples, later in this chapter under "Carriage Control".

Arrays in iolists

If you include an unsubscripted array name in an iolist, F77 reads values for or writes values from all of its elements. The order it uses for elements is the traditional column/row order; e.g., for an array declared as C(2,3) the order is C(1,1), C(2,1), C(1,2), C(2,2), C(1,3), C(2,3).

In a subprogram (Chapter 7), you cannot use an assumed-size array in an iolist. The section "Array Arguments and Declarations" in Chapter 7 explains assumed-size arrays.

Edit-Directed Format Specifications

You can specify an edit-directed format via a character expression or FORMAT statement. The results may be the same, but the syntax of the two differs.

Character Expressions as Format Specifications

In a character expression, the format identifier can be a character constant that specifies the format or the name of a character entity that specifies the format. It can also be the name of a character array or a character array element.

To use a character constant as a format identifier/specifier, enclose the constant in either apostrophes or quotation marks and in parentheses. For example:

```
'(I8, F9.2)'
"(I8, F9.2)"
```

Within I/O statements, these constants might look like:

```
READ  (2,  '(I8, F9.2)')   IVAL, RVAL
WRITE (49, '(I8, F9.2)')   IVAL, RVAL
PRINT      '(I8, F9.2)',   IVAL, RVAL
```

To use a character entity (variable or array element), assign the constant to the entity and refer to the entity name in the I/O statement.

The entity must have at least as many characters as the format requires. For example:

```
CHARACTER*10   CX
CX = '(I8, F9.2)'
```

Within I/O statements, the character entity CX might look like:

```
READ  (2,  CX)   IVAL, RVAL
WRITE (49, CX)   IVAL, RVAL
PRINT      CX,   IVAL, RVAL
```

You can use a character array name as a format specification; if so, the first element need not be large enough to hold the entire specification. F77 will access the elements in order to get the specification. For example:

```
CHARACTER*4   CHF(3)
CHF(1) = '(1X,'
CHF(2) = 'F9.2'
CHF(3) = ')'
...
WRITE (*, CHF)   XX
```

Using the character constant allows you to do I/O in one statement. The character entity or array approach allows variable formats through character variables or array elements.

The following example with Hollerith constants is equivalent.

```
INTEGER*4 IHF(3)
IHF(1) = 4H(1X,
IHF(2) = 4HF9.2
IHF(3) = 1H)
...
WRITE (*, IHF)   XX
```

If you want a quote or apostrophe edit descriptor within the format itself (e.g., for a text message on output), you need to use extra quotation marks/apostrophes, following the rules of character delimiters; e.g.,

```
PRINT '('' Value is: '', I6)', IVAL
PRINT '( " Value is: " , I6)', IVAL
```

Nesting character delimiters this way can be confusing; it is one disadvantage of using character expressions as format specifications.

FORMAT Statements as Format Specifications

With the FORMAT approach use a separate FORMAT statement to specify the format. For example:

```
100 FORMAT (I8, F9.2)
```

The I/O statement identifies the format via the label of the pertinent FORMAT statement; e.g.,

```
    READ  (2,  100)   IF00, RF00
    WRITE (49, 100)   IF00, RF00
    PRINT      100,   IF00, RF00
100 FORMAT (I8, F9.2)
```

Using the separate FORMAT statement has its own advantages. It is familiar to experienced FORTRAN programmers, it may execute faster than the equivalent character expression, and it is somewhat clearer to read; e.g.,

```
    WRITE (2, 100)   IF00, RF00
100 FORMAT ('Values are: ', I8, F9.2)
```

Anatomy of the Format Specification

The format specification — however it is supplied — can contain the following items:

- a *repeat count* for one or more edit descriptors, nested in parentheses; e.g., 2(I8, F9.2) .
- one or more *edit descriptors*, separated by commas or other legal separators. Edit descriptors are the heart of format specifications. The repeatable edit descriptors each accept a repeat count; for example 2I8. If the I/O statement's iolist is not empty, the format specification *must* contain at least one repeatable edit descriptor.
- a slash, to tell F77 to read or write a new record, described later.

The iolist entities, which need not be present, are not part of the format specification although they are formatted by it.

Edit Descriptors

Edit descriptors in F77 do many different things. This chapter divides them functionally into categories: number-oriented, logical, character, positioning, colon (terminates format control conditionally), and multiple record.

All number-oriented descriptors are repeatable except those that control scale factor, plus sign, and blank interpretation. Logical and alphanumeric descriptors are also repeatable. *Note that every format specification whose I/O statement has one or more iolist entities must include at least one repeatable descriptor.*

Table 6-1 names and briefly describes all the F77 edit descriptors.

Table 6-1. F77 Edit Descriptors

Category	Descriptor	What It Does
Number-oriented	Iw*[.m]*	Formats a type integer iolist entity as Integer data, with a field width of w characters. The optional *.m* instructs F77 to pad with zeros to the minimum number of digits *m* on output. Repeatable.
	Ow*[.m]*	Formats a type integer unsigned iolist entity as Octal (base 8) data, with a field width of w characters. The optional *.m* works the same as for I. Repeatable.
	Zw*[.m]*	Formats a type integer unsigned iolist entity as Hexadecimal (base 16) data with a field width of w characters. The optional *.m* works the same way as for I. Repeatable.
	Bw*[.m]*	Formats a type integer unsigned iolist entity as Binary (base 2) data with a field width of w characters. The optional *.m* works the same way as for I. Repeatable.
	Fw.d	Formats a type real or type complex iolist entity as Floating-point (real) data with a field width of w characters, including d characters to the right of the decimal point. Repeatable.
	Ew.d*[Ee]* or Dw.d	Formats a type real or type complex iolist entity with Explicit exponent (real or double-precision), with a field width of w characters, including d characters to the right of decimal point. *Ee* gives a nondefault width for the exponent field. Repeatable.
	Gw.d*[Ee]*	Formats a type real or type complex iolist entity using Generalized format. Inputs real data as type F. Outputs real data as type F or E, depending on the magnitude of the iolist value. *Ee* is as for the E descriptor. Repeatable.
	kP	With F, E, D, or G, imposes *scale factor* to move decimal point k positions to the right or left.
	BZ, BN	With I, O, Z, B, F, E, D, or G, BZ interprets blanks as zeros; BN restores interpretation of blanks as nulls.
	S,SS,SP	With I, F, E, D, or G, S or SS outputs positive numbers with a leading blank for sign; SP outputs these values with a plus sign (+).
Logical	Lw	Formats a type logical iolist entity as Logical data, true or false, with a field width of w characters. Repeatable.
Character	A*[w]*	Formats any type of iolist entity as Alphanumeric (character) data. If you omit *w*, F77 uses the declared length of the iolist entity.
	'c...c' or "c...c"	Formats and outputs the literal character string enclosed in apostrophes or quotation marks.
	nHc...c	Formats and outputs the literal character string composed of the n characters that follow H.
Positioning	nX	Positions n columns to the right in the record.
	Tn	Positions at column n in the record.
	TLn	Positions n columns to the left in the record.
	TRn	Positions n columns to the right in the record.
	$ (dollars)	Suppresses NEW LINE after record output.
Colon	:	Ends format control if iolist entities are exhausted.
Multiple records	/ (slash)	Starts a new record.

File Positioning

Generally, a new I/O statement positions the file at the next record; a slash in a format specification *always* positions the file at the next record.

After F77 processes an I, O, Z, F, E, G, D, L, A, H or apostrophe/quote edit descriptor, the current file position is after the rightmost character read from or written to by that edit descriptor.

After F77 processes an X or T-series descriptor, the file is positioned left or right in the record, described under "Positioning Descriptors."

Field Width on Input

As mentioned earlier, F77 scans both the iolist and the format specification. It associates the repeatable edit descriptors with the iolist entities in the order that you wrote them.

On input, if a field width you give in the descriptor's w is less than the number of characters remaining in the record, F77 formats w characters. For example, if the characters 12345 remain in the current record, the edit descriptor is I2, and the iolist entity is J, F77 reads 12 (2 characters) into J and ignores 345.

If your field width w is greater than the number of characters F77 will read, F77 may report an error. For example, assume that F77 reads the characters 1234 using the edit descriptor I5.

1. If a record terminator follows the 4 in 1234, and the unit was opened with PAD='NO', F77 signals an "...end of record" runtime error. If PAD='YES' was specified, F77 pads the record with blanks to the maximum record length (MAXRECL). If the MAXRECL length satisfies the format (here, MAXRECL=5 or greater), F77 reads the padded value into the iolist entity. In the current case, if the iolist entity is JJ, F77 reads "1234□" and by default places 1234 into JJ. If the MAXRECL length does not satisfy the format (here, MAXRECL=4 or less), F77 signals an "...end of record" error. (File @INPUT is preconnected to units 5 and 11 with PAD='YES'. But the default on user OPENs is PAD='NO'.)
2. If a character immediately follows 4 in the record, F77 attempts to read the five characters into the iolist entity. But if the fifth character is an alphabetic character or special symbol, F77 signals an error.

All this means that the input descriptor must be appropriate and correct for the length and type of data read.

On output, the results depend on the edit descriptor you select. See the individual descriptor for an explanation.

Number-Oriented Editing (I, O, Z, B, F, E, D, G, kP, BZ, BN, S-Series)

Aside from control descriptors kP, S-series, and BZ/BN, all number-oriented descriptors are repeatable.

The I, O, Z, and B descriptors handle logical*1/byte data as integers in AOS/VS F77.

I (Integer) Editing

The I descriptor transmits integer data to or from an iolist entity in integer format. The iolist entity must be of type integer; if not, F77 signals an error. The form of the I descriptor is

[r] Iw *[.m]*

where:

- *r* is a repeat count to repeat the edit descriptor for more than one iolist entity.
- w is an unsigned positive integer constant that specifies the field's width in characters. w cannot exceed 255.
- *m* is an unsigned integer constant that gives the minimum number of digits for output. If the number of digits to be output (excluding sign) would be less than *m*, F77 will insert leading zeros to pad it to *m* characters. (But if *m* is 0 and the entity is 0, F77 will output blanks for the output field.) The constant *m* cannot be larger than width w. F77 ignores *m* on input.

Input to I

F77 reads w characters from the record in integer repesentation and stores the value in the iolist entity. A leading minus sign indicates a negative value; a leading plus sign or no sign indicates a positive value. A noninteger character, like a decimal point or letter, causes an error.

If the iolist entity is a standard (4-byte) integer, it can accept a range of values from -2,147,483,648 through 2,147,483,647. If the iolist entity is a short (2-byte) integer, it can accept only a value range from -32,768 through 32,767. If the iolist entity is logical*1/byte (AOS/VS only), it can accept only a value range from -128 through 127. If the value read is out of the iolist entity's range, F77 signals a runtime error.

Blanks count as characters but leading blanks have no effect on the value. Embedded blanks have no effect on the value unless you specified BLANK='ZERO' on the OPEN statement or via the BZ descriptor. (We cover this later in this section.) An input field containing all blanks has a value of 0.

Output from I

F77 formats the iolist entity and outputs it in a field w characters wide, right-justified, with leading blanks, if needed, to fill the field w. A minus sign precedes the integers if the value is negative, no sign precedes them if the value is positive (unless you specified plus via the SP descriptor, described in this section). The sign, if negative or specified plus, occupies one character in the output field. If you specified *m*, F77 inserts leading zeros to pad to *m* characters — but if *m* is 0 and the entity contains 0, F77 outputs a field of blanks.

If the field width w is too small for the iolist value, F77 outputs a field of asterisks (*).

I Examples

INPUT		
Edit Descriptor	**Characters Read**	**Value Stored in iolist Entity**
I2	50	50
I3	50□	50 (blank=null)
I5	-5□□3	-53 (blank=null)
I5	-5□□3	-5003 (blank= zero)
I4	□□□□	0
I3	□12	12
I3	+12	12
I3	-12	-12
I3	31A	*Error (illegal character)*
I10	3111111111	*Error (too big)*

OUTPUT		
Value in iolist Entity	**Edit Descriptor**	**Characters Written to Record**
50	I2	50
50	I3	□50
50	I3.3	050
-50	I2	** *(I2 too small)*
-50	I3	-50
-50	I5	□□-50
50	I6.4	□□0050
0	I4	□□□0
0	I4.4	0000
0	I4.3	□000
0	I4.0	□□□□
1234567890	I15	□□□□□1234567890

O (Octal Editing)

The O descriptor transmits integer data to or from an iolist entity in octal format. The iolist entity must be type integer; if not, F77 signals an error. The form of the O descriptor is

*[r]*Ow *[.m]*

where:

r is a repeat count.

w is an unsigned integer constant that specifies the field's width in characters; it cannot exceed 255.

m is an unsigned integer constant that gives the minimum number of digits for output. If the number of digits to be output would be less than *m*, F77 will insert leading zeros to pad it to *m* (but if *m* is 0 and the entity is 0, F77 will output blanks for the value). *m* cannot be larger than w. F77 ignores *.m* on input.

Input to O

For any octal field, F77 can store only the range of values that will fit into the iolist integer (standard 4 bytes, 2 bytes, or, on AOS/VS systems, logical*1/byte). If the value read is greater than 37777777777 (standard integer) or 177777 (2-byte integer) or 377 (logical*1/byte), or if it contains a nonoctal character, F77 signals an error. If the input value is signed, F77 also signals an error. Legal octal characters are 0 through 7 and blank.

Blanks count as characters but leading blanks have no value significance. Embedded blanks have no value significance unless you specified blanks as 0s on the OPEN statement or via the BZ descriptor. (We cover this later in this section.) An input value of all blanks has a value of 0.

Output from O

F77 converts the value to octal and outputs it as w octal digits, right-justified, with leading blanks if needed to fill field w. If you specified *.m*, F77 inserts leading 0s to pad to *m* — unless *m* is 0 and the entity contains 0, in which case F77 outputs a field of blanks.

If the field width w is too small for the iolist value, F77 outputs a field of asterisks (*).

Z (Hexadecimal) Editing

The Z descriptor transmits integer data to or from an iolist entity in hexadecimal format. The iolist entity must be of type integer; if not, F77 signals an error. The form of the Z descriptor is

*[r]*Zw *[.m]*

where:

r is a repeat count.

w is an unsigned integer constant that specifies the field's width in characters. It cannot exceed 255.

m is an unsigned integer constant that gives the minimum width in digits for output. If the number of digits to be output would be less than *m*, F77 will insert leading 0s to pad it to *m* (but if *m* is 0 and the entity is 0, F77 will output blanks for the value). *m* cannot be larger than w. F77 ignores *.m* on input.

Input to Z

For any hexadecimal field, F77 can store only the range of values that will fit into the iolist integer (standard 4 bytes, 2 bytes, or, on AOS/VS systems, logical*1/byte). If the value read is greater than FFFFFFFF (standard integer) or FFFF (short integer) or FF (logical*1/byte), or if it contains an illegal character, F77 signals an error. If the input value is signed, F77 also signals an error. Legal hexadecimal characters are 0 through 9, A, B, C, D E, F, a, b, c, d, e, f, and blank.

Blanks count as characters but leading blanks have no value significance. Embedded blanks have no value significance unless you specified blanks as 0s on the OPEN statement or via the BZ descriptor. (We cover this later in this section.) An input value of all blanks has a value of 0.

Output from Z

F77 converts the value to hexadecimal and outputs it as w hex digits, right-justified, with leading blanks if needed to fill the field. If you specified *.m*, F77 inserts leading 0s to pad to *m* — unless *m* is 0 and the entity contains 0, in which case F77 outputs a field of blanks.

If the field width w is too small for the iolist value, F77 outputs a field of asterisks (*).

O Examples

INPUT		
Edit Descriptor	**Characters Read**	**Value Stored in iolist Entity**
O2	34	28
O11	40000000000	*Error (value too big)*
O3	25□	21 (blank=null)
O2	61	49
O3	615	397
O6	177777	65535 *(integer*4)*
O6	177777	-1 *(integer*2)*
O6	000038	*Error (illegal character)*
O6	□□□-37	*Error (illegal character)*
O6	□□□+37	*Error (illegal character)*
O6	□□□□37	31

OUTPUT		
Value in iolist Entity	**Edit Descriptor**	**Characters Written to Record**
+33	O5	□□□41
+100	O2	**
+100	O3	144
-3 (=177775)	O6	177775 *(integer*2)*
9	O5.3	□□011
-433 (=177117)	O6	177117 *(integer*2)*
0	O4	□□□0
0	O4.4	0000
0	O4.0	□□□□

Z Examples

INPUT		
Edit Descriptor	**Characters Read**	**Value Stored in iolist Entity**
Z2	43	67
Z9	FFFFFFF1	*Error (value too big)*
Z3	10□	16 (blank=null)
Z2	1D	29
Z2	1d	29
Z3	1D2	466
Z4	-1D2	*Error, illegal sign*
Z4	+1D2	*Error, illegal sign*

OUTPUT		
Value in iolist Entity	**Edit Descriptor**	**Characters Written to Record**
128	Z5	□□□80
433	Z2	**
433	Z3	1B1
433	Z4.4	01B1
-433	Z5	□FE4F *(integer*2)*
0	Z4	□□□0
0	Z4.4	0000
0	Z4.0	□□□□

B (Binary) Editing

The B descriptor transmits integer data to or from an iolist entity in binary format. The iolist entity must be of type integer or, in AOS/VS, logical*1/byte; if not, F77 signals an error. The form of the B descriptor is

*[r]*Bw *[.m]*

where:

r is a repeat count.

w is an unsigned integer constant that specifies the field's width in characters. It cannot exceed 255.

m is an unsigned integer constant that gives the minimum width in digits for output. If the number of digits to be output would be less than *m*, F77 will insert leading 0s to pad it to *m* (but if *m* is 0 and the entity is 0, F77 will output blanks for the value). *m* cannot be larger than w. F77 ignores *m* on input.

Input to B

For any binary field, F77 can store only the range of values that will fit into the iolist integer (standard 4 bytes, 2 bytes, or logical*1/byte). If the value read is greater than 111...1 (32 1s; standard integer) or 111...1 (16 1s; 2-byte integer) or 11111111 (logical*1/byte in AOS/VS), or if it contains an illegal character, F77 signals an error. If the input value is signed, F77 also signals an error. Legal binary characters are 0, 1, and blank.

Blanks count as characters but leading blanks have no value significance. Embedded blanks have no value significance unless you specified blanks as 0s on the OPEN statement or via the BZ descriptor. (We cover this later in this section.) An input value of all blanks has a value of 0.

Output from B

F77 converts the value to binary and outputs it as w binary digits, right-justified, with leading blanks if needed to fill the field. If you specified *.m*, F77 inserts leading 0s to pad to *m* — unless *m* is 0 and the entity contains 0, in which case F77 outputs a field of blanks. If the value is negative and w exceeds the number of digits, it does *not* move the sign bit (high order bit) to the leftmost of the w positions. See the fifth and seventh OUTPUT examples below. They illustrate the logical*1/byte data type and apply only to AOS/VS F77.

If the field width w is too small for the iolist value, F77 outputs a field of asterisks (*).

B Examples

INPUT		
Edit Descriptor	**Characters Read**	**Value Stored in iolist Entity**
B2	10	2
B9	111111111	*Error (value too big)* for logical*1/byte; 511 otherwise
B3	10□	2 (blank=null)
B2	00	0
B5	□1101	13
B3	121	*Error, illegal character (2)*
B4	-111	*Error, illegal sign*
B4	+111	*Error, illegal sign*

OUTPUT		
Value in iolist Entity	**Edit Descriptor**	**Characters Written to Record**
12	B5	□1100
33	B2	**
33	B7	□100001
-1	B3	***
-1 (logical*1/byte)	B9	□11111111
-1 (integer*2,*4)	B9	*********
-128 (logical*1/byte)	B9	□10000000
-128 (integer*2,*4)	B9	*********
3	B4.4	0011
0	B4	□□□0
0	B4.4	0000
0	B4.0	□□□□

F (Floating-Point, Real) Editing

The F descriptor reads real data with or without an explicit exponent; it writes floating-point, real data without an explicit exponent. The form of the F descriptor is

*[kP] [r]*Fw.d

where:

k gives a scale factor to position the decimal point. *k* is an optionally signed integer that on input will move the decimal point to the left *k* positions and on output will move it to the right *k* positions. The scale factor, *kP*, is detailed later in this section.

r is a repeat count.

w is an unsigned integer constant that specifies the field's width in characters. w cannot exceed 255.

d is an unsigned integer constant that specifies the number of characters to the right of the decimal point.

Input to F

F77 handles input to the F, E, D, and G edit descriptors the same way.

The iolist entity must be type real, double precision (real*8) or complex (standard or complex*16); if it is any other type, F77 will signal an error. To format complex data, use two F, E, D, or G descriptors — one for the real part and one for the imaginary part.

F77 reads w characters from the record; these include the sign and the exponent (if any). If there is an exponent, F77 disregards any scale factor you specify. The exponent, if present, can be

- an E or D followed by a signed or unsigned integer constant; e.g., E2 or D2. F77 handles the D and E exponents the same way.
- a signed integer constant; e.g., +2.

The characters read can (but need not) include a decimal point. If they do, this overrides the decimal position given in .d. A decimal point, if present, requires one character position. If the characters read do not include a decimal point, F77 inserts one, placing the rightmost d digits to the right of the decimal point. Then F77 stores the entire value in the iolist entity.

Since w is a count of all characters in the field, it includes the sign (if any), exponent (if any), and the decimal point (if any).

For any real field, F77 can store accurately only the precision of approximately 6.7 digits (real*4) or 16.4 digits (double-precision/real*8). If the value read exceeds F77's storage *range* (approximately 5.4*10**-79 to 7.2*10**75), F77 will signal an error.

If a minus sign precedes the value, F77 interprets the value as negative; if a plus sign or no sign precedes the value, F77 interprets it as positive. If F77 reads an illegal character, it signals an error. Legal characters are 0 through 9, decimal point, plus, minus, E, D, and blank. If the record or remainder of the record contains less than w characters, and the file was not opened with PAD='YES', F77 signals an "End of record" runtime error.

Blanks count as characters but leading blanks have no value significance. Embedded blanks have no value significance unless you specified blanks as 0s on the OPEN statement or via the BZ descriptor. (We cover this later in this section.) An input value of all blanks has a value of 0.

Output from F

F77 right justifies w characters from the iolist entity, inserts leading blanks if needed, signs the value if negative or if you specified SP control, places the decimal point so that d digits follow it, and outputs this string of ASCII characters to the record. The decimal point and sign (if needed) each occupy one position in width w. If the number of digits following the decimal point in the iolist entity is greater than d, F77 rounds the fraction at the d position.

No sign precedes a positive value unless you specified plus via the SP descriptor, described in this section.

If the field width w is too small for the formatted iolist value, F77 outputs a field of asterisks (*).

F Examples

INPUT		
Edit Descriptor	**Characters Read**	**Value Stored in iolist Entity**
F7.2	-123.41	-123.41
F7.2	□ -12341	-123.41
F6.2	12345□	123.45
F6.5	012345	.12345
F6.5	123.45	123.45
F8.0	2.032E3□	+2032.
F8.0	-2.032-3	-.002032
F8.3	+6□3.□53	+63.53 (blank= null)
F8.3	+6□3.□53	+603.053 (blank= zero)
F5.2	6□3.□	+63.0 (blank= null)
F6.2	107.8A	Error, illegal char.

OUTPUT		
Value in iolist Entity	**Edit Descriptor**	**Characters Written to Record**
-987.	F7.2	-987.00
100.37	F8.1	□□□100.4 *(note rounding)*
123.54	F5.2	***** *(F5.2 too small)*
+42.35	F9.3	□□□42.350
+109967.1	F13.1	□□□□□109967.1
+109967.1	F14.2	□□□□□109967.13
+109967.1	F14.2	□□□□□109967.10 *(double precision)*
0.0	F5.2	□□.00
0.007	F8.3	□□□□.007
-0.007	F8.3	□□□-.007

E or D (Real, Explicit Exponent) Editing

The E or D descriptor reads real data with or without an explicit exponent; it writes data with an explicit exponent. The D descriptor is functionally identical to E in FORTRAN 77. If you use D, F77 will interpret or output it as E. The forms of these descriptors are

[kP] [r] Ew.d *[Ee]*

[kP] [r] Dw.d

where:

k is a scale factor that moves the decimal point to the left or right by *k* places, described later in this section.

r is a repeat count.

w is an unsigned integer constant that specifies the field's width in characters. w cannot exceed 255.

d is an unsigned integer constant that specifies the number of characters to the right of the decimal point.

e is an unsigned integer constant that specifies the number of digits for the exponent on output. The default, which is up to three digits, will suffice for any representable value. *Ee* is ignored on input.

Input to E or D

F77 handles input to the F, E, D, and G edit descriptors the same way.

The iolist entity must be type real*4, double precision/real*8, or complex (standard or complex*16); if it is any other type, F77 will signal an error. To format complex data, use two F, E, D, or G descriptors, one for the real part and one for the imaginary part.

F77 reads w characters from the record; these include the sign and can include a specific exponent. If there is an exponent, F77 disregards any scale factor, *kP*, you specify. The exponent can be

- an E or D followed by a signed or unsigned integer constant; e.g. E2 or D2. F77 handles D and E exponents the same way.
- a signed integer constant; e.g., +2.

As with F, the characters read can (but need not) include a decimal point. If they do include a decimal point, this overrides the decimal position given in .d. The decimal point, if present, requires one character position. If the characters read do not include a decimal point, F77 inserts one, placing the rightmost d digits to the right of the decimal point. Then, F77 converts the characters to a floating-point number and stores it in the iolist entity.

Since w is a count of all characters in the field, it includes the sign (if any), decimal point, and exponent.

For any real field, F77 can store accurately only the precision of approximately 6.7 digits (real) or 16.4 digits (double precision/real*8). If the value read exceeds F77's storage *range* (approximately 5.4*10**-79 to 7.2*10**75), F77 will signal an error.

If a minus sign precedes the value or exponent, F77 interprets it as negative; if a plus sign or no sign precedes any of these, F77 interprets it as positive. If F77 reads an illegal character, it signals an error. Legal characters are 0 through 9, decimal point, plus, minus, E, D, and blank.

Blanks count as characters, but leading blanks have no value significance. Embedded blanks have no value significance unless you specified blanks as 0s on the OPEN statement or via the BZ descriptor, covered later in this section. An input value of all blanks has a value of 0.

Output from E or D

F77 formats w characters, including the iolist item value, sign (if needed), and exponent field. If the sum of these is less than w, F77 inserts leading blanks to pad to w. Then F77 outputs this string of characters to the record. The sign is needed if the value is negative or if you specified SP control. The value field includes the decimal point and d characters to the right of the decimal point. If the number of digits following the decimal point in the iolist entity is greater than d, F77 rounds the fraction at the d position.

If you omit *Ee*, the exponent field is 4 characters long, of the form E sign nn.

Thus, the field width w must be large enough to accommodate a minus sign for a negative value (or a plus sign if SP is on), a decimal point, and an exponent. Therefore, w must equal or exceed d+6 (d+e+4 if you specified *Ee*) or F77 will output a field of asterisks.

E and D Examples

In the following examples, the D descriptor would have the same effect as E. (A comparison of E to G is shown under the G descriptor, described next.)

INPUT		
Edit Descriptor	**Characters Read**	**Value Stored in iolist Entity**
E6.4	-.21E5	-21000.
E4.3	.456	.456
E10.1	-1□ 3.99E+3	-13990. (blank= null)
E6.1	□31-01	.31
E6.1	.31-01	.031
E13.0	1234.56789012	1234.57
E9.1	-1□3.99E+	*Error, invalid exp.*
E9.3	24.42E80	*Error, exp. too big*
E7.1	□761348	76134.8
E7.1	76.1348	76.1348
E10.1	761348E-02	761.348
E4.1	□□□□	000.0 = 0.

OUTPUT		
Value in iolist Entity	**Edit Descriptor**	**Characters Written to Record**
+12.34	E10.3	□□.123E+02
-19.54	E6.2	****** *(E6.2 too small for exponent)*
-123.4567	E12.6	-.123457E+03
123.4567	E12.7	.1234567E+03
0.0	E10.4	□.0000E+00
0.007	E10.4	□.7000E-02
-0.007	E10.4	□-.7000E-02

G (Generalized) Editing

On input, the G descriptor is functionally identical to F, E, and D. For output, it uses either F or E editing, depending on the relationship between the field width and exponent. The form of G is

[kP] [r] Gw.d *[Ee]*

where:

- *k* is a scale factor to move the decimal point, described later in this section.
- *r* is a repeat count.
- w is an integer constant that specifies the field's width in characters. It cannot exceed 255.
- d is an integer constant that specifies the number of characters to the right of the decimal point.
- *e* is an integer constant that specifies the number of digits for the exponent on output. The default, which is 2 digits, will suffice for any representable value. *Ee* is ignored on input.

The G mode is useful when you want to use F, but are afraid that some values may be too small or large for a reasonably sized F field to represent. For example, suppose that most of the elements in a real*4 array are greater than or equal to 0.1 and less than 100000.0=10.0**5. You'd like to choose F8.1 for an output edit descriptor since, for example, 48638.1 prints as □48638.1 — but you know that a value such as 1460000.0 would print as ********. The G edit descriptor comes to the rescue! Here, you can use the edit descriptor G12.6 to specify a general field width of 12 characters with 6 to the right of the decimal point if exponential notation is needed. Then the following formatted output occurs:

iolist Entity Value	Output Using F8.1	Output Using G12.6
0.043	□□□□□□.0	□.430000E-01
48638.1	□48638.1	□48638.1□□□□
1460000.0	********	□.146000E+07

Continue reading for the necessary details about the G edit descriptor.

Input to G

On input, G functions exactly as F does.

Output from G

Generally, G works much like F for "reasonable" values (neither very large nor very small), but works like E for numbers that exceed the F field width w, or where insignificant digits are output to the right of the decimal point.

If the absolute value of the iolist entity is equal to or greater than .1 and less than 10**d, F77 uses the F mode but decreases the value of w by 4. It outputs the 4-character exponent field as 4 blanks *after the value.* Generally, F77 outputs d digits. But if the scale factor *kP* is positive and the value is large enough, F77 outputs d+1 digits.

Table 6-2 illustrates the information in the previous paragraph.

On the other hand, if the absolute value of the iolist entity is less than .1 or equal to/greater than 10**d, F77 outputs the number with an explicit exponent, E editing. The exponent has the form E sign nn unless you specified a different exponent field in *Ee*.

Table 6-2. Examples of Output from the G Edit Descriptor

iolist Entity Value, V	Equivalent Conversion, Where n=4 for Gw.d and n=e+2 for Gw.dEe	Example
$0.1 \leq V < 1.0$	F(w-n).d, n('□')	V=0.42, G9.3: F5.3,4X, □.420□□□□
$0.1 \leq V < 1.0$	F(w-n).d, n('□')	V=0.42, 1PG9.3: F5.3,4X, □.420□□□□
$1.0 \leq V < 10.0$	F(w-n).d-1, n('□')	V=4.20, G9.3: F5.2,4X, □4.20□□□□
$1.0 \leq V < 10.0$	F(w-n).d-1, n('□')	V=4.20, 1PG9.3: F5.2,4X, □4.20□□□□
$10.0 \leq V < 100.0$	F(w-n).d-2, n('□')	V=42.0, G9.3: F5.1,4X, □42.0□□□□
$10.0 \leq V < 100.0$	F(w-n).d-2, n('□')	V=42.0, 1PG9.3: F5.1,4X, □42.0□□□□
$100.0 \leq V < 1000.0$	F(w-n).d-3, n('□')	V=420.0, G9.3: F5.0,4X, □420.□□□□
$100.0 \leq V < 1000.0$	F(w-n).d-3, n('□')	V=420.0, 1PG9.3: F5.0,4X, □420.□□□□
$1000.0 \leq V < 10000.0$	F(w-n).d-4, n('□')	V=4200.0,F5.0,4X 4200.□□□□
$1000.0 \leq V < 10000.0$	F(w-n).d-4, n('□')	V=4200.0,G9.3: □.420E+04
$1000.0 \leq V < 10000.0$	F(w-n).d-4, n('□')	V=4200.0, 1PG9.3: 4.200E+03

G Examples

INPUT

See the F input examples and substitute G for F.

OUTPUT

This is a comparison of E and G descriptor output, using a format of 11.4.

Value in iolist Entity	E11.4 Output	G11.4, G11.4E2 Output	G11.4E1 Output
-8977.0	□-.8977E+04	□-8977.□□□□	□□-8977.□□□
23.4	□□.2340E+02	□□23.40□□□□	□□□23.40□□□
7.8	□□.7800E+01	□□7.800□□□□	□□□7.800□□□
-23.1	□-.2310E+02	□-23.10□□□□	□□-23.10□□□
-0.12	□-.1200E+00	□-.1200□□□□	□□-.1200□□□
-1.395	□-.1395E+01	□-1.395□□□□	□□-1.395□□□
-20320.	□-.2032E+05	□-.2032E+05	□□-.2032E+5
0.02032	□□.2032E-01	□□.2032E-01	□□□.2032E-1
-0.02032	□-.2032E-01	□-.2032E-01	□□-.2032E-1
0.00004999	□□.4999E-04	□□.4999E-04	□□□.4999E-4
-0.000000123	□-.1230E-06	□-.1230E-06	□□-.1230E-6
0.0	□□.0000E+00	□□.0000E+00	□□□.0000E+0
0.0123456	□□.1235E-01	□□.1235E-01	□□□.1235E-1
0.123456	□□.1235E+00	□□.1235□□□□	□□□.1235□□□
1.2346	□□.1235E+01	□□1.235□□□□	□□□1.235□□□
12.346	□□.1235E+02	□□12.35□□□□	□□□12.35□□□
123.46	□□.1235E+03	□□123.5□□□□	□□□123.5□□□
-1234.6	□-.1235E+04	□-1235.□□□□	□□-1235.□□□
-123460.0	□-.1235E+06	□-.1235E+06	□□-.1235E+6
123456E+18	□□.1235E+24	□□.1235E+24	**********
123456E-18	□□.1235E-12	□□.1235E-12	**********

P Editing (Scale Factor)

The scale factor moves the decimal point in real numbers. You can use it in conjunction with the F, E, D, or G edit descriptors. Its form is

kP

where:

k is an optionally signed integer constant called the *scale factor*. It specifies the number of positions you want to move the decimal point and in what direction. The range is -127 to +127.

You can affix a scale factor directly to an F, E, D, or G edit descriptor (e.g., 2PF15.5) or let the scale factor stand alone (e.g, 2P,F15.5). The scale factor has no effect on other descriptors.

The scale factor is 0 at the beginning of each I/O statement. After you set a scale factor, it affects all F, E, D, and G (with real numbers) formatting within this format specification unless you reset it. If F77 rescans the specification after encountering the rightmost parenthesis, it will not reset the scale factor. To reset the scale factor to 0, specify 0P.

On input with F, E, D, or G, a positive scale factor has the effect of dividing the number read by 10**k. On output with F, a positive scale factor has the effect of multiplying the iolist number by 10**k.

Input with P

If the number read has an explicit exponent (e.g., 2.3E+1), F77 ignores any scale factor. For example, if F77 reads the number 3.0E02, it stores 300.0 regardless of the scale factor.

If the number read has no explicit exponent, F77 inserts it in the iolist entity as value*(1/10**k), which moves the decimal point left for a positive k and right for a negative k. (On output with F editing, you can undo the input conversion simply by specifying the same scale factor.)

Output with P

You can use a scale factor for all real numbers on output.

For F editing, F77 muliplies the iolist number by 10**k, which moves the decimal point left for a negative k and right for a positive k — the opposite effect from input. For example, to output the iolist number 12.987 as 1298.7, use the edit descriptors 2P,F6.1. This is useful for numeric conversions, like meters to centimeters (use 2P) and cents to dollars and cents (use -2P).

With an E or D edit descriptor, F77 multiplies the real number by 10**k, then subtracts k from the exponent — which does not change the value but does change the output representation; e.g., the iolist value 0.32E05, written with a format specification of 2P,E6.1, would be output as 32.E+03.

For E and D output editing, the scale factor k must be greater than -d and less than d+1, where d is the decimal field width given in the E or D descriptor. If k is greater than 0 and less than d+2, there will be exactly k digits to the left of the decimal point and d-k+1 digits to the right of the decimal point. For example, the value 23.402 with descriptor E11.4 would be output as □□.2340E+02. But with the scale factor 1P, it would be output as 2.3402E+01. An E descriptor with a scale factor of 1 produces classic scientific notation, with one significant figure to the left of the decimal point.

If k is less than 0 and greater than -d, there will be the absolute value k, ABS(k), with leading 0s left of the decimal point and d-ABS(k) digits to the right of the decimal point. For example, the value 0.2340E+02 with descriptor -1P,E11.4 would be output as □□.0234E+03.

With G output editing, F77 chooses either the F or E descriptor for output, as follows, *before* it examines the scale factor. If the absolute (ABS) value of the number would be between .1 and 10**d inclusive, F77 ignores the scale factor and outputs the number in F format. If the exponent of the output number would be less than 0 or more than d, F77 outputs the number in E format, using the scale factor to determine the number of leading zeros after the decimal point.

P Examples

INPUT		
Edit Descriptor	**Characters Read**	**Value Stored in iolist Entity**
2P, F10.2	-25.44□□□□	-.2544
2P, F9.2	3457.1□□□	34.571
2P, F9.2	345.71□□□	3.4571
2P, F9.2	34571□□□□	3.4571
2P, F9.2	□□□□34571	3.4571
3P, F6.3	12.345	.012345
-3P, F6.3	12.345	12345.
-3P, E8.3	12.345E0	12.345 (*P ignored*)
-3P, E8.3	1.2345E0	1.234 (*P ignored*)
-3P, E8.3	12.345E6	12345000 (*P ignored*)
-3P, E8.3	1.2345E6	1234500 (*P ignored*)

OUTPUT		
Value in iolist Entity	**Edit Descriptor**	**Characters Written to Record**
501.33	E10.3	□□.501E+03
501.33	1P, E10.3	□5.013E+02
501.33	2P, E10.3	□50.13E+01
501.33	-1P, E10.3	□□.050E+04
+12.217	F7.3	□12.217
+12.217	1P, F7.3	122.170
+12.217	-1P, F7.3	□□1.222

BZ and BN Editing (Blank Interpretation)

The nonrepeatable edit descriptors BZ (Blank Zero) and BN (Blank Null) control the way F77 handles blanks in *numeric input values.*

F77 always ignores *leading* blanks in numeric values. By default, unless you opened the unit with BLANK='ZERO', F77 also ignores *embedded and trailing* blanks in numeric input values. When it encounters one or more embedded blanks, it treats the value as if the blanks didn't exist, right justifies the numbers, then places the blanks in front of the value. For example, if the edit descriptor is I4 and the characters are 2□□1, F77 assigns the value □□21 to the iolist entity. An edit descriptor of I4 and input characters 21□□ result in the assignment of 21 to the iolist entity.

The BZ descriptor changes this, instructing F77 to interpret blanks as 0s. So if the descriptor is I4 and the characters are 2□□1, F77 assigns 2001 to the iolist entity. Similarly, an edit descriptor of I4 and input characters 21□□ would result in the assignment of 2100 to the iolist entity. The BN descriptor restores normal blank handling (as described in the previous paragraph).

The forms of the BZ and BN edit descriptors are

BZ

BN

Input with BZ, BN

After encountering a BZ descriptor in a format specification, F77 interprets embedded blanks in I, O, Z, F, E, D, and G fields as 0s *for the remainder of the I/O statement.* A slash within the format specification does not change the BZ setting.

After encountering a BN descriptor, F77 resumes normal blank processing, ignoring all embedded blanks in I, O, Z, F, E, D, and G fields.

Output with BZ, BN

F77 ignores the BZ and BN descriptors on output statements.

BZ, BN Examples

Assume a format descriptor of F6.2.

Value in record	Blank Setting	Value Stored in iolist Entity
3□26E2	null	326.
3□26E2	zero	3026.
326E2□	null	326.
326E2□	zero	3.26E20
3□2□6□	null	3.26
3□2□6□	zero	3020.6
3□□□26	null	3.26
3□□□26	zero	3000.26
326□□□	null	3.26
326□□□	zero	3260.
32.□6□	null	32.6
32.□6□	zero	32.06

S-Series Descriptors (Sign Control)

By default, when F77 outputs a numeric field from an iolist entity, it does not insert a leading plus sign (+) for a positive value. (It does insert a leading minus sign (-) before a negative value.)

The S-series descriptors allow you to force output of a plus sign with any numeric field except O, Z, or B. Their forms and meanings are

S Restore default

SP Sign Produce

SS Sign Suppress

The S-series descriptors work on output only, with numeric fields produced by an I, F, E, D, or G descriptor; they are ignored on input.

Output from S, SS, SP

The SP descriptor forces plus-sign output for all positive I, F, D, E, and G values *within this format specification.* (A slash within the specification cannot change plus-sign control.) You should provide space for the plus sign in all numeric fields.

The S or SS descriptor simply restores normal plus-sign handling within this format specification; it does nothing if normal sign handling is in effect.

S-Series Examples

```
      WRITE (10, 100)  2,     3,    4,
     +                -5,    -6,   -7
100   FORMAT  ( 2 (SS, I4, SP, I4, S, I4))
```

The output from this will be

□□2□+3□□4□-5□-6□-7

Complex Data Editing

Because a complex value consists of two values (a real part and an imaginary part), it requires two real F, E, D, or G edit descriptors. The descriptors can differ. The F77 intrinsic function CMPLX (described in Chapter 7) can produce a complex number from two reals; e.g.,

```
      COMPLEX Z

      RR = 3.652
      RI = 5.0E4
      Z = CMPLX(RR, RI)
      WRITE (*, 50) Z
50    FORMAT (1X, 'Real  part:', F10.3, /,
     +        1X, 'Imag. part:', F10.3)

C     Simulate classic imaginary notation.

      WRITE (*, 60) Z
60    FORMAT (1X, F8.3, SP, F9.2, 'i')
      ...
      STOP
      END
```

The output would appear as:

```
Real  part:     3.652
Imag. part: 50000.000
  3.652+50000.00i
```

Note that the WRITE with FORMAT statement 60 produces the classic real-sign-imaginary-i notation for complex numbers.

L (Logical) Editing

The L descriptor transmits a logical value (.TRUE. or .FALSE.) to or from an iolist entity. The entity is logical*4, logical*2, or (AOS/VS) logical*1/byte. The form of the L descriptor is

[r] Lw

where:

r is a repeat count.

w is an integer constant that specifies the field's width in characters.

Input to L

The iolist entity into which F77 reads the value must be type logical or byte; if not, F77 signals an error. F77 reads w characters from the record. The first nonblank character(s) can be T, F, .T, or .F . If T or .T, F77 stores the value .TRUE.; if F or .F, F77 stores the value .FALSE. . F77 ignores characters that follow the T or F. But if the first nonblank characters read are not T, F, .T, or .F, F77 signals an error.

Output from L

If the iolist entity contains the value .TRUE., F77 writes w-1 blanks followed by a T to the record. If the entity contains .FALSE., F77 writes w-1 blanks followed by an F to the record.

L Examples

INPUT		
Edit Descriptor	**Characters Read**	**Value Stored in iolist Entity**
L6	□□TRUE	.TRUE.
L5	□F120	.FALSE.
L2	.FALSE.	.FALSE.
L10	.TRUE.	.TRUE.
L5	12345	*Error, illegal value.*

OUTPUT		
Value in iolist Entity	**Edit Descriptor**	**Characters Written to Record**
.TRUE.	L4	□□□T
.FALSE.	L1	F

Character Editing (A, ' , " , H)

The edit descriptors A, ' (apostrophe), " (quotation mark), and H all are designed to handle character data. The A edit descriptor can also handle logical*1/byte data.

Only the A edit descriptor can transmit character data to or from an iolist entity. The apostophe, quotation mark, and H transfer character constants directly *from* the format specification; they are nonrepeatable and can be be used for output only.

A (Alphanumeric) Editing

The A descriptor transmits characters in alphanumeric format to or from an iolist entity. It transmits the characters simply as characters, which means that it can transmit any string. Numbers transmitted via A cannot be used directly for computations. The form of the A descriptor is

[r] A *[w]*

where:

r is a repeat count.

w is an unsigned integer constant that specifies the field's width in characters. If you omit *w*, the field assumes the width declared for the corresponding iolist entity in character storage units. One standard numeric storage unit equals four character storage units. If you include *w*, it cannot exceed 255; if you omit *w*, the field width can be up to 32,767.

F77 transmits characters according to their ASCII values, without alteration; it does not convert them in any way.

Input to A

The characters read can be any of those in the ASCII character set (Appendix A).

If you omit *w*, F77 reads the number of characters declared for the iolist entity. For example, if CA is CHARACTER*5, and CA is the iolist entity, F77 reads five characters from the record into CA.

If you specify *w*, and your *w* is smaller than the iolist entity, F77 left justifies the characters, padding on the right with blanks. If your *w* is larger than the iolist entity, F77 reads the rightmost characters from the record into the entity. For example, take CHARACTER*3 MM, the edit descriptor A6, and record ABCDEF. A6 specifies six characters, but the entity MM can take only three, so F77 puts the rightmost characters DEF in MM.

The number of characters that F77 uses to represent a *whole* value (e.g., real or integer) depends on the value's data type, as follows:

Data Type of Value	Number of Characters Needed for Whole Value
Integer	4
Integer*4	4
Integer*2	2
Real	4
Real*8	8
Double-Precision	8
Complex	8
Complex*16	16
Logical	4
Logical*4	4
Logical*2	2
Logical*1/byte (AOS/VS)	1
Character*n	n

In F77, you may read numeric or logical values via A format into character iolist entities. However, this has limited utility because F77 transmits the characters in ASCII and you cannot use them for computation directly. Before you can use them for computation, you must convert them to numeric values. You can use internal files to perform this conversion.

Reading numeric or logical values via A *does* offer compatibility with older FORTRANs, which did not have the character data type. Consider, for example, the statements

```
INTEGER*2 FIRST_NAME(5)
READ (11, '(5A1)') FIRST_NAME
...
```

If at runtime you respond with PHIL↵, then FIRST_NAME contains 'P□H□I□L□□□'.

Output from A

If you omit *w*, F77 writes all characters in the iolist entity to the record.

If you include *w*, F77 writes *w* characters from the iolist entity to the record. If *w* is greater than the number of characters in the iolist entity, F77 right justifies the value, inserts leading blanks, and writes *w* characters to the record. *Unlike the numeric descriptors*, if the iolist entity contains more than *w* characters, F77 outputs the leftmost *w* characters and ignores the remaining characters. (With numeric descriptors, F77 outputs a field of asterisks.)

A Examples

A practical example with the A descriptor is

```
...
CHARACTER*20   CFN, CSTA, COFORMAT
PRINT *,   'About to open file.'
PRINT *,   'What filename? '
READ (*, '(A)')   CFN
PRINT *,   'Unit number? '
READ (*, '(I3)')   IU
PRINT *,   'File status desired? '
READ (*, '(A)')   CSTA
...
PRINT *, 'What output format would you like? '
READ (*, '(A)')   COFORMAT

OPEN (IU, FILE= CFN, STATUS= CSTA)
...
...
```

The dialog on the terminal might be

About to open file.
What filename? PFILE ↵

Unit number? 28 ↵

File status desired ? FRESH ↵

What output format would you like? (1X, 2F20.2) ↵

OPEN then proceeds with the filename (actually pathname), unit number, and status given; the output format would be the one given.

For another example of the use of the A descriptor, suppose you want to verify that input consists of a three-digit nonnegative integer. One way is to accept three characters and verify that each is either blank or 0–9. If so, then extract the three-digit integer; if not, display an error message.

```
      CHARACTER*3 IN_DATA
      INTEGER*2 CODE_NUMBER, OK_SWITCH

      PRINT *, 'TYPE AN INTEGER WITH AT MOST 3 DIGITS '
      READ (5, 10) IN_DATA
10    FORMAT (A)

      OK_SWITCH = 1  ! ASSUME GOOD DATA
      DO 20 I = 1, 3
      IF ( IN_DATA(I:I) .EQ. " " )
     +     IN_DATA(I:I) = "0"
      IF ( IN_DATA(I:I) .LT. "0" )
     +     OK_SWITCH = 0
      IF ( IN_DATA(I:I) .GT. "9" )
     +     OK_SWITCH = 0
20    CONTINUE
      IF ( OK_SWITCH .EQ. 1 ) THEN
           READ(IN_DATA, 30) CODE_NUMBER
30         FORMAT (I3)
           PRINT *, 'NUMBER EXTRACTED IS ', CODE_NUMBER
      ELSE
           PRINT *, 'ILLEGAL INPUT'
      END IF
      ...
```

The following example reads a string of characters into an integer*2 array. The right byte of each word in the array is a blank <040>.

```
      INTEGER*2 RECORD(80)
      OPEN (4, FILE='MY_DATA', PAD='YES')

   10 READ (4, 20, END=30) RECORD
   20 FORMAT (80A1)
      IF ( RECORD(1) = 'A ' ) THEN
C     IF ( RECORD(1) = 1HA  ) IS EQUIVALENT
           PRINT *, RECORD
      END IF
      GOTO 10

   30 STOP
      END
```

INPUT		
Assume CHARACTER*6 CH1, current record contains ABCDEFGHIJ.		
Edit Descriptor	**Characters Read**	**Value Stored in iolist Entity**
A	ABCDEF	ABCDEF
A4	ABCD	ABCD□□
A5	ABCDE	ABCDE□
A6	ABCDEF	ABCDEF
A7	ABCDEFG	BCDEFG
A10	ABCDEFGHIJ	EFGHIJ

OUTPUT		
Assume CHARACTER*10 CH2.		
Value in iolist Entity	**Edit Descriptor**	**Characters Written to Record**
Value□is:□	A	Value□is:□
Value□is:□	A2	Va
Value□is:□	A4	Valu
Value□is:□	A6	Value□
Value□is:□	A10	Value□is:□
Value□is:□	A12	□□Value□is:□

Some formatting examples of A are

```
      BYTE INITIALS(3)    ! AOS/VS only
      INITIALS(1) = 'J'
      INITIALS(2) = 'T'
      INITIALS(3) = 'M'
      WRITE (10, 20) INITIALS
   20 FORMAT('INITIALS ARE *', 3A1, '*', 3A2, '*')
      STOP
      END
```

The output from the WRITE and FORMAT statements is

*INITIALS ARE *JTM* J T M**

Apostrophe (') and Quotation Mark ('') Editing

As edit descriptors, the apostrophe or quotation mark characters (' and '') simply output a string of characters to the record. The apostrophe is the standard descriptor; the quotation mark is an extension and functions exactly the same way. The forms are

'string'

"string"

where string is any string of characters. In a source file, these can include any of the F77 special mnemonics (<NL> for NEW LINE, <FF> for form feed, etc.) or the angle-bracketed octal code of any character. The mnemonics and octal codes are described in Chapter 2, "Character Constants". To have these function as control characters, you must place them in a source file so the compiler can translate them; they will not operate as control characters if you read them at runtime.

For example:

```
    PRINT 100,   RF00
100 FORMAT (' <BEL>Value is: ',F9.2)
```

If, using apostrophes only, you wish to enclose an apostrophe in the character string, use two successive apostrophes; F77 will then transfer one apostrophe. This also applies to quotation marks. For example,

```
    PRINT *, 'Owned by O''Leary'
```

results in the output of

```
Owned by O'Leary
```

For another example, a way to output the message

Ohio State's football team is called the "Buckeyes."

is with the pair of statements

```
   WRITE (*, 100)
100 FORMAT (1X, 'Ohio State''s football team is ',
  +          "called the ""Buckeyes.", 1H'')
```

This FORMAT statement uses a Hollerith edit descriptor (1H") as described later in this chapter in the section "H Editing."

Alternatively, you can use one type of descriptor to enclose the character that delimits the other:

```
    WRITE (*, 105)   RF00
105 FORMAT (" RF00's value is: ",F9.2)
```

Input to ' or ''

You cannot use these descriptors for input.

Output from ' or ''

F77 writes the characters *enclosed within the descriptors*. Either of the examples

```
    WRITE (*, 107)   266.45
107 FORMAT (1X, 'Value is: $',F9.2)
```

or

```
    WRITE (*, "(1X,'Value is: $',F9.2)")   266.45
```

would write

```
Value is $   266.45
```

on to the file connected to unit 6, by default file @OUTPUT.

' and '' Examples

The following two I/O statements do the same thing:

```
    PRINT 111,   N**2
111 FORMAT (" N squared is: ",I10)

    WRITE (*, "(' N squared is: ',I10)")   N**2
```

The next example uses both the apostrophe and A edit descriptors.

```
    ...
C   I and J get values, for example:
    I = 25
    J = 999
    WRITE (*, 113)   'I', I, 'J', J
113 FORMAT (1X, A, ' is ', I8, A, ' is ', I8)
```

Output from this example would appear as:

```
I is      25 J is      999
```

H Editing

The H descriptor writes a literal string of characters to a record; you can use it for output only. The form is

nHc...c

where:

n is a positive, unsigned integer constant that specifies the number of h characters.

c is any character from the ASCII character set.

The H descriptor retains some of the old (pre-F77) Hollerith functionality. But you cannot use it for input (READs).

There is no such thing as a Hollerith data type or a Hollerith variable. Hollerith *constants* exist. Recall from Chapter 2 that you can assign them to variables via statements such as I2VAR=2HAB.

Output from H

F77 writes n characters. It outputs any character, including an apostrophe or quotation mark, without interpretation. Angle brackets have no special meaning as part of an H edit descriptor.

H Examples

```
    WRITE (*, 115)    45
115 FORMAT (11H Value is: , I8)

    RY = 24667.4
    WRITE (*, 120)   RY
120 FORMAT (19H The 'result' is:  , F10.2)

    WRITE (*, 125)
125 FORMAT (1H , 'NO BELL: ', 5H<BEL>)
```

Each WRITE statement specifies unit 6, preconnected to file @OUTPUT with FORTRAN carriage control. With FORTRAN carriage control, the first character is used for carriage control, and is not printed. So, during execution, F77 will display

```
Value is:□□□□□□45
The□'result'□is:□□□24667.40
NO BELL: <BEL>
```

Positional Editing (X, T-Series, $)

This section describes the edit descriptors that control positioning *within* a record. These are X, T-series (T, TL, TR), and $ (dollar sign); they are all nonrepeatable. (The slash, which positions the file at a *new* record, is described after "Colon Editing", next.)

The position within a record is the current column number; the initial position is column 1.

The X and T-series (Tab) descriptors set position within a record, allowing you to reread characters on input or rewrite fields on output.

The $ descriptor suppresses output of the NEW LINE character. (F77 prints a NEW LINE after each record in files opened with the default LIST carriage control.) $ is effective on output only, with LIST carriage control only. It allows you to effectively concatenate records and print multiple records on one line. (To get the $ effect with FORTRAN carriage control, use the # character in column 1, described under "Carriage Control".)

X (Skip Position) Editing

The X descriptor skips one or more character positions. Its form is

nX

where:

n is an unsigned positive integer constant that specifies a number of character positions to the right of the current position. You *must* include a number less than 256 for n, even if it is 1.

Input with X

F77 moves the current column position n columns to the right within the current record.

Output with X

F77 moves the current column position n columns to the right within the record. At record output time, if there are any gaps (unfilled positions) between column 1 and the last character supplied by the format, F77 inserts blanks in the gap positions. For example, for the statements

```
    WRITE (*, 128)   1000, 2000
128 FORMAT (1X, I4, 3X, I4)
```

F77 inserts one blank before 1000 and three blanks before 2000. If CARRIAGECONTROL='FORTRAN', then the blank before 1000 is the carriage control character, so the "1" of "1000" appears in column 1.

By using X to skip the first position in each record, you can force F77 to insert a blank at the first position. If carriage control is FORTRAN (for CARRIAGECONTROL='FORTRAN'), the first character is used for carriage control, not output as data. Thus the blank inserted by 1X — and not the first character of the record — serves for carriage control, and the first character of the record is output as is.

X can move the current column position to a column outside of the record if you don't try to write characters there.

X Examples

```
C   Compute value of variable and write to
C      printable output file.
    ...
    OPEN (4, CARRIAGECONTROL='FORTRAN',
   +   RECFM='DS', FILE='PFILE')
    ...
    CREDIT = 25.00
    DEBIT  = 5.00
    BALANCE = CREDIT - DEBIT
    WRITE (4, 80)   CREDIT, DEBIT, BALANCE
80  FORMAT (1X, '$', F7.2, 2X, '$',F7.2, 4X,'$', F7.2)
    ...
```

F77 will print this record as

```
$□25.00□□$□□5.00□□□$□□20.00
```

Another example, also with CARRIAGECONTROL='FORTRAN', is

```
    ...
    B = 367.82
    C = 1.21
    ...
    WRITE (*, 90)   B, C
    WRITE (*, 95)   B, C
90  FORMAT (1X, 'Values are: ', F6.2, F6.2)
95  FORMAT (    'Values are: ', F6.2, F6.2)
    ...
```

When variables B and C are printed, the output will look like

```
Values are: 367.82  1.21
alues are: 367.82  1.21
```

The second line is truncated because the first character (V) was used for carriage control — detailed later.

T-Series (Tab) Editing (T, TL, TR)

The T-series positioning descriptors set absolute position (T), tab left (TL), and tab right (TR). TR is functionally identical to X. The forms are

Tn

TLn

TRn

where:

n is an unsigned integer constant. For T, n indicates the absolute position within the record. For TL, n indicates the number of positions to the left of the current position. For TR, n indicates the number of positions to the right of the current position.

T-Series Input

For T, F77 positions at character position n of the current record. For TL, it positions n positions left of the current position, which enables you to reread characters (but if n is more than the current position, F77 positions to the first character; it does not back up into the previous record). For TR, F77 skips over n characters, tabbing right.

T-Series Output

For T, F77 positions at column n within the record.

For TL, F77 tabs n column positions left, which enables you to overwrite characters already placed in the record. As for input, if n is greater than the current position, F77 positions at column 1; it cannot back up into the previous record.

For TR, F77 positions right as for X.

At record output time, if there are any gaps (unfilled positions) between column 1 and the last character supplied by the format, F77 inserts blanks in the gap positions (as for X).

T, TL, and TR can help format columnar output. With T and TR you can set position outside the current record, but you cannot write there.

T-Series Examples

```
C    For T descriptor:
     ...
     C = 4.11
     WRITE (*, 124)   C
124  FORMAT ( T2, '12345', T4, F6.2, ' abcdef')
     ...
```

F77 will print this record as:

```
12  4.11 abcdef
```

The asterisk in the WRITE statement specifies unit 6, file @OUTPUT, preconnected with FORTRAN carriage control — so the first character is used for carriage control.

```
C   For TL descriptor.
    PRINT *, 'Type 10 digits<NL>'
    READ (*, 95)  J, K, L
95  FORMAT (I8, TL4, I2, TL30, I3)
    WRITE (*, 99)  J, K, L
99  FORMAT (1X, 'J= ', I8, ',K= ', I4, ',L=', I3)
    ...
```

If, when the code above executes, the person at the terminal types the ten characters 1234567890 ↓, the output will be

```
J= 12345678,K=   56,L=123
```

For a TR example, see the X examples.

$ (Suppress NEW LINE) Editing

If a file is opened with CARRIAGECONTROL='LIST' (the default) and data-sensitive record organization (also default), F77 will output a NEW LINE after each record written. This means that records will be single spaced, each on its own line.

But if F77 encounters a $ descriptor in an output format specification, it will not output a NEW LINE character after the record. This allows you to write more than one record on a line. The form of $ is

```
$
```

The $ descriptor has no effect on input, or when CARRIAGECONTROL='FORTRAN' or 'NONE'. With FORTRAN carriage control, the # character has a similar effect, described later under "Carriage Control."

$ Examples

Assume that unit 3 is opened with LIST carriage control, the default. The statements

```
    WRITE (3, '(I6, $)')  125
    WRITE (3, '(I6)'   )  133
```

produce this output when the file opened on unit 3 is printed:

```
□□125□□133<NL>
```

Without the dollar sign in the first WRITE, the output would have been

```
□□125
□□133
```

Another example is

```
    WRITE (10, 36)
36  FORMAT ('Type Answer ', $)
    READ (11, '(F6.2)')  ANSWER
    WRITE (10, FMT="('The answer is ', F6.2)") ANSWER
    STOP
```

The dialog from the statements above might be

Type Answer 5.00 ↓
The answer is 5.00
(blank line)
STOP

Unit 10 is preconnected to file @OUTPUT with LIST carriage control, so $ suppresses the NEW LINE that would normally be printed after *Type Answer*.

Colon (:) Editing (Terminate Format Control Conditionally)

The colon descriptor terminates format control if the iolist entities have all been processed; it has no effect if more iolist entities remain in the list.

F77 can ignore extra repeatable edit descriptors in a specification. But by default it will process *all* nonrepeatable descriptors, even if the iolist is empty or if all entities have been formatted.

A colon in a specification tells F77 to terminate format control if no iolist entities remain. It can help you avoid printing superfluous text messages. It is also useful with empty iolists (which are often used to print messages, titles, and headers or to control position within a record). On input, it can be used to prevent an extra record from being read.

The form of the colon descriptor is

```
:
```

Colon Input and Output

The colon descriptor terminates format control only if the iolist contains no more entities for formatting, preventing F77 from processing subsequent nonrepeatable edit descriptors.

Colon Examples

```
   WRITE (*, 75)
   WRITE (*, 75)    10, 20, 30
75 FORMAT (1X, : ,'1st', I4, : ,
  +             ';2nd', I4, : , ';3rd', I4)
```

The output will be

```
(blank record)
1st□□10;2nd□□20;3rd□□30
```

If the colons had been omitted from statement 75, the output would have been

```
1st
1st□□10;2nd□□20;3rd□□30
```

Another example is

```
   ...
   READ (2, 3 F9.2, : , /)   A, B, C
```

The colon prevents execution of the slash edit descriptor.

Carriage Control

Often, when you want to print a file, you will want the first character to be used for traditional FORTRAN vertical carriage control, rather than for printing. (Printing means transferring the record to a printing device, either a terminal or line printer.)

One way to get traditional carriage control is to create a file with data-sensitive records (default), opened with CARRIAGECONTROL='FORTRAN'.

Alternatively, utility program F77PRINT can process files created by other means; the output is a printable file. See Appendix D for details.

With FORTRAN carriage control, each of the following characters, inserted via format specification or any other means, has the following effect when the record is printed.

First Character in Record	Result when Record Is Printed
□	Prints record on next line (single spacing); <NL>record.
0	Prints record on line after next (double spacing); <NL><NL>record.
1	Prints record on top of next page; <FF>record.
+	Prints record on same line, starting with column 1 (overprints); <CR>record.
#	Prints record on same line, starting with current column position; appends to the record; <record>.
other character	Prints record on next line (single spacing); <NL>record.

Special Properties of 1 (New Page)

Be especially careful about using 1 with FORTRAN carriage control. It makes F77 output a form-feed character. If you mistakenly use 1 instead of a space and your output file has several hundred records, then the printed (QPRINT) file may have several hundred pages with just one line at the top of each page! Additionally, the printer responds to a form-feed character by advancing the paper rapidly — typically several inches per second — and a spectacular paper jam may occur.

While a form-feed character does indeed advance the paper to the top of the next page, a "page" does not have to be 11 inches and 66 lines. You AOS/VS, F7716, and AOS programmers can use the Forms Control Utility (FCU) program to specify a page in a file to contain a specific number of lines (typically between 6 and 144). See your operating system CLI user's manual for information on the FCU program and the flexibility it offers you to print special forms, such as mailing labels.

Placing Carriage Control Characters in Records

There are several ways to get these characters into records. You can insert them literally with apostrophe or quotation mark edit descriptors; e.g.,

```
WRITE (39, '(" ", 2 F9.2)')   B, B4
WRITE (49, '("1", 2 F9.2)')   C, C1
```

Or, if you want a NEW LINE before a record, you can skip one or more character positions before writing so that F77 will insert one or more blanks; e.g.,

```
WRITE (39, '(1X, 2 F9.2)')    B, B4
```

Or, you can make the field width w large enough to ensure that the first character will be a blank; e.g.,

```
WRITE (39, '(2 F20.2)')    B, B4
```

Or, you can use a Hollerith constant to specify carriage control; e.g.,

```
! Double space before printing.
WRITE (39, '(1HO, 2 F9.2)')    B, B4
```

Preconnected units for output that are opened with CARRIAGECONTROL='FORTRAN' are 6 (* specifier) and 12.

Thus far in this chapter, most examples assume FORTRAN carriage control. If you omit the CARRIAGECONTROL= property on the OPEN statement, F77 uses LIST carriage control.

LIST (Default) Carriage Control

Unless you open a file with CARRIAGECONTROL='FORTRAN', F77 uses the default carriage control, LIST.

With LIST carriage control, F77 outputs the first character of each record literally — and, if the file involved is data sensitive, by default it outputs a NEW LINE after each record.

If you want other printing control in records, you can insert NEW LINEs and form feeds manually via angle-bracketed mnemonics; e.g.,

```
    WRITE (49, 100)   A, B, C
100 FORMAT ('<FF>', 3 F10.2)
```

Other common angle-bracketed mnemonics are <NL> and <TAB>; all of them are described in Chapter 2, "Character Constants." You should be careful when placing control characters within records because F77 and other utilities may treat them as record delimiters on input — making two records from one.

The preconnected unit for output opened with LIST carriage control is 10; the file is @OUTPUT.

Some examples of FORTRAN and LIST carriage control, with data-sensitive record organization, are shown in Table 6-3. In the table, "Printed Output" means the characters that F77 places in the file connected to unit 4 in response to the WRITE(4,... statement.

Table 6-3. Carriage Control Examples with Data Sensitive Records

Statement with Format Specification	Printed Output, FORTRAN Carriage Control. Unit 6	Printed Output, LIST Carriage Control. Unit 10
C Integer examples.		
WRITE (4, '(I1)') 1	*<form feed>*	*1<NL>*
WRITE (4, '(I1)') 125	*<NL>*	**<NL>*
WRITE (4, '(I2)') 125	*<NL>**	***<NL>*
WRITE (4, '(I3)') 125	*<form feed>25*	*125<NL>*
WRITE (4, '(I4)') 125	*<NL>125*	*□125<NL>*
WRITE (4, '(1X, I4)') 125	*<NL>□125*	*□□125<NL>*
WRITE (4, '(" ",I4)') 125	*<NL>□125*	*□□125<NL>*
WRITE (4, '(1H ,I4)') 125	*<NL>□125*	*□□125<NL>*
WRITE (4,'("<NL>",I4)') 125	*<NL>□125*	*<NL>□125<NL>*
WRITE (4, '(I12)') 125	*<NL>□□□□□□□□□125*	*□□□□□□□□□125<NL>*
C Character examples.		
CHARACTER*6 CI / 'abcdef' /		
WRITE (4, '(A5)') CI	*<NL>bcde*	*abcde<NL>*
WRITE (4, '(A6)') CI	*<NL>bcdef*	*abcdef<NL>*
WRITE (4, '(A7)') CI	*<NL>abcdef*	*□abcdef<NL>*
WRITE (4, '(A)') CI	*<NL>bcdef*	*abcdef<NL>*
C # and $ examples.		
WRITE (4, '(1X,I3)') 25	*<NL>□25*	*□□25<NL>*
WRITE (4, '("#",I3)') 33	*□33*	*#□33<NL>*
WRITE (4, '(I3, $)') 25	*<NL>25*	*□25*
WRITE (4, '(I3)') 33	*<NL>33*	*□33<NL>*

Descriptor Separators

If you put multiple descriptors in a format specification, you must separate them clearly. The comma is the traditional separator.

Several edit descriptors themselves serve as separators, and you can omit comma separators around these (although the commas will do no harm). The descriptors that act as separators are

- colon (:); for example,

```
(1X, '1st ', F9.2, : '2nd ', F9.2)
```

executes the same way as

```
(1X, '1st ', F9.2, :, '2nd ', F9.2).
```

- slash (/) also works as a separator but starts a new record. Use it only if you want to start a new record (as on output). The slash is described later under "Multiple Record Formatting."

Repeat Count and Nested Format Specifications

You can repeat any repeatable descriptor individually as needed. You can also nest groups of repeatable and nonrepeatable descriptors within parentheses and precede *them* with a repeat count. A repeat count that precedes a parenthesis pair affects all descriptors and nested descriptors within that parenthesis pair. If you omit a repeat count, each descriptor is used only once, unless format reversion (described next) occurs.

For example, each of the following pairs of format specifications have the same effect if no reversion occurs:

```
( 2I2, 3F11.2 )
( I2, I2, F11.2, F11.2, F11.2 )

( F9.2, 1 (I2, I3) )
( F9.2, (I2, I3) )

( 2 (I2), 4 (1X, I3), 2 (I4) )
( I2, I2, 1X, I3, 1X, I3, 1X, I3, 1X, I3, I4, I4 )
```

You can nest up to 16 levels of groups with parentheses. The level delimited by the outermost parentheses is level 0, the level delimited by the first nested parentheses is level 1, the level nested within level 1 is level 2, and so on.

The levels also determine where the format rescan begins if format reversion occurs.

Multiple Record Formatting

F77 can process more than one record in a single I/O statement if either of the following conditions exists:

1. If the repeatable edit descriptors (I, F, A, etc.) are exhausted and there are more iolist entities to process, F77 starts a new record and *reuses* the format specification. This is *format reversion.*
2. If there is a slash (/) in the format specification, F77 starts a new record, using the repeatable edit descriptors to the right of the slash for the remaining iolist entities.

You can determine the number of records processed by *any* formatted I/O statement with the following rules. F77 reads or writes a record:

- at the beginning of each I/O statement (even if the iolist is empty), and
- each time a format reversion (rescan) occurs, and
- each time it encounters a slash in a format specification.

The number of records processed for any I/O statement is the sum of all these.

Format Reversion

F77 *must* process each iolist entity that uses a repeatable edit descriptor (I, F, A, etc.). If all repeatable edit descriptors in the format specification have been used and more entities remain in the iolist, format reversion occurs: F77 starts the next record and reuses the format specification. (On the other hand, if there are *extra* repeatable edit descriptors after F77 has formatted all iolist entities, F77 simply ignores the extra descriptors.)

If reversion occurs and there are no nested parentheses in the specification, the nesting level is 0 and F77 reuses the entire format. F77 starts at the first repeatable edit descriptor and proceeds through the repeatable descriptors until it has formatted all iolist entities. For example:

```
      READ (88, 100)   B1, B2, B3
100   FORMAT (F9.2)
```

Here, F77 formats the first nine characters from the record according to F9.2 and puts them in B1. Then, having run out of repeatable edit descriptors, it reads in the next record, formats the first nine characters per F9.2 and puts them in B2. Once more, being out of repeatable descriptors, it reads in the *next* record, formats the first nine characters via F9.2, and puts *them* in B3. In sum, it reads a constant number of characters from each record into an iolist entity.

If reversion occurs and there *are* nested parentheses in the format specification, F77 starts the rescan before the *rightmost level-1* left parenthesis, including the repeat count for this rightmost group if any. (It never starts a rescan at a level deeper than 1.)

For examples of both level-0 and level-1 reversion, consider the following statements:

```
20    FORMAT (F10.2, 2E15.7, I5, 10X, A)
40    FORMAT (I4, 2F10.2, 3(1X, I6), 4(A) )
```

```
20      FORMAT  (F10.2, 2E15.7, I5, 10X, A)
                 ↑          0
```

```
40      FORMAT  (I4, 2F10.2, 3 (1X, I6), 4 (A)  )
                             1        1
                 0
```

For statement 20, the rescan begins before the first repeatable descriptor: F10.2. For statement 40, rescan begins before the rightmost level-1 left parenthesis: before 4(A).

Reversion Example

```
    A = 1.0
    B = 2.0
    C = 3.
    WRITE (*, 135)   A, B, C
135 FORMAT (T2, 'Values:', (T14, F9.2) )
    ...
```

In the example, reversion occurs within level 1, at the leftmost parenthesis of the level-1 descriptors: at T14. Output with FORTRAN carriage control is

```
Values: □□□□□□□□1.00
□□□□□□□□□□□□□□2.00
□□□□□□□□□□□□□□3.00
```

Slash (/) Editing (Start a New Record)

A slash in a format specification directs F77 to stop processing the current record and start the next. It is a nonrepeatable edit descriptor that controls file position and does not format characters. Primarily, the slash allows you to format a group of zero or more iolist entities as a separate record and to skip records. The form is

/

Input with Slash

On a unit connected for sequential access, a slash (/) causes F77 to skip the rest of the current record; it then reads the next record. For direct-access connections, F77 simply adds 1 to the record number and reads it, which has the same effect.

Sequential slashes allow you to bypass input records. The first slash terminates input of the current record and reads in the next record, even if the file is positioned at the beginning of the current record. Each slash that follows skips a record.

Output with Slash

Each slash terminates a record. Two sequential slashes output a record consisting of a blank line.

Slash Examples: INPUT

Assume that the current record and the next sequential records on unit 45 contain the following (14) characters:

```
current   □111111□111111
next      □222222□222222
next      □333333□333333
next      □444444□444444
```

```
    READ (45, 100)   L, M, N
100 FORMAT (I4, /, I4, /, I4 )
```

The first four characters of the current record (1s) are read into L, the slash skips the rest; the first four characters of the next record (2s) are read into M, the slash skips the rest; the first four characters of the next record (3s) are read into N, and the I/O statement terminates. (Note that 100 FORMAT (I4) would do the same thing through format reversion.) After the READ executes, the iolist entities have the following values:

L = 111 from first record
M = 222 from second record
N = 333 from third record

If the first slash and comma had been omitted, F77 would have filled L and M from the first record. The iolist entity values would have been

L = 111 from columns 1–4 of first record
M = 111 from columns 5–8 of first record
N = 222 from columns 1–4 of second record

If *all* slashes and their commas had been omitted, F77 would have read only the first record and the iolist values would have been

L = 111 from columns 1–4 of first record
M = 111 from columns 5–8 of first record
N = 1111 from columns 9–12 of first record

Slash Examples: OUTPUT

```
    WRITE (4, 200)    1.0, 2.0
    WRITE (4, 300)    1.0, 2.
200 FORMAT (1X, F4.2, /, 1X, F4.2)
300 FORMAT (1X, F4.2, 1X, F4.2)
```

The output (with FORTRAN carriage control) is

```
1.00
2.00
1.00□2.00
```

The slash in format 200 outputs the second value as a new record. Format 300, without the slash, outputs both values as a single record.

The next slash output example outputs the values of entities L, M, and N. The output file is unit 4. L, M, and N contain the following values:

```
    L = 1111
    M = 2222
    N = 3333
    WRITE (4, 120) L, M, N
120 FORMAT (1X, I4, /,
   +        1X, I4, /, /,
   +        1X, I4, /
   +        )
```

F77

- formats the value in L and starts a new line
- formats the value in M and starts a new line for the first slash and outputs a blank line for the second slash
- formats the value in N and starts a new line
- outputs one blank line for completion of the I/O statement.

Assuming FORTRAN carriage control, the output looks like this:

```
1111
2222
(1 blank line)
3333
(1 blank line)
```

If the first slash and comma had been omitted, F77 would have output the values of L and M as one record:

```
1111 2222
(1 blank line)
3333
(1 blank line)
```

If *all* slashes and their commas had been omitted, F77 would have output all iolist values as one record:

```
1111 2222 3333
```

Format Expressions

As an extension to the standard, DG AOS/VS F77 allows *expressions* within FORMAT statements.

Traditionally, FORMAT statements contain integer constants to indicate field widths, repeat specifiers, and scale factors. DG's F77 allows this usage as shown so far in this chapter. Some examples are

```
10 FORMAT (I4, F7.3)  ! Field widths are 4, 7, 3
20 FORMAT (7F4.1)     ! Repeat specifier is 7
30 FORMAT (2P6.0)     ! Scale factor is 2
```

DG F77 also allows expressions in these three places. You delimit the expressions by angle brackets, < >. Some examples, based on the previous three FORMAT statements, follow.

```
   CHARACTER*4 CH  / '12' /
   K2 = 2
   K3 = 3
   K4 = 4
   K7 = 7
   R = 5.8
10 FORMAT ( I<K4>, F<K7>.<K3> )   ! ( I4, F7.3 )
11 FORMAT ( I<4>, F<K7>.3 )       ! ( I4, F7.3 )
12 FORMAT ( I<K7-3>, F7.<K2+1> )  ! ( I4, F7.3 )
20 FORMAT ( <K7>F4.1 )            ! ( 7F4.1 )
21 FORMAT ( <K7>F<K4>.1 )         ! ( 7F4.1 )
22 FORMAT ( <R>F4.1 )             ! ( 5F4.1 )
30 FORMAT ( <K2>PF6.0 )           ! ( 2PF6.0 )
40 FORMAT ( A<LEN(CH)> )          ! ( A4 )
41 FORMAT ( T<40+K2+K3> )         ! ( T45 )
```

Additional properties of expressions in FORMAT statements are:

- An expression may not replace the character count before the H edit descriptor. Thus, <K7>Habcdefg is wrong, even though variable K7 contains 7.
- F77 evaluates the expression each time it encounters the expression at runtime.
- If the expression is not type integer, F77 converts it to an integer. See the previous FORMAT statement whose label is 22.

Implied DO loops don't work with FORMAT statements containing expressions because of optimization that the FORTRAN 77 compiler performs on these loops. An example is

```
      WRITE (10, 20) (MINE(m), m = 1, 4)
   20 FORMAT ( I<m> )
```

MINE(1) through MINE(4) do *not* appear with successive edit descriptors I1, I2, I3, and I4. Instead, a runtime error occurs.

Format/Entity Mismatch

As an extension to the standard, DG F77 allows certain runtime mismatches between I/O entities and their edit descriptors. This mismatch occurs only when you specify it with the F77LINK switch /FMTMM (ForMaT MisMatch), and only in formatted I/O.

A simple example introduces a mismatched I/O entity and edit descriptor.

```
      PROGRAM ENT_FMT_MISMATCH
      RVAR = 18.75
      WRITE (10, 20) RVAR
   20 FORMAT ('*', L5, '*')
      STOP
      END
```

This program compiles and links with no errors. By default, its execution results in the runtime error message "Real item may only be formatted using the A,F,E,D, or G edit descriptors." This message corresponds to the symbol F77ERFSMR in F77 error file, F77ERMES.SR.

On the other hand, you can give the CLI command F77LINK with the /FMTMM switch. At runtime program ENT_FMT_MISMATCH outputs

*□□□□□*F**

F77 handles the case of real-entity-output/L-edit-descriptor by converting the real entity to an integer, examining its low order bit, and outputting either F (the bit is 0) or T (the bit is 1). Here, F77 obtained the integer 18 from RVAR's value of 18.75. Since 18 is an even number, its low order bit is 0 and F77 outputs the letter F (right justified with leading spaces).

Classes of Edit Descriptors

In general, F77 separates its edit descriptors into classes for integer, logical, floating-point, and alphanumeric editing. By default F77 formatted I/O requires each data type to have a specific class of edit descriptors. The following lists the FORTRAN data types and their allowable edit descriptors. "Allowable" means "without the F77LINK switch /FMTMM."

Data Type	Allowable Edit Descriptor
Integer	I, O, Z, B, A
Logical	L, A
Real, Double Precision, Complex	F, E, D, G, A
Character	A

If an iolist entity doesn't have a corresponding allowable edit descriptor in the accompanying FORMAT statement, F77 reports a task-fatal runtime error for the I/O statement.

Mismatch Operations

Table 6-4 describes the operations F77 performs in various cases of mismatches (and matches) between iolist entities and edit descriptors. The table assumes a program was linked with a F77LINK/FMTMM command. The following abbreviations appear in the table.

FX	Fix the floating-point number to produce an integer number of the appropriate size (2 or 4 bytes).
FL	Float the integer number to produce a double precision real number in floating-point format.
DE	Dynamic Equivalence — Transfer the bit pattern without modification to an equivalent length destination location.
RB	Right Byte — Transfer the right-most byte of the bit pattern to the 1-byte destination location; all the bits to the left of the right-most byte must be off.
IN	"Integerize" the floating-point number. That is, convert the number to an integer, and then the integer to a character string of digits.
LG	Logical conversion of a real number. Convert the real number to an integer, and then test the right-most bit of the integer to return the character T or F.
X_in	The input data. X is A (alphanumeric), I (integer), F (floating point), or L (logical).
X_out	The output data. X is A (alphanumeric), I (integer), F (floating point), or L (logical).

The examples in Tables 6-5 and 6-6 illustrate the abbreviations in Table 6-4.

The examples in Table 6-5 illustrate the input operations in Table 6-4. The examples assume F77 programs are linked with the /FMTMM switch.

The examples in Table 6-6 illustrate the output operations in Table 6-4. The examples assume F77 programs are linked with the /FMTMM switch.

Table 6-4. Operations Performed During Formatted Input and Output

Format / Data Type, Input or Output	Integer I,O,Z,B	Logical L	Floating F,E,D,G	Alphanumeric A
Integer:				
Input:	Standard	I ← DE(L_in)	I ← FX(F_in)	I ← DE(A_in)
Output:	Standard	DE(I) → L_out	FL(I) → F_out	DE(I) → A_out
Logical*1:				
Input:	L ← RB(I_in)	Standard	L ← RB(FX(F_in))	L ← DE(A_in)
Output:	DE(L) → I_out	Standard	FL(DE(L)) → F_out	DE(L) → A_out
Logical:				
Input:	L ← DE(I_in)	Standard	L ← DE(FX(F_in))	L ← DE(A_in)
Output:	DE(L) → I_out	Standard	FL(DE(L)) → F_out	DE(L) → A_out
Real:				
Input:	R ← FL(I_in)	R ← FL(DE(L_in))	Standard	R ← DE(A_in)
Output:	IN(R) → I_out See note 1.	LG(R) → L_out	Standard	DE(R) → A_out
Character:				
Input:	Error	Error	Error	Standard
Output:	Error	Error	Error	Standard

Note 1: The only integer format you may use with a floating-point variable is I.

Table 6-5. Examples of Input Entities, Edit Descriptors, and Values Stored

Character String Input	Input Entity and Edit Descriptor	Value Placed in Input Entity	Comments
2916	Integer*2, I4	2916	Standard
T	Integer*2, L1	-1	L_in contains all 1s for T; this bit pattern is -1 as a twos-complement one-word integer.
7.9	Integer*2, F3.1	7	F_in contains 7.9; FX(F_in) is 7.
T	Integer*2, A1	21536	A_in contains <124><040>; this bit pattern is 21536.
2916	Real*4, I4	2916.0	I_in contains 2916; FL(I_in) is 2916.0.
T	Real*4, L1	-1.0	L_in contains all 1s for T; this bit pattern is the integer*4 number -1; FL(-1) is -1.0.
7.9	Real*4, F3.1	7.9	Standard
T	Real*4, A1	1.517083E+23	A_in contains <124><040><040><040>; this bit pattern is 1.517083E+23.
2916	Logical*1, I4	——	Error; the bits to the left of the right-most byte of 2916 are not all zero.
2916	Logical*4, I4	.FALSE.	I_in ends in 0.
T	Logical*1, L1	.TRUE.	Standard (The entity contains -1 = all 8 bits on.)
T	Logical*4, L1	.TRUE.	Standard
7.9	Logical*1, I3	——	Error; the three characters '7.9' do not form a valid integer.
7.9	Logical*4, I3	——	Error; the three characters '7.9' do not form a valid integer.
T	Logical*1, A1	.FALSE.	A_in contains <124>; this bit pattern ends in 0.
T	Logical*4, A1	.FALSE.	A_in contains <124><040><040><040>; this bit pattern ends in 0.
29160	Character*5, A5	'29160'	Standard
29160	Character*5, I5	——	Error — Only A edit descriptor handles data declared as character.

Table 6-6. Examples of Output Entities, Edit Descriptors, and Character String Output

Values in Output Entity	Output Entity and Edit Descriptor	Character String Output	Comments
2916	Integer*2, I4	2916	Standard
35	Integer*2, L1	T	L_out contains <000><043>; its rightmost bit is 1.
8	Integer*2, F3.1	8.0	FL(8) is 8.0 which, under F3.1, is the string '8.0'.
21536	Integer*2, A1	T	21536 is <124><040> = 'T□'.
2916.2	Real*4, I4	2916	IN(2916.2) and I4 create the character string '2916'.
16.75	Real*4, L1	F	The resulting integer is 16; its rightmost bit is 0.
7.9	Real*4, F3.1	7.9	Standard
0.0	Real*4, A4	<0><0><0><0>	A_out contains the bit pattern of 0.0. Nulls to @OUTPUT don't affect the cursor.
.FALSE.	Logical, I4	□□□0	L_out contains all 0s.
.TRUE.	Logical, L1	T	Standard
.FALSE.	Logical, F3.1	□.0	The entity contains all 0s, which is true zero as a floating-point number.
.FALSE.	Logical, A1	<0>	The high order byte for .FALSE. in any logical entity is all 0s. Nulls to @OUTPUT don't affect the cursor.
'29160'	Character*5, A5	29160	Standard
'29160'	Character*5, I5	——	Error — only A edit descriptor handles data declared as character.

List-Directed Formatting

List-directed formatting provides easy I/O and is ideal for program interaction with the terminal. You can use list-directed formatting for input or output.

A list-directed I/O statement has an asterisk (*) for a format identifier; for example:

```
READ  (5, *)   B2, B3, B4
WRITE (6, *)   B2, B3, B4
```

or the equivalent with the short form of a READ and PRINT statement:

```
READ  *,   B2, B3, B4
PRINT *,   B2, B3, B4
```

A list-directed format transfers record values to and from iolist entities according to the iolist-entity data type — freeing you from concern with format syntax. (However, see the last paragraph in this section.) In the examples above, F77 would expect to read values that are real (by the name rule) from the terminal.

List-directed formatting makes terminal I/O quite easy; for example:

```
WRITE (*, *)   'Type 3 real values. '
READ  (*, *)   B2, B3, B4
WRITE (*, *)   'You typed ',   B2, B3, B4
```

The dialog from these three statements might appear this way on a terminal:

Type 3 real values. 3 4.5, 999 ↵

You typed 3. 4.5 999.

The short form of a READ implies unit 5; a PRINT implies unit 6. The WRITE statement specifies unit 6; the first asterisk in the READ statement specifies unit 5. Unit 6 is preconnected with FORTRAN carriage control, but this is not really important for list-directed formatting. That's because F77 outputs a blank before each record and the blank serves for carriage control. FORTRAN carriage control allows the prompt to appear on the same line as the response. This means that an extra blank line may appear after the response.

As with edit-directed formatting, records written via list-directed formatting are formatted records. On input, F77 reads until it has acquired values for all iolist entities; on output, it writes until it has written values from all iolist entities.

But unlike edit-directed formatting, the *value separators* divide the values in each list-directed record. (In edit-directed formatting, there are no value separators; F77 simply reads or writes the number of characters given in the format.) A value separator is one of the following:

- one or more contiguous blanks between values or following the last value, *except for* blanks within character strings or blanks between the parts of a complex value. Leading blanks in a record are not treated as value separators.
- a comma, optionally preceded and/or followed by one or more contiguous blanks.
- a slash, optionally preceded and/or followed by one or more contiguous blanks.

F77 always writes a blank as the first character of each output record and writes two blanks as a separator between numeric values. The way the blank in the first position is treated depends on the unit's carriage-control property. The preceding example and sample dialog show how F77 handles blanks on input and output via list-directed formatting.

Although records written via list-directed formatting are formatted, reading them back requires care. It will read *numeric* records back identically via list-directed reads but will not read character values back unless you embed character delimiters in the value, as described later. Also, records written via list-directed formatting may vary widely in length. Thus, aside from terminal I/O, you may want to avoid reading records written via list-directed formatting.

List-directed formatting treats logical*1/byte data as character*1 data. For example, the output from

```
BYTE VAL1 / 'A'    /
BYTE VAL2 / 65     /
BYTE VAL3 / .TRUE. /
PRINT *, '*', VAL1, '*', VAL2, '*', VAL3, '*'
```

is

A*A*

The bit patterns for the letter A and the integer 65 are identical in a byte. .TRUE. is stored as 8 1s in VAL3. Since this bit pattern forms a nonprinting character, the output ends with two consecutive asterisks.

List-Directed Input

On each READ statement, F77 attempts to read values until it acquires a value for each iolist entity. Ignoring leading blanks, it assigns the first value (up to the first separator) to the first iolist entity, then assigns the next value to the second iolist entity, and so on.

For numeric input, the value separators are: one or more contiguous blanks, *one* comma surrounded by zero or more contiguous blanks, or one or more NEW LINEs. (NEW LINE does not terminate the READ statement unless it follows the input values for all iolist entities.) For example:

```
WRITE (*, *)   'Type 3 integer values. '
READ  (*, *)   J5, J6, J7
...
```

Any of the following sequences would assign values to the three J entities:

1,2,3 ↵

or

1□2□3 ↵

or

1,□□□2 ↵
3↵

or

1, ↵
2,□□□ ↵
3 ↵

Character input requires apostrophes or quotation marks to start and end the character string. You'll learn about this soon in the section "Character Input".

Repeat Count

You can enter a repeat count, to assign a value to more than one entity, via the form `r*c`, where `r` is an integer repeat count and `c` is the constant to assign. This is handy for arrays; for example:

```
DIMENSION RRY (100)
CHARACTER*10 CHN(20)
...
PRINT *,  'Enter values for RRY(100). '
READ *,   RRY
PRINT *,  'Enter char. values for CHN(20). '
READ *,   CHN
...
```

Dialog at the terminal might be

Enter values for RRY(100). `25*50.0, 75*2.` ↵

Enter char. values for CHN(20). `20*'abcd'` ↵

This assigns the first 25 elements of RRY the value 50.0, assigns the next 75 elements of RRY the value 2.0, and assigns all 20 elements of CHN the value abcd□□□□□□. Note how the terminal operator enclosed `abcd` with apostrophes.

Nulls and Slashes

A null value tells F77 to leave the value of the corresponding iolist entity as is. You can specify a null value for an entity via a comma or via a repeat count with an empty constant (`r*`). For example:

```
    CHARACTER*20  CHZ
    DIMENSION RXY (100)
    ...
C   Program assigns values to
C      CHZ and RXY elements.
    ...
    PRINT *, 'Current value of CHZ is ', CHZ
    PRINT *, 'Type new value or null.'
    READ  *, CHZ
    PRINT *, 'Current elements of RXY are: ', RXY
    PRINT *, 'Enter new RXY values or nulls.'
    READ  *, RXY
    ...
```

Dialog at the terminal might be

Current value of CHZ is abc
Type new value or null. , ↵

Current elements of RXY are: 1.10 13.4
457.6 2.63E20 ... ...
Enter new RXY values or nulls. `100*` ↵

In the dialog, the null values entered preserve the current values of both the character entity and real array.

While assigning values to more than one iolist entity, you can specify a null for the first value via *one* comma as shown above. For any subsequent value, use two successive commas to specify a null. For example:

```
READ (*, *)   J, K, L
```

If you type	Values in J, K, L will be:
1, 2, 3 ↵	J=1, K=2, L=3
, 2, 3 ↵	J is unchanged, K=2, L=3
1 , , 3 ↵	J = 1, K is unchanged, L = 3
1, 2, , ↵	J = 1, K = 2, L is unchanged.

As for edit-directed formatting, F77 tries to assign values to each iolist entity before it terminates the statement. But if F77 encounters a slash (/) on list-directed input, it skips the rest of the iolist entities and terminates the READ statement. The iolist entities skipped retain the values they had before F77 encountered the slash. For example:

```
N = 888
RX = 45.67
RZ = 377.4
PRINT *,   'Type values for N, RX, RZ.'
READ (*, *)    N, RX, RZ
WRITE (*,*)   'N=',N,' RX=',RX,' RZ=',RZ
...
```

At the terminal, you could enter a slash at any point:

Type values for N, RX, RZ. 0 / ↵

N= 0 RX=45.67 RZ=377.4

N received the new value 0 but the slash then terminated input; RX and RZ retained their old values and program execution continued.

Data Types

Generally, the rules for integer, real, double-precision and logical input are the same as for edit-directed formatting — the value read must match the iolist data type — but one or more blanks signal the end of the value. When F77 has read a value into each entity, or encountered a slash, the I/O statement terminates. Values unused in the record are ignored. For example:

```
PRINT *,   'Type 2 integer values.'
READ *,    J8, J9
...
```

F77 will accept only two (integer) values and ignore others. If you type 1 2 3 ↵, J8 will equal 1, J9 will equal 2, and F77 will ignore the 3. The next READ statement will start a new record and the 3 will be lost.

Values for complex data differ from those for edit-directed formatting. The input value must have a left parenthesis, a numeric value, a comma, another numeric value, and a right parenthesis. One iolist entity receives the value and it must be type complex. One or more blanks around either parenthesis or the comma are ignored; they are not treated as separators. The end of a record may occur between the real part and the comma or between the comma and the imaginary part. The real and imaginary parts themselves must be suitable for type real input. For example:

```
COMPLEX CMPX
PRINT *, 'Enter a complex value. '
READ (*, *) CMPX
...
```

At the terminal, you might type *any* of the following sequences to assign the complex value:

```
( 3.E2, 5.) ↵
```

or

```
(3.E2 ↵
,5.) ↵
```

or

```
( 3.E2, ↵
5. ) ↵
```

Character Input

For list-directed input of type character, the character string read must start and end with apostrophes or quotation marks (unlike the A edit descriptor). F77 transfers all characters within the apostrophes or quotation marks without interpreting them — embedded blanks, commas, and slashes do not act as separators within character strings. The special angle-bracketed mnemonics (e.g., <NL>) receive no special handling on input; they too are transferred literally. NEW LINE characters within character strings *are ignored*. For example, F77 would read each of the following character strings as ABCD□□□/EFGH:

```
'ABCD□□□/EFGH'
```

or

```
'ABCD□ ↵
□□/EFGH'
```

Depending on the character length given when the iolist entity was declared, F77 transfers the leftmost characters read, either truncating the value to fit or filling it on the right with blanks. It does *not* transfer the delimiting apostrophes or quotation marks to the iolist entity. For example:

```
CHARACTER*6   CHP   ! Length of 6.
...
PRINT *,  'Type 6 chars. in quotation marks.'
READ (*, *)   CHP
...
```

If, when it runs, you type	Entity CHP will receive the value
'A' ⤸	A□□□□□
'ABC' ⤸	ABC□□□
'ABCDEFG' ⤸	ABCDEF
'A ⤸ BCDEF' ⤸	ABCDEF
'A ⤸ B,,□□' ⤸	AB,,□□

(Regardless of the PAD= setting given on the OPEN statement, F77 does not pad input records on list-directed reads. This is most relevant for character strings that contain NEW LINE characters. In the last line of the example above, if padding had occurred, the value read into CHP would have been A□□□□□□ instead of AB,,□□.)

F77 does not output the character delimiters on a list-directed write, but requires them on a list-directed read. Thus you cannot do a list-directed read directly on a character string written with a list-directed write. On the write, you must insert an extra pair of delimiters as part of the iolist entity so that the value read will have character delimiters. For example:

```
CHARACTER*10 CFOO    / '"abcde"'/
CHARACTER*10 CFOOX
...
OPEN (2, FILE='FOO')
WRITE (2, *) CFOO  ! 7 chars. "abcde" written
REWIND 2
READ (2, *) CFOOX  ! 5 chars. abcde read
...
```

Here, the string in CFOO is written to the record as "abcde", allowing it to be read later via the list-directed read. The value read into CFOOX is abcde. If the inner delimiters in CFOO had been omitted, the string would have been written to the record as abcde — and the list-directed read would have failed because there were no character delimiters in the record.

Another example, which shows list-directed input of a character string from the terminal, is

```
CHARACTER*10 CC   / 'List' /
...
PRINT *,   'Enter "Fortran" or / '
PRINT *,   '   for default cc property.'
READ *,    CC
OPEN (3, CARRIAGECONTROL = CC,...)
...
```

When this runs, the terminal will display:

Enter "Fortran" or /
for default cc property.

If you type "Fortran" ⤸, the value in CC will be Fortran□□□; if you type ⤸ or , ⤸, F77 retains the original value of CC — List□□□□□□. Unit 3 will then be opened with the given property. The A edit descriptor is easier to use for this kind of thing — but the examples make the point.

List-Directed Output

On each WRITE or PRINT statement, F77 writes an entire record consisting of the iolist entities. It writes more than one record if the characters written exceed the maximum record length (MAXRECL, 136 bytes by default for data-sensitive files).

F77 always writes one blank at the beginning of each record and two blanks between each pair of numeric values. It inserts no blanks between a character value and any other value.

The leading blank will be used for carriage control if the unit was opened with FORTRAN carriage control. Otherwise, with the default LIST carriage control, the leading blank will print as a blank. (Unit 6, * specifier, is preconnected with FORTRAN carriage control, and unit 10 is preconnected with LIST carriage control.)

For fancier output, you can insert any of the special control character mnemonics described in Chapter 2; e.g.,

```
PRINT *,   '<FF><BEL> Attention!<NL>'
```

For numeric values, F77 outputs only as many numbers as needed; it does not output trailing zeros to the right of the decimal point. For example,

```
PRINT *,   02.04, 1.2000
```

prints as 2.04□□1.2.

If the iolist entity is type integer, F77 uses the form of the Iw descriptor, with w determined by the number of digits. If the entity is type real or double precision, F77 uses a variation of generalized format Gw. Gw is format F for numbers .1 through 10**6, format E otherwise. (But, unlike Gw, 0 prints in F format via list-directed formatting.)

If the iolist entity is type logical, F77 uses the form of the L descriptor. It outputs T for the value .TRUE. and F for the value .FALSE..

If the iolist entity is type complex, F77 outputs it as a left parenthesis, the real part of the value, a comma, the imaginary part of the value, and a right parenthesis. F77 does not break a complex entity between records unless it will not fit in one record. For example:

```
COMPLEX   CMPX2   / (3., .6) /
PRINT *,  CMPX2
```

prints as

(3.,.6)

If an iolist entity is type character or a character constant, F77 outputs it without the delimiting (outer) apostrophes or quotation marks, as described under "Character Input".

If the iolist entity is an unsubscripted array name, F77 outputs all the array elements in the traditional column/row order. If the array values overflow the maximum record length, F77 continues on the next line until it has output all the elements. For example, assume array JARRY with 140 elements, where the values run sequentially from 1 through 140. Also assume the terminal — which as a data-sensitive file has a maximum record length of 136 — as the output file:

```
      DIMENSION JARRY(140)
      DO 300   I = 1, 140
      JARRY (I) = I
300   CONTINUE
      PRINT *,   JARRY
      ...
```

F77 would output the JARRY values as:

```
1□2□3□4□5□6□7... 36
37□38□39□40□ ... 69
... ... ... ... ... ... ... ... ... ...
... 137□138□139□140
```

(On a terminal screen the lines would wrap around after 80 characters.)

Summary of Rules for Formatted I/O

This section summarizes the rules for edit-directed formatting (general, input, and output) and then gives the rules for list-directed formatting (general, input, and output).

Edit-Directed Formatting

The general rules for edit-directed formatting are

1. Each I/O statement must include a format identifier.
2. A format identifier can be a character expression (entity name or character constant like '(1X,F9.2)') or the name of an array that contains the format in Hollerith, within the I/O statement. Or, the identifier can be the label of a separate FORMAT statement. If it's a character constant, you must enclose the constant in apostrophes or quotation marks and in parentheses. If the identifier is the label of a FORMAT statement, the FORMAT statement must specify the format and must be in the current program unit.
3. There are short forms of READ and PRINT statements that do not use a control information list (cilist). In these, the format identifier must be the first argument to the statement. The general form of READ and every form of WRITE must include a cilist in parentheses. The cilist must include a unit specifier and format identifier; it can include other specifiers. You can precede the format identifier with FMT=, or you can omit FMT= if the identifier is the second argument in the cilist and if you omitted UNIT= from the first argument.
4. If a formatted I/O statement contains an I/O list (iolist), its format specification must include at least one repeatable edit descriptor: I, O, Z, B, F, E, D, G, L, or A. If not, F77 will signal an error.
5. You may not precede a nonrepeatable edit descriptor with a repeat count (r). The nonrepeatable descriptors are P, S-series, BZ/BN, apostrophe or quoted string, H, X, T-series, $, colon, and slash. These descriptors may, however, appear within a group in parentheses that is repeated.
6. Repeat counts (r), field width specifiers (w), and decimal positions (d) in edit descriptors must all be unsigned integer constants or expressions that evaluate to integers. The field width w cannot exceed 255. d cannot be larger than w.
7. An iolist entity's corresponding edit descriptor must come from a class of allowable edit descriptors. The /FMTMM F77LINK switch expands the class for an iolist entity, thus eliminating many entity/descriptor mismatch error messages at runtime.

Edit-Directed Input

1. The data type of an iolist entity must match the kind of value read into it from the record. F77 will make what appears to be an integer value into a real value under format control; but in all other mismatch situations (e.g., real value→integer entity), F77 by default will signal a runtime error. You can override this error in many mismatch situations with the /FMTMM (ForMaT MisMatch) F77LINK switch.

2. F77 ignores leading blanks before numeric values. Each embedded and trailing blank has a value of null by default; it has a value of 0 if you opened the file with BLANK='ZERO' or if the BZ descriptor is in effect. An input value of blanks has a value of 0.
3. A decimal point in a real value overrides the decimal point position specified by d of an edit descriptor.
4. If the width (w) given in an edit descriptor is greater than the number of characters remaining in the record, the results may be undesirable. If the file was opened with PAD='YES', the characters will be padded with blanks up to the maximum record length. If the file was opened with PAD='NO' (default for user opens) and the remaining characters in the record are less than w, F77 will signal an "End of record" error.
5. If the field width (w) given in an edit descriptor is smaller than the number of characters remaining in the record, F77 reads w characters into the iolist entity.
6. An *exponent* in an input value does not override the decimal point implied by a decimal position descriptor (d); however, the exponent does override any given scale factor.

Edit-Directed Output

1. For all numeric editing, the field width (w) must be large enough to accommodate a sign (if needed) and the value in the iolist entity or F77 will output a field of asterisks.

   For real numeric editing, other items that may require character positions in field w are a decimal point and an exponent. If w is too small for these and they are needed, F77 will output a field of asterisks.

   If the field width (w) is greater than required by the value, sign, exponent, etc., F77 right justifies the numbers, precedes them with the appropriate number of blanks, and outputs them.
2. For real numeric editing, the field width (w) must be greater than the decimal position (d) or F77 will signal an error.
3. For real numeric editing: suppose the number of digits following the decimal point in an iolist value is greater than d in the pertinent edit descriptor. Then, F77 outputs d digits to the right of the decimal point, rounding the extra numbers at the d position.
4. For A editing, if *w* is not large enough for the number of characters in the iolist entity, F77 outputs the leftmost *w* characters, inserting blanks on the right.
5. Files with data-sensitive records (default) can be printed directly. If they were opened with CARRIAGECONTROL='FORTRAN', the first character of each record will be used for carriage control. If they were opened with default LIST carriage control, F77 will supply a NEW LINE character after each record, unless you suppress this with the $ descriptor.

   Files whose records are not data sensitive usually cannot be printed directly. The F77PRINT utility can make a printable copy of such files (or data-sensitive files), with the carriage control specified on the files' OPEN statements. F77PRINT will also make the files suitable for text editing.

List-Directed Formatting

The general rules of list-directed formatting are

1. You specify list-directed formatting by using an asterisk (*) as a format identifier in the I/O statement.
2. One or more blanks and/or commas between values are used as value separators.
3. Under AOS/VS, F77 treats logical*1/byte data as character*1 data.

List-Directed Input

1. The iolist entity data type must match the value read from the record.
2. F77 ignores leading blanks. Commas, blanks, and/or NEW LINEs *between* values read act as value separators.
3. You can enter a repeat count for a value via the form r*c, to assign value c to r iolist entities.
4. A null value tells F77 to leave the original iolist entity as is. One comma specifies a null value as the first value read; two successive commas specify a null value for the second, third, etc., value read.
5. A slash terminates a read, effectively assigning null values to iolist entities that have not received values before the slash.

6. Integer, real, and double-precision values are read into iolist entities much as they are for edit-directed formatting. Complex values must have the form **(value,value)**.

7. Character values must have the form **del string del**, where **del** is an apostrophe or quotation mark. F77 reads blanks, commas, and/or slashes within character strings without interpretation, storing them in the pertinent iolist entity.

List-Directed Output

1. F77 always writes one blank at the beginning of each record and two blanks between each pair of numeric values. It inserts no blanks between a character value and any other value.

2. F77 outputs no more numbers than needed for a numeric iolist entity.

3. F77 outputs character values without the outer string delimiters (apostrophes or quotation marks).

4. If the number of characters output would exceed the maximum record length (as with all of an array's elements), F77 starts a new output record and proceeds until it has output all the values.

End of Chapter

Chapter 7
Functions and Subroutines

Functions and subroutines allow you to create and use procedures for your own needs. They help provide structured application programs where each program unit does its own job.

F77 has three kinds of functions and one kind of subroutine. They are

- Intrinsic Functions, supplied with F77. These perform trigonometric, conversion, logarithmic, and other operations.
- Statement Functions, single statements that you provide.
- Function Subprograms, also called external functions. You write these as separate program units and make reference to them from another program unit; they return their own value to the calling unit.
- Subroutine Subprograms or Subroutines. These are also separate program units but do not return their own value to the calling unit.

This chapter describes these and their associated statements and arguments. Then it covers statements that relate to all multiple unit applications: ENTRY, RETURN, SAVE, and EXTERNAL. Next it explains common storage — a vehicle for communication between program units — and it ends with the BLOCK DATA statement and a summary.

The major sections proceed:

- Intrinsic Functions
- Subroutine MVBITS
- System Functions
- Statement Functions
- Function Subprograms (includes FUNCTION)
- Arguments to Function and Subroutine Subprograms (includes INTRINSIC)
- Subroutine Subprograms (includes SUBROUTINE and CALL)
- Using ENTRY to Define a Procedure within a Subprogram
- Using SAVE to Preserve Subprogram Entities
- Using RETURN to Return to the Calling Unit
- Using EXTERNAL to Identify a Subprogram Name
- Using PROGRAM to Name the Main Program
- Using COMMON Storage
- BLOCK DATA Subprograms
- Summary of Rules for Functions and Subroutines

NOTE: Certain symbols have restrictions because of MP/AOS and MP/AOS-SU requirements. In particular, external names — such as COMMON block and subprogram names — must be unique with respect to each other and within their first 10 characters. For example, you cannot have subroutines SUBROUTINE_1 and SUBROUTINE_2 or labelled COMMON names COMMON_AREA_A and COMMON_AREA_B in one or more program units. The system linker program (BIND.PR) causes this restriction.

Intrinsic Functions

F77 includes a set of intrinsic (built-in) functions. To use an intrinsic function, you specify its name in an arithmetic expression, followed by the arguments on which you want it to act. For example, the statement `Y=SIN(X)` invokes the intrinsic function `SIN` to compute the sine of `X`, then assigns this value to `Y`. `PRINT *, SIN(X)` computes the sine of `X` and prints it on the terminal.

To refer to an intrinsic function, use the form:

`fref ( arg` *[,arg]* ... `)`

where:

`fref` is an intrinsic function name. The intrinsic function names are covered in this section. These functions are built into F77; an IMPLICIT statement does not change the data type of an intrinsic function.

`arg` is an argument on which you want the function to operate. This can be any expression. Some intrinsic functions accept only one argument, others two or more. Use a comma to separate multiple arguments. Frequently, an argument must be a certain data type.

There is a *generic name* for most sets of intrinsic functions. If you use the generic function name, F77 will scan the data type of the arguments and select the proper *specific name*. Using a generic name simplifies function references because you can use the function name with more than one type of argument.

For either generic or specific functions that take multiple arguments, all arguments must be of the appropriate data types. If not, the compiler will signal an error. For example:

```
INTEGER*2  ITWO, ITWO2
INTEGER*4  IFOUR
REAL   RX
...
IFOO = MOD(ITWO, ITWO2)   ! OK.
IFOO = MOD(ITWO, IFOUR)   ! Error.
FOO = MOD(RX, 5.)         ! OK.
FOO = MOD(RX, 5)          ! Error.
```

The compiler would reject the two erroneous function references in the previous example because the arguments are of differing data types. F77 will *not* convert incorrectly typed arguments. If need be, you can use a conversion function to produce the correct argument type(s).

You can't use Hollerith constants as arguments to any intrinsic functions. You can use character constants as arguments only to intrinsic functions that expect character arguments. Several examples are:

Valid	Invalid	Because
	`Y = SIN(1HA)`	Hollerith constants can never be arguments to any intrinsic function.
	`Y = SIN('A')`	The generic SIN function requires a real or complex variable or constant as its argument.
`ILEN = LEN('ABCDEF')`		The generic LEN function can handle a character constant. (ILEN contains 6.)
`CHARACTER*5 CSTRING /'AB'/` `  ILEN = LEN(CSTRING)`		The generic LEN function can handle a character variable. (ILEN contains 5.)
	`J = IBCLR('ABCD',2)`	The generic IBCLR function requires an integer variable.

`INTEGER*4 CHAR4 /'ABCD'/`
`INTEGER*4 J, C2 /2/`
`J = IBCLR(CHAR4,C2)`
		CHAR4 is an integer argument (J contains 'ABC@' because its right byte is <100>; the right byte of CHAR4 contains <104>).

If a function reference fref includes assignment to an entity (e.g., RFOO=SIN(X)), the data type of the entity on the left of the equals sign should be of the proper data type. If the types do not match, F77 will convert the derived value before assigning it according to the rules for any assignment statement (described in Chapter 3, Table 3-1). But if the types are incompatible (e.g., character-integer), F77 will report an error.

When you use a logical*1/byte entity as an argument to an intrinsic function that expects an integer, F77 converts the entity to an equivalent integer of the default length (generic name) or of the expected length (specific name). When you use a logical*1/byte entity as an argument to an intrinsic function that expects a character argument, F77 interprets the entity as character.

If you want to use an intrinsic function name as an argument to a subprogram, use the specific name. The program unit that refers to the function must include the specific name in an INTRINSIC statement; for example:

```
INTRINSIC  DSIN  ! Specific name.
...
PRINT *,  DP_FUNC (DSIN, X)  ! Get DSIN of X.
```

The following tables show all F77 intrinsic functions by category. Table 7-1 contains the trigonometric functions, Table 7-2 the arithmetic and conversion functions, Table 7-3 the lexical functions, Table 7-4 the word and bit functions, and Table 7-5 the system functions. Within each table, the functions proceed alphabetically by generic name. And, "Int, def len" in these five tables means the default integer length — 4 bytes unless the F77 command includes the /INTEGER=2 switch. The F77 macro supplied with AOS, F7716, MP/AOS, and MP/AOS-SU FORTRAN 77s includes this switch.

Trigonometric Intrinsic Functions

Table 7-1 shows the F77 trigonometric intrinsic functions. All angular quantities are expressed in radians. To convert from degrees to radians or vice versa, use the formula

pi radians = 180 degrees

Table 7-1. Trigonometric Intrinsic Functions

Generic Name	Meaning	No. of Args	Specific Name	Argument Data Type	Result Data Type	Function Reference Example
ACOS	Arccosine. Absolute value of argument must be LE to 1. Result ranges from 0 through pi.	1	ACOS DACOS	Real*4 Double	Real*4 Double	X=ACOS(arg)
ASIN	Arcsine. Absolute value of argument must be LE to 1. Result ranges from -pi/2 through pi/2.	1	ASIN DASIN	Real*4 Double	Real*4 Double	X=ASIN(arg)
ATAN	Arctangent. Result ranges from -pi/2 through pi/2.	1	ATAN DATAN	Real*4 Double	Real*4 Double	X=ATAN(arg)
ATAN2	Arctangent, arg1/arg2; result ranges from -pi through pi. See note.	2	ATAN2 DATAN2	Real*4 Double	Real*4 Double	X=ATAN(arg1,arg2)
COS	Get cosine. Absolute value of COS and DCOS argument need not be LT 2*pi.	1	COS DCOS CCOS DCCOS	Real*4 Double Complex Complex*16	Real*4 Double Complex Complex*16	X=COS(arg)
COSH	Hyperbolic cosine.	1	COSH DCOSH CCOSH DCCOSH	Real*4 Double Complex Complex*16	Real*4 Double Complex Complex*16	X=COSH(arg)
SIN	Sine. Absolute value of SIN and DSIN arguments need not be LT 2*pi.	1	SIN DSIN CSIN DCSIN	Real*4 Double Complex Complex*16	Real*4 Double Complex Complex*16	X=SIN(arg)
SINH	Hyperbolic sine.	1	SINH DSINH CSINH DCSINH	Real*4 Double Complex Complex*16	Real*4 Double Complex Complex*16	X=SINH(arg)
TAN	Tangent. Absolute value of TAN and DATAN arguments need not be LT 2*pi.	1	TAN DTAN CTAN DCTAN	Real*4 Double Complex Complex*16	Real*4 Double Complex Complex*16	X=TAN(arg)
TANH	Hyperbolic tangent.	1	TANH DTANH CTANH DCTANH	Real*4 Double Complex Complex*16	Real*4 Double Complex Complex*16	X=TANH(arg)

Note: For arctangent, ATAN2 or DATAN2. If arg1 is positive, the result is positive. If arg1 is negative, the result is negative. If arg1 is 0 and arg2 is positive, the result is 0; if arg1 is 0 and arg2 is negative, the result is pi. Always if arg2 is 0, the absolute value of the result is pi/2. Both arguments must not be 0.

Arithmetic and Conversion Intrinsic Functions

Table 7-2 has the arithmetic and conversion functions. The term "Int, def len" means integer, default length. The default integer length is 4 bytes for AOS/VS and 2 bytes for AOS, F7716, MP/AOS, and MP/AOS-SU.

"Real/Real*4" stands for the standard 4-byte real type. "Dble" means Double Precision. "Dble/Real*8" are functionally identical — each producing a double-precision real entity.

The notes mentioned in Table 7-2 appear after the table.

Table 7-2. Arithmetic and Conversion Intrinsic Functions

Generic Name	Meaning	Num. of Args.	Specific Name	Argument Data Type (* bytes if nonstandard)	Result Data Type (* bytes if nonstandard)	Function Reference Example
ABS	Absolute value. For complex, see note 1.	1	ABS IABS IABS2 IABS4 DABS CABS DCABS	Real*4 Int, def len Integer*2 Integer*4 Double Complex Complex*16	Real*4 Int, def len Integer*2 Integer*4 Double Real*4 Double	R=ABS(arg)
AINT	Integral value.	1	AINT DINT	Real*4 Double	Real*4 Double	R=AINT(arg)
AIMAG	Imaginary part of complex argument. See note 1.	1	AIMAG DIMAG	Complex Complex*16	Real*4 Double	R=AIMAG(arg)
AMAX0	Largest value of 2 or more arguments.	2 or more	— —	Integer(any)	Real*4	R=AMAX0(arg1,arg2 *[,...argN]*)
AMIN0	Smallest value of 2 or more arguments.	2 or more	— —	Integer(any)	Real*4	R=AMIN0(arg1,arg2 *[,...argN]*)
ANINT	Round to nearest integral value.	1	ANINT DNINT	Real*4 Double	Real*4 Double	R=ANINT(arg)
AREAL	Real part of complex argument.	1	AREAL DREAL	Complex Complex*16	Real*4 Double	R=AREAL(arg)
CABS	Complex modulus: SQRT(REAL(arg)**2 + IMAG(arg)**2); see note 1.	1	CABS DCABS	Complex Complex*16	Real*4 Double	R=CABS(arg)

(continues)

Table 7-2. Arithmetic and Conversion Intrinsic Functions

Generic Name	Meaning	Num. of Args.	Specific Name	Argument Data Type (* bytes if nonstandard)	Result Data Type (* bytes if nonstandard)	Function Reference Example
CEIL	Least integer value greater than or equal to argument.	1	CEIL DCEIL	Real*4 Double	Real*4 Double	R=CEIL(arg)
CEXP	Exponential: *e***argument. See note 1.	1	CEXP DCEXP	Complex Complex*16	Complex Complex*16	Z=CEXP(arg)
CHAR	Get character whose ASCII value is passed in argument. See note 4.	1	— —	Integer(any)	Character	CH=CHAR(arg)
CMPLX	Convert one or two integer or real arguments to complex. Convert a dble/real*8 or complex*16 argument to complex. See note 6.	1 or 2	— —	Integer(any) Real*4 Double Complex*16	Complex Complex Complex Complex	Z=CMPLX(arg1 *[,arg2]*) Z=CMPLX(arg)
CLOG	Natural logarithm: $\log_e$ argument. See notes 1 and 2.	1	CLOG DCLOG	Complex Complex*16	Complex Complex*16	Z=CLOG(arg)
CONJG	Conjugate of complex argument: CMPLX(REAL(argument) -IMAG(argument)). See note 1.	1	CONJG DCONJG	Complex Complex*16	Complex Complex*16	Z=CONJG(arg)
CSQRT	Square root. See note 3.	1	CSQRT DCSQRT	Complex Complex*16	Complex Complex*16	Z=CSQRT(arg)
DBLE	Convert an integer, real, or complex value to double precision (real*8). See note 7.	1	—	Any numeric type	Double	DP=DBLE(arg)
DCMPLX	Convert one or two integer, real, or dble/real*8 values to complex*16. Convert one complex value to complex*16. See note 6.	1 or 2 1	— — — — —	Integer(any) Real*4 Double Complex Complex*16	Complex*16 Complex*16 Complex*16 Complex*16 Complex*16	Z=DCMPLX(arg1 *[,arg2]*) Z=DCMPLX(arg)
DIM	Positive difference: arg1 – arg2 if arg1 GT arg2; 0 if arg1 LE arg2.	2	DIM IDIM IDIM2 IDIM4 DDIM	Real*4 Int, def len Integer*2 Integer*4 Double	Real*4 Int, def len Integer*2 Integer*4 Double	R=DIM(arg1, arg2)
DPROD	Compute double precision product of 2 real values.	2	DPROD	Real*4	Double	DP=DPROD(arg1, arg2)

(continued)

Table 7-2. Arithmetic and Conversion Intrinsic Functions

Generic Name	Meaning	Num. of Args.	Specific Name	Argument Data Type (* bytes if nonstandard)	Result Data Type (* bytes if nonstandard)	Function Reference Example
EXP	Exponential: *e***argument.	1	EXP DEXP CEXP DCEXP	Real*4 Double Complex Complex*16	Real*4 Double Complex Complex*16	R=EXP(arg)
FLOAT	Convert integer argument to real. See note 8.	1	— —	Integer(any)	Real*4	R=FLOAT(arg)
FLOOR	Obtain greatest integral value less than or equal to argument.	1	FLOOR DFLOOR	Real*4 Double	Real*4 Double	R=FLOOR(arg)
FRAC	Obtain fractional part of a real or double-precision number.	1	FRAC DFRAC	Real*4 Double	Real*4 Double	R=FRAC(arg)
IABS	Absolute value.	1	IABS IABS2 IABS4	Int, def len Integer*2 Integer*4	Int, def len Integer*2 Integer*4	J=IABS(arg)
ICHAR	Get the integer ASCII value of the character passed in argument. See note 5.	1	ICHAR ICHAR2 ICHAR4	Character Character Character	Int, def len Integer*2 Integer*4	J=ICHAR(arg)
IDNINT	Obtain the nearest integer from real or double-precision argument.	1	IDNINT IDNINT2 IDNINT4	Double Double Double	Int, def len Integer*2 Integer*4	J=IDNINT(arg)
INDEX	Within character entity argument1, return character position at which character entity argument2 begins. Argument1 is entity to be searched; argument2 is entity to search for. No match returns 0. If argument2 is longer than argument1, also returns 0.	2	INDEX INDEX2 INDEX4	Character Character Character	Int, def len Integer*2 Integer*4	J=INDEX(arg1, arg2)
IDINT	Convert a dble/real*8 value to integer. See note 9.	1	—	Double	Int, def len	J=IDINT(arg)
IFIX	Convert a real/real*4 value to integer. See note 9.	1	—	Real*4	Int, def len	J=IFIX(arg)

(continued)

Table 7-2. Arithmetic and Conversion Intrinsic Functions

Generic Name	Meaning	Num. of Args.	Specific Name	Argument Data Type (* bytes if nonstandard)	Result Data Type (* bytes if nonstandard)	Function Reference Example
INT	Convert a numeric value to integer. See note 9.	1	—	Any numeric type	Int, def len	J=INT(arg)
INT2	Convert a numeric value to integer*2. See note 9.	1	—	Any numeric type	Integer*2	J=INT2(arg)
INT4	Convert a numeric value to integer*4. See note 9.	1	—	Any numeric type	Integer*4	J=INT4(arg)
ISIGN	Transfer sign.Return argument1 if argument2 GT 0; return -argument1 if argument2 LT 0. If argument1 is 0, return 0.	2	ISIGN ISIGN2 ISIGN4	Int, def len Integer*2 Integer*4	Int, def len Integer*2 Integer*4	J=ISIGN(arg1, arg2)
LEN	Return the number of characters in character entity.	1	LEN LEN2 LEN4	Character Character Character	Int, def len Integer*2 Integer*4	J=LEN(arg)
LOG	Natural logarithm: $\log_e$) argument. Argument to ALOG or DLOG must be GT 0.	1	ALOG DLOG CLOG DCLOG	Real*4 Double Complex Complex*16	Real*4 Double Complex Complex*16	R=ALOG(arg)
LOG10	Common logarithm: $\log_{10}$ argument. Argument must be GT 0.	1	ALOG10 DLOG10	Real*4 Double	Real*4 Double	R=ALOG10(arg)
LOGICAL1	Get a logical value, a single character, or an integer between -128 and +127 from the argument. (AOS/VS only)	1	— —	Any logical or byte Any numeric type	Logical*1 Logical*1	L=LOGICAL1(arg)

(continued)

Table 7-2. Arithmetic and Conversion Intrinsic Functions

Generic Name	Meaning	Num. of Args.	Specific Name	Argument Data Type (* bytes if nonstandard)	Result Data Type (* bytes if nonstandard)	Function Reference Example
LOGICAL2	Get logical value of argument: .TRUE. or .FALSE..	1	— —	Logical*2 Logical*4	Logical*2 Logical*2	L=LOGICAL2(arg)
LOGICAL4	Get logical value of argument: .TRUE. or .FALSE..	1	— —	Logical*2 Logical*4	Logical*4 Logical*4	L=LOGICAL4(arg)
MAX	Choose the largest value of two or more arguments.	2 or more	— — — AMAX1 DMAX1	Int, def len Integer*2 Integer*4 Real*4 Double	Int, def len Integer*2 Integer*4 Real*4 Double	J=MAX(arg1, arg2 *[,...,argN]*)
MAX0	Same as MAX.	2 or more	— —	Integer*2 Integer*4	Ineger*2 Integer*4	J=MAX0(arg1, arg2 *[,...,argN]*)
MAX1	Same as MAX.	2 or more	—	Real*4	Int, def len	J=MAX1(arg1, arg2 *[,...,argN]*)
MIN	Choose the smallest value of two or more arguments.	2 or more	— — — AMIN1 DMIN1	Int, def len Integer*2 Integer*4 Real*4 Double	Int, def len Integer*2 Integer*4 Real*4 Double	J=MIN(arg1, arg2 *[,...,argN]*)
MIN0	Same as MIN.	2 or more	— —	Integer*2 Integer*4	Integer*2 Integer*4	J=MIN0(arg1, arg2 *[,...,argN]*)
MIN1	Same as MIN.	2 or more	—	Real*4	Int, def len	J=MIN1(arg1, arg2 *[,...,argN]*)
MOD	Get remainder: (arg1-(INT(arg1/arg2) * arg2)).arg1 should be GT arg2. If arg1 is 0, result is undefined. See note 9.	2	MOD MOD2 MOD4 AMOD DMOD	Int, def len Integer*2 Integer*4 Real*4 Double	Int, def len Integer*2 Integer*4 Real*4 Double	J=MOD(arg1, arg2)

(continued)

Table 7-2. Arithmetic and Conversion Intrinsic Functions

Generic Name	Meaning	Num. of Args.	Specific Name	Argument Data Type (* bytes if nonstandard)	Result Data Type (* bytes if nonstandard)	Function Reference Example
NINT	Get the nearest integer value from real or dble/real*8 value. Returns INT(argument+.5) if argument GT 0; returns INT(argument-.5) if argument LT 0.	1	NINT NINT2 NINT4 IDNINT IDNINT2 IDNINT4	Real*4 Real*4 Real*4 Double Double Double	Int, def len Integer*2 Integer*4 Int, def len Integer*2 Integer*4	J=NINT(arg)
REAL	Convert integer, real, dble/real*8, or complex value to real. See note 8.	1	— — FLOAT SNGL	Integer(any) Real*4 Integer*4 Double	Real*4 Real*4 Real*4 Real*4	R=REAL(arg)
SIGN	Transfer sign. Return argument1 if argument2 GT 0; return -argument1 if argument2 LT 0. If argument1 is 0, return 0.	2	SIGN ISIGN ISIGN2 ISIGN4 DSIGN	Real*4 Int, def len Integer*2 Integer*4 Double	Real*4 Int, def len Integer*2 Integer*4 Double	R=SIGN(arg1, arg2)
SQRT	Get square root. Argument to SQRT or DSQRT must be GE 0.	1	SQRT DSQRT CSQRT DCSQRT	Real*4 Double Complex Complex*16	Real*4 Double Complex Complex*16	R=SQRT(arg)

(continued)

Table 7-2. Arithmetic and Conversion Intrinsic Functions

Notes

1. AIMAG, CABS, CEXP, CLOG, etc. Express the argument for a complex value as a pair of reals, form **(ar,ai)** where **ar** is the real part and **ai** is the imaginary part. For example, suppose AR=3.0, AI=2.0 complex variable ARG=CMPLX(AR,AI); then R=AIMAG(ARG)=2.0. For another example, suppose AR=3.0, AI=4.0, complex variable ARG=CMPLX(AR,AI); then R=CABS(ARG) = SQRT(AR**2+AI**2) = SQRT(3.0**2+4.0**2) = SQRT(9.0+16.0) = SQRT(25.0) = 5.0.

2. CLOG (complex logarithms). The argument to CLOG must not be (0.,0.). The range of the imaginary part of the result is from -pi through pi. The imaginary part of the result is pi only when the real part of the argument is negative and the imaginary part of the argument is 0; e.g., (-3.,0.) would yield an imaginary result of pi.

3. CSQRT (complex square root). The value of CSQRT(arg) is the principal value with the real part greater than or equal to 0. When the real part of the result is 0, the imaginary part is greater than or equal to 0.

4. CHAR. CHAR returns the character that has that ASCII value of the argument. The argument must contain an integer between 0 and 255 (decimal). CHAR always returns one chararacter. For example, if the argument contains 65, CHAR will return A. Appendix A is a chart that describes all ASCII characters and their octal and decimal codes.

5. ICHAR. ICHAR returns the ASCII decimal value of a character. The argument must contain one character; ICHAR will return its ASCII value, range 0 through 255. For example, if the argument contains 'A', ICHAR will return 65 or if the argument contains 'B', ICHAR will return 66. Appendix A gives the ASCII octal and decimal values of all characters.

   For example:

```
CHARACTER C1 / 'A' /, C2
IF00 = 65 ! or 101K.
C2 = CHAR(IF00)
IF002 = ICHAR(C1)
PRINT *, C2, ' ', IF002
```

   F77 will print A□65 on unit 6.

6. CMPLX and DCMPLX. CMPLX/DCMPLX allow one or two arguments; e.g., **CMPLX(argument)** or **CMPLX(argument1, argument2)**. If you include *one* argument, it can be of type real, double precision, complex, or complex*16. F77 treats **CMPLX(argument)** as **CMPLX(argument,0.)** and treats **DCMPLX(argument)** as if it were **DCMPLX(argument, 0.)**.

   If you include *two* arguments, they must be of the same type and can be type integer, real, or double precision. With two arguments, CMPLX/DCMPLX yields a complex value whose real part is **REAL(argument1)** and whose imaginary part is **REAL(argument2)**. See note 8 for REAL details.

7. DBLE. If your argument is double precision, **DBLE(argument)** yields **argument**. If your argument is type integer or real, **DBLE(argument)** yields as much precision of the significant part of **argument** as will fit in a double-precision entity. If your argument is type complex, **DBLE(argument)** yields as much precision of the *real part* of **argument** as will fit in a double-precision entity.

8. FLOAT, REAL, etc. If your argument is type real, either function yields real. If your argument is type integer or double precision, **FLOAT(argument)/REAL(argument)** yields as much precision of the real part of **argument** as will fit in a real entity. If your argument is type complex, **REAL(argument)** yields as much of the real part of **argument** as will fit in a real entity.

9. INT, IDINT, IFIX, INT2, INT4, MOD — all called **func** here for brevity. If your argument is type integer, **func(argument)** yields type integer. If your argument is type real and its absolute, real value is less than 1, **func(argument)** yields 0. If your argument is type real and its absolute, real value is greater than 1, **func(argument)** yields the largest integer value that does not exceed *the real representation* of **argument**. For example, INT(-3.7) yields integer -3. (Also for example, INT(-2.0) may yield 1 because the decimal representation of fractions is not always exact.) If your argument is type complex, **func(argument)** deals only with the real part, using the same rules as for a type real argument.

(concluded)

Lexical Comparison Intrinsic Functions

The F77 lexical functions compare the values of characters and return a logical value, .TRUE. or .FALSE. . The data type of the arguments to these functions must be character. As mentioned in Chapter 4, DG's F77 allows you to use the arithmetic relational operators (.GT., etc.) instead of the lexical functions — and get identical results. The lexical functions described *here* allow you to write processor-independent code.

All lexical functions require two arguments that are character expressions. If the strings are different lengths, F77 treats the shorter string as if it had been padded on the right with blanks to the length of the longer string.

Word and Bit Functions — Introduction

The word and bit functions allow you to compare, AND, OR, shift, set, extract, and clear bits within integer entities. Word functions process words; bit functions process subsets of words. By default, the word and bit functions are intrinsic (built-in). You may specify external counterparts with the EXTERNAL statement.

Intrinsic Word and Bit Functions

For these functions, you can use the generic function name, or else the explicit function name ending in 2 for 2-byte integers or ending in 4 for 4-byte integers.

For example, suppose you want to set bit number 1 (the second from the right) in integer variable IFOO (default length) to zero. The following statements use the intrinsic function IBSET to do this:

```
INTEGER IFOO, K1
...
K1 = 1
IFOO = IBSET(IFOO, K1)
...
```

External Word and Bit Functions

All external word and bit functions have integer*4 and logical*4 arguments and results. Furthermore, you select an external word/bit function by placing its name in an EXTERNAL statement. This statement must appear in a program unit before any executable statements.

For example, suppose you want to set bit number 1 (the second from the right) in integer*4 variable IFOO to zero. The following statements use the external function IBSET to do this:

```
EXTERNAL IBSET
INTEGER*4 IFOO, K1
...
K1 = 1
IFOO = IBSET(IFOO, K1)
...
```

Table 7-3. Lexical Comparison Intrinsic Functions

Generic Name	Specific Name	Meaning	Num. of Args.	Argument Data Type	Result Type	Function Reference Example
—	LGE	Test for greater than or equal to. If argument1 .GE. argument2, return .TRUE.	2	Character	Logical, declared or default length	L=LGE(arg1, arg2)
—	LGT	Test for greater than. If argument1 .GT. argument2, return .TRUE.	2	Character	Logical, declared or default length	L=LGT(arg1, arg2)
—	LLE	Test for less than or equal to. If argument1 .LE. argument2, return .TRUE.	2	Character	Logical, declared or default length	L=LLE(arg1, arg2)
—	LLT	Test for less than. If argument1 .LT. argument2, return .TRUE.	2	Character	Logical, declared or default length	L=LLT(arg1, arg2)

Word and Bit Functions — Details

Each function treats its arguments as unsigned values; i.e., it interprets the leftmost bit as a value, not a sign indicator. For each function, the bits in each argument are numbered n to 0 where n is the leftmost bit: bit 31 for 4-byte integers; bit 15 for 2-byte integers — as shown in Figure 7-1. Note that the bit numbers are opposite of those in Figure 2-1.

For Boolean operations (AND, OR, inclusive OR, and exclusive OR), F77 proceeds on a bit-by-bit basis. That is, the *n*th bit of argument1 and the *n*th bit of argument2 together determine the *n*th bit of the value returned. All bits of each argument participate in the function.

Note that the length of integer and logical entities used in these functions must be consistent. For example:

```
LOGICAL  LOG
INTEGER*2  IWORD
...
LOG = BTEST(IWORD, 5)
```

IWORD is 2 bytes long, but 5 may be either 2 or 4 bytes long, depending on the default integer length. If the default integer length is 4 bytes, the compiler will signal an error.

Table 7-4 describes the word and bit intrinsic functions. Table 7-5 describes the word and bit external functions. The fourth column of Table 7-5 (headed by "Specific Name") exists solely to show that the external word and bit functions have no length-dependent names.

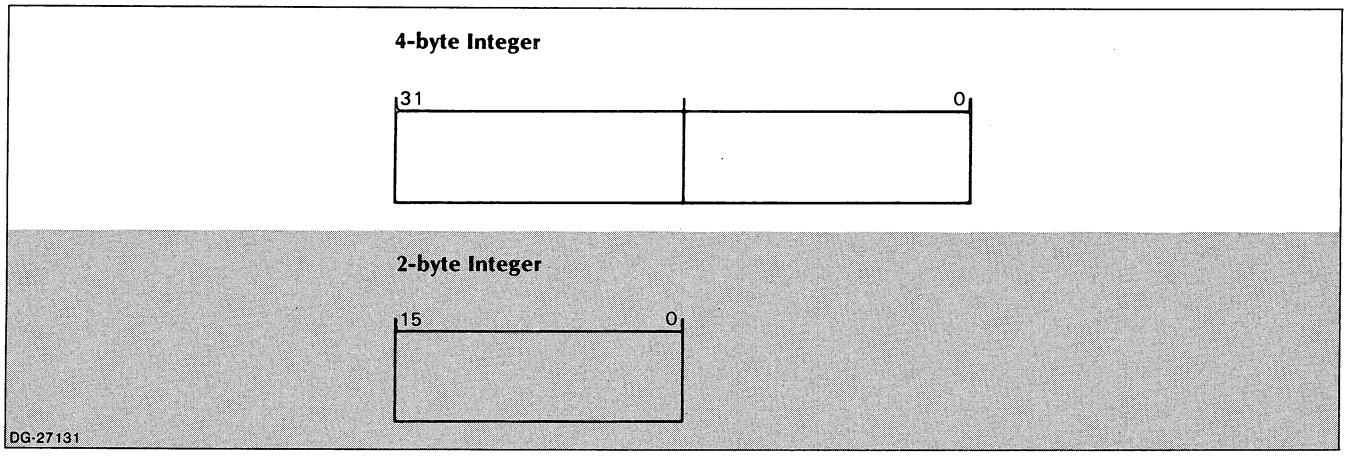


Figure 7-1. Order of Bits for Word and Bit Functions

Table 7-4. Word and Bit Intrinsic Functions

Generic Name	Meaning	Num. of Args.	Specific Name	Argument Data Type (* bytes if nonstandard)	Result Data Type	Function Reference Example
BTEST	Bit test. Test the bit in a variable (argument1) in position argument2. Returns .TRUE. if bit is 1, otherwise returns .FALSE. .	2	BTEST BTEST2 BTEST4	Int, def len Integer*2 Integer*4	Logical, as declared	L=BTEST(IVAR,3)
IAND	AND. For each bit position in each argument, insert binary 1 if both bits are 1; otherwise insert 0.	2	IAND IAND2 IAND4	Int, def len Integer*2 Integer*4	Int, def len Integer*2 Integer*4	J=IAND(IV1,IV2)
IBCLR	Bit clear. Clear the bit in a variable (argument1) in position argument2. Returns result with given bit cleared.	2	IBCLR IBCLR2 IBCLR4	Int, def len Integer*2 Integer*4	Int, def len Integer*2 Integer*4	J=IBCLR(IV,8)
IBITS	Extract bits. Extract a subfield of argument3 bits from argument1, starting at bit position argument2 and extending left. The result field is right justified and padded on the left with zero bits. The value of argument2+argument3 must be LE 32 (or 16). Argument2 must be GE 0 and LT 32 (or 16). Argument3 must be GT 0.	3	IBITS IBITS2 IBITS4	Int, def len Integer*2 Integer*4	Int, def len Integer*2 Integer*4	J=IBITS(K,4,2)
IBSET	Bit set. Set the bit in a variable (argument1) in position argument2. Returns result with given bit set.	2	IBSET IBSET2 IBSET4	Int, def len Integer*2 Integer*4	Int, def len Integer*2 Integer*4	J=IBSET(IV,4)
IEOR	Exclusive OR. For each bit position in each argument, insert binary 1 if only one bit is 1; if both bits or neither bit is 1, insert 0.	2	IEOR IEOR2 IEOR4	Int, def len Integer*2 Integer*4	Int, def len Integer*2 Integer*4	J=IEOR(IV1,IV2)
IOR	Inclusive OR. For each bit position in each argument, insert binary 1 if either both bits or one bit is 1; else insert binary 0.	2	IOR IOR2 IOR4	Int, def len Integer*2 Integer*4	Int, def len Integer*2 Integer*4	J=IOR(IV1,IV2)

(continues)

Table 7-4. Word and Bit Intrinsic Functions

Generic Name	Meaning	Num. of Args.	Specific Name	Argument Data Type (*bytes if nonstandard)	Result Data Type	Function Reference Example
ISHFT	Bit shift. Argument2 gives the number of positions to shift: a negative value shifts right, a positive value shifts left. Valid argument2 values are -16 through 16 (2-byte), -32 through 32 (4-byte). F77 sets each vacated bit to 0.	2	ISHFT ISHFT2 ISHFT4	Int, def len Integer*2 Integer*4	Int, def len Integer*2 Integer*4	J=ISHFT(IV,3)
ISHFTC	Circular bit shift. Shift the rightmost argument3 bits of argument1 circularly by |argument2| positions. If argument2 LT 0, it is a right shift; if argument2 GT 0, it is a left shift. |argument2| must be LE argument3. Argument3 must be GT 0 and LE 32 (or 16).	3	ISHFTC ISHFTC2 ISHFTC4	Int, def len Integer*2 Integer*4	Int, def len Integer*2 Integer*4	J=ISHFTC(K,2,5)
IXOR	Exclusive OR, same as IEOR.	2	IXOR IXOR2 IXOR4	Int, def len Integer*2 Integer*4	Int, def len Integer*2 Integer*4	J=IXOR(IV1,IV2)
NOT	Bit complement. For each bit in the argument, insert the complement (opposite) binary value.	1	NOT NOT2 NOT4	Int, def len Integer*2 Integer*4	Int, def len Integer*2 Integer*4	J=NOT(IV)

(concluded)

Table 7-5. Word and Bit External Functions

Name	Meaning	Num. of Args.	Argument Data Type (* bytes if nonstandard)	Result Data Type	Function Reference Example
BTEST	Bit test. Test the bit in a variable (argument1) in position argument2. Returns .TRUE. if bit is 1, otherwise returns .FALSE. .	2	Integer*4	Logical*4	EXTERNAL BTEST L=BTEST(IVAR,3)
IAND	AND. For each bit position in each argument, insert binary 1 if both bits are 1; otherwise insert 0.	2	Integer*4	Integer*4	EXTERNAL IAND J=IAND(IV1,IV2)
IBCLR	Bit clear. Clear the bit in a variable (argument1) in position argument2. Returns result with given bit cleared.	2	Integer*4	Integer*4	EXTERNAL IBCLR J=IBCLR(IV,8)
IBITS	Extract bits. Extract a subfield of argument3 bits from argument1, starting at bit position argument2 and extending left. The result field is right justified and padded on the left with zero bits. The value of argument2+argument3 must be LE 32. Argument 2 must be GE 0 and LT 32. Argument3 must be GT 0.	3	Integer*4	Integer*4	EXTERNAL IBITS J=IBITS(K,4,2)
IBSET	Bit set. Set the bit in a variable (argument1) in position argument2. Returns result with given bit set.	2	Integer*4	Integer*4	EXTERNAL IBSET J=IBSET(IV,4)
IEOR	Exclusive OR. For each bit position in each argument, insert binary 1 if only one bit is 1; if both bits or neither bit is 1, insert 0.	2	Integer*4	Integer*4	EXTERNAL IEOR J=IEOR(IV1,IV2)
IOR	Inclusive OR. For each bit position in each argument, insert binary 1 if either both bits or one bit is 1; else insert binary 0.	2	Integer*4	Integer*4	EXTERNAL IOR J=IOR(IV1,IV2)
ISHFT	Bit shift. Argument2 gives the number of positions to shift: a negative value shifts right, a positive value shifts left. Valid argument2 values are -32 through 32. F77 sets each vacated bit to 0.	2	Integer*4	Integer*4	EXTERNAL ISHFT J=ISHFT(IV,3)
ISHFTC	Circular bit shift. Shift the rightmost argument3 bits of argument1 circularly by argument2 positions. If argument2 LT 0, it is a right shift; if argument2 GT 0, it is a left shift. argument2 must be LE argument3. Argument3 must be GT 0 and LE 32.	3	Integer*4	Integer*4	EXTERNAL ISHFTC J=ISHFTC(K,2,5)
IXOR	Exclusive OR, same as IEOR.	2	Integer*4	Integer*4	EXTERNAL IXOR J=IXOR(IV1,IV2)
NOT	Bit complement. For each bit in the argument, insert the complement (opposite) binary value.	1	Integer*4	Integer*4	EXTERNAL NOT J=NOT(IV)

Word and Bit Function Examples

```
C   Trigonometric examples:

    X = SIN(R)

    PRINT *,   COS(Y)

    B = C(I) + (SIN(Y) + 6)

    IF (SIN(X) .LE. 0.5)  IF00 = IF00*2


C   Numeric examples:

    HYP = SQRT(A**2 + B**2)

    L4 = INT4(L2)

C   See if Real F00 is an INTEGRAL value:
    IF (REAL (INT(F00)) .EQ. F00)
   +    PRINT *, 'Integral'


C   Lexical comparison examples:

    CHARACTER*10  TX   / 'abcdefghij' /
    CHARACTER*10  TX1  / '1234567890' /
    LOGICAL  L
    ...
    L = LGT ( TX(1:1), TX1(1:1) )
    IF (L)  PRINT *,  'TX is 2nd alphabetically.'
    ...


C   Word and bit examples:

C   Test bit 3 (4th from rightmost).
    LOGICAL  L1
    I3 = 3
    L1 = BTEST (IF00, I3)
    IF (L1)  PRINT *,  'Bit 3 of IF00 is 1.'

C   Shift all bits left 4 positions.
    ISF00 = ISHFT (IF00, 4)
    ! Bits 3, 2, 1, and 0 of ISF00 are now 0.

C   Circular right shift by 3 bit positions of
C      the rightmost 5 bits using intrinsic
C      function ISHFTC; for example,
C         01000110.00100101
C      becomes
C         01000110.00110100
C      where the period separates the bits
C      into groups of 8.
    K3 = -3
    K5 = 5
    NVAR = ISHFTC(IVAR, K3, K5)
    ...

C   Swap the two bytes of 'AB' using specific
C      intrinsic function ISHFTC2
    INTEGER*2 IVAR, K8, K16
    IVAR = 'AB'
    K8 = 8
    K16 = 16
    IVAR = ISHFTC2(IVAR, K8, K16)
    ...
```

Subroutine MVBITS

This subroutine moves a bit field from one INTEGER*4 entity to another. The form of a reference to this subroutine is

CALL MVBITS(SOURCE, SBEGIN, LEN, DEST, DBEGIN)

where:

SOURCE	is an integer*4 expression that contains the 32 bits of the source field. If SOURCE is a different entity from DEST, then SOURCE is unchanged after MVBITS has executed.
SBEGIN	is an integer*4 expression that contains the position in SOURCE of the first bit that will be moved.
LEN	is an integer*4 expression that contains the number of consecutive bits that will be moved from SOURCE. Thus, the position of the last bit moved is SBEGIN+LEN-1.
DEST	is an integer*4 variable or array element that receives the LEN bits from SOURCE.
DBEGIN	is an integer*4 expression that specifies the first bit in DEST that receives its value from SOURCE. The number of affected bits in DEST is LEN.

For example, suppose that integer*4 variables SOR and DES contain 'ABCD' and 'WXYZ' respectively. Then the following statements result in DES containing 'WAYZ' (and in 'ABCD' remaining in SOR) after MVBITS executes:

```
INTEGER*4 SOR, SBEG, SLEN, DES, DBEG
SOR = 'ABCD'
SBEG = 24     ! HIGH ORDER BYTE IS IN
SLEN = 8      !   THE 8 BITS 24-31
DES = 'WXYZ'
DBEG = 16     ! 8 AFFECTED BITS BEGIN AT 16
CALL MVBITS (SOR, SBEG, SLEN, DES, DBEG)
...
```

System Functions

Table 7-6 summarizes the system functions BYTEADDR and WORDADDR. They let you obtain runtime byte and word addresses of entities.

The main purpose of these two system functions is to supply arguments or build packets for the external function ISYS. For example, suppose you want your F77 program at runtime to change the working directory to the initial working directory. The ISYS function, with the system call code for ?DIR as an argument, is the way to make this change. The outline of the necessary FORTRAN 77 statements is as follows:

```
      ...
      IAC0 = 0
      IER = ISYS (ISYS_DIR, IAC0, IAC1, IAC2)
C     THE VALUES OF ARGUMENTS IAC1 AND IAC2 AREN'T
C       IMPORTANT FOR THIS USE OF FUNCTION ISYS.
```

ISYS_DIR has the value of the ?DIR system call code from SYSID.32.SR (AOS/VS) or SYSID.SR (AOS). See your operating system's *FORTRAN 77 Environment Manual* for an explanation of how your program obtains the values of operating system symbols such as ?DIR and how in general your program interfaces with its operating system at runtime via function ISYS.

As another example, suppose in the same program you want to change the working directory to :UDD:MAC:CARS. Suppose your operating system's *Programmer's Manual* states that the ?DIR system call needs in AC0 a byte pointer to the pathname of the new working directory. The important FORTRAN 77 statements are as follows:

```
      ...
      IADDRESS_NEW_DIR = BYTEADDR(":UDD:MAC:CARS<0>")
      IER = ISYS (ISYS_DIR, IADDRESS_NEW_DIR, IAC1, IAC2)
C     THE VALUES OF ARGUMENTS IAC1 AND IAC2 AREN'T
C       IMPORTANT FOR THIS USE OF FUNCTION ISYS.
```

Of course, it's much easier for you to simply give the commands

```
DIR/I
```

or

```
DIR :UDD:MAC:CARS
```

to the CLI. But the BYTEADDR, WORDADDR, and ISYS functions are important software modules that work to let your programs give many runtime commands equivalent to direct CLI commands. Furthermore, use of BYTEADDR, WORDADDR, and ISYS lets your programs go beyond CLI commands to interact with the system. One such interaction is obtaining process statistics bye the ?PSTAT system call.

Table 7-6. System Intrinsic Functions

Generic Name	Meaning	Num. of Args.	Specific Name	Arg. Data Type	Result Data Type	Function Reference Example
BYTEADDR	Obtain the runtime byte address of (i.e., byte pointer to) the first byte of the argument.	1	—	Any: See note 1.	Integer, default length	LOC = BYTEADDR ("WHERE I AM")
WORDADDR	Obtain the runtime address of the first word (i.e., word address) of the argument. See note 2.	1	—	Any: See note 1.	Integer, default length	LOC = WORDADDR (IAC2)

Notes: 1. The argument is any

- arithmetic, logical, or character variable
- arithmetic, character, or Hollerith constant
- array name or array element
- substring

2. All arguments except character variables and character constants begin on word boundaries. Character variables and character constants may or may not begin on word boundaries. In either character case, WORDADDR returns the address of the word that contains the first byte of the character argument. This means the first byte of the word whose address WORDADDR(character-entity) obtains may contain data unrelated to the character variable or character constant given as the argument.

Statement Functions

A *statement function* is a procedure that you define in much the same way as an assignment statement. After defining the statement function, you execute it by specifying its name in an expression and following the name with a list of arguments, in parentheses. Like an intrinsic function, a statement function returns a single value to the program.

Defining a Statement Function

To define a statement function, use the form:

fname (*[d [,d] ...]*) = expr

where:

fname — is the symbolic name of the statement function. The data type of the name specifies the data type returned by the function; e.g., an integer name for an integer value, a character name for a character value. You cannot use the fname to identify any other entity in the current program unit except for a common block.

d — is a dummy argument name that will hold an actual argument's value during execution. The dummy argument list indicates the order, number, and type of actual arguments whose values will be used. A dummy argument name has meaning only in a statement function definition. You can also use a dummy name to identify variables of the same type, as a dummy argument to FUNCTION, SUBROUTINE, or ENTRY statements, or as a common block name. But each *d* name must be unique in the function definition; e.g., FOO(DUM1,DUM1,DUM2)=expr is illegal. Each dummy argument name must be of the data type of the actual value that will replace it during the function reference. Dummy arguments are further described below.

expr — is an expression. The component(s) of this expression can legally be

- a constant or symbolic name of a constant (assigned with PARAMETER);
- a variable;
- an array element;
- an intrinsic function;
- a statement function that has been defined in the current program unit;
- an external function reference;
- a dummy procedure reference;
- an expression within parentheses that meets all these requirements.

The fname (*[d, ...]*) = expr statement is the statement function definition; it is a nonexecutable statement and simply defines the function. The statement function definition must precede *all executable statements* within the program unit.

The function *reference* has the form:

fref (*[a [,a] ...]*)

where:

fref — is the function name, fname.

a — is an actual argument whose value the function may receive. The actual argument may not be a Hollerith constant.

Each time the function executes, F77 will use the expression given in the definition and the actual arguments to compute a value. Then it will return this value as the value of the function reference.

The following example illustrates a statement function reference.

```
C     Statement function definition:

      AVG(DUM1,DUM2,DUM3)=(DUM1+DUM2+DUM3)/3.0
      ...
      ...
C     Statement function reference:

      PRINT *,  AVG (TIME1,TIME2,TIME3)
```

AVG is the function name, real by the name rule. DUM1, DUM2, and DUM3 in the function definition are dummy arguments, serving as place holders for the actual arguments that will be used in the function reference. The dummies are type real by the name rule. TIME1, TIME2, and TIME3 are the actual arguments whose values can be used by the function.

At compilation time, the compiler encounters the function reference and associates the reference's actual arguments with the dummy arguments of the companion statement function definition. It associates actual argument TIME1 with dummy argument DUM1, TIME2 with DUM2, and TIME3 with DUM3.

When the program executes, F77 takes the following steps (as it does for every program that includes statement functions):

1. F77 evaluates all actual arguments that are expressions. Here, it need not evaluate anything because the actual arguments are simple variables that contain values. (The statements that produce and assign the TIME values aren't shown in the example.)

2. It uses the association established by the compiler to substitute values in TIME1, TIME2, and TIME3 for the dummy arguments DUM1, DUM2, and DUM3 respectively.
3. It then evaluates the expression **expr**: here, (TIME1+TIME2+TIME3)/3.0
4. It assigns the value of the expression to the function name, **fname**. If the data types of the value and **fname** are numeric and the types differ, F77 converts the value before assigning it to **fname**. If the data types are incompatible (e.g., integer and character), F77 reports an error. Here, F77 assigns the value to AVG and prints the value of AVG on unit 6. A function reference can also be an assignment statement, like FOO = AVG(TIME1,TIME2,TIME3).

Actual arguments and dummy arguments must agree in number, order, and data type. The program unit must define the actual arguments before referencing the statement function.

Rules for Statement Functions

- A statement function is local to a program unit; no other unit can use it.
- Before a program unit can use a statement function, it must
  - define the statement function before all executable statements and
  - assign values to the actual arguments that will be passed to the statement function.
- A statement function can make a reference to a different statement function that has already been defined.
- The name of a statement function must not appear in any specification statement except a type statement or as a common block name in a COMMON statement.
- The name of a statement function cannot be an actual argument or appear in a EXTERNAL statement.
- A function reference must include parentheses, even if the actual argument list is empty.
- The actual argument supplied for each dummy argument must match the data type and length of the dummy argument. The number of actual and dummy arguments must also match. And, an argument may not be a Hollerith constant.
- An external function reference used in a statement function definition must not cause a dummy argument in this statement function to be defined or redefined.
- A statement function in a function subprogram must not make reference to either the name of the function subprogram or an entry name in the function subprogram.

The statement function definition and reference are really a short, concise way to define and use a function within a program unit. Function subprograms, described next, define the function in a separate program unit.

Statement Function Example

The following statement function example finds one real root of a quadratic equation.

```
C   Statement function definition:

    ROOT (A,B,C) =
   +   ( -B + SQRT(B**2-4.0*A*C) ) / (2.0*A)

C   Statement function Reference:

    VALUE = ROOT (D(5), 9.9, X-Y) + Z**3
```

The compiler substitutes D(5) for A, 9.9 for B, and (X-Y) for C. The expression then becomes

```
(-9.9 + SQRT(9.9**2 - 4.0*D(5)*(X-Y))) / (2.0*D(5))
```

When F77 evaluates this expression at runtime, it returns the value to the function reference, adds to it Z**3, and assigns the result to VALUE.

Function Subprograms

This section gives an overview of function subprograms, then explains the FUNCTION statement itself.

A *function subprogram* — like an intrinsic or statement function — is a procedure that you refer to by name in an expression and follow with a list of arguments, in parentheses.

But — unlike an intrinsic or statement function — a function subprogram (often called a function) is a separate program unit. The program that uses it, the *calling program unit*, refers to the function subprogram; its statements execute; and it returns a value to the calling program unit. A function subprogram and its calling program unit are compiled separately and thus yield separate object (.OB) files. In contrast, a statement function and its related program are compiled together and result in one object file.

The calling program unit refers to a function subprogram in much the same way as a statement function:

func (*[a [,a] ...]*)

where:

func — is the name of the function.

a — is an actual argument. An actual argument can be a variable, array element, array, intrinsic function, constant, expression, or other subprogram name. The function subprogram can use each actual argument. The parentheses are mandatory.

A function subprogram itself begins with a FUNCTION statement, defines the function, and contains a RETURN or END statement to return control to the calling unit; it ends with an END statement. The structure of a function is

[typ] FUNCTION func (*[d [,d] ...]*)
...
func = expr
...
[RETURN]
...
END

where *typ* is the data type, func is the function name, and *d* is a dummy argument that will hold the value of an actual argument *a* in the calling unit.

The expr is an expression that defines the function's value; you must assign a value to the function's name before having the function execute a RETURN or END statement. Either RETURN or END statements pass control back to the function reference in the calling unit.

The function returns its own value to the calling unit. It may or may not change the values of the arguments. A function can contain any statement except a statement that defines it as a different program unit; i.e., a SUBROUTINE, BLOCK DATA, PROGRAM, or a second FUNCTION statement.

Using FUNCTION to Start a Function Subprogram

The FUNCTION statement starts a function subprogram. It specifies the function name and, optionally, its data type, and dummy arguments. Its form is

[typ] FUNCTION func (*[d [,d] ...]*)

where:

typ — is the data type of the function: INTEGER, INTEGER*2, INTEGER*4, REAL, REAL*4, REAL*8, DOUBLE PRECISION, COMPLEX, COMPLEX*16, LOGICAL, LOGICAL*1/BYTE (AOS/VS), LOGICAL*2, LOGICAL*4, or CHARACTER*[*len*], where *len* is the length of the character string that the function will produce. *len* can have any of the forms allowed for the CHARACTER type statement except that it cannot use the symbolic name of a constant in an integer expression. If you omit *len*, the length is 1. If you use a *typ* of CHARACTER*(*), the function's character string will assume the size declared in the calling unit (as detailed under "Character Arguments" in the next section).

func — is the symbolic name of this function subprogram. If you omitted *typ* and do not declare the type in a later type statement, the name implies the data type of the function. A valid symbolic name is 1 to 32 characters and includes letters, numbers, question marks, and underscores; the first character must be a letter or the question mark. MP/AOS and MP/AOS-SU names follow the same rules except that they are unique only up to the first ten characters.

d — is a dummy variable name, dummy array name, or dummy procedure name. Each *d* can hold the value of an actual argument from the calling unit. The parentheses are mandatory, even if you omit dummy arguments.

The FUNCTION statement must be the first statement in a function subprogram, although it may follow comment lines. FUNCTION is a nonexecutable statement and the compiler ignores any label on it.

When the calling program unit makes a reference to the function, F77 transfers control to the function, using any actual arguments for the function's dummy arguments. The function executes its statements. Upon a RETURN or END statement, the function returns its value to the function reference in the calling unit.

Within a calling unit, do not assign a value to the name of a function. Assigning a value implies that the name belongs to a variable, not to a function; the compiler will signal an error if you also use the name as a function reference. The previous two sentences do not apply if a function subprogram calls itself.

If a function contains a READ, WRITE, or PRINT statement, the function cannot be specified by a READ, WRITE, or PRINT that specifies the same unit. For example, if MYFUNC was invoked by PRINT *, MYFUNC, then MYFUNC cannot use a PRINT statement. If both program units try to use a PRINT statement, F77 will report a runtime error.

You must assign a value to the function name at least once before the function returns control to the function reference in the calling unit. For example, a function beginning with the statement FUNCTION ICAN might contain a statement like ICAN=5, which assigns a value to the function.

The dummy arguments in a FUNCTION statement must correspond in order, number, and type with the actual arguments of the function reference that invoked it. (But you can use any subroutine name as a dummy argument because subroutines do not have a data type.) A dummy argument can hold the value of an expression, an array name, or an external subprogram name. It can also hold an intrinsic function name if the calling unit declared the function INTRINSIC, described later in this chapter.

If the dummy argument will hold an array name, the function must dimension the array —not larger than the original array in the calling unit. This is detailed under "Actual and Dummy Arguments" in the next section.

You cannot include dummy argument names in EQUIVALENCE, PARAMETER, INTRINSIC, SAVE, or DATA statements within the subprogram. You can include a dummy argument name in a COMMON statement, but only as a common block name (the two names will be treated as different entities).

Rule Summary for Function Subprograms

- A function subprogram must begin with the FUNCTION statement and can contain only one FUNCTION statement. It cannot contain a PROGRAM, SUBROUTINE, or BLOCK DATA statement.
- If you include a *typ* in the FUNCTION statement, the function type is *typ*. If you omit a *typ*, you can imply a type via the name rule or an IMPLICIT statement that follows the FUNCTION statement. Alternatively, if you omit a *typ*, you can assign a type in a type statement within the function.
- The dummy argument names (if any) must match the actual arguments in the calling argument in number, order, and type.
- Between a function's FUNCTION and END statements, the function's name (or name of an ENTRY statement in the function, described later) may appear only as a variable — and, if you omitted *typ*, in *one* type statement.
- A function must assign a value to its symbolic name at least once.
- If a function subprogram is type character, either via *typ* or type statement, each ENTRY name must be type character. All character ENTRY names and the character function name must have the same length, which can be an integer (CHARACTER*60 FUNCTION CH) or asterisk (CHARACTER*(*) FUNCTION CH1). See the next major section for more detail.
- The dummy argument names used must not appear in a DATA, EQUIVALENCE, PARAMETER, INTRINSIC, SAVE, or COMMON statement within the subprogram, except as a common block name.
- A function cannot use an assumed-size array (described next) in an iolist.

Function Subprogram Examples

Some function statement examples are

```
FUNCTION  IFIND ( )

FUNCTION  MARGO(L)

INTEGER FUNCTION ROYCE (K,L,M, CSTRING)

DOUBLE PRECISION FUNCTION  FLAG (X,Y)

CHARACTER*30 FUNCTION  TX (REALVAR)
```

The following function is assigned the value 0.0 if the value it receives is negative; otherwise, the function is assigned the value 1.0.

```
C   Main program.

    ! Program assigns a real value to A.

C   Make a reference to the function:

    PRINT *, 'SWITCH(A) equals ', SWITCH(A)
    ...
    END


C   Function Subprogram.

    FUNCTION   SWITCH (A2)

    IF ( A2 .LE. 0.0 ) THEN
           SWITCH = 0.0
    ELSE
           SWITCH = 1.0
    END IF

    RETURN
    END
```

Function SWITCH is assigned the value 0.0 or 1.0 and returns this value to the main program. Dummy argument A2 holds the value of A from the calling program. All data types involved are real.

The next example involves a function that computes the scalar product of two vectors (i.e., arrays) up to a given element.

```
C   Main program.
    PROGRAM  VECTORS
    REAL  B(40), C(40)
10  PRINT *,  'Type number of elements, 1-40.'
    READ (*, *)   LEN
    IF ((LEN .GT. 40) .OR. (LEN .LT. 1))  GO TO 10
    DO 30  K = 1, LEN
       PRINT *, 'Type B and C values, element ', K
       READ (*,*)   B (K), C(K)
30  CONTINUE
    PRINT *
    PRINT *, 'Values in array B and C are: '
    PRINT *, (B(I), I=1, LEN), (C(I), I=1, LEN)
    PRINT *, 'Scalar product is ', SCALE (B,C,LEN)
    END


C   Function.
    FUNCTION  SCALE (B2, C2, LEN2)
    REAL  B2(LEN2), C2(LEN2)
    SCALE = 0.0
    DO 20  I = 1, LEN2
       SCALE = SCALE + B2(I) * C2(I)
20  CONTINUE
    RETURN
    END
```

Here, function SCALE takes the products of the values in an accumulating total, then returns its value to the main program.

The next example shows how a main program (FIND_AST_CHARACTER.F77) uses a quoted string to pass information to a function subprogram (CHAR_LOCATION.F77). For it to work correctly, compile this program with the /HOLLERITH=NON_DG switch.

```
C     Program  FIND_AST_CHARACTER.F77  to find
C       the location of the first occurrence of an
C       asterisk in a string of text.
C
C     This program calls on function subprogram
C       CHAR_LOCATION  to do the actual searching.
C
      INTEGER*2 LINE(80)
      INTEGER*2 CHAR_LOCATION   ! Note this type
C       statement with its name of a function
C       subprogram.
      INTEGER*2 ASTERISK_LOCATION
      WRITE (10, 20)
   20 FORMAT (/,1X,'PLEASE TYPE IN A LINE OF ',
     +            'TEXT AND THEN NEW-LINE')
      READ (11, 30) LINE
   30 FORMAT (80A1)   ! The user doesn't have
C       to type 80 characters.
      ASTERISK_LOCATION =
     +  CHAR_LOCATION(LINE, 80, '*')  ! Note use
C       of the character constant in the
C       arguments to function subprogram
C       subprogram CHAR_LOCATION.
C       We also could have used  1H* .
      IF (ASTERISK_LOCATION .EQ. 0) THEN
          PRINT *, "ASTERISK ISN'T IN THIS TEXT"
      ELSE
          PRINT *, "FIRST ASTERISK IS IN ",
     +             "LOCATION ", ASTERISK_LOCATION
      END IF
      PRINT *, " "    ! Blank line before the
C                         terminal displays STOP.
      STOP
      END
```

```
      INTEGER*2 FUNCTION CHAR_LOCATION
     +                  (TEXT, N, CHAR)
C     This function subprogram gets  N
C       characters in array  TEXT  and determines
C       the location of the first occurrence of
C       the variable  CHAR  in the array.  It then
C       returns this location to the caller.
      INTEGER*2 TEXT(N)  ! See the explanation
C                          of adjustable arrays.
      INTEGER*2 CHAR
C
      CHAR_LOCATION = 0   ! Assume that the
C       character isn't in the line of text.
      DO 10 I = 1, N
         IF ( TEXT(I) .EQ. CHAR )
     +       GO TO 20   ! found a match!
   10 CONTINUE
C     We now know  CHAR  doesn't exist in
C       array TEXT.  So, return to the caller with
C       CHAR_LOCATION   equal to its initial value
C       of zero.
      RETURN
   20 CHAR_LOCATION = I   ! CHAR_LOCATION contains
C       the location of the first occurrence
C       of  CHAR .
      RETURN
      END
```

Arguments to Function and Subroutine Subprograms

This section describes actual-dummy argument association, which applies to function and *subroutine* subprograms. The material here applies to both functions and subprograms, and we present it here to avoid duplication.

Functions and Subroutines

Functions and subroutines are both *subprograms*. Aside from the CALL and SUBROUTINE statements, all statements in the rest of this chapter apply to both. The main differences between function subprograms and subroutine subprograms are as follows:

Function Subprograms	Subroutine Subprograms
are invoked from the calling unit via the form: fref name (args)	are invoked from the calling unit via the form: CALL name (args)
begin with a FUNCTION statement.	begin with a SUBROUTINE statement.
have a data type.	have no data type.
are assigned a value which they return to the calling unit.	cannot be assigned a value; rely entirely on arguments or common storage to pass values.
return to the point of reference in the calling unit.	return to the point of reference or to other points in the calling unit.

Traditionally, functions are used to get one value, the function value, and the values received in dummy arguments are not changed. (However, you may alter these values.) Subroutines work either by changing values received in dummy arguments or by changing values in common storage.

Actual and Dummy Arguments

F77 associates the values of actual arguments in the calling program's subprogram reference with dummy arguments in the subprogram. It associates the correct values only if the arguments agree in order, number, and type. However, F77 *may not* report an error if they do not agree.

Each actual argument is really a pointer to the memory location(s) that holds the original value(s). The subprogram receives the pointer through its dummy argument and can thus find the location(s) that holds the original value(s). Technically speaking, DG F77 passes arguments by reference instead of by value.

An *actual* argument can be an expression, array name, intrinsic function name, function name, or subroutine name. If an actual argument is an expression, array name, or function name, the dummy argument name must match its data type. F77 evaluates an expression before passing control to the subprogram. If an argument is a subroutine name, the actual and dummy data types need not match because subroutines do not have data types. You cannot use a statement function as an actual argument.

An actual argument name can appear without restrictions — according to the rules for its use — within the calling program unit. Be careful when using the same name more than once in an actual argument list (e.g., CALL SUB (C, C)). Because a pointer to the value — not the actual value — is passed, the corresponding dummy arguments will refer to the same value, although the arguments have different names. This can produce confusing results.

A dummy argument name cannot appear in a DATA, EQUIVALENCE, PARAMETER, or SAVE statement within its program unit. It can appear as a COMMON block name, but F77 will treat the names as different entities. Because the names originate in different units, a dummy argument can have the same name as its corresponding actual argument.

Logical*1/byte arguments (AOS/VS) are treated exactly the same as character*1 arguments. This means that type and size agreement exists if the actual argument is logical*1/byte and the dummy argument is character*1. (Technical note: F77 passes dope vectors for logical*1/byte entities.) Type and size agreement also exists if the actual/dummy argument correspondence is logical*1/byte-logical*1/byte, character*1-logical*1/byte, and character*1-character*1.

The /HOLLERITH Compiler Switch: Its Rules and Effects

A calling program unit and its called subprograms must obey these rules (see also Chapter 9):

1. The members of each pair of corresponding variable arguments must agree in order, number, and type.
2. When Hollerith constants are used as actual arguments, the corresponding dummy arguments must be non-CHARACTER.
3. When character constants are used as actual arguments, and /HOLLERITH=ANSI, the corresponding dummy arguments must be CHARACTER.
4. When character constants are used as actual arguments, and /HOLLERITH=OLD_DG or /HOLLERITH=NON_DG, the corresponding dummy arguments must be non-CHARACTER.

The three values of the /HOLLERITH compiler switch (ANSI, NON_DG and OLD_DG) affect the storage of character constants and Hollerith constants that are used as subprogram arguments. In the explanation below, we cover all possible combinations of the three /HOLLERITH compiler switch values and the two types of constants (character and Hollerith).

1. By default or under /HOLLERITH=ANSI,

   ```
   CALL FOO_SUB_1 ('<NL>', 'ABCDE')
   ```

   passes two addresses (byte pointers) to memory areas whose respective contents are <012><000> and <101><102><103><104><105><000>. Also by default and according to Appendix C of the FORTRAN 77 standard,

   ```
   CALL FOO_SUB_2 (2HAB)
   ```

   passes an address (word pointer) to a 16-byte memory area whose contents are <101><102><040>...<040>. The memory area is 16 bytes long because the compiler doesn't know the type of the corresponding argument in subroutine FOO_SUB_2 and must allow for the largest possible type of variable: complex*16.

2. You may now use the /HOLLERITH=NON_DG switch to pad with blanks to 16 bytes every character constant and Hollerith constant that a program unit passes to a subprogram. With this switch, the statement

   ```
   CALL FOO_SUB_1 ('<NL>', 'ABCDE')
   ```

   passes two addresses (word pointers) to 16-byte memory areas whose respective contents are <012><040>...<040> and <101><102><103><104><105> <040>...<040>. Also with this switch, the statement

   ```
   CALL FOO_SUB_2 (2HAB)
   ```

   passes an address (word pointer) to a 16-byte memory area whose contents are <101><102><040>...<040>. Note that this result is identical to that obtained under default conditions.

3. You may now use the /HOLLERITH=OLD_DG switch to pad with nulls to 16 bytes the character constants and Hollerith constants a program unit passes to a subprogram. With this switch,

   ```
   CALL FOO_SUB_1 ('<NL>', 'ABCDE')
   ```

   passes two addresses (word pointers) to 16-byte memory areas whose respective contents are <012><000>...<000> and <101><102><103><104><105> <000>...<000>. Also with this switch,

   ```
   CALL FOO_SUB_2 (2HAB)
   ```

   passes an address (word pointer) to a 16-byte memory area whose contents are <101><102><000>...<000>.

If a character constant or Hollerith constant is 16 or more bytes long, the compiler stores a null byte at the end of the constant, making it suitable for use as input to a system call. This storage occurs for all values of /HOLLERITH=.

Finally — a general comment applies to this discussion of subprogram arguments regardless of which /HOLLERITH switch setting you select. "Subprogram" includes function subprograms as well as the subroutine subprograms that appear in the previous six examples. Thus the first example could have been

```
FOO_FUNC_1_RESULT = FOO_FUNC_1('<NL>', 'ABCDE')
```

Variable Arguments

To pass a variable to a subprogram, use its name as an actual argument and include a dummy argument of the same data type in the subprogram's argument list. (Character variables are described further on.) You can pass a single array element by making the array name and subscript(s) an actual argument; if so, F77 evaluates the subscript expression before assigning the element value to the dummy argument. The following example shows how to pass both a variable and array element.

```
C   Main program.
    DIMENSION  ARY(100)
    ARY(9) = 1.00     ! Array element.
    J = 999           ! Variable.
    VAL_OF_SUBPROG = SUBPROG(J,ARY(9)) ! Reference
    PRINT *, VAL_OF_SUBPROG
    ...
    END

C   Function subprogram.
    FUNCTION  SUBPROG (J2, ARYVAR)
C   J2 equals 999, ARYVAR equals 1.00: values of
C      actual arguments in calling program.
    ...
    SUBPROG = ... (assign a value to SUBPROG)
    RETURN
    END
```

Array Arguments and Declarators

To pass an array to a subprogram, use the array name as an actual argument. The subprogram must dimension the array to use it. There are three ways you can have the subprogram dimension the array:

1. You can use integer constant expressions. In this case, be sure that you dimension the array to have *no more* than the number of elements in the actual array. For example, if the calling unit declared the array with 100 elements, a subprogram declaration of more than 100 elements (like (10,11)) would be an error. *However, F77 may not detect this error.*

2. You can use an *adjustable array declarator.* An adjustable array declarator is an integer passed to the subprogram via a dummy argument (or an integer variable within a common block). The declarator can include integer constant expressions. For example:

```
C   Calling unit.
    DIMENSION  IA(100), IB(100)
    J = 100
    K = 8
    ...
    PRINT *, SUBPROG1 (IA, IB, J, K)  ! Reference.
    ...
    END

C   Subprogram.
    FUNCTION  SUBPROG1 (IA2, IB2, J2, K2)
    DIMENSION  IA2(J2)
    DIMENSION  IB2(K2, K2)
    ...
    END
```

   With either an adjustable or constant declarator, a subprogram can structure the array elements according to its needs — so long as the total number of elements remains less than or equal to the number in the original array.

3. The third and easiest way to pass an array is to use an *assumed-size* array declarator. In an assumed-size array declarator, you specify the upper bound of the last array dimension as an asterisk. Thus, for one-dimensional arrays, you need use only an asterisk dimension; e.g.,

```
C   Main program.
    REAL ARY(200)
    PRINT *, SUBPROG2 (ARY)   ! Reference.
    ...
    END

C   Subprogram.
    FUNCTION  SUBPROG2 (ARY2)
    DIMENSION ARY2(*)   ! Assumed size
                        !    declarator.
    ...
    END
```

If the array has more than one dimension, you can use an assumed-size declarator only for the last dimension. For example:

```
C   Calling unit.
    DIMENSION  JX(10,10)
    ...
    PRINT *,  SUBPROG3 (JX)   ! Reference.
    ...
    END

C   Subprogram.
    FUNCTION  SUBPROG3 (JX2)
    DIMENSION  JX2(10, *)   ! Assumed size
                  ! for last dimension.
    ...
    END
```

Note that a subprogram cannot use an assumed-size array in an iolist.

If you use an array *element* name as an actual argument, that element becomes the first element in the subprogram array; but you may not dimension this array so that it would extend past the original upper bound. For example, for IFOO(100), if you passed IFOO(50) as a dummy argument, dimensioning the subprogram array with more than 51 elements would be an error (but F77 might not detect this error).

Character Arguments

Any subprogram can receive character values in one or more of its dummy arguments. The actual argument in the calling program must — naturally — be of type character. The subprogram cannot process more characters than were declared for the entity in the calling unit. If the dummy argument is an array, the subprogram can declare an element length that differs from the caller's, but the subprogram cannot refer to a character beyond the last character reserved by the caller for the array.

An easy way to handle character variables and arrays is to use an asterisk as a length specifier; if you do, F77 will assign all the characters in the actual argument to the dummy. Here are two examples that do the same thing. The first example shows a function, the second a subroutine.

```
C   Example with function.
C   Calling unit.
    PROGRAM  MAIN
    ...
    CHARACTER*10  CF  / '1234567890' /
    CHARACTER*10  CARY(4)
    JY = 8
    ...
    X = FUNC (CF, CARY, JY)   ! Reference.
    ...
    END

C   Function.
    REAL FUNCTION  FUNC (CF2, CARY2, JY2)
    CHARACTER*(*)  CF2   ! Define var CF2 and
    CHARACTER*(*)  CARY2(*)   ! array CARY2.
    ...
C   When referenced by program unit MAIN, CF2
C      contains '1234567890'; CARY2 is the
C      4-element array; JY2 is an integer.
    ...
    FUNC = nnn  ! Assign function value.
    RETURN
    END

C   Subroutine Example.
C   Calling unit.
    PROGRAM  MAIN
    ...
    CHARACTER*10  CF  / '1234567890' /
    CHARACTER*10  CARY(4)
    JY = 8
    ...
    CALL  SUBPROG (CF, CARY, JY)
    ...
    END

C   Subroutine.
    SUBROUTINE  SUBPROG (CF2, CARY2, JY2)
    CHARACTER*(*)  CF2   ! Define var CF2 and
    CHARACTER*(*)  CARY2(*)   ! array CARY2.
    ...
C   When called by program unit MAIN, CF2
C      contains '1234567890'; CARY2 is the
C      4-element array; JY2 is an integer.
    ...
    RETURN
    END
```

For a function subprogram of type character, you can define the function length in the calling unit, then use an asterisk as a len specifier in the function subprogram. The function will then assume the length given in the calling unit. For example,

```
C   Calling unit.
    PROGRAM  MAIN
    CHARACTER*20  CFUNC
    ...
    ...
    PRINT *,  CFUNC( )  ! Reference.
    ...
    END

C   Function subprogram.
    CHARACTER*(*)  FUNCTION  CFUNC()
    ...
C   CFUNC contains the 20 characters
C      declared in the calling unit.
    ...
    CFUNC = 'xxxx'      ! Mandatory assignment.
    ...
    RETURN
    END
```

The asterisk length specifier for character entities allows you to manipulate character strings easily in discrete program units. As an extension, DG's F77 allows you to concatenate strings passed via asterisk specifiers.

Subprogram Arguments

You can use certain intrinsic functions as actual arguments if you declare them in an INTRINSIC statement (described next).

You can also use subprogram names as actual arguments if the subprograms exist. You must declare each subprogram name you plan to pass as EXTERNAL in the calling program (to let F77 know that the name is a subprogram — not a variable — name).

Using a subprogram name as an actual argument implies that you will have the subprogram call another subprogram. In other words, the original caller (say MAIN) calls a subprogram (say S1), passing the name of another subprogram (say S2). S1 will act as an intermediary, and call the subprogram(s) whose name(s) is (are) passed in the dummy argument.

The data type of S1 is not important because it will act as an intermediary. Only the data types of values that S1 passes back to MAIN are important. If the value passed from MAIN is a function name, S1 must make reference to it as a function; if it is a subroutine, S1 must CALL it as a subroutine. The following example shows S1 as a subroutine but would be identical (aside from the CALL and SUBROUTINE statements) if S1 were a function.

```
PROGRAM  MAIN
EXTERNAL  SUB, IFUNC
...
CALL  S1 (SUB, IFUNC)
! S1 returns control here.
...
END


SUBROUTINE  S1 (NAME1, NAME2)
...
CALL  NAME1 (args)  ! SUB is subroutine.
...
PRINT *,  NAME2(args) !  IFUNC is function.
...
RETURN   ! Return to MAIN.
END
```

Using the INTRINSIC Statement for Intrinsic Functions

Normally, intrinsic functions (SIN, INT, etc.) work only within a program unit. If you try to pass their names as actual arguments, F77 will treat the names as variables. If you declare them EXTERNAL, they will lose their intrinsic properties.

But if you declare certain intrinsic functions INTRINSIC in a program unit, you *can* pass them as actual arguments to the dummy arguments in subprograms. INTRINSIC is a specification statement and as such must precede statement function definitions and executable statements in a program unit. The form of INTRINSIC is

INTRINSIC ifunc *[,ifunc]* ...

where:

ifunc is one of the F77 intrinsic function names.

If you declare a generic function name INTRINSIC, it will retain its intrinsic properties in the subprogram. Do not use a function name more than once in an INTRINSIC statement in any program unit.

Although you can declare them INTRINSIC, you cannot pass the names of intrinsic functions for type conversion, maximum/minimum value (e.g., AMIN), and lexical comparison functions. You *can* pass the names of *all* trigonometric functions (ACOS, etc., in Table 7-1), generic functions (ABS, AIMAG, CEIL, CEXP, CLOG, CONJG, CSQRT, DIM, EXP, FLOOR, FRAC, IABS, ISIGN, LEN, LOG, LOG10, SIGN, and SQRT in Table 7-2), and all word/bit functions (IAND, etc., in Table 7-4).

INTRINSIC Example

The following example uses a subroutine to compute the sine of array elements.

```
C   Calling program.

    PROGRAM  CALLER
    INTRINSIC  SIN
    ...
    DIMENSION  RRY(100)
    ...
C   Assign values to RRY.
    ...
    CALL  APPLY (RRY, 100, SIN)
C   Process array with sines here.
    ...
    END

    SUBROUTINE  APPLY (RRY2, J2, SIN2)
    REAL  RRY2(J2)
    DO 90 I = 1, J2
      RRY2(I) = SIN2(RRY2(I))
90  CONTINUE
    RETURN
    END
```

Subroutine Subprograms

This section describes subroutines and the statements that pertain only to them: SUBROUTINE and CALL.

A *subroutine subprogram*, generally referred to as a *subroutine*, is a procedure external to the main program unit. A CALL statement in the calling program unit invokes it.

As sketched earlier, subroutines resemble function subprograms: actual and dummy arguments are handled the same way in both, and a RETURN statement in both returns control to the calling program. Furthermore, both subroutine subprograms and function subprograms may be compiled separately from their calling program units, thus yielding distinct object (.OB) files.

But subroutines differ from functions in that a CALL statement, not a function reference, invokes them. They never assign a value to themselves; instead they operate on the values in their arguments. They have no data type. And, within the subroutines, you can specify different points of return to the calling program.

The calling program unit invokes a subroutine this way:

CALL sub *[([a [,a] ...])]*

where CALL passes control to the subroutine, **sub** is the subroutine name, and *a* is an actual argument: variable, array, array element, constant, expression, intrinsic function, or other subprogram name — just as for functions. An *a* can also be an alternate return specifier. The subroutine can change the *a* values but need not do so.

A subroutine itself starts with a SUBROUTINE statement; its statements execute; and it issues RETURN at any number of points to return control to the calling unit. It must end with an END statement. The form of a subroutine is

SUBROUTINE sub *[([d [,d] ...])]*
...
...
[RETURN]
...
END

where **sub** is the subroutine name and each *d* is a dummy argument that holds a place for an actual argument *a* in the calling unit. If the subroutine makes a reference to or changes a dummy argument, it makes a reference to or changes the corresponding actual argument.

Either RETURN or END statements pass control back to the statement following CALL in the calling unit, unless a RETURN specifies an alternate return point.

A subroutine can contain any statement except a statement that defines it as a different program unit; i.e., a FUNCTION, BLOCK DATA, PROGRAM, or a second SUBROUTINE statement. A subroutine has no data type of its own but its dummy arguments do.

Using SUBROUTINE to Start a Subroutine

The SUBROUTINE statement starts and names a subroutine. Its form is

SUBROUTINE sub *[([d [,d] ...])]*

where:

sub is the symbolic name of the subroutine. A subroutine has no data type, thus the name rule doesn't apply. A valid symbolic name is 1 to 32 characters and includes letters, numbers, question marks, and underscores; the first character must be a letter or the question mark. MP/AOS and MP/AOS-SU names follow the same rules except that they are unique only up to the first ten characters.

d is a dummy variable name, dummy array name, or dummy procedure name. *d* can be an asterisk (*) if the subroutine will use alternate returns.

The SUBROUTINE statement must be the first statement of a subroutine, but it may follow comment lines. It is a nonexecutable statement and the compiler ignores any label on it.

On the subroutine CALL statement in the calling program unit, F77 transfers control to the subroutine, using any actual arguments for the subroutine's dummy arguments. The subroutine executes its statements, which may or may not modify the arguments. On a RETURN or END statement, control returns to the statement following the caller's CALL statement, unless the subprogram specifies an alternate return.

Each dummy argument may be a variable name, an array name, a subprogram name, or an asterisk.

If a dummy argument is an asterisk, the corresponding actual argument in the calling unit must be an asterisk followed by the label of an executable statement within the calling unit. In a RETURN statement within the subroutine, you can specify an alternate return by giving the position of this asterisk among other asterisks in the dummy argument list. For example,

```
C   Calling program.
    ...
    CALL  SUB (A, *80, B, *90)
    ! Normal return here.
80  xxx    ! RETURN 1 returns here.
    ...
90  xxx    ! RETURN 2 returns here.
    ...
    END


C   Subroutine.
    SUBROUTINE  SUB (A2, *, B2, *)
    ...
    RETURN    ! Returns after CALL statement.
    ...
    RETURN 1  ! Returns to caller at 80.
    ...
    RETURN 2  ! Returns to caller at 90.
    END
```

You cannot include dummy argument names in EQUIVALENCE, PARAMETER, INTRINSIC, SAVE, or DATA statements within the subprogram. You can include a dummy argument name in a COMMON statement, but only as a common block name (the two names will be treated as different entities).

A subroutine cannot use an assumed-size array in an iolist.

Subroutine Examples

Some subroutine statement examples are

```
SUBROUTINE  ELECT (X, YARD, *, J)

SUBROUTINE  NAME9
```

The following example uses a subroutine that reverses N elements of a 200-element array.

```
C   Calling program.
    PROGRAM  TESTIT
    DIMENSION  ARY(200)
    PRINT *,  'Type number of elements and'
    PRINT *,  '   values for them. '
    READ (*,*)  NUM,  (ARY(I), I = 1, NUM)
    CALL  REVERS (ARY, NUM)
    PRINT *, 'Here they are: ', (ARY(I), I=1,NUM)
    END

C   Subroutine.
    SUBROUTINE  REVERS (ARY2, NUM2)
    DIMENSION  ARY2(NUM2)  ! Adj. declarator.
    MIDDLE = NUM2 / 2      ! Get middle element.
    DO 90  I = 1, MIDDLE   ! Reverse.

C       Use a temporary variable to reverse
C          array elements.

        TEMP = ARY2(I)
        ARY2(I) = ARY2( NUM2 + 1 - I )
        ARY2( NUM2 + 1 - I ) = TEMP
90  CONTINUE
    RETURN
    END
```

Using CALL to Invoke a Subroutine

The CALL statement invokes and passes control to a subroutine. Its form is

CALL sub *[([a [,a] ...])]*

where:

sub is the name of a subroutine.

a is an actual argument: variable, array element, array, substring, constant, expression, intrinsic function, or subprogram name. For an alternate return, *a* can be an asterisk followed by the label of an executable statement in the current unit.

CALL invokes your subroutine, passing control to the first executable statement and using any *a* arguments for the subroutine's dummy arguments. When the subroutine executes a RETURN statement, control returns to the statement in the calling unit that follows the CALL statement — unless the subroutine specifies an alternate return. Return from the subroutine completes execution of the CALL statement.

A subroutine can call another subroutine or itself.

You must supply an actual argument for each dummy argument in the called subroutine's SUBROUTINE statement. An actual argument can be a variable or array element, or array; if the argument is any of these, the data type must match that of the dummy argument. Character arguments are described earlier. An actual argument can be a statement label in the current unit if the subroutine has an asterisk as its corresponding dummy argument (to specify an alternate return). If an argument is the name of a subprogram, the data type is irrelevant, but you must have declared this name EXTERNAL in this program unit. If an argument is an intrinsic function name and the subroutine will use this as an intrinsic function, you must have declared the function INTRINSIC in the current unit.

CALL Examples

Examples of CALL statements are

```
CALL QUAD (9, Q/R, 20, R-S**2, X1, HALF)

CALL OPTIONS
```

An example that shows different arguments passed is

```
C   Calling unit.
    CHARACTER*15  CH  / 'a passed string' /
    DIMENSION  ARRY(200)
    A = 90.0
    CALL  SUB (CH, ARRY, A, *40)
    ! Normal return here.
40  A = 100.0   ! RETURN 1 here.
    ...
    END


C   Subroutine.
    SUBROUTINE  SUB (CH2, ARRY2, A2, *)
    CHARACTER*(*)  CH2
    DIMENSION  ARRY2 (*)
    ...
    RETURN 1
    ...
    RETURN
    END
```

Using ENTRY to Define an Entry Point in a Subprogram

Normally within a subprogram, execution begins at the first executable statement after the initial FUNCTION or SUBROUTINE statement. The ENTRY statement allows you to define other starting points in a subprogram.

You make reference to an ENTRY much the same way as you make reference to the subprogram that contains it. For an ENTRY in a function, use a function reference; for an ENTRY in a subprogram, use CALL. You can make reference to an ENTRY directly from any program unit except the subprogram that contains it.

ENTRY's form within the subprogram is

ENTRY en *[([d [,d] ...])]*

where:

en is the symbolic name of the entry point. If the entry is in a function, the en name has a data type; you can imply the type or specify it in a type statement before or after the ENTRY statement. If the entry is in a subroutine, it has no data type. A valid symbolic name is 1 to 32 characters and includes letters, numbers, and underscore; the first character must be a letter or the question mark. MP/AOS and MP/AOS-SU names follow the same rules except that they are unique only up to the first ten characters.

d is a dummy argument. An entry's dummy arguments can differ in number and type from the dummy arguments of other entries or those in its parent subprogram. Each *d* must match the data type of the corresponding actual argument in the statement that refers to or calls the entry — unless the actual argument is a subprogram name. If the entry is in a subroutine, *d* can be an asterisk to allow an alternate return. You can omit the parentheses for an empty argument list in a subroutine entry; *but you must always include the parentheses in a function entry and in the entry reference.*

ENTRY is a nonexecutable statement that F77 skips when it executes the subprogram. You can label ENTRY statements but cannot refer to the label. In a function subprogram, the entry name en must not appear before the ENTRY statement, except within *one* type statement (or a COMMON statement). Also within any program unit, you cannot declare an entry name EXTERNAL or use it as a dummy argument.

An entry can appear anywhere between the initial FUNCTION or SUBROUTINE statement and the END statement, but *not* within a block IF or in the range of a DO loop.

On an entry reference or a CALL statement, F77 transfers control from the calling unit to the first executable statement following ENTRY, using any actual arguments for the entry's dummy arguments.

F77 does not retain the actual-dummy argument association between references — the association is brand new for each reference to the subprogram or any entry within it. Thus an entry's dummy argument list need not be the same as its parent subprogram's, but it must match the actual argument list in the statement that refers to or CALLs the entry.

A dummy argument name that appears in an ENTRY statement must not appear in an executable statement that precedes that ENTRY statement — unless the name also appears in a FUNCTION, SUBROUTINE, or ENTRY statement that precedes the executable statement.

If the entry is in a function, the statements after ENTRY must assign a value to the entry name (as is true for any function).

On a RETURN or END statement, control passes back to the calling unit at the function reference or after CALL — unless, in a subroutine, the RETURN statement specifies a different return point.

ENTRY Rules for Function Subprograms

In a function, an ENTRY cannot be type character unless the function itself is type character (CHARACTER*len FUNCTION func or type statement within the function). All character entries must have the same length as the function itself: integer or asterisk. Each of the following examples is valid:

```
C   Character function, integer length specifier.
    CHARACTER*60 FUNCTION  CFUNC (args)
    CHARACTER*60 C_ENTRY
    ...
    CFUNC = 'abc'
    ...
    RETURN
    ENTRY C_ENTRY  (args)
    ...
    C_ENTRY  = 'def'
    ...
    RETURN
    END
```

```
C   Character function, asterisk length specifier.
    CHARACTER*(*) FUNCTION  CFUNC (args)
    CHARACTER*(*) C_ENTRY
    ...
    CFUNC = 'abc'
    ...
    RETURN
    ENTRY C_ENTRY (args)
    ...
    C_ENTRY = 'def'
    ...
    RETURN
    END
```

Within a function, all variables whose names are also the names of entries are associated with each other and with the variable that has the name of the function subprogram. Defining any of these variables also defines all associated variables of the same data type. The associated variables need not be of the same type unless the function is type character, but the variable whose name is used to refer to the function must be defined before a RETURN or END statement appears in the subprogram. An associated variable of a different type must not become defined within the subprogram.

ENTRY Examples

Examples of ENTRY statements are

```
ENTRY  CX

ENTRY  SUB3 ( A2, I2, CH3 )

ENTRY  ISUB4

ENTRY  LX ( X2, Y2, ARY2, *)   ! In subroutine.
```

The following example, Figure 7-2, includes a main program, a subroutine called READ_WATCH, and a function called CURRENT_TIME (not shown). CURRENT_TIME simply gets the number of tenths of a second since midnight on the system clock.

Subroutine READ_WATCH models a stopwatch with three buttons. Each button is an entry. A call to the *subroutine* READ_WATCH simply returns the value on the watch. The call to the first *entry* starts the watch; a call to the second entry stops the watch; and a call to the third entry resets the watch to 0.

```
C   Calling program, pertinent statements.

    PROGRAM   MAIN
    INTEGER*4   TIME          ! Variable to hold time.
    ...
C   Start the watch with call to entry START_WATCH.
    CALL  START_WATCH
C   Here, program does operation that it wants to time.
    ...
C   Operation is done; use entries to stop and read watch.
    CALL  STOP_WATCH       ! Stop it.
    CALL  READ_WATCH (TIME)   ! Read the elapsed time.
    PRINT *,  'Watch reads ', TIME, ' tenths of a second.'
C   Reset the watch for the next timed operation.
    CALL  RESET_WATCH
    PRINT *,  'Watch reset.'
    ...
    END


C   Subroutine READ_WATCH. Models a stopwatch with three buttons, each
C   an entry. The entry-buttons are:
C       START_WATCH - Starts the timer; if pressed (entered) when the
C               watch is running, has no effect.
C       STOP_WATCH - Stops the timer; if pressed (entered) when the
C               watch is stopped, has no effect.
C       RESET_WATCH - Resets the time on the watch to 0.
C   Gets accumulated time in tenths of a second from a function called
C       CURRENT_TIME.

    SUBROUTINE  READ_WATCH (TIME)

    LOGICAL        RUNNING         ! .TRUE. if watch is running.
    INTEGER*4      TIME,           ! For time value.
   +               START_TIME,     ! For start time.
   +               CURRENT_TIME,   ! For time from external function.
   +               DISPLAY_TIME    ! To hold time if watch not running.

    SAVE  RUNNING, START_TIME, DISPLAY_TIME   ! Save entries.

    DATA  RUNNING, START_TIME, DISPLAY_TIME  / .FALSE., 0, 0 /

C ---------------------------------------------------------------------
C   READ_WATCH, returns current time on watch.

    IF (RUNNING) THEN
        TIME = CURRENT_TIME() - START_TIME    ! Use function to get time.
    ELSE
        TIME = DISPLAY_TIME                   ! Else initialize time.
    END IF
    RETURN
```

Figure 7-2. ENTRY Program Example (continues)

```
C ------------------------------------------------------------------------
C   Entry to start watch.

    ENTRY  START_WATCH
    IF (.NOT. RUNNING) THEN
C       Compute new "virtual" starting time.
        START_TIME = CURRENT_TIME () - DISPLAY_TIME
        RUNNING = .TRUE.
     END IF
    RETURN

C ------------------------------------------------------------------------
C   Entry to stop watch.

    ENTRY  STOP_WATCH

    IF (RUNNING) THEN
        DISPLAY_TIME = CURRENT_TIME() - START_TIME
        RUNNING = .FALSE.
    END IF
    RETURN

C ------------------------------------------------------------------------
C   Entry to reset watch.

    ENTRY  RESET_WATCH

    IF (RUNNING) THEN
        START_TIME = CURRENT_TIME ()
    ELSE
        DISPLAY_TIME = 0
    END IF
    RETURN

    END                         ! End of stopwatch routines.
```

Figure 7-2. ENTRY Program Example (concluded)

Using SAVE to Preserve Subprogram Entities

Normally, when F77 executes a RETURN or END statement in a subprogram, all local entities in the subprogram (except DATA- or constant-list-assigned values and common blocks) become undefined.

The SAVE statement preserves the local variables and arrays with the names and values assigned to them by the subprogram. Use SAVE primarily to preserve values for reuse within the subprogram on subsequent references or calls to the subprogram.

Values saved are accessible only to the subprogram that saved them, unless these values are in common. Thus a SAVE statement doesn't provide communication between units; the best vehicles for communication between units are actual-dummy arguments and common storage.

SAVE is a specification statement and as such must precede statement function definitions and executable statements in a program unit. The form of SAVE is

SAVE *[v [,v] ...]*

where:

v is any variable name, array name, or named common block name surrounded by slashes. *v* cannot be a dummy argument name, subprogram, or statement function name, or name of an entity *within* a common block. You cannot save a *v* name more than once in a program unit; e.g., SAVE A,B,A is illegal.

If you include a SAVE statement in a subprogram, F77 will preserve the values of the entity names given. If you omit arguments, it will preserve the values of *all* entities defined by the subprogram. You can use more than one SAVE statement in a subprogram.

One or more SAVE statements in the main program saves main program entities in static storage rather than in the F77 runtime stack.

It is not necessary to save any common block with DG's F77; DG's F77 always preserves the values of both named and blank common.

As an alternative to the SAVE statement, you can use the compiler switch /SAVEVARS; for example, F77/SAVEVARS MYPROG. This switch has the effect at runtime of preserving all local values in the program units compiled — precisely the effect of a global SAVE in each program unit.

Whether or not you use SAVE or the /SAVEVARS switch, DG's F77 preserves the values of the following items on return from any subprogram:

- entities specified in SAVE statements;
- entities in both blank and named common blocks;
- entities whose values were assigned with a DATA or type statement *clist*.

SAVE Examples

```
SAVE

SAVE IARRAY, J
```

A contextual example is

```
      SUBROUTINE  JFIX (I2, K2)
      REAL  XRY(200)
      SAVE        ! Save XRY and other locals.
      ...
C     Manipulate data in local array XRY.
      ...
      RETURN

      ENTRY  LENTRY     ! Enter here.
C     Use data SAVEd in local array XRY.
      ...
      RETURN
      END
```

Using RETURN to Return to the Calling Unit

The RETURN statement returns control from a subprogram to the calling program unit. Its form is

RETURN *[expr]*

where:

expr is an integer expression that indicates an alternate statement in the calling program to receive control. *expr* applies to subroutines only; you cannot specify alternate returns in a function subprogram.

A subprogram can have one, more than one, or no RETURN statement. If you omit a RETURN statement, the subprogram's END statement will have the same effect as RETURN.

Normal RETURN Statement

You can use RETURN without an *expr* in either a function or subroutine. From a function, RETURN passes control to the function reference in the calling unit and also returns the function's value. From a subroutine, RETURN passes control to the statement following the CALL statement in the calling program unit.

Alternate RETURN Statement

You may use the alternate RETURN statement, with *expr*, only in a subroutine. This allows you to return control to any labelled statement in the calling program whose label you gave as an alternate return specifier to the subprogram. The *expr* must be an integer expression.

The *expr* represents the number of the corresponding asterisk, among other asterisks, in the subroutine's dummy argument list. For example, the statement RETURN 2 indicates the alternate return specifier held in the second asterisk in the list.

If *expr* has a value of less than 1 or greater than the total number of asterisks in the dummy argument list, F77 returns control as for a normal RETURN.

To route an alternate RETURN, supply an alternate return specifier as an actual argument in the CALL statement, and insert an asterisk as the corresponding argument in the SUBROUTINE dummy argument list. An alternate return specifier is an asterisk followed by the label of the executable statement; e.g., *90.

Execution of any RETURN (or END) statement in a subprogram terminates the association between the subprogram's dummy arguments and the current actual arguments. In addition, all entities within the subprogram become undefined, except for the following:

- entities saved by SAVE statements;
- entities initialized by DATA or type statement *clists*;
- entities declared in named or blank common.

RETURN Example

This example shows alternate returns.

```
C   Calling Unit.

    CALL SUBR2 (A, *100, B, *200)
    ! Normal RETURN here.
    ...
100 ...  ! RETURN 1 returns here.
    ...
200 ...  ! RETURN 2 returns here.
    ...
    END


C   Subprogram.

    SUBROUTINE  SUBR2 (A2, *, B2, *)
    ...
    IF (A2 .GT. 1.0) THEN
        RETURN 1
    ELSE IF ( INT(A2) .EQ. 1 ) THEN
        RETURN
    ELSE
        J = INT ( A2 / Y*COS(A2) )
        RETURN J
    END IF
    RETURN
    END
```

Subroutine SUBR2 returns selectively to the calling unit. If F77 executes the statement RETURN 1, control returns to the statement labelled 100 in the calling unit, because 100 was passed into the first asterisk of the argument list. If F77 executes the RETURN, control returns to the statement following CALL in the calling unit. If F77 executes the RETURN J, when the value of J is 1, control returns to the statement labelled 100 in the calling unit; if J has the value 2, control returns to the statement labelled 200 in the calling unit. If J has neither 1 nor 2, F77 will return to the statement following CALL.

Using EXTERNAL to Identify a Subprogram Name

EXTERNAL allows you to use an externally defined subprogram as an actual argument. You can omit EXTERNAL unless you want to pass a subprogram name to a subprogram. EXTERNAL is a specification statement and as such must precede statement function definitions and executable statements in a program unit. The form of EXTERNAL is

EXTERNAL name *[,name]* ...

where:

name — is the name of a function, subroutine, or block data subprogram or of a dummy procedure. You can declare this name EXTERNAL only once in a program unit.

The EXTERNAL statement tells F77 that a given name is the name of a subprogram instead of a variable or array name. You must use EXTERNAL for a subprogram or dummy procedure name that will appear as an actual argument in a function reference or in a CALL statement.

If you declare an intrinsic function name as EXTERNAL, you cannot use it as an intrinsic function within this program unit. Generally, if you want to pass an intrinsic function to a subprogram, you should use INTRINSIC in the calling program. If you want to *use* an intrinsic function in any unit, do so directly.

You cannot use a statement function name in an EXTERNAL statement.

EXTERNAL Example

The following example passes a subprogram name to a subprogram as an actual argument. The subprogram then uses this name to make reference to the subprogram.

```
C   Main program.
    PROGRAM  PASS
    EXTERNAL  FUNC_REAL
    ...
    CALL  SUB6 (X, Y, FUNC_REAL)
    ! SUB6 returns here.
    END


C   Subroutine.
    SUBROUTINE  SUB6 (X1, Y1, F_NAME)
    ...
    PRINT *,  F_NAME(Z)   ! Function reference.
    ...
    RETURN      ! To caller.
    END


C   Function.
    FUNCTION  FUNC_REAL (Z2)
    ...
    FUNC_REAL = nnn
    ...
    RETURN  ! To caller.
    END
```

For another example, see "Subprogram Arguments", earlier.

Using PROGRAM to Name the Main Program

Naming a main program may help you visually to distinguish its listing from a subprogram's. If you omit a PROGRAM statement, the default main program name is .MAIN. Either the name assigned by a PROGRAM statement or the default name provides a reference for the high-level debugger, SWAT, and for the operating system's debugger.

Only one program unit of the unit(s) you link to form the application program can be a main program. If that unit has a PROGRAM statement, PROGRAM must be the first statement in the program unit.

The form of PROGRAM is

PROGRAM pgm

where:

pgm is the symbolic name for the main program. This name must differ from all subprogram and common block names within this program unit. A valid symbolic name is 1 to 32 characters and includes letters, numbers, question marks, and underscores; the first character must be a letter or the question mark. MP/AOS and MP/AOS-SU names follow the same rules except that they are unique only up to the first ten characters.

A main program can include only one PROGRAM statement and cannot include statements that identify it as a different unit; e.g., FUNCTION, SUBROUTINE, or BLOCK DATA.

Note that a name given with PROGRAM, FUNCTION, SUBROUTINE, or BLOCK DATA does not necessarily relate to the program unit's *filename*. You determine each filename when you create the file, via a text editor or otherwise; throughout program development, the compiler and link programs maintain the major part of this name (described in Chapter 9).

PROGRAM Example

```
PROGRAM  SUPERVISOR
COMMON  / CBLK /  ARRY(200), ACOM, BCOM
EXTERNAL  SUB, SUB1, FUNC, FUNC1
...
END
```

Using COMMON Storage

The COMMON statement is useful for all application programs that include subprograms. COMMON allows multiple program units to declare one or more blocks of storage for their common use. Each declaring unit can then use part or all of the blocks as needed.

There are two main reasons for using common storage. The first is to make values available to any program unit that needs them. (Subprograms with actual-dummy arguments can also do this, but represent a different approach to data sharing.) The second reason is to conserve storage space by allowing several program units to allocate storage only once for commonly used variables and arrays.

COMMON statements are specification statements; as such, they must appear before function statements and executable statements. If a COMMON statement will use IMPLICIT or PARAMETER information, it must follow the statements that specify the implicit data types or parameters. The general order is as given in Chapter 1.

The form of COMMON is

COMMON *[/ [bname] /]* vlist *[[,] / [bname] / vlist] ...*

where:

bname is the symbolic name of a common block. If you include *bname*, this block is *named common.* A valid symbolic name is 1 to 32 characters and includes letters, numbers, question marks, and underscores; the first character must be a letter or the question mark. MP/AOS and MP/AOS-SU names follow the same rules except that they are unique only up to the first ten characters.

If you omit *bname*, this block is *blank common.* If you omit the first *bname*, the first two slashes are optional.

vlist is a list of variable names, array names, and array declarators.

What COMMON Does

A *common block* is a storage area shared by two or more program units. F77 allots the block to variables and arrays in the order you specify in each vlist. We explain the two kinds of common blocks, named and blank, in the next section. A common block *bname* is a symbolic name but has no data type. A common block *bname*, but not a vlist entity name, may appear more than once in a COMMON statement and more than once in a program unit. F77 simply continues to allot the block to variables and arrays in the vlist order. For example, the statements

```
COMMON  /AA/ B,C,D  /ZZARY/ PINT, QT

COMMON /ZZARY/ E,K,Q   /AA/ MOND,SATD
```

allot the entities to common blocks AA and ZZARY in the same order as:

```
COMMON  /AA/ B,C,D,MOND,SATD
+   /ZZARY/ PINT,QT,E,K,Q
```

A common block may have the same name as any local entity except a constant, intrinsic function, or — in a function — a variable that has the same name as the function. If a common block and variable have the same name, all references to the name outside of COMMON or SAVE statements apply *only to the variable.*

You cannot use a vlist entity name more than once in a COMMON statement within a program unit. The vlist names used in different program units have no direct relation to one another; you can use different names within different units. But, to avoid confusion, you might want to use the same vlist item name in all units for an item that the units will have access to. For example, either of the following approaches allows the two program units to share a 200-element array, but the example that uses the same vlist names is clearer.

Same vlist Names

```
PROGRAM  MAIN
COMMON /A/ RY(200)
...
END

SUBROUTINE  XX
COMMON /A/ RY(200)
...
END
```

Different vlist Names

```
PROGRAM  MAIN
COMMON /A/ RY(200)
...
END

SUBROUTINE  XX
COMMON /A/ YY(200)
...
END
```

Within a program unit, you can dimension an array with a DIMENSION, type, or COMMON statement. But you can dimension the same array only once within the program unit (as with any array). For example, either of the following sequences sets up a real array in common:

```
COMMON /BK/ RY1(50)
```

```
REAL RY1(50)
COMMON /BK/ RY1
```

Unless a vlist entity has already been typed when you declare it with COMMON, it is typed according to an IMPLICIT statement. Or, if you omit IMPLICIT, it is typed according to the name rule.

You can increase the size of a common storage area by using either another COMMON statement or an EQUIVALENCE statement, but you cannot extend the area backwards (as further described under the EQUIVALENCE statement).

As an extension, DG's F77 allows you to declare both character and noncharacter data types within the same common block; e.g.,

```
COMMON  / BOTH /   CX, B
CHARACTER*20  CX
CX = 'abc'
B = 1.21
```

is legal. Note that if you put numeric and character data in one common block, each numeric entity *must begin on a 16-bit word boundary*. This means that if numeric entities follow character entities, the number of characters in all the character entities must be even. As F77 builds a COMMON block, it treates logical*1/byte entities the same as character entities: both may begin on a word or byte boundary.

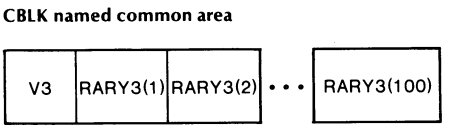
Named and Blank Common Storage

You specify named common storage by preceding the vlist entities with a *bname* within slashes. Different program units can then declare *bname* common with their own vlist entities. The named common blocks declared in different program units must all be the same size, although their vlist entities can be different. (If the blocks are different sizes, F77 may use the largest size and not report an error.)

For example, the following two programs share an array and variable in named common block CBLK:

```
PROGRAM  TEST6
COMMON / CBLK /   V3, RARY3(100)
...
END

SUBROUTINE  SUB6
COMMON / CBLK /   V3, RARY3(100)
...
RETURN
END
```

Whichever program unit has control can have access to and/or modify variable V3 and array RARY3 — these COMMON entities will retain the last values placed in them. Within the different program units, these entities need not have the same name as long as they have the same data type. They look like this in storage:



DG-02302

Within named area CBLK, V3 is the first entity, RARY3(1) is the second, RARY3(2) is the third, and RARY3(100) is the 101st. Of course, you process each entity by its entity name, not by its position within the common area. But keeping track of the position is important if the subprogram sets up the entities differently; for example,

```
COMMON /CBLK/ RX, RY, RZ(90), R1(9)
```

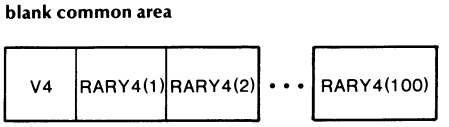
Note that all common blocks that have the same name must have the same size. In the example, if subroutine SUB6 described entities that would need more or less than 101 real storage locations, the program would be in error. For example, COMMON /CBLK/ V3, RARY3(99) would be an error. (F77 might not report this error, and instead allot the largest block declared.)

To specify *blank* common, omit the *bname*. If the blank common immediately follows the COMMON statement (no named common precedes it), you can omit the slashes. F77 will store the vlist entities in blank common, without a name. The blank common blocks declared by different program units can differ in size.

The following programs share an array and variable in blank common storage:

```
      PROGRAM  TEST7
      COMMON  V4, RARY4(100)
      ...
      END

      SUBROUTINE  SUB7
      COMMON    V4, RARY4(100)
C     COMMON // V4, RARY4(100) would
C        also specify blank common.
      ...
      RETURN
      END
```

As with named common, the variable and array will retain the last values assigned to them by any program unit. The names need not be the same in different program units, as long as they indicate the same data type. The blank common area shared by the two programs looks like this in storage:



DG-02303

In the blank common area, V4 is the first entity, RARY4(1) is the second, RARY4(2) is the third, and RARY4(100) is the 101st. As with named common storage, the position within the block is not critical to programming unless one or more subprograms reconfigures the common block entities.

For either named or blank common, program units can have access to the vlist entities in the order in which they were placed in the named common block or in the blank common area.

For either named or blank common, DG's F77 preserves the values of all local variables and arrays on a subprogram RETURN or END statement.

DG's F77 also allows you to DATA-initialize entities in blank common as well as named common.

COMMON Examples

The following example places a double-precision variable and 100-element array in named common, where two program units have access to them.

```
C   Main program.
    PROGRAM  XX
    DOUBLE PRECISION  DP
    COMMON / BLOK1 /   DP, IRY(10,10)
    ...
    END


C   Subprogram.
    SUBROUTINE  XXSUB
    COMMON / BLOK1 /   R(2), IRX(100)
    ...
    RETURN
    END
```

Double-precision value DP requires the space of two real values, so the subprogram uses a two-element real array to provide this. The subprogram could also have used two real variables or a double-precision variable to provide the space. Storage in common area BLOK1 is as shown in Figure 7-3.

The next example associates variable names in both named and blank common storage. Code not shown might process the data in the common blocks.

```
C   Main program.
    PROGRAM  EXCHANGE
    COMMON PESO,Q  /MONEY/ CHNG,J // FRANC,P
    ...
    CALL  SUBR1
    ...
    END


C   Subprogram.
    SUBROUTINE  SUBR1
    COMMON  YEN,S  /MONEY/ RANK,ILIST // PENCE,H,
   + FOO(50)
    ...
    RETURN
    END
```

The subprogram SUBR1 above associates its YEN and S with the main program's variables PESO and Q; it also associates its PENCE and H with the main program's FRANC and P. All four associations are in blank common. The subprogram also places FOO(50) in blank common (illustrating that blank common areas in different units need not be the same size).

In the common block named MONEY, the subprogram associates variables RANK and ILIST with the main program's variables CHNG and J. Note that all data types correspond.

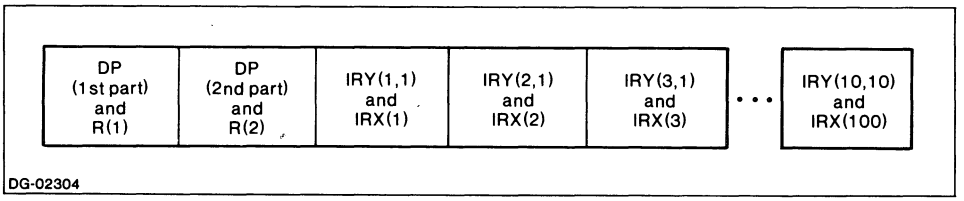


Figure 7-3. COMMON Storage in Named Common Block BLOK1

The final COMMON example is a variation of the program-subprogram example REVERS shown earlier in this chapter under "Subroutines." It has an array in common and a subprogram that reverses the elements of an array. It uses the common array instead of dummy arguments to pass values.

```
C   Calling program.
    PROGRAM  TESTIT
    COMMON   / BLOK2 / NUM, ARY(200)
    PRINT *,  'Type number of elements and'
    PRINT *,  '   values for them. '
    READ (*,*)  NUM,  (ARY(I), I= 1, NUM)
    CALL  REVERS
    PRINT *, 'Here they are: ', (ARY(I), I=1, NUM)
    END


C   Subprogram.
    SUBROUTINE  REVERS
    COMMON  / BLOK2 / NUM2, ARY2(200)
    MIDDLE = NUM2 / 2   ! Get middle element.
    DO 90  I = 1, MIDDLE ! Reverse.

C       Use a temporary variable to reverse
C          array elements.

        TEMP = ARY2(I)
        ARY2(I) = ARY2( NUM2 + 1 - I )
        ARY2( NUM2 + 1 -I ) = TEMP
90  CONTINUE
    RETURN
    END
```

Block Data Subprograms

A *block data subprogram* is really a set of DATA statements used to assign initial values to entities in named and blank common blocks.

You never make a reference to or call a block data subprogram from other program units; you simply compile it, which assigns initial values to the named blocks. Then link it along with other program unit modules. F77 allows more than one BLOCK DATA subprogram in the program units that make up the application program.

DG's F77 allows you to DATA-initialize COMMON entities in *any* program unit, thus you are not *required* to use a BLOCK DATA subprogram for this.

A BLOCK DATA subprogram begins with a BLOCK DATA statement of the form:

BLOCK DATA *[name]*

and it ends with an END statement. Only one BLOCK DATA statement is allowed in the subprogram. The name is optional, but if you include the name, you cannot use this name in the subprogram or as the name of another subprogram or common block in the application program. Like FUNCTION, BLOCK DATA is a nonexecutable statement, so the compiler ignores a label.

Between the BLOCK DATA and END statements, you can use only specification statements: IMPLICIT, DIMENSION, type statements, DATA, EQUIVALENCE, PARAMETER, SAVE, and COMMON.

The common block names you give in a BLOCK DATA subprogram should be names of blocks declared in other program units (they will have no effect on other program units if they are not). When you specify a named common block, its total size (but not necessarily its vlist names) should be the same as the size declared in each program unit. (But if the sizes differ, F77 may use the largest size and not report an error.) You need not initialize all vlist entities in each block. For example, if a named common block will use 200 storage units, and you want to initialize the last 100 of these to the value 3.5, either of the following two BLOCK DATA subprograms would do the job:

```
    BLOCK DATA  INIT1
    REAL  RXY(200)
    COMMON   / BLK6 / RXY
    DATA   (RXY(I), I=101,200)   / 100*3.5 /
    ...
```

or

```
    BLOCK DATA  INIT1
    REAL  R1, R2, RX(98), RZ(100)
    COMMON   /BLK6/  R1, R2, RX(98), RZ(100)
    DATA  RZ   / 100*3.5 /
    ...
```

You can place character and noncharacter data types in the same common block as long as the noncharacter data begins on a 16-bit word boundary. As F77 builds a common block, it treates logical*1/byte entities the same as character entities: both may begin on a word or byte boundary. The common block names pertain to every subprogram, but the variable and array names are irrelevant (except as a vehicle to help you assign values in the block data subprogram).

BLOCK DATA Example

```
C   Program unit whose COMMON values
C      will be initialized.
    ...
    character*60  CH1, CH2
    common / CHAR_INT / CH1, CH2,  J, JRY(100)
    double precision  DP1, DP2
    common / REAL_DP /  R, RRY(100), DP1, DP2
    logical  LOG1, LOG2
    common / LOG / LOG1, LOG2


C   Block data subprogram.
    block data  INIT8
    character*60  CH1, CH2
    common / CHAR_INT / CH1, CH2,  J, JRY(100)
    double precision  DP1, DP2
    common / REAL_DP / R, RRY(100), DP1, DP2
    logical  LOG1, LOG2
    common / LOG / LOG1, LOG2

    data   CH1, CH2  / 'Init val1', 'Init val2' /
    data   J, JRY  / 26, 100*192 /
    data   R,RRY /1.0,100*0.8 /  DP1,DP2 /2*1.2D2/
    data   LOG1, LOG2  / .TRUE., .FALSE. /
    end
```

Note that the DATA statements above could have been included in the original program unit, eliminating the need for the BLOCK DATA subprogram.

Summary of Rules for Functions and Subroutines

1. There are three kinds of functions and one kind of subroutine:
   - Intrinsic functions, supplied with F77 (e.g., Y=SIN(X)).
   - Statement functions, which you define in a statement with optional dummy arguments, then refer to using actual arguments; for example, define as X(D1,D2)=D1+D2/2.0, make reference to as R=X(var1,var2).
   - Function subprograms, which are separate program units that begin with a FUNCTION statement, must assign a value to their names. You invoke a function subprogram by making a reference to it; for example, X=FUNC(A,B) or PRINT *, FUNC(A,B).
   - Subroutines, like functions, are separate program units, but they begin with a SUBROUTINE statement, cannot assign a value to themselves, and they have no data type. The calling unit invokes them via a CALL statement.
2. Function and subroutine subprogram references can contain actual arguments, whose values will be used for the subprogram's dummy arguments. The subprogram can change these values. Dummy argument names must match the data type of the corresponding actual argument. Arguments can include numeric, logical, character, or Hollerith data. An argument can be an intrinsic function name if the caller declares the intrinsic function name INTRINSIC. An argument can be a subprogram name if the caller declares the subprogram name EXTERNAL.
3. Normally, execution begins at the beginning of a subprogram but you can define an ENTRY at one or more points within any subprogram with the ENTRY statement — then refer to the ENTRY directly from the caller. If an ENTRY is in a function, the caller invokes it with a function reference; if the ENTRY is in a subroutine, the caller invokes it with CALL.
4. Normally, most variables local to a subprogram (aside from common blocks) become undefined when the subprogram returns. The SAVE statement, or compiler switch /SAVEVARS, preserves the values of some or all of these local variables for later reuse within the subprogram.
5. The PROGRAM statement allows you to name the main program.
6. The COMMON statement, like the actual-dummy argument mechanism, can provide communication between program units and allows them to share data. Each program unit can declare named or blank common blocks of variables or arrays. Each unit can then use the entities within the block as needed, assigning new values for other units. Common blocks with the same name must be declared to occupy the same size in each unit that uses them. Blank common, declared without slashes, can be declared with different sizes in different units. All common areas are accessible to program units according to the variables and arrays given in their COMMON lists. Any common block — named or blank — can contain character and noncharacter data.

End of Chapter

Chapter 8
Statements and Directives, Alphabetically

Chapters 1–7 of this book describe F77 features in a traditional, contextual way.

This chapter describes all F77 statements and directives alphabetically. It is designed for fast reference. Each statement is a concise, condensed presentation of relevant material given in earlier chapters. It does not include introductory and contextual material given earlier. It has no new information.

The only statement described earlier in more substantive detail is FORMAT (Chapter 6). In this chapter, the FORMAT edit descriptors are sketched alphabetically — the noncharacter edit descriptors (like /) follow the alphabetic descriptors (like A).

Certain constructs are *defined* in preceding chapters. These are entities, arrays, variables, data types, iolists and cilists (Chapter 5); files, records, edit descriptors, format reversion, and list-directed formatting (Chapter 6); and intrinsic functions (like SIN) and dummy-actual argument handling (Chapter 7). The page references are shown in the index.

This chapter describes the following statements and compiler directives:

= (assignment)
ASSIGN
BACKSPACE
BLOCK DATA
CALL
CHARACTER
CLOSE
COMMON
COMPLEX
CONTINUE
DATA
DIMENSION
DO
DO WHILE
DOUBLE PRECISION
ELSE
END
ENDFILE
END DO
END IF
ENTRY
EQUIVALENCE
EXTERNAL
FORMAT
FUNCTION
GO TO (assigned)
GO TO (computed)
GO TO (unconditional)
IF (arithmetic)
IF (logical)
IF...THEN (block IF)
IMPLICIT
IMPLICIT NONE
%INCLUDE (compiler directive)
INCLUDE
INQUIRE
INTEGER
INTRINSIC
%LIST (compiler directive)
LOGICAL
OPEN
PARAMETER
PAUSE
PRINT
PROGRAM
READ
REAL
RETURN
REWIND
SAVE
SUBROUTINE
WRITE

LOGICAL*1 has BYTE as a synonym (AOS/VS). See the description of the LOGICAL statement.

NOTE: Certain symbols have restrictions because of MP/AOS and MP/AOS-SU requirements. In particular, external names — such as common block and subprogram names — must be unique with respect to each other and within their first 10 characters. For example, you cannot have subroutines SUBROUTINE_1 and SUBROUTINE_2 or labelled COMMON names COMMON_AREA_A and COMMON_AREA_B in one or more program units.

= (assignment statement)

Assigns a value to a variable, array element, or function, or defines a statement function.

Form

v = expr

where:

v — is a variable, array element, or function name. It should be the appropriate data type for the value produced by expr.

expr — is an arithmetic, logical, or character expression.

What It Does

The = assignment operator assigns the current value of expr to v. For example,

```
A = A + B
```

means "evaluate the expression on the right of the = sign (A+B), and store the result in the entity to the left of the = sign (A)." The value of expr must match the general class of v: numeric-numeric, logical-logical, or character-character. You may also assign a character constant expr to a numeric v and a Hollerith constant expr to a numeric v.

The program must have assigned a value to each part of the expression expr before this assignment statement executes.

See Chapter 7 for the explanation of statement functions and their use of the = symbol.

Numeric Values

If expr yields an arithmetic value, v must be numeric, else F77 will signal an error. v can be type integer, integer*4, integer*2, logical*1/byte, real, real*4, real*8, double precision, complex, complex*8, or complex*16. If the numeric types are the same (e.g., RFOO = 2.2), F77 assigns the value of expr to v directly. If the data types differ (e.g., RFOO = 2), F77 converts the expr after evaluation to v's data type and assigns it to v. In the second case, RFOO contains 2.0 after the statement executes. Conversion rules are detailed in Chapter 3, Table 3-1.

Logical Values

If the data type of expr is logical, v must be type logical.

Character Values

If the data type of expr is character, v must be type character or logical*1/byte (AOS/VS). A character expr can be any character string delimited with apostrophes (') or quotation marks ("), or it can be the name of a character entity already assigned a value; or it can be one or more concatenated strings, substrings, or entities. The delimiters ' or " are not assigned to v. The total number of characters assigned to v must be from 1 through 32,767.

F77 replaces the bytes reserved in the CHARACTER or IMPLICIT statement with the characters in expr — starting with the first (leftmost) character position. If the number of assigned characters is smaller than the entity, F77 inserts blanks after the characters to fill v. If the number of assigned characters won't fit, F77 uses the leftmost characters and drops the remaining ones. If v is logical*1/byte, then expr should be 1 character long to avoid truncation of the second through last characters.

To concatenate two or more character exprs, place the concatenation operator (//) between them.

You cannot use the same substring specifier within both v and expr; e.g., for character entity C, C3(1:2) = C3(2:3) is illegal and will yield unpredictable results. However, you can use *differing* substrings of an entity on both sides; e.g., C3(1:2) = C3(3:4) is legal.

= Examples

```
C  Numeric examples.

   LOGICAL*1 LOG1A      ! Or, BYTE LOG1A
                        !   (AOS/VS).
   LOG1A = -120

   J = 1446

   AVG = (A+B+C) /3

   B = C(I) + (SIN(Y)+6)

   B(I,J) = A ((K-1) * 10 + K)

   IVAR = 'POT'         ! 4th byte of IVAR
                        !    is blank (<040>)

   RVAR(2,3) = 4HYEAR

   TOO_SMALL = 5HABCDE ! TOO_SMALL contains only
                        !    'ABCD'; E is ignored.
```

```
C  Logical examples.

   LOGICAL  L, LG(10), Y   ! Logical entities.
   LOGICAL*1 LOG1B         ! Or, BYTE LOG1B
                           !   (AOS/VS).
   ! ...
   L = I .GT. 4            ! .TRUE. or .FALSE.,
                           !    depending on I.

   LG(1) =(X.GT.5.0) .OR. (V.LT.Z) ! .TRUE. if
                                   !   X>5 or V<Z.

   Y = .TRUE.              ! .TRUE.

   LOG1B = .FALSE.         ! .FALSE.



C  Character examples.
C     Declare 2 10-char. and 1 20-char. variables.

   CHARACTER*10  CH,  CH1,  CH2*20
   LOGICAL*1 LOG1C         ! Or, BYTE LOG1C
                           !   (AOS/VS).
   ! ...
   LOG1C = 'Z'
   CH = 'Message is'       ! Assign 10 chars.
   CH1 = 'Goodbye'         ! Assign 7 chars.
   CH1(8:9)='!<BEL>'       ! Assign substring.
   CH2 = CH // CH1         ! Concatenate.
   print *, CH2            ! Print.

C  Compare:
   if ( CH(1:1) .GT. CH1(1:1) )
  +   go to 7
   ! ...
 7 print *,'CH1 leads alphabetically, is ', CH1
```

ASSIGN

Assigns a statement label to an integer variable for subsequent use in an assigned GO TO statement.

Form

ASSIGN s TO v

where:

- s is the label of an executable statement or FORMAT statement in the current program unit.
- v is an integer*4 variable name. It must be integer*4 for AOS/VS but can be either length for AOS, F7716, MP/AOS, and MP/AOS-SU.

What It Does

ASSIGN places the statement label s in variable v, allowing you to use the variable's symbolic name in subsequent assigned GO TO statements or in formatted I/O statements. ASSIGN has no meaning unless you plan to use one or more assigned GO TOs (described under GO TO) or assigned format statements as a format identifier.

ASSIGN Example

```
   ! ...
   ASSIGN 5 TO IPROC ! Assign.
   ! ...
   GO TO IPROC
15 CALL  SUBR1       ! Should label the statement
                     !   following assigned GOTO.
   ! ...
 5 FLAG = 3.0        ! IPROC routine.
   ! ...
```

Control goes to the statement labelled 5.

BACKSPACE

Positions a file before the current record.

Forms

BACKSPACE iu

BACKSPACE (*[UNIT=]* iu *[,IOSTAT=ios]* *[,ERR= s]*)

where:

UNIT= iu — iu is an expression that evaluates to the integer number of the unit you want to BACKSPACE. You can omit UNIT= if the unit specifier is the first argument.

IOSTAT= ios — *ios* is an integer variable or array element that returns a status indicator: 0 for normal, an error code if an error occurred. If you include IOSTAT= without ERR=, execution will continue at the next statement on an error.

ERR= s — *s* is a statement label to which control will go on an error. This must be an executable statement within the current program unit. If you omit both IOSTAT= and ERR=, the program will terminate on an error.

What It Does

BACKSPACE is an auxiliary I/O statement that positions a unit's file before the current record, thus allowing you to reread or rewrite the record. The record can be a formatted, unformatted, or endfile record.

BACKSPACE works only for files connected for sequential access. It can backspace over the endfile record of any file opened for sequential access.

BACKSPACE works on intermediate records (between the START and END positions) with any organization except dynamic. However, backspacing a data-sensitive or variable file takes a lot of time. So, if you want to use BACKSPACE, and care about speed, use it with a FIXED file. BACKSPACE works on a data-sensitive file by treating a sequence of characters that end with a <NL> character as a record.

Backspacing from the first (START position) record or on a device that cannot position a file has no effect.

BACKSPACE Examples

```
BACKSPACE 2

BACKSPACE (IU)
```

BLOCK DATA

Begins a subprogram that initializes blank and named common blocks.

Form

BLOCK DATA *[name]*

where:

name is an optional symbolic name. A valid symbolic name includes 1 to 32 alphanumeric characters, question marks, and underscores; it must begin with a letter or the question mark. MP/AOS and MP/AOS-SU names follow the same rules except that they are unique only up to the first ten characters. The name is optional and for your convenience only. You cannot use *name* elsewhere in the subprogram, nor as a subprogram, main program, common block, or other block data subprogram name.

What It Does

BLOCK DATA starts a BLOCK DATA subprogram; it must be the first statement of a BLOCK DATA subprogram.

BLOCK DATA is a nonexecutable statement on which the compiler ignores a label. Between the BLOCK DATA and END statements, you can use only specification statements: IMPLICIT, IMPLICIT NONE, DIMENSION, type statements, DATA, EQUIVALENCE, PARAMETER, SAVE, and COMMON.

The common block names you give in a BLOCK DATA subprogram should be names of blocks declared in other program units; they will have no effect on other program units if they are not the same.

A BLOCK DATA subprogram is a group of DATA statements that specifies initial values for common blocks. You never refer to or call a BLOCK DATA subprogram from other program units; you simply compile it, which assigns initial values to the named blocks, and then link it along with other program units that form the application program. F77 allows more than one BLOCK DATA subprogram per application program.

DG's F77 allows you to DATA-initialize common entities in *any* program unit; thus you are not *required* to use BLOCK DATA for this.

BLOCK DATA Example

```
C   Program unit whose COMMON values
C   will be initialized.
    ! ...
    character*60   CH1, CH2
    common / CHAR_INT / CH1, CH2, J, JRY(100)
    logical   LOG1, LOG2
    common / LOG / LOG1, LOG2


C   BLOCK DATA subprogram.
    block data  INIT8
    character*60   CH1, CH2
    common / CHAR_INT / CH1, CH2, J, JRY(100)
    logical   LOG1, LOG2
    common / LOG / LOG1, LOG2

    data    CH1, CH2  / 'Init val1', 'Init val2' /
    data    J, JRY    / 26, 100*192 /
    data    LOG1, LOG2    / .TRUE., .FALSE. /
    end
```

CALL

Invokes and passes control to a subroutine subprogram.

Form

CALL sub *[([a [,a] ...])]*

where:

sub is the name of a subroutine.

a is an actual argument.

What It Does

CALL invokes your subroutine, passing control to the first executable statement and using any *a* arguments for the subroutine's dummy arguments. When the subroutine returns, control returns to the statement in the calling unit that follows the CALL statement — unless the subroutine specifies an alternate RETURN statement. Return from the subroutine completes execution of the CALL statement.

You must supply an actual argument, *a*, for each dummy argument in the subroutine's SUBROUTINE statement. An actual argument can be a variable, constant, array element, substring, expression, or array. It can also be an intrinsic function or subprogram name. For an alternate return, *a* must be an asterisk followed by the label of an executable statement in the current program unit.

The data type of each actual argument must match that of the subprogram's corresponding dummy argument. If the data types do not match, results are undefined.

If an actual argument is the name of a subroutine, the data type is irrelevant, but you must have declared this name EXTERNAL in this program unit. If an argument is an intrinsic function name and the subroutine will use this as an intrinsic function, you must have declared the function INTRINSIC in the current unit.

To set up an alternate return, use an asterisk followed by the label of an executable statement as an actual argument. The subprogram's corresponding dummy argument must be an asterisk.

A subroutine can CALL or refer to another subprogram or itself.

CALL Examples

Examples of CALL statements are

```
CALL QUAD (9, Q/R, A(3), 20, R-S**2, X1, HALF)

CALL OPTIONS
```

An example that shows different arguments passed is

```
C   Calling unit.
    CHARACTER*15   CH   / 'a passed string' /
    DIMENSION  ARRY(200)
    A = 90.
    CALL  SUB (CH, ARRY, A, *40)
    ! Normal return here.
40  A = 100.    ! RETURN 1 here.
    ...
    END


C   Subroutine.
    SUBROUTINE  SUB (CH2, ARRY2, A2, *)
    CHARACTER*(*)   CH2
    DIMENSION   ARRY2(*)
    ...
    RETURN 1
    ...
    RETURN
    END
```

CHARACTER

Declares and sizes one or more character variables or arrays.

Form

CHARACTER *[*len] v [(d)] [*len] [/ clist/]*
*[, v [(d)] [*len] [/ clist /]]...*

where:

len specifies the length of a character variable or each element of a character array. If you omit *len*, F77 gives the entity the default length of 1 character. The maximum length of a character entity is 32,767 units (bytes). The *len* specifier can also be a parenthesized asterisk [(*)] as described below.

v is the name you want for the variable, array, function, or dummy argument. This can be any valid symbolic name; the CHARACTER statement overrides IMPLICIT rules, if any.

d applies only to an array and is a dimension declarator; it specifies the dimensions for the array. The declarator can be an integer, integer constant, or, in a subprogram, a dummy expression. For a multiply dimensioned array, include an expression for each dimension. For a lower array bound other than 1, you must specify both upper and lower bounds; e.g., CHARACTER*10 CH(2:9). Declarators are further described under DIMENSION and in Chapter 2.

clist is a constant list that includes one or more values for one or more v entities. The entity(ies) will be initialized to the *clist* value(s) when the program is built. A *clist* value can be a literal constant, enclosed in apostrophes (') or quotation marks ("); e.g., CHARACTER*11 VAL/'balance is:'/; or it can be a named constant defined with a PARAMETER statement. If you initialize an array via *clist*, the *clist* must include precisely enough values to fill the entire array. In a subprogram, the current values of entities initialized via *clist* are automatically saved on return.

What It Does

The CHARACTER statement declares one or more character entities with the names, lengths, and (for arrays) the numbers of elements given.

As a type statement, CHARACTER overrides an IMPLICIT statement. It is a nonexecutable statement; if you label it, no other statement can refer to the label. All type statements are specification statements and as such must precede statement function definitions and executable statements in a program unit.

You can establish the data type of a variable or array, and dimension an array, only once in a program unit.

The Asterisk (*) len

You can use a parenthesized asterisk [(*)] for CHARACTER *len* in two cases: 1) where the current program unit names the entity as a constant with PARAMETER; or 2) where the calling program has declared the *len* explicitly and passes the entity to a subprogram; the subprogram can then use an asterisk for *len* in its CHARACTER statement. Examples of both are

```
C   Case 1. PARAMETER use:
    ...
    CHARACTER*(*)  CABC
    ...
    PARAMETER  (CABC = 'abc')

C   Case 2. Dummy argument use.
C   Calling program.

    PROGRAM  MAIN
    CHARACTER*20  CXX
    ...
    CALL  SPROG (CXX)
    ...
    END

C   Subprogram.
    SUBROUTINE  SPROG (CXX)
    CHARACTER*(*)  CXX
    ...
    RETURN
    END
```

In the first use, the PARAMETER statement makes CABC a three-character entity with the value abc. In the second use, the subprogram picks up the 20-character variable CXX through a dummy argument.

CHARACTER (continued)

CHARACTER Examples

```
      CHARACTER CR / 'N' /  ! Variable, initialized.

      CHARACTER*10  C1      ! Variable.

      CHARACTER*20 AMY(6,8) ! 48-element array.

C     Name a character constant and use it to
C        initialize a character variable.
      character*(*)  C_CON
      parameter   (C_CON= 'abc')
      character*3  C_VAR   /C_CON/

C     Assign values to character entities:
      C1 = 'abcdef'
C     C1 now contains 'abcdef   '.
      AMY(1,8) = 'Goodbye,Ruby Tuesday'
```

The = (assignment statement) shows other examples of character assignment.

CLOSE

Severs the unit/file connection established by an OPEN statement.

Form

CLOSE (*[UNIT=]* iu *[,IOSTAT=ios] [,ERR=s] [,STATUS=sta]*)

where:

UNIT= iu — iu is an expression that evaluates to the integer number of the unit you want to close. You can omit UNIT= if the expression is the first argument.

IOSTAT= ios — *ios* is an integer variable or array element that returns a status indicator: 0 for normal, an error code if an error occurred. If you include IOSTAT= without ERR=, execution will continue at the next statement on an error.

ERR= s — *s* is statement label to which control will go on an error. This must be an executable statement within the current program unit. If you omit both IOSTAT= and ERR=, the program will terminate on an error.

STATUS= sta — *sta* is a character expression that, after F77 has ignored any trailing blanks, is 'KEEP' or 'DELETE'. If 'KEEP' (the default), the file will continue to exist after a CLOSE unless it is a scratch file. (Specifying 'KEEP' for a file you opened with STATUS='SCRATCH' will cause an error.) 'KEEP' will not create a file that does not exist. 'DELETE' will delete the file if it exists.

What It Does

CLOSE is an auxiliary I/O statement that severs the connection between a unit and file, regardless of the number of times you've reopened this unit/file connection.

You can close a unit from any program unit. F77 closes all units automatically when the program terminates. Unless you specify STATUS='DELETE', F77 keeps all files except scratch and unnamed files on a CLOSE.

If you close a preconnected unit, *the preconnection will be severed while this program runs*; for the next program run, the preconnection will be re-established. Opening a different file on any connected unit (including preconnected ones) closes the previous connection and establishes a new connection.

CLOSE Examples

```
CLOSE (2)

CLOSE (IU, IOSTAT=IOS)

CLOSE (22, ERR=99, STATUS = 'DELETE')
```

COMMON

Allots a storage area for access by any program unit and names the arrays and variables that will reside in this area.

Form

COMMON *[/ [bname] /]* vlist *[[,] / [bname] / vlist]]...*

where:

bname is the symbolic name of a common block. If you include *bname*, this block is *named common.* A valid symbolic name is 1 to 32 characters and includes letters, numbers, question marks, and underscores; the first character must be a letter or the question mark. MP/AOS and MP/AOS-SU names follow the same rules except that they are unique only up to the first ten characters. Common block names have no data type.

If you omit *bname*, this block is *blank common.* If you omit the first *bname*, the first two slashes are optional.

vlist is a list of variable names, array names, and array declarators. You cannot use the same name twice within a vlist, nor can you use the names of dummy arguments.

What It Does

COMMON allows multiple program units to declare one or more blocks of storage for their common use.

COMMON statements are specification statements; as such, they must appear before statement function statements and executable statements. The general order is given in Chapter 1.

F77 allots each common block to variables and arrays in the order that you specify them in each vlist.

A common block *bname*, but not a vlist entity name, may appear more than once in a COMMON statement and more than once in a program unit. A common block may have the same name as any local entity except a constant, intrinsic function, or — in a function — the function name. If a common block and variable have the same name, all references to the name outside of COMMON or SAVE statements apply *only to the variable.*

COMMON (continued)

You cannot use a vlist entity name more than once in a COMMON statement within a program unit. The vlist names used in different program units have no direct relationship to one another; you can use different names within different units. But, to avoid confusion, you might want to use the same vlist item name in all units for an item that the units will access.

Within a program unit, you can use either a DIMENSION, type, or COMMON statement to dimension an array, but can dimension the same array only once within the program unit (as with any array).

Unless a vlist entity has already been typed when you declare it COMMON, it is typed by an IMPLICIT statement or — if you omit IMPLICIT — by the name rule.

You can increase the size of a common storage area by using either another COMMON statement or an EQUIVALENCE statement, but you cannot extend the area backwards with EQUIVALENCE. This is further described under EQUIVALENCE.

For either named or blank common, DG's F77 preserves the current values of all local variables and arrays on a subprogram RETURN or END statement.

DG's F77 allows you to DATA-initialize entities in blank common as well as named common.

DG's F77 allows you to declare both character and noncharacter data types within the same common block; e.g.,

```
COMMON  / BOTH /   CX, B
CHARACTER*20  CX
CX = 'ABC'
B = 1.21
```

is legal. Note that if you put numeric and character data in one common block, each numeric entity *must begin on a 16-bit word boundary*. This means that if numeric entities follow character entities, the number of characters in all the character entities must be even. As F77 builds a common block, it treates logical*1/byte entities the same as character entities: both may begin on a word or byte boundary.

Named and Blank Common Storage

You specify named common by preceding the vlist entities with a *bname* within slashes. Different program units can then declare *bname* common with their own vlist entities. The named common blocks declared in different program units must all be the same size, although their vlist entities can be different.

To specify blank common, omit the *bname*. The blank common blocks declared by different program units can differ in size.

Program units can have access to the vlist entities in the order in which they were placed in the specific named common block or the blank common area.

COMMON Examples

The following two programs share an array and variable in named common block CBLK:

```
PROGRAM  TEST6
COMMON / CBLK /  V3, RARY3(100)
...
END


SUBROUTINE  SUB6
COMMON / CBLK /  V3, RARY3(100)
...
RETURN
END
```

Whichever program unit has control can have access to and/or modify variable V3 and array RARY3 — common entities will retain the last values placed in them. Within the different program units, these entities need not have the same name as long as they have the same data type.

Within named area CBLK, V3 is the first entity, RARY3(1) is the second, RARY3(2) is the third, and RARY3(100) is the 101st. All common blocks that have the same name must have the same size.

The following programs share an array and variable in *blank* common:

```
PROGRAM  TEST7
COMMON   V4, RARY4(100)
...
...
END


SUBROUTINE  SUB7
COMMON   V4, RARY4(100)
...
RETURN
END
```

In the blank common area, V4 is the first entity, RARY4(1) is the second, RARY4(2) is the third, and RARY4(100) is the 101st. Blank common blocks in different program units need not be the same size. The next example associates variable names in both blank and named common.

```
C   Main program.
    PROGRAM  EXCHANGE
    COMMON // LIRA,Q,  /MONEY/ CHGE,J,
   +       // FRANC,P
    ...
    CALL  SUBR1
    ...
    END


C   Subprogram.
    SUBROUTINE  SUBR1
    COMMON // MARK,S,  /MONEY/ RANK,ILIST,
   +       // PENCE,H,FOO(50)
    RETURN
    END
```

The subprogram above associates its MARK and S with the main program's variables LIRA and Q; it also associates its PENCE and H with the main program's FRANC and P in blank common. The subprogram also places FOO(50) in blank common (illustrating that blank common areas in different units need not be the same size).

In the common block named MONEY, the subprogram associates variables RANK and ILIST with the main program's variables CHGE and J. Note that all data types correspond.

COMPLEX

Declares one or more variables or arrays as type complex.

Form

COMPLEX *[*len]* v *[(d)]* *[*len]* *[/ clist/]*
[, v [(d)] *[*len]* *[/ clist /]]...*

where:

len is an optional byte length specifier for the variable or array elements. F77 reserves 8 units (bytes) for a standard complex entity (COMPLEX or COMPLEX*8): 4 bytes for the real part and 4 bytes for the imaginary part. For double precision, you can specify a *len* of 16, which reserves 16 bytes: 8 for the real part and 8 for the imaginary part.

v is the name you want for the variable, array, function, or dummy argument. This can be any valid symbolic name; the COMPLEX statement overrides any IMPLICIT rules.

d applies only to an array and is a dimension declarator; it specifies the dimensions for the array. The declarator can be an integer, integer constant, or dummy expression. For a multiply dimensioned array, include an expression for each dimension. For a lower array bound other than 1, you must specify both upper and lower bounds; e.g., COMPLEX ZX(2:9). Declarators are further described under DIMENSION and in Chapter 2.

clist is a constant list that includes one or more values for one or more v entities. The entity(ies) will be initialized to the *clist* value(s) when the program is built. A *clist* value can be a complex constant of the form (real, imag); e.g., COMPLEX Z /(2.3,6.1E4)/; or it can be a named complex constant defined with PARAMETER. If you initialize an array via *clist*, the *clist* must include precisely enough values to fill the entire array. DG's F77 preserves the current values of all entities initialized via *clist* when it executes a subprogram's RETURN or END statement.

What It Does

The COMPLEX statement declares one or more complex entities with the names, lengths, and (for arrays) the numbers of elements given.

COMPLEX (continued)

As a type statement, COMPLEX overrides an IMPLICIT statement. It is a nonexecutable statement; if you label it, no other statement can refer to the label. All type statements are specification statements and as such must precede statement function definitions and executable statements in a program unit.

You can establish the data type of a variable or array, and dimension an array, only once in a program unit.

COMPLEX Examples

```
COMPLEX  ZX              ! Variable.

COMPLEX  ZY /(1.,3.E4)/ ! Variable with both
                         !   parts initialized.

complex*16  ZZ  /(1.0D4, 4.3D5)/ ! Complex*16
                 ! variable initialized to
                 !    double-precision values.

COMPLEX  Z2(20,20)       ! Array.

COMPLEX COMPX_1
COMPX_1 = 'POTATO'       ! 7th and 8th bytes of
                         !   COMPX_1 are blank
                         !   (<040>).
```

CONTINUE

Provides a place for a statement label.

Form

CONTINUE

What It Does

CONTINUE is an executable statement but has no effect. It is designed for, and generally used as, the terminating statement of a DO loop. It is a placeholder that makes the program clearer and easier to understand.

You must label a CONTINUE statement when using it as a DO terminator. Otherwise, you need not label CONTINUE and it can appear anywhere in your program where an executable statement is allowed.

CONTINUE Example

```
    ...
    TOTAL = 0.0
    DO 80  J = 1, L
       TOTAL = TOTAL + ARY(J)
80  CONTINUE
    ...
```

DATA

Initializes variables and array elements to given values.

Form

DATA vlist / clist / *[,vlist / clist /]...*

where:

vlist is a list of one or more variable, array element, character substring, and/or array names. It can also include implied DO lists.

clist is a list of constants that will be assigned to the vlist items. A constant can be a numeric, character, logical, or Hollerith constant; it can also be the name of a constant whose value you will assign with PARAMETER. The clist can also include a repeat count, described below. In a subprogram, the current values of entities initialized via clist will be saved when F77 executes a RETURN or END statement.

What It Does

The DATA statement initializes variables, substrings, and array elements to the values you specify before the program executes. Having initialized an entity with DATA, you cannot reinitialize that entity with DATA.

DATA is a nonexecutable statement. If you label it, you cannot refer to that label in another statement.

The number of entity names in vlist must match the number of constants in clist; if not, the compiler signals an error. For example, if IVAR is a variable name, the statement DATA IVAR/1,2/ would evoke an error message. The order and grouping of names and value lists is unimportant as long as the correct name/value association is maintained.

In the DATA statement, you can either insert the entity names one by one, then their values; e.g.,

```
DATA V1, ARY(1),ARY(2),ARY(3) /99., 8., 9., 6./
```

Or, to insert the same values in multiple entities, you can use a repeat count. A repeat count has the form:

n*constant

where n is the number of times you want the constant inserted. For example:

```
DATA V1, ARY(1), ARY(2), ARY(3) / 2*1., 2*4.0 /
```

Or you can use only the array name and give a repeat count for some or all its elements:

```
DIMENSION  IX(100)     ! A 100-element array.
DATA  IX  / 100*3 /    ! Init each element to 3.
```

If the repeat count specifies too many elements for the array, the compiler signals an error. You can specify *a portion* of an array in a repeat count, to initialize part of the array.

You can also use an implied DO list with DATA.

If the data types of a variable and its corresponding constant do not match, F77 converts the constant to the variable's type. Rules for this are detailed in Chapter 3, Table 3-1.

To initialize a variable or element of type complex, enclose the complex value in parentheses; e.g., DATA X/(1.0,1.4)/ for the complex number $X=1.0+1.4*i$.

DATA Examples

```
DIMENSION  R(8)
DATA  A, R   / 7.1, 8*3.3 /
```

This initializes A to 7.1 and the 8 elements of R to 3.3.

```
REAL  R(10)
CHARACTER*10  C, C1
DATA  R(3), C, C1 /7E4, 'Size is', 'Size was'/
```

This initializes element R(3) to 7.0*10**4 (=70000.), variable C to Size□is□□□, and variable C1 to Size□was□□.

```
COMMON / BLOK1 /  RARY(100),  R2
DATA  RARY, R2   / 100*0.0,  8.99 /
```

This initializes each element of common array RARY to 0.0 and common variable R2 to 8.99.

DATA (continued)

```
      DIMENSION AY(24), BY(24)
      DATA  (AY(I), BY(I), I = 1,21)  / 22*1., 20*2. /
```

This example uses an implied DO. It initializes the first 11 elements of arrays AY and BY to 1.0 and the next 10 elements of AY and BY to 2.0. The last 3 elements of AY and BY are undefined.

```
      INTEGER*4 VALID_GRADES(8)
      DATA VALID_GRADES
     1    / 'A', 'B', 'C', 'D', 'E',
     2      'WP', 'WF', 'INC' /
C           The last 3 are respectively
C             withdrawn-passing, withdrawn-failing,
C             and incomplete.
```

This statement pair initializes the first element of VALID_GRADES to <101><040><040><040>, the sixth element to <127><120><040><040>, and the last element to <111><116><103><040>. The *clist* elements may also be 1HA, 1HB, ..., 3HINC for the same results.

DIMENSION

Names and dimensions arrays.

Form

DIMENSION a (d) *[/ clist /][, a (d) [// clist //]]...*

where:

a — is the symbolic name of an array. If a preceding IMPLICIT or type statement has not specified a data type for this name, the type is integer or real by the name rule. The array data type determines the size of each element.

d — is a dimension declarator: usually one or more integers or integer expressions that establishes the dimensions of the array.

clist — is a list of values that you want F77 to insert in the elements of the array.

What It Does

DIMENSION names and dimensions arrays of the data types given by IMPLICIT or a preceding type statement. DIMENSION statements are nonexecutable. You can label a DIMENSION statement, but no other statement can refer to the label.

DIMENSION is a specification statement and as such must precede statement function definitions and executable statements in a program unit, as detailed in Chapter 1.

You can establish an array's dimensions only once per program unit, in a DIMENSION, COMMON, or type statement.

Dimension Declarators

If you use only one expression per dimension (e.g., IFOO(3)), it is the upper bound, and the lower bound is 1. To specify a lower bound other than 1, insert both a lower and upper bound expression, separated by a colon; e.g., DIMENSION IFOOA(-1:9). In this case, IFOOA has 11 elements: IFOOA(-1), IFOOA(0), IFOOA(1), IFOOA(2), ..., IFOOA(9).

If the array will have more than one dimension, separate the dimension expressions with commas; e.g., DIMENSION RARRAY(9,10). F77 calculates the number of storage locations for an array by taking the product of the dimensions. For example, array AMY(4,6,6) gets 144 (4*6*6) elements and array FRED(0:3,-1:2) gets 16 (4*4) elements.

In a subprogram, instead of using constant integer expressions to establish the dimensions, you can use integer dummy arguments; then use the dummy arguments to dimension the array. For example:

```
C   Calling program.
    DIMENSION  ARRY(20, 30)
    I = 20
    J = 30
    CALL  SUB (ARRY, I, J)
    ...
    END


C   Subprogram.
    SUBROUTINE  SUB (ARRY2, I2, J2)
    DIMENSION  ARRY2 (I2, J2)
    ...
    END
```

This is called an "adjustable array" and is detailed in Chapter 7, "Arguments to Function and Subroutine Subprograms".

Initial Values

The initial values you give in *clist* are stored when the program is built; they cannot be reassigned via a DATA statement or a second DIMENSION *clist*.

Between the slashes (/.../), you can specify initial single values for sequential elements, separated by commas (e.g., DIMENSION IARY(4)/9,5,-3,2/). Or you can specify the same initial value for multiple elements with the number of elements, an asterisk, and the value; e.g., DIMENSION ORAY(4)/4*0.0/ initializes each element to zero while DIMENSION ORAY(4)/3*0.0,1./ initializes elements 1–3 to zero and element 4 to one.

Before you assign a value to a variable or array element, it may contain any value; it is not initialized before you use it. So for some applications you may want to set it to zero. In a subprogram, the current values of entities initialized via *clist* are saved on return.

DIMENSION Examples

```
    DIMENSION  ZRY(10,10), IRY(30)

    CHARACTER  CHR
    DIMENSION  CHR(10)


    INTEGER*2 NEW_ENGLAND_STATES
C       The New England States are either
C          northern (Vermont, New Hampshire,
C          Maine) or southern (Massachusetts,
C          Connecticut, Rhode Island).
    DIMENSION NEW_ENGLAND_STATES(2,3)
   1     / 'VT', 'MA', 'NH', 'CT', 'ME', 'RI' /
C       NEW_ENGLAND_STATES contains  VT NH ME
C          in its first row and      MA CT RI
C          in its second row.
C
C       The second line in the DIMENSION statement
C          could have been
C  1     / 2HVT, 2HMA, 2HNH, 2HCT, 2HME, 2HRI /
```

DO

Executes a group of statements zero or more times.

This statement has two forms. The first is the traditional (pre-F77) DO statement, sometimes called the labelled DO statement. We explain it in detail. Then, we describe the second form of the DO statement, known as the unlabelled DO statement.

Form: (Labelled) DO

DO s [,] i = e_1, e_2 [, e_3]

sts

s terminal statement

where:

s is the label of an executable statement within the current program unit. This is the *terminal statement*. It can be any statement except these: another DO, an unconditional or arithmetic GO TO, IF...THEN, ELSE IF, ELSE, END DO, END IF, RETURN, STOP, or END. CONTINUE is a good choice for a terminal statement. If the terminal statement is a logical IF, it may contain any executable statement except DO, IF...THEN, ELSE IF, ELSE, END IF, END, or other logical IF.

i is an integer (including logical*1/byte) or real variable name, called the *DO-variable*.

e_1 is an integer (including logical*1/byte) or real expression that is the initial value for the DO-variable, called the *initial parameter*.

e_2 is an integer (including logical*1/byte) or real expression that is the terminal value for the DO-variable, called the *terminal parameter*.

e_3 is an integer (including logical*1/byte) or real expression that gives the number by which the DO-variable will be incremented in each pass through the loop; it is called the *incrementation parameter*. If you omit this parameter, the DO-variable will be incremented by the default value, 1, in each pass. It can have any value valid for its data type, *except 0*. An increment of 0 would run the DO loop forever.

sts are any statements, except one that redefines i.

What It Does

DO sets up a loop that begins at the DO and ends at the terminal statement labelled s. This set of statements is "the range of the DO". By default, the range of the DO can execute zero or more times; but you can override this at compilation time to ensure that the loop executes at least once (F77/DOTRIP=1 MYPROG).

DO Loop Execution

When it encounters a DO or implied DO, F77 takes the following steps:

- It examines the data types of i, e_1, e_2, and e_3 (if present), evaluates expressions in any of these, then converts the data types of each e to the data type of i (if needed).
- It assigns initial parameter e_1 to the DO-variable, i.
- It effectively establishes the *trip count* (number of passes through the range of the DO).
- If the trip count is not 0, F77 executes the range of the DO. Unless a statement transfers control out of the loop, F77 executes the terminal statement s (implied in an implied DO) and increments the DO-variable by the value of the incrementation parameter e_3.
- F77 then tests the DO-variable against the terminal parameter e_2. If the DO-variable, in positive or negative steps according to the increment, has not passed the terminal parameter, F77 repeats the loop. If the DO-variable has passed the terminal parameter, F77 exits from the loop and passes control to the first executable statement *after* the terminal statement.

The DO-variable i and e parameters can be any integer or real data type. They can have any valid value except 0 for e_3. It is good practice to make all these arguments one data type even though F77 will convert them to match the type of i. Note that the trip count may not be exact for real arguments because the representation of fractions is often a close approximation, not absolutely exact. For example,

```
C   Exactly 10 times.
    DO 20 I = 1, 10
       PRINT *, I
20  CONTINUE
    STOP
    END
```

```
C   Perhaps 10 times.
    DO 30 X = .3, 1.2, .1
       PRINT *, X
30  CONTINUE
    STOP
    END
```

After a DO loop terminates, control goes to the first executable statement after the terminating statement s (or to the next statement in an implied DO). The e arguments retain the values they had when the loop began. The i argument has the value set during the final incrementation. If you try to change the increment i value within the loop, F77 will report an error at compilation time. You *can* change the value of the e parameters but this will not affect the trip count; it will have other effects. See the explanation of the /DOTRIP F77 switch in Chapter 9 for additional information.

Nesting DO Loops

You can nest DO loops to any depth. The range of an inner DO loop cannot extend beyond the range of the outer DO loop, but they may share the same terminal statement. The following example shows correct nesting:

```
      DO 10  J = 1,90          ! Outer loop.
      ...
             DO 20  K = 1,6    ! Inner loop.
             ...
20           CONTINUE          ! Inner terminator.
      ...
10    CONTINUE                 ! Outer terminator.
```

Two loops may use the same terminal statement if they execute in normal sequence. However, a statement in the outer loop may not transfer control to the terminal statement.

Implied DO Lists

You can imply a DO loop without using a DO statement by using an *implied DO list*. An implied DO list exists entirely within an I/O or DATA statement and has the form:

(dlist , i = e_1, e_2 *[*, e_3*]*)

where dlist is a list of array element names for a DATA statement or an iolist for a READ, WRITE, or PRINT statement. The i and e parameters are exactly the same as for loops with DO statements.

Implied DO loops don't work with expressions in FORMAT statements because of optimization that the FORTRAN 77 compiler performs on these loops. An example is

```
      WRITE (10, 20) (MINE(K), K = 1, 4)
   20 FORMAT ( I<>  )
```

MINE(1) through MINE(4) do *not* appear with successive edit descriptors I1, I2, I3, and I4. Instead, a runtime error occurs.

DO Loop Restrictions

- You may not transfer control into the range of a DO from elsewhere in the program unit.
- You may not terminate the range of a DO with another DO or any statement described in the DO form description. CONTINUE is a traditional choice for the terminal statement.
- If you use a real value for i, the trip count may not be precise because the internal representation of real numbers is often approximate. This restriction does not apply to implied DOs because the loop variable must be an integer.
- You may not redefine the increment i within the range of a DO loop. You can redefine the other parameters but this will not affect the trip count.
- If you place a DO loop in a block IF, you must place the entire range of the loop within the IF block, ELSE IF block, or ELSE block.
- If you place a block IF in a DO loop, you must terminate the block IF (with END IF) within the loop.

DO Loop Examples

```
C   Compute the sum of all elements in an array.
C      Assume array ARY(24) exists and has real
C      values.

      ...
      TOTAL = 0.0
      DO 7  I = 1, 24
         TOTAL = TOTAL + ARY(I)
7     CONTINUE
      ...
```

An implied DO list example is

```
      DIMENSION  RY(10)
      READ *,   (RY(J),  J = 10, 1, -1)
      WRITE  (1, '(1X, E10.2)' )  (RY(K), K = 1, 10)
```

This accepts 10 values from the terminal and assigns them to RY elements in descending order: RY(10), RY(9), ..., RY(1). Then it writes each value as a record to unit 1. (To assign values in *ascending* order, no implied DO would be needed; you'd simply use the array name as the iolist entity.)

DO (continued)

Form: Unlabelled DO

DO i = e_1, e_2 *[, e_3]*

sts

END DO

where:

i, e_1,
e_2, are as explained for the labelled DO statement.
e_3,
sts

Labelled and Unlabelled DO Statement Differences

The functionality of these two statements is identical. Their names imply their difference: One has a labelled statement as a terminator and the other has an unlabelled statement (END DO) as a terminator.

Two example program segments show the difference between labelled and unlabelled DO statements. The programs yield identical results.

```
      PROGRAM LABEL_DO
      ...

      DO 10 I = 1, 5
         SUM = SUM + ARY(I)
   10 CONTINUE

      END



      PROGRAM UNLABEL_DO
      ...

      DO I = 1, 5
         SUM = SUM + ARY(I)
      END DO

      END
```

DO WHILE

Executes a group of statements zero or more times.

Form

DO *[s [,]]* WHILE (log_expr)

where:

s	is the statement label of an END DO statement. This END DO statement appears within the current program unit, after the DO WHILE statement.
log_expr	is a logical expression.

What It Does

The DO WHILE statement functions similarly to the plain iterative DO statement. However, a logical expression, not a numeric variable, controls the loop.

The terminating statement of a DO WHILE loop must be an END DO statement. If *s* appears in the DO WHILE statement, then *s* must be the label of the corresponding END DO statement. Like iterative DO loops, DO WHILE loops may nest within each other. Also like iterative DO loops, you can't transfer into the range of a DO WHILE loop from outside the loop.

When the DO WHILE statement executes, it evaluates log_expr. If log_expr is true, execution continues with the first statement in the range of the DO loop. When this execution arrives at the corresponding END DO statement, control returns to the DO WHILE statement, which retests the logical expression. On the other hand, if the value of log_expr is false, the DO loop becomes inactive. Then execution continues with the next statement after the corresponding END DO statement.

DO WHILE Restrictions

- You must not transfer control into the range of a DO WHILE from elsewhere in the program unit. You may transfer control out of the range of a DO WHILE.
- You must terminate the range of a DO WHILE statement only with an END DO statement, not with a CONTINUE statement.
- If you place a DO WHILE loop in a block IF, you must place the entire range of the loop within the IF block, ELSE IF block, or ELSE block.
- If you place a block IF in a DO WHILE loop, you must close the block IF (with END IF) within the loop.

DO WHILE Examples

```
      INTEGER ARRAY(5) ! TO CONTAIN 99, 98, 97,
                       !    96, AND 95.
      I = 1
      DO WHILE (I .LE. 5)
             ARRAY(I) = 100 - I
             I = I + 1
      END DO
      STOP
      END



      INTEGER ARRAY(5) ! TO CONTAIN 99, 98, 97,
                       !    96, AND 95.
      I = 1
      DO 10 WHILE (I .LE. 5)
             ARRAY(I) = 100 - I
             I = I + 1
10    END DO
      STOP
      END



      CHARACTER*80 LINE
      OPEN (2, FILE="MY_DATA_LINES")
      IER = 0
      DO WHILE (IER .EQ. 0)
             READ (2, 10, IOSTAT=IER) LINE
10           FORMAT (A)
C            Process LINE.
      END DO
      ...
```

DOUBLE PRECISION

Declares one or more variables or arrays of type double precision.

Form

DOUBLE PRECISION v *[(d)] [/ clist /]*
[, v [(d)] [// clist //]]...

where:

v — is the name you want for the variable, array, function, or dummy argument. This can be any valid symbolic name; the DOUBLE PRECISION statement overrides any IMPLICIT rules.

d — applies only to an array and is a dimension declarator; it specifies the dimensions for the array. The declarator can be an integer, integer constant, or dummy expression. For a multiply dimensioned array, include an expression for each dimension. For a lower array bound other than 1, you must specify both upper and lower bounds; e.g., DOUBLE PRECISION Z(2:9). Declarators are further described under DIMENSION and in Chapter 2.

clist — is a constant list that includes one or more values for one or more v entities. The entity(ies) will be initialized to the *clist* value(s) when the program is built. A *clist* value can be a numeric constant, character constant, Hollerith constant, or named double-precision constant defined with PARAMETER. If you initialize an array via *clist*, the *clist* must include precisely enough values to fill the entire array. DG's F77 preserves the current values of all entities initialized via *clist* when it executes a subprogram's RETURN or END statement.

DOUBLE PRECISION (continued)

What It Does

The DOUBLE PRECISION statement declares one or more double-precision entities of the names and (for arrays) the numbers of elements given. The double-precision data type is the same as REAL*8.

A double-precision number must include the letter D and can include an exponent field. It can range from 5.4*10**-79 to 7.2*10**75. It has significance to about 16.4 decimal digits.

As a type statement, DOUBLE PRECISION overrides an IMPLICIT statement. It is a nonexecutable statement; if you label it, no other statement can refer to the label. All type statements are specification statements and as such must precede statement function definitions and executable statements in a program unit.

You can establish the data type of a variable or array, and dimension an array, only once in a program unit.

DOUBLE PRECISION Examples

```
      DOUBLE PRECISION  DP

      DOUBLE PRECISION  DX  / 403.2D-4 /

      DOUBLE PRECISION  DY(100)

      DOUBLE PRECISION  ENTREE / 'Lobster' /
C     Eighth byte of ENTREE is padded with
C        a blank (<040>).
```

ELSE

Starts an alternative block of statements in a block IF (IF...THEN).

What It Does

The ELSE IF(expr) or ELSE statements introduce a block of statements that will execute conditionally. They are components of a block IF, described under IF...THEN.

ELSE Example (in block IF)

```
...
IF (HOURS .GT. 50.) THEN
        DTIME = 2.0 * RATE * (HOURS - 50.)
        OTIME = 1.5 * RATE * 10.
        REGPAY = RATE * 40.
ELSE IF (HOURS .GT. 40.) THEN
        DTIME = 0.
        OTIME = 1.5 * RATE * (HOURS - 40.)
        REGPAY = RATE * 40.
ELSE
        DTIME = 0.
        OTIME = 0.
        REGPAY = RATE * HOURS
END IF
PAY = DTIME + OTIME + REGPAY
...
```

If the HOURS is greater than 50.0, the IF block statements execute and control passes through the END IF to the PAY assignment statement. If HOURS is greater than 40.0 and less than or equal to 50.0, the ELSE IF block statements execute and control goes to END IF and out of the block IF. If HOURS is not greater than 40.0, the IF and ELSE IF block statements are skipped, the ELSE statements execute, and control passes on through the END IF.

END

Returns to the calling program unit.

Form

END

What It Does

The END statement marks the end of the program unit and returns to the calling unit. END must be the last statement in a program unit and you must include it in every program unit.

The letters E, N, and D must be on the same line, in that order.

In the main program, END returns to the initial program environment, which halts the program *without* a termination message. STOP also returns to the initial program environment and halts the program, but *with* a termination message.

In a function or subroutine subprogram, END returns control to the calling program. RETURN in a subprogram has the same effect.

END Examples

```
C   Main program.
    PROGRAM  MONITOR
    ...
    ...
    END


C   Function subprogram.
    CHARACTER FUNCTION  ALPHA (D1, D2)
    ...
    ...
    END


C   Subroutine subprogram with multiple
C      RETURN points.
    SUBROUTINE  BETA (A, B, C)
    ...
    ...
    RETURN
    ...
    RETURN
    END
```

ENDFILE

Writes an end-of-file marker to a file opened for sequential access.

Forms

ENDFILE iu

ENDFILE (*[UNIT=]* iu *[,IOSTAT=ios] [,ERR=s]*)

where:

UNIT= iu	iu is an expression that evaluates to the integer number of the unit to which you want to write an endfile record. You can omit UNIT= if the unit specifier is the first argument.
IOSTAT= ios	*ios* is an integer variable or array element that returns a status indicator: 0 for normal, an error code if an error occurred. If you include IOSTAT= without ERR=, execution will continue at the next statement on an error.
ERR= s	*s* is a statement label to which control will go on an error. This must be an executable statement within the current program unit. If you omit both IOSTAT= and ERR=, the program will terminate on an error.

What It Does

ENDFILE is an auxiliary file I/O statement. It writes an end-of-file (endfile) record as the file's next record and positions the file *after* this end of file. ENDFILE is not required to establish an end of file, but does allow you to truncate an existing file.

As you write records to a file, the end of file is implicitly just after the last record written. After ENDFILE writes an endfile record to the file, you cannot read or write the file's records unless you execute a BACKSPACE or REWIND statement.

ENDFILE is designed for files connected for sequential access; it will return an error on a file connected for direct access.

You cannot write an ENDFILE record if the file is open on another unit *or* if another process has the file open. If either condition exists, F77 will return an error from the ENDFILE statement.

ENDFILE (continued)

Furthermore:

- When ENDFILE executes, any records past the record that ENDFILE writes are truncated. Thus, ENDFILE's effect persists beyond the life of the file's connection during the program's execution.
- When ENDFILE executes, normally your program must execute a BACKSPACE or REWIND statement to read or write the file's records. The exception to this rule occurs with a terminal (a file with type CON). This exception allows simple conversational I/O between you and your program to continue.

ENDFILE Examples

```
ENDFILE 10

ENDFILE (UNIT = IUN, IOSTAT = IERR)
```

END DO

Serves as the terminal statement of a DO WHILE statement.

What It Does

END DO is always the terminal statement for a DO WHILE or unlabelled DO statement. As its name implies, END DO simply ends a DO loop.

If a DO WHILE statement contains a statement number, the corresponding END DO statement must have that number as its label. The unlabelled DO statement does not contain a statement number, so its corresponding END DO statement has no need for a label.

END DO Examples

```
      INTEGER ARRAY(5) ! TO CONTAIN 99, 98, 97,
                       !    96, AND 95.
      I = 1
      DO WHILE (I .LE. 5)  ! DO WHILE LOOP.
             ARRAY(I) = 100 - I
             I = I + 1
      END DO
      STOP
      END



      INTEGER ARRAY(5) ! TO CONTAIN 99, 98, 97,
                       !    96, AND 95.
      I = 1
      DO 10 WHILE (I .LE. 5)  ! DO WHILE LOOP.
             ARRAY(I) = 100 - I
             I = I + 1
10    END DO
      STOP
      END



      INTEGER ARRAY(5) ! TO CONTAIN 99, 98, 97,
                       !    96, AND 95.
      DO I = 1, 5      ! UNLABELLED DO LOOP.
             ARRAY(I) = 100 - I
             I = I + 1
      END DO
      STOP
      END
```

END IF

Terminates a block IF (IF...THEN).

What It Does

END IF closes a block IF, introduced by an IF...THEN statement. You can use it only in a block IF, described under IF...THEN.

END IF Example (in block IF)

```
C   Swap the values of 1 and 2.

    READ *,  K
    IF (K .EQ. 1) THEN
       K = 2
    ELSE IF (K .EQ. 2) THEN
       K = 1
    ELSE
       PRINT *,  'The value of K is incorrect.'
    END IF

    END
```

ENTRY

Defines an entry point within a subprogram.

Form

ENTRY en *[([d [,d] ...])]*

where:

en is the symbolic name of the entry point. If the entry is in a function subprogram, the en name has a data type; you can imply the type or specify it in a type statement before or after the ENTRY statement. If the entry is in a subroutine subprogram, it has no data type. A valid symbolic name is 1 to 32 characters and includes letters, numbers, question marks, and underscores; the first character must be a letter or the question mark. MP/AOS and MP/AOS-SU names follow the same rules except that they are unique only up to the first ten characters.

d is a dummy argument. An entry's dummy arguments can differ in number and type from the dummy arguments in other entries or those in its parent subprogram. Each *d* must match the data type of the corresponding actual argument in the statement that makes reference to or calls the entry — unless the actual argument is a subprogram name. If the entry is in a subroutine, *d* can be an asterisk to allow an alternate return. You can omit the parentheses for an empty argument list in a subroutine entry; *but you must always include the parentheses in a function entry and in the entry reference.*

What It Does

Normally within a subprogram, execution begins at the first executable statement after the initial FUNCTION or SUBROUTINE statement. The ENTRY statement allows you to define other starting points as entries in a subprogram.

An entry can appear anywhere between the initial FUNCTION or SUBROUTINE statement and the END statement, but *not* within a block IF or range of a DO loop.

ENTRY is a nonexecutable statement that F77 skips when it executes the subprogram. You can label ENTRY statements but cannot refer to the label. In a function subprogram, the entry name en must not appear before the ENTRY statement, except within *one* type statement (or a COMMON statement). Within one program unit, you cannot declare an entry name EXTERNAL or use it as a dummy argument.

ENTRY (continued)

You refer to an ENTRY much the same way as you do to the subprogram that contains it. For an ENTRY in a function, use a function reference; for an ENTRY in a subprogram, use a CALL statement. You refer to an ENTRY directly from any program unit except the subprogram that contains it.

On an entry reference or CALL, F77 transfers control from the calling unit to the first executable statement following ENTRY, using any actual arguments for the entry's dummy arguments. An entry's dummy argument list need not be the same as other entries' or its parent subprogram's — but it must match the actual argument list in the statement that makes reference to or CALLs the entry.

A dummy argument name that appears in an ENTRY statement must not appear in an executable statement that precedes that ENTRY statement — unless the name also appears in a FUNCTION, SUBROUTINE, or ENTRY statement that precedes the executable statement.

If the entry is in a function, the statements after ENTRY must assign a value to the entry name (as is true for any function).

On a RETURN or END statement, control passes back to the calling unit at the function reference or after CALL; that is, unless in a subroutine the RETURN statement specifies a different return point.

ENTRY Rules for Function Subprograms

In a function, an ENTRY cannot be type character unless the function itself is type character (CHARACTER*len FUNCTION func or type statement within the function). All character entries must have the same length as the function itself: integer or asterisk.

Within a function, all variables which are the names of entries in that function are associated with each other and with the variable named for the function subprogram. Defining any of these variables also defines all associated variables of the same data type. The associated variables need not be of the same data type unless the function is of type character, but the variable whose name is used when referring to the function must be defined before a RETURN or END statement appears in the subprogram. An associated variable of a different type must not become defined within the subprogram.

ENTRY Examples

```
      ENTRY  CX2

      ENTRY  SUB3 (A2, I2, CH3)

      ENTRY  ISUB4

      ENTRY  JFIX (J, K, ARY, *)   ! In subroutine.


C     A contextual example.
C     Calling program.
      PROGRAM  MAIN
      ...
      CALL  ENT (CX)
      ...
      CALL  ENT1 (X, Y)
      ...
      CALL  ENT2 (Z, CH)
      ...
      END

C     Subroutine.
      SUBROUTINE  ENT (CX1)
      ...
      ENTRY  ENT1 (X1, Y1)
      ...
      ENTRY  ENT2 (Z1, CH1)
      ...
      END
```

EQUIVALENCE

Associates two or more entities within the same storage area.

Form

EQUIVALENCE (vlist) *[, (vlist)]..*

where:

vlist is a list of *two or more* variables, arrays, and/or array elements that have integer constants for subscripts. Each vlist must contain at least two names. In a subprogram, names of dummy arguments must not appear in a vlist. If a variable name is also a function subprogram name, this name must not appear in a vlist.

What It Does

EQUIVALENCE associates one or more variables, array elements, or arrays within one storage area.

EQUIVALENCE is a specification statement and as such must precede statement function definitions and executable statements in a program unit.

For each vlist, the compiler allots storage to the largest entity, then associates other items in this vlist with this storage area. You can include any noncharacter data type in an EQUIVALENCE statement with any other noncharacter data type, or any character type with another character type, or any noncharacter entity with a character entity. The logical*1/byte datatype (AOS/VS) may share storage with character and noncharacter data.

You must have dimensioned an array, via DIMENSION, type statement, or COMMON statement, before it or any of its elements can appear in an EQUIVALENCE statement.

You must have all numeric, logical*2, and logical*4 entities beginning on a word boundary. Character and logical*1/byte entities may begin on a word or byte boundary.

You can specify the first array element in an EQUIVALENCE expression by either:

- A Complete Subscript. For example, A(1,1,1) indicates the first element of three-dimensional array A (assuming all three lower bounds are 1).
- No Subscript. This (e.g., A) indicates the first element of the array.

Having an element of one array in an EQUIVALENCE statement with an element of another array implicitly creates an EQUIVALENCE relation with all other corresponding elements of the two arrays. For example, the statements

```
DIMENSION     A(5,4), B(20)
EQUIVALENCE   (A(1,1), B(1) )
```

not only indicate that array elements A(1,1) and B(1) share the same storage locations, but also that A(2,1) and B(2), A(3,1) and B(3), etc., also share the same storage locations.

```
DIMENSION     C(20), D(20)
EQUIVALENCE   (C(20), D(1) )
```

indicates that C(20) and D(1) share the same storage location.

EQUIVALENCE in Common Blocks

If an entity is part of a common block, the entity cannot be in an EQUIVALENCE statement with another entity in that or any other common block.

An array element in the program unit can be in an EQUIVALENCE statement with an entity of a common block, if doing so does not add locations to the beginning of the block. For example, the statements

```
COMMON  B, C, D
DIMENSION  ARRAY(4)
EQUIVALENCE   (B, ARRAY(1))
```

extend the common block validly. The entire array ARRAY is equivalenced to common; but, since locations are added to the end of the common block, this is allowed.

An array element and an entity in a common block can not be in an EQUIVALENCE statement if — via the insertion of the whole array — it would extend the block by placing array elements before the block's beginning. The statements

```
COMMON   B, C, D
DIMENSION   ARRAY(3)
EQUIVALENCE   (B, ARRAY(3))
```

attempt to extend the common block invalidly by placing elements ARRAY(1) and ARRAY(2) before the block's beginning. This kind of mistake would evoke a compiler error message.

EQUIVALENCE Examples

```
DIMENSION    A(100), B(0:99)
EQUIVALENCE  ( A(1), B(0)   )
```

Array B, elements 0:99, and array A, elements 1:100, share the same storage locations. A(1) and B(0) will contain the same value, as will A(2) and B(1), A(3) and B(2), ..., and A(100) and B(99).

```
DIMENSION  VALUES(10), SUBTOTALS(9)
EQUIVALENCE  (VALUES, SUBTOTALS),
+    (VALUES(10), TOTAL)
```

This specifies that elements 1 through 9 of VALUES and SUBTOTALS share the same locations and that VALUES(10) and variable TOTAL share the same location.

EXTERNAL

Identifies a subprogram name for use as an actual argument.

Form

EXTERNAL name *[,name]...*

where:

name — is the name of a function subprogram, subroutine subprogram, BLOCK DATA subprogram, or dummy procedure. You can declare this name EXTERNAL only once in a program unit.

What It Does

The EXTERNAL statement tells F77 that a given name is the name of a subprogram instead of a variable or array name. You must use EXTERNAL for a subprogram or dummy procedure name that will appear as an actual argument in a function reference or in a CALL statement.

If you declare an intrinsic function name as EXTERNAL, you cannot use it as an intrinsic function within this program unit. Use INTRINSIC, not EXTERNAL, if you will want to pass an intrinsic function name to a subprogram.

EXTERNAL is a specification statement and as such must precede statement function definitions and executable statements in a program unit.

You cannot use a statement function name in an EXTERNAL statement.

EXTERNAL Example

```
C   Calling program.
    EXTERNAL  MYFUNC
    ...
    CALL  SUB (MYFUNC)
    ...
    END


C   Subprogram.
    SUBROUTINE  SUB (EXTFUN)
    ...
    PRINT *, EXTFUN(Z)   ! Uses function passed.
    ...
    END
```

FORMAT

Specifies a format for a READ, WRITE, or PRINT statement.

Form

f FORMAT (format-specification)

where:

f	is the format identifier. In the context of a FORMAT statement, f is always a statement label.
format-specification	is the format specification, which includes edit descriptors that format input or output, control positioning, sign, or blank interpretation, or start new records, as described below.

What It Does

The FORMAT statement (or character expression that contains a format) establishes a format for transfer of one or more formatted records. A FORMAT statement must be labelled. A READ, WRITE, or PRINT statement specifies the unit, FORMAT statement label, and records to be transferred; F77 then transfers the records according to the FORMAT specification.

FORMAT is a nonexecutable statement and can occur anywhere in a program unit. The FORMAT statement identified in any I/O statement must be in the same program unit as the I/O statement. FORMAT gets more detail in Chapter 6.

As an extension, Data General F77 allows expressions such as F<KWIDTH>.<KDECIMALS> in FORMAT statements. Chapter 6 contains the details of this extension.

Edit Descriptors

A FORMAT specification includes zero or more edit descriptors. These fall into two categories: repeatable and nonrepeatable. Each FORMAT statement must include *at least one repeatable edit descriptor,* if its corresponding I/O statement has an iolist. The F77 edit descriptors appear alphabetically in this section. In summary, they are

A	Alphanumeric (character) editing, repeatable.
B	Binary integer editing, repeatable.
BN/BZ	Blank interpretation (blank null, blank zero), nonrepeatable.
D, E	Real-number, explicit-exponent editing, repeatable.
F	Real-number, floating-point editing, repeatable.
G	Real-number, generalized (F or E) editing, repeatable.
H	Character-constant output editing, nonrepeatable.
I	Integer editing, repeatable.
L	Logical editing, repeatable.
O	Octal integer editing, repeatable.
P	Scale factor control, nonrepeatable.
S,SP,SS	Sign control for numeric output editing, nonrepeatable.
T,TL,TR	Tab (position) control, nonrepeatable.
X	Position right control, nonrepeatable.
Z	Hexadecimal integer editing, repeatable.
' "	(apostrophe or quotation mark) Character output editing, nonrepeatable.
$	(dollar) Suppress NEW LINE after record output, with LIST carriage control, nonrepeatable.
:	(colon) Terminate format control conditionally, nonrepeatable.
/	(slash) Start a new record, nonrepeatable.

A (alphanumeric) edit descriptor, form

[r] A*[w]*
where *r* is integer repeat count;
w is integer field width in characters.

F77 transfers characters via A without interpretation; you cannot use these directly for computations. If you omit *w*, F77 uses the width declared for the corresponding I/O list entity, which can be as large as 32,767 bytes. If you include *w* for A or any descriptor, it cannot exceed 255.

For input to A, the iolist entity can be of any data type. If you include optional *w*, and *w* is smaller than the iolist entity, F77 left justifies *w* characters, padding on the right with blanks. If *w* is larger then the iolist entity, F77 reads the rightmost characters from the record into the entity.

For output from A, if you include optional *w*, and *w* is larger than the number of characters in the entity, F77 inserts leading blanks and outputs *w* characters. If *w* is less than the number of characters in the entity, F77 outputs the leftmost *w* characters in the entity.

The following example shows the A, I, and ' descriptors; it also shows list-directed formatting (PRINT * sequences).

FORMAT (continued)

A Example, Input and Output

```
      CHARACTER*20  CFN, CSTA*10
      PRINT *,  'Type filename for OPEN. '
      READ (*, 60)  CFN
60    FORMAT (A)   ! A descriptor without w.
      WRITE (*, 70)  CFN
70    FORMAT (1X, 'You specified ', A)
      PRINT *,  'Unit number? '
      READ (*, 80)   IU
80    FORMAT (I3)   ! Integer (I) descriptor.
      WRITE (*, 90)  IU
90    FORMAT (1X, 'You specified', I5)    ! I again.
      PRINT *,  'File status desired? '
      READ (*, 60)  CSTA
      WRITE (*, 70)   CSTA
      OPEN (IU, FILE=CFN, STATUS=CSTA)
      PRINT *,  'File OPEN completed.'
      ...
```

The dialog on the terminal might be

Type filename for OPEN.□:UDD:JACK:PFILE ⤸
You specified□*UDD:JACK:PFILE*□□□□□□
Unit number?□28 ⤸
You specified□□□*28*
File status desired?□FRESH ⤸
You specified□*FRESH*□□□□□
File OPEN completed.

B (binary integer) edit descriptor, form
*[r]*Bw *[m]*
where *r* is an integer repeat count;
w is an integer that gives the field width in digits;
m is an integer that gives the minimum width for output; F77 will insert leading 0s to pad to *m*. But if *m* and the entity are 0, F77 will output blanks for the value.

The B descriptor transmits binary integer data to or from an iolist entity. The entity must be type integer or, for AOS/VS F77 only, logical*1/byte.

On input, F77 reads w digits from the record; a sign or any characters other than digits 0 and 1 will produce an error. If the value read is greater than 111...1 (32 1s for a 4-byte integer) or 111...1 (16 1s for a 2-byte integer) or 11111111 (for an AOS/VS logical*1/byte entity), F77 signals an error.

On output from B, F77 converts the iolist value to binary and outputs it as w digits. If the number of digits is less than w, F77 pads the value with leading blanks to w. If the number exceeds w, F77 outputs w asterisks. If you included *m*, F77 inserts leading 0s to pad the value to *m* characters.

B Examples, Input

Edit Descriptor	Characters Read	Value Stored in iolist Entity
B2	11	3
B5	01101	13
B8	11111111	255 *(integer*2)*
B8	11111111	-1 *(logical*1/byte)*
B8	10000000	-128 *(logical*1/byte)*

B Examples, Output

Value in iolist Entity	Edit Descriptor	Characters Written to Record
+11	B5	□1011
+100	B2	**
+100	B7	1100100

BN/BZ (blank interpretation) edit descriptor, form

BN or BZ

For numeric values, F77 interprets leading blanks as nulls. It interprets embedded blanks as specified by BLANK= on the OPEN statement. If you omit BLANK= on the OPEN statement, it interprets embedded blanks as nulls. The BZ descriptor tells it to interpret embedded blanks as 0; the BN descriptor restores null blank handling. These nonrepeatable descriptors apply for input only; they are ignored for output.

BN/BZ Example, Input

```
C     UNIT 5 IS PRECONNECTED WITH BLANK='NULL'.
C     ASSUME THE INPUT CHARACTERS ARE (b=blank):
C                   37b4bb37b4bb5bb1
      READ (5,10) IVAR1,IVAR2,IVAR3
   10 FORMAT (I4, 2X, BZ, I4, 2X, BN, I4)
C     IVAR1 CONTAINS 374, IVAR2 CONTAINS 3704, AND
C     IVAR3 CONTAINS 51.
```

D or E (real, explicit exponent) edit descriptors, forms

*[kP][r]*Ew.d*[Ee]* or *[kP][r]*Dw.d

where *k* is an integer scale factor to move decimal point right or left;
r is an integer repeat count;
w is an integer that gives the field width in digits;
d is the integer number of digits to the right of the decimal point;
Ee is an alternate way to express the exponent, ignored on input and optional for output.

The D and E descriptors are functionally identical for F77. For use with either, the iolist entity must be type real*4, double precision/real*8, or complex (standard or complex*16). To format complex data, use two type D, E, F, or G edit descriptors.

On input to D, E, F, or G, F77 reads w characters from the record, including sign (if any), decimal point (if any), and exponent (if any). The exponent, if present, can be either an E or D followed by an unsigned integer constant or simply an unsigned integer constant. Blanks count as characters, but leading blanks have no value significance; embedded blanks have no value significance unless you specified blank-zero interpretation. For any real field, F77 can store accurately only the precision of approximately 6.7 digits (real*4) or 16.4 digits (double precision/real*8). If the value read exceeds F77's storage *range* (approximately 5.4*10**-79 to 7.2*10**75), F77 will signal an error.

On output from D or E, F77 formats w characters, including the sign (if negative or SP descriptor is on), the iolist value with a decimal point (if any), and the exponent field of four characters. If the sum of these is less than w, F77 pads the value with leading blanks to w. If the sum of these exceeds w, F77 outputs w asterisks. F77 uses E in the exponent even if you specify D.

E and D Examples, Input

Edit Descriptor	Characters Read	Value Stored in iolist Entity
E6.4	-.21E5	-21000.
E4.3	.456	.456
E5.1	31-01	.31

E and D Examples, Output

Value in iolist Entity	Edit Descriptor	Characters Written to Record
+12.34	E10.3	□□.123E+02
-19.54	E6.2	****** *(E6.2 too small for exponent)*
-123.4567	E12.6	-.123457E+03

F (real, floating-point) edit descriptor, form
*[kP][r]*Fw.d
where *k* is an integer scale factor to move the decimal point right or left;
r is an integer repeat count;
w is an integer that gives the field width in digits;
d is the integer number of digits to the right of the decimal point.

The F descriptor reads real data with or without an explicit exponent; it writes floating-point, real data without an explicit exponent. The iolist entity must be type real*4, double precision/real*8, or complex (standard or complex*16). To format complex data, use two D, E, F, or G edit descriptors.

On input to F, F77 reads w characters from the record, including the sign (if any), decimal point (if any), and exponent (if any). It handles F input the same way as D and E input. Any decimal point among the w characters overrides a conflicting value of d in the F descriptor.

On output from F, F77 formats w characters, including the sign (if negative or SP descriptor is on) and iolist value, placing the decimal point so that d digits follow it. If the iolist entity has more than d digits following the decimal point, F77 rounds the fraction at the d position. If the number of characters including the sign (if any), decimal point, and digits is less than w, F77 pads the value with leading blanks to w. If this number exceeds w, F77 outputs w asterisks.

F Examples, Input

Edit Descriptor	Characters Read	Value Stored in iolist Entity
F7.2	-123.41	-123.41
F5.2	12345	123.45
F8.0	2.032E3□	+2032.
F8.0	-2.032-3	-.002032

F Examples, Output

Value in iolist Entity	Edit Descriptor	Characters Written to Record
-98.7	F7.2	□-98.70
100.67	F8.1	□□□100.7 *(note rounding)*
123.54	F5.2	***** *(F5.2 too small)*
68.3	F4.1	68.3

G (real, generalized) edit descriptor, form

*[kP][r]*Gw.d*Ee*

where *k* is an integer scale factor to move the decimal point right or left;
r is an integer repeat count;
w is an integer that gives the field width in digits;
d is the integer number of digits to the right of the decimal point;
Ee is an alternate way to express the exponent, ignored on input and optional for output.

For G editing, the iolist entity must be type real*4, double precision/real*8, or complex (standard or complex*16). To format complex data, use two D, E, F, or G edit descriptors.

On input, G is identical to D, E, and F.

G Examples, Input

See F.

On output, G uses either F or E editing. If the absolute value of the iolist entity is equal to or greater than .1 and less than 10**d, F77 uses the F mode but decreases the value of w by 4. It outputs the four-character exponent field as four blanks *after the value*. Generally, F77 outputs a total of d digits. If the scale factor *kP* is positive, F77 outputs d+1 digits.

But if the absolute value of the iolist entity is less than .1 or equal to/greater than 10**d, F77 outputs the number with an explicit exponent, E editing.

In either F or E format, if the total number of characters is less than w, F77 pads the value to w with leading blanks. If the total number of characters is greater than w, F77 outputs w asterisks.

G Examples, Output Compared to E

Value in iolist Entity	E 11.4 Output	G11.4 Output
-8977.	□-.8977E+04	□-8977.□□□□
23.40	□□.2340E+02	□□23.40□□□□
-20320.	□-.2032E+05	□-.2032E+05
0.02032	□□.2032E-01	□□.2032E-01
0.	□□.0000E+00	□□.0000E+00

G Examples, Output

Value in iolist Entity	Edit Descriptor	Characters Written to Record
.0123456	G11.4	□□.1235E-01
.123456	G11.4	□□.1235□□□□
1.2346	G11.4	□□1.235□□□□
12.346	G11.4	□□12.35□□□□

FORMAT (continued)

H (Hollerith constant) edit descriptor, form

nHc*[c][...]*
where n is the count of c characters;
c is an ASCII character.

The H descriptor writes a literal string of n characters to a record; you can use it for output only.

On output, F77 writes n characters without interpreting them; angle brackets have no special meaning. An H example for output is

```
    WRITE (*, 115)   45
115 FORMAT (11H Value is: , I2)
```

F77 will display

(blank line)
Value is: 45

I (integer) edit descriptor, form

*[r]*Iw*[.m]*
where *r* is an integer repeat count;
w is an integer that gives the field width in digits;
m is an integer that gives the minimum width for output; F77 will insert leading 0s to pad to *m*. But if *m* and the entity are 0, F77 will output blanks for the value.

The I descriptor transmits integer data to or from an iolist entity. The entity must be type integer or, for AOS/VS only, logical*1/byte.

On input, F77 reads w characters from the record, including sign (if negative or if SP sign control is on). Blanks count as characters but leading blanks have no value significance; embedded blanks have no value significance unless you specified blank-zero interpretation. If the iolist entity is a 4-byte integer, it can accept a range of values from at least -2,147,483,647 through 2,147,483,647. A 2-byte integer can accept a value only from -32,768 through 32,767. A logical*1/byte entity (AOS/VS only) can accept a value only from -128 through 127. If the value is out of the entity's range, F77 signals an error.

On output from I, F77 formats w characters, including sign (if negative or if SP sign control is on) and iolist value. If the number of characters is less than w, F77 pads the value with leading blanks to w. If the number exceeds w, F77 outputs w asterisks. If you included *m*, F77 inserts leading 0s to pad the value to *m* characters.

I Examples, Input

Edit Descriptor	Characters Read	Value Stored in iolist Entity
I3	□50	50
I4	□□□□	0
I10	3111111111	*Error (too big)*

I Examples, Output

Value in iolist Entity	Edit Descriptor	Characters Written to Record
50	I3	□50
50	I3.3	050
-50	I2	** *(I2 too small)*
-50	I3	-50
-50	I5	□□-50
0	I6.0	□□□□□□
1111111111	I15	□□□□□1111111111

L (logical) edit descriptor, form
*[r]*Lw
where *r* is an integer repeat count;
w is an integer that gives the field width in characters.

The entity used with L must be type logical*4, logical*2, or (AOS/VS F77 only) logical*1/byte.

On input, F77 reads w characters. If the first nonblank characters are .T or T, F77 places the value .TRUE. in the entity. If the first nonblank characters are .F or F, F77 places the value .FALSE. in the entity. If the first nonblank characters are neither of these, F77 reports an error.

On output from L, F77 writes w-1 blanks to the record, followed by T if the entity contains the value .TRUE. or followed by F if the entity contains the value .FALSE..

L Examples, Input

Edit Descriptor	Characters Read	Value Stored in iolist Entity
L6	□□TRUE	.TRUE.
L5	□F120	.FALSE.
L2	.FALSE.	.FALSE.

L Examples, Output

Value in iolist Entity	Edit Descriptor	Characters Written to Record
.TRUE.	L4	□□□T
.FALSE.	L1	F

FORMAT (continued)

O (octal integer) edit descriptor, form
*[r]*Ow*[.m]*
where *r* is an integer repeat count;
w is an integer that gives the field width in digits;
m is an integer that gives the minimum width for output; F77 will insert leading 0s to pad to *m*. But if *m* and the entity are 0, F77 will output blanks for the value.

The O descriptor transmits octal integer data to or from an iolist entity. The entity must be type integer or, for AOS/VS F77 only, logical*1/byte.

On input, F77 reads w digits from the record; a sign or any characters other than digits 0–7 will produce an error. If the value read is greater than 37777777777 (4-byte integer) or 177777 (2-byte integer) or 377 (AOS/VS logical*1/byte entity), F77 signals an error.

On output from O, F77 converts the iolist value to octal and outputs it as w digits. If the number of digits is less than w, F77 pads the value with leading blanks to w. If the number exceeds w, F77 outputs w asterisks. If you included *m*, F77 inserts leading 0s to pad the value to *m* characters.

O Examples, Input

Edit Descriptor	Characters Read	Value Stored in iolist Entity
O2	34	28
O2	61	49
O6	177777	65535 *(integer*4)*
O6	177777	-1 *(integer*2)*

O Examples, Output

Value in iolist Entity	Edit Descriptor	Characters Written to Record
+33	O5	□□□41
+100	O2	**
+100	O3	144

P (scale factor) edit descriptor, form
kP
where k is an optionally signed integer constant, called the scale factor, that moves the decimal position left or right. The range of k is -127 through 127.

The scale factor moves the decimal point in real numbers; it is nonrepeatable. You can use it with a D, E, F, or G descriptor.

On input with D, E, F, or G, a positive scale factor has the effect of dividing the number read by 10**k. F77 ignores any scale factor if the value read has an explicit exponent; e.g., 2.4E10.

On output with F, a positive scale factor has the effect of *multiplying* the number read by 10**k. On output with D or E, F77 multiplies the number by 10**k and subtracts k from the exponent, which does not change the value but does change the output representation. On output with G, F77 decides on E or F editing, then applies the scale factor according to E or F rules. See "P Editing" in Chapter 6 for more on the scale factor.

P Examples, Input

Edit Descriptor	Characters Read	Value Stored in iolist Entity
2P, F10.2	-25.44□□□□	-.2544
2P, F9.2	345.71□□□	3.4571
3P, F6.3	12.345	.012345
-3P, F6.3	12.345	12345.
-3P, E8.3	12.345E0	12.345 (*P ignored*)

P Examples, Output

Value in iolist Entity	Edit Descriptor	Characters Written to Record
501.33	E10.3	□□.501E+03
501.33	1P, E10.3	□5.013E+02
501.33	2P, E10.3	□50.13E+01
+12.217	F7.3	□12.217
+12.217	-1P, F7.3	□□1.222

S, SP, SS (sign control) edit descriptors, form

S or SP or SS

By default, F77 outputs positive numeric values without a leading sign. S and SS (sign suppress) restore default output; SP (sign produce) forces output of a plus sign before positive numeric values (except octal, O and hexadecimal, Z and binary, B). You should provide space for this sign in the output field, if needed. These descriptors work on output only; they are ignored on input.

For example, after encountering SP,I2 in a format, F77 would output the value 3 as +3.

T, TL, TR (tab position) edit descriptors, forms

Tn or TLn or TRn
where n is an integer that specifies:
an absolute column position for Tn (tab),
column positions to the left for TLn (tab left),
or column positions to the right for TRn (tab right).

The T-series descriptors set column position within the current record. They are nonrepeatable.

On input with Tn, F77 positions at column n in the current record. For TLn, it positions n columns left but will not back up into the previous record. For TRn, it skips n positions, tabbing right.

On output from Tn, F77 positions at column n. With TLn, F77 positions n columns left, enabling you to overwrite characters already placed in the record. Attempting to set position before the first character only sets position at the first character. For TR, F77 positions n columns right, skipping columns.

At record output time, if there are any gaps (unfilled positions) between column 1 and the last character supplied by the format, F77 inserts blanks in the gap positions.

T, TL Examples, Input and Output

```
C   For T descriptor:
    ...
    C = 4.11
    WRITE (*, 95)   C
95  FORMAT ( T2, '12345', T4, F6.2, ' abcdef')
```

F77 will print this record as:

```
12 4.11 abcdef
```

With the default preconnection, the first blank is used for carriage control.

```
C   For TL descriptor.
    READ (*, 96)   J, K, L
96  FORMAT (I8, TL4, I2, TL30, I3)
    WRITE (*, 99)   J, K, L
99  FORMAT (1X, 'J= ', I8, ',K= ', I4, ',L=', I3)
    ...
```

If, when the code above executes, the user types

123456789 ↓

the output will be

J=□12345678,K=□□□56,L=123

For a TRn example, see the X example.

X (skip position) edit descriptor, form

nX
where n is the integer that gives the number of character positions to the right of the current position.

On input, F77 skips n column positions, tabbing right (as for TRn).

On output, F77 moves right n column positions. If, as for TR and Tn, there are any unfilled positions, F77 inserts blanks in these columns before outputting the record. Thus 1X at the beginning of a format specification will have the effect of inserting a blank in column 1 of the record. This blank can serve for FORTRAN carriage control.

X Example, Output

```
    WRITE (*, 100)   B1, B2, B3
100 FORMAT (1X, F11.2, 10X, F11.2, 10X, F11.2)
```

Z (hex integer) edit descriptor, form

*[r]*Zw*[.m]*
where *r* is an integer repeat count;
w is an integer that gives the field width in digits;
m is an integer that gives the minimum width for output; F77 will insert leading 0s to pad to *m*. But if *m* and the entity are 0, F77 will output blanks for the value.

The Z descriptor transmits hexadecimal integer data to or from an iolist entity. The entity must be type integer or, for AOS/VS only, logical*1/byte.

On input, F77 reads w digits from the record; a sign or any characters other than 0–9, A–F, and a–f will produce an error. If the value read is greater than FFFFFFFF (4-byte integer) or FFFF (2-byte integer) or FF (AOS/VS logical*1/byte entity), F77 signals an error.

On output from Z, F77 converts the iolist value to hex and outputs it as w digits. If the number of digits is less than w, F77 pads the value with leading blanks to w. If the number exceeds w, F77 outputs w asterisks. If you included *m*, F77 inserts leading 0s to pad the value to *m* characters.

Z Examples, Input

Edit Descriptor	Characters Read	Value Stored in iolist Entity
Z2	42	66
Z3	1D2	466

Z Examples, Output

Value in iolist Entity	Edit Descriptor	Characters Written to Record
433	Z3	1B1
433	Z4.4	01B1

FORMAT (continued)

' and '' (apostrophe and quotation mark) edit descriptors, forms

'string' or "string"

where string is any string of characters.

As edit descriptors, apostrophe or quotation mark work for output only; they will cause an error if used for input. F77 outputs the enclosed string to the record. The string can include the special angle-bracketed codes, like <BEL>, described in Table 2-2.

' Example, Output

```
    PRINT 111,  N**2
111 FORMAT ( ' N squared is: ', I10)
```

$ (suppress NEW LINE) edit descriptor, form

$

The default carriage control for user opens is LIST. With LIST carriage control in a data-sensitive file, F77 outputs a NEW LINE after each record. The $ descriptor supresses this NEW LINE, allowing you to write more than one record on each line. $ is a nonrepeatable edit descriptor, effective on output only and ignored on input. It has no effect on files that have other than LIST carriage control. (You can get the same effect with FORTRAN carriage control in a data-sensitive file by using # in column 1 of a record, as shown under the OPEN statement, CARRIAGECONTROL= property.)

$ Example, Output

```
    WRITE   (10, 100)
100 FORMAT  ( ' Type a value for A. ', $)
    READ *,  A
```

A sample dialog is

Type a value for A.□999.0↓

The $ in the FORMAT statement suppressed output of NEW LINE after the prompt.

: (colon, terminate format control conditionally) edit descriptor, form

:

Normally, F77 must process all nonrepeatable edit descriptors in a format specification, even if all iolist entities have been processed. The colon (:) descriptor tells F77 to terminate format control if all iolist entities have been processed. It allows you to avoid printing superfluous text messages.

On input and output, the colon terminates format control if, and only if, no more iolist entities remain to be formatted.

Colon Example, Output

```
    WRITE (*, 75)   I
    WRITE (*, 75)   I, J, K
75  FORMAT (1X, : ,'1st' , I8, : , ';2nd', I8, : ,
   +              ';3rd', I8)
```

/ (slash, start a new record) edit descriptor, form

/

Normally, F77 processes one record for each I/O statement. It processes more than one record if: 1) all repeatable descriptors have been used and more entities remain in the iolist (this is called format reversion and is detailed in Chapter 6 under "Multiple Record Formatting"); or 2) if it encounters a slash in a format specification.

A slash directs F77 to stop processing the current record and start the next, using the edit descriptors to the right of the slash. Slash is a nonrepeatable edit descriptor.

On input, the slash causes F77 to skip the rest of the current record and read in a new record. Sequential slashes allow you to bypass input records.

On output, each slash terminates a record. Two sequential slashes output a record of a blank line.

/ Example, Input

```
    READ (3, 90)   I, J, K
90  FORMAT (I4, I4, I4)
    READ (3, 100)   L, M, N
100 FORMAT (I4, /, I4, /, I4 )
```

Using format 90, F77 would attempt to read three values from the same record; using format 100, it would attempt to read three values from three successive records.

/ Example, Output

```
    WRITE (4, 190)   I, J, K
190 FORMAT  (1X, I8, I8, I8)
    WRITE (4, 200)   L, M, N
200 FORMAT  (1X, I8, /, I8, /, I8)
```

Using format 190, F77 would output the three values as one record; using format 200, it would output the three values as three records.

Other Format Considerations

To separate edit descriptors, you can always use a comma. Slash and colon also act as separators but have other effects.

Each repeatable edit descriptor accepts a repeat count directly. You can repeat groups of descriptors by placing them in parentheses. Examples of both are

```
      WRITE (*, 100)  CHA, CHB
100   FORMAT  (1X, 2A)

      WRITE (10, 200)  J1, CHC, J2, CHD
200   FORMAT  ( 2 ('Values: ', I8, A) )
```

If you specified FORTRAN carriage control on the OPEN statement, F77 will use the first character of each output record for carriage control; it will not output the first character as data. If you print the file, the first character has printer control as shown under OPEN, CARRIAGECONTROL= property.

The default carriage control is LIST. In a data-sensitive file with LIST carriage control, F77 outputs a NEW LINE after each record by default; it outputs the first character as data.

For either FORTRAN or LIST carriage control, a file that you want to print directly must have been opened with data-sensitive record organization (default); if not, you must run the F77PRINT postprocessor on it.

In AOS/VS F77 you may use *expressions* along with constants for field widths, repeat specifiers, and scale factors. For example, the statements

```
      K2 = 2
      K3 = 3
      K7 = 7
      WRITE (6, 10) VAR1, VAR2
10    FORMAT (1X, <K2>F<K7>.<K3+1>)
```

and

```
      WRITE (6, 10) VAR1, VAR2
10    FORMAT (1X, 2F7.4)
```

produce identical results. However, don't use expressions in format statements associated with implied DO loops.

If you specify the /FMTMM (ForMaT MisMatch) F77LINK switch, you may have certain mismatches between iolist entities and their edit descriptors. An example is

```
      IVAR = 16
      WRITE (10, 20) IVAR
20    FORMAT ('*', F6.2, '*')
```

The output (with the /FMTMM F77LINK switch) is

*□*16.00**

instead of a runtime error message and task termination (without the /FMTMM F77LINK switch).

Each of the descriptors and topics described in this section receives more detail in Chapter 6.

FUNCTION

Begins a function subprogram.

Form

[typ] FUNCTION func (*[d [,d] ...]*)

where:

typ is the data type of the function: INTEGER, INTEGER*2, INTEGER*4, REAL, REAL*4, REAL*8, DOUBLE PRECISION, COMPLEX, COMPLEX*16, LOGICAL, LOGICAL*1/BYTE (AOS/VS), LOGICAL*2, LOGICAL*4, or CHARACTER*/*len/*, where *len* is the length of the character string that the function will produce. *len* can have any of the forms allowed for the CHARACTER type statement except that it cannot use the symbolic name of a constant in an integer expression. If you omit *len*, the length is 1. If you use a *typ* of CHARACTER*(*), the function's character string will assume the size declared in the calling program unit.

func is the symbolic name of this function subprogram. If you omitted *typ* and do not declare the type in a later type statement, the name implies the data type of the function. A valid symbolic name is 1 to 32 characters and includes letters, numbers, question marks, and underscores; the first character must be a letter or the question mark. MP/AOS and MP/AOS-SU names follow the same rules except that they are unique only up to the first ten characters.

d is a dummy variable name, dummy array name, or dummy procedure name. Each *d* can hold the value of an actual argument. The parentheses are mandatory, even if you omit dummy arguments.

What It Does

The FUNCTION statement starts a function subprogram and must be the first statement in a function subprogram. A function subprogram can contain only one FUNCTION statement and cannot contain a PROGRAM, SUBROUTINE, or BLOCK DATA statement. FUNCTION is a nonexecutable statement and the compiler ignores any label on it.

On the function reference in the calling program unit, F77 transfers control to the function, using any actual arguments for the function's dummy arguments. The function executes its statements. Upon a RETURN or END statement, the function returns its value to the function reference in the calling unit.

If a function contains a READ, WRITE, or PRINT statement, the function cannot be referred to by a READ, WRITE, or PRINT that specifies the same unit. For example, if MYFUNC contains a PRINT * statement, then your program cannot invoke MYFUNC via a PRINT *, MYFUNC() statement.

A dummy argument can hold the value of an expression, an array name, an external subprogram name, or dummy procedure name. The dummy argument names (if any) in a FUNCTION statement must correspond in order, number, and type with the actual arguments of the function reference in the calling unit. But you can use any subroutine name as a dummy argument because subroutines do not have a data type.

Before the function returns control to the calling unit, it must assign a value to its symbolic name at least once. This value must match the function's data type.

Between a function's FUNCTION and END statements, the function's name or name of an ENTRY in the function may appear only as a variable — if you omitted *typ*, in *one* type statement.

If a function subprogram is type character, either via *typ* or type statement, each ENTRY name must be type character. All character ENTRY names and the character function name must have the same length, which can be an integer (CHARACTER*60 FUNCTION CH) or asterisk (CHARACTER*(*) FUNCTION CH1).

The dummy argument names used must not appear in a DATA, EQUIVALENCE, PARAMETER, INTRINSIC, SAVE, or COMMON statement within the function, except as a common block name.

A subprogram cannot use an assumed-size array (described next) in an iolist.

Actual-Dummy Argument Summary

Generally, you can use any expression as an actual argument if the corresponding dummy argument is the appropriate data type. Other rules sketched in this section are detailed in Chapter 7, "Arguments to Function and Subroutine Subprograms."

Variables

To pass a variable, use its name as an actual argument in the calling unit and include a dummy argument of the same data type in the subprogram's dummy argument list. To pass an array element, use the *array name and element subscript* as an actual argument.

Arrays

To pass a whole array, use its name as an actual argument and a dummy argument of the same data type in the subprogram. The subprogram must dimension the array. It can do so in one of three ways:

1. Via one or more integer constant expressions; if so, the total number of elements must not exceed the total number declared originally in the calling unit.
2. Via one or more integer expressions that include an integer dummy argument and can include an integer constant.
3. Via an asterisk assumed-size array declarator. For this, the caller specifies the array name as an actual argument, and the subprogram specifies a dummy argument of the proper data type to receive the name. Then the subprogram uses an asterisk as the high bound of the last dimension expression. For example:

```
C   Calling unit.
    DIMENSION  ARY(200)
    PRINT *,  MYFUNC(ARY)   ! Reference.
    ...
    END


C   Subprogram.
    INTEGER FUNCTION  MYFUNC(ARY2)
    DIMENSION ARY2(*)
    ...
    END
```

An assumed-size declarator can specify only the *last* dimension of a multiply dimensioned array. The subprogram cannot use an assumed-size array in an iolist.

Character Values

The data types of both actual and dummy arguments must be character. The subprogram cannot have access to more characters than were declared for the entity in the calling unit. If the dummy argument is an array, the subprogram can declare an element length that differs from that of the caller; but the subprogram cannot have access to a character beyond the last character reserved by the caller for the array.

For either a character variable or array, you can use an asterisk length specifier when you have the subprogram declare the character entity. An asterisk specifier directs F77 to use the length declared by the calling unit. For example:

```
C   Calling unit.
    PROGRAM  MAIN
    CHARACTER*10  CF
    ...
    X = FUNC(CF)   ! Function reference.
    ...


C   Function.
    FUNCTION  FUNC(CF2)
    CHARACTER*(*)  CF2
C   When referenced by program unit MAIN,
C      CF2 is 10 characters long.
    ...
```

If the function is type character, a CHARACTER*(*) *typ* specifies the length given in the calling unit. For example:

```
C   Calling unit.
    CHARACTER*20  CFUNC
    ...
    PRINT *,  CFUNC( )    ! Reference.
    ...


C   Function subprogram.
    CHARACTER*(*)  FUNCTION  CFUNC( )
    ...
C   CFUNC contains the 20 characters
C      declared in the calling unit.
    ...
    CFUNC = 'xxxx' ! Mandatory assignment.
    ...
```

As an extension, DG's F77 allows you to concatenate strings passed via asterisk specifiers.

Subprogram Arguments

You can use certain intrinsic functions as actual arguments if you declare them in an INTRINSIC statement; these are described under INTRINSIC.

You can use subprogram names as actual arguments if you declare each subprogram name to be passed as EXTERNAL in the calling program. This lets F77 know that the name is a subprogram name and not a variable name.

FUNCTION Examples

```
FUNCTION  IFIND ( )

FUNCTION  MARGO(L)

INTEGER FUNCTION  ROYCE(K,L,M, CSTRING)

DOUBLE PRECISION FUNCTION  FLAG(X,Y)

CHARACTER*30 FUNCTION  TX(REALVAR)
```

The following example shows a function that computes the scalar product of two vectors up to a given element.

```
C   Main program.
    PROGRAM  VECTORS
    REAL  B(40), C(40)
10  PRINT *, 'Type number of elements, 1-40.'
    READ (*, *)  LEN
    IF ((LEN.GT.40) .OR. (LEN.LT.1))  GO TO 10
    DO 30  K = 1, LEN
       PRINT *,  'Type B and C values, element ',K
       READ (*,*)  B(K), C(K)
30  CONTINUE
    PRINT *,  'Values in array B and C are: '
    PRINT *
    PRINT *,  (B(I), I=1, LEN)
    PRINT *
    PRINT *,  (C(I), I=1, LEN)
    PRINT *
    SCALE_PROD = SCALE(B,C,LEN)  ! Call subprogram
    PRINT *, 'Scalar product is ', SCALE_PROD
    END


C   Function.
    FUNCTION  SCALE(B2, C2, LEN2)
    REAL  B2(LEN2), C2(LEN2)
    SCALE = 0.0
    DO 20  I = 1, LEN2
       SCALE = SCALE + B2(I) * C2(I)
20  CONTINUE
    RETURN
    END
```

Here, function SCALE takes the products of the values in an accumulating total, then returns its value to the main program.

GO TO (assigned)

Transfers control to a previously assigned statement label.

Form

GO TO v *[[,] (s_1, s_2 ,..., s_n)]*

where:

v is an integer variable name: integer*4 for AOS/VS, integer*4 or integer*2 for AOS, F7716, MP/AOS, and MP/AOS-SU.

s is the label of an executable statement within the current program unit.

What It Does

An assigned GO TO transfers control to a previously assigned statement label.

Before using this form of GO TO, you must have assigned a statement label to the integer variable v with the ASSIGN statement. The label assigned to v must belong to an executable statement within this program unit.

F77 will always pass control to the statement label assigned to v, regardless of the other labels in the *s* list. The compiler does not check the label for validity. Be very careful with assigned GO TOs.

You should label the first executable statement that follows an assigned GO TO statement. If you don't, the compiler will signal a warning.

GO TO (assigned) Example

```
    ...
    ASSIGN 5 TO IPROC          ! Assign.
    ...
    GO TO IPROC
15  CALL  SUBR1                ! The label is
                               !   recommended.
    ...
5   FLAG = 3.0                 ! IPROC routine.
    ...
```

In the example, control goes to statement 5.

GO TO (computed)

Transfers control to one of several statements depending on the value of a variable.

Form

GO TO s_1 *[, s_2 ,..., s_n])* *[,]* iexpr

where:

s — is the label of an executable statement within this program unit.

iexpr — is an integer (including logical*1/byte) expression.

What It Does

A computed GO TO transfers control to one of several statements depending on the value of an integer entity.

F77 transfers control to the statement *whose position* in the s list corresponds to iexpr. If iexpr is 1, control goes to statement label s_1; if iexpr is 2, control goes to statement label s_2; if iexpr is *n*, control goes to statement label s_n. If iexpr is out of range (less than 1 or more than *n*), control passes to the next executable statement. You can use the same statement label in more than one s item. However, a computed GO TO statement cannot have more than 63 labels. Exception: AOS/VS F77 accepts up to 254 labels in a computed GO TO statement.

A statement label must refer to an executable statement within the same program unit, or the compiler will signal an error.

GO TO (computed) Example

```
GO TO (10, 100, 50, 10, 9)   K
PRINT *, 'NO BRANCH OCCURS'
```

The variable K must be 1, 2, 3, 4, or 5 for a branch to occur. If K is 1, control passes to the statement labelled 10; if K is 2, control passes to the statement labelled 100, and so on. If K is less than 1 or greater than 5, control passes to the next statement (PRINT).

GO TO (unconditional)

Transfers control to a statement.

Form

GO TO s

where:

s — is the label of an executable statement within the current program unit.

The unconditional GO TO transfers control to the statement s. Statement s must be the label of an executable statement in this program unit. If the label refers to a nonexistent label or nonexecutable statement, the compiler signals an error.

You should label the first executable statement that follows an unconditional GO TO statement, or the compiler will signal a warning.

GO TO (unconditional) Example

```
    GO TO 25
10  R = 2.0*A
    ...
    ...
25  I = I + 1
    ...
```

The GO TO transfers control unconditionally to the statement labelled 25. Unless this program unit directs control to the statement labelled 10, it and the statements that immediately follow it will never be executed.

IF (arithmetic)

Transfers control to one of three statements depending on the value of an arithmetic expression.

Form

IF (expr) s_1, s_2, s_3

where:

expr is an arithmetic expression of any type except complex.

s is the label of an executable statement within the current program unit.

What It Does

An arithmetic IF transfers control conditionally to one of three statements based on the value of expr.

F77 evaluates the arithmetic expression. If the value is less than 0, F77 passes control to statement s_1; if the result is 0, it passes control to statement s_2; or if it is greater than 0, F77 passes control to s_3.

You must specify all three statement labels. They need not refer to different statements, but they must refer to executable statements within the same program unit. If not, the compiler signals an error.

An arithmetic IF statement tests for exact 0. If the expr is type real and involves arithmetic, it may not produce true 0 because the results of real computations are approximate. Thus a real arithmetic expression that might appear to produce exact 0 will probably not produce an exact 0. For example, the next statement to execute after

```
IF ( 3.0*(1.0/3.0) - 1.0 ) 10, 20, 10
```

is the statement labelled 10, *not* 20, because (1.0/3.0) is not exactly .3333... .

IF (arithmetic) Examples

```
IF (INDEX)   100, 200, 300

IF ( K(I,J) - L )   10, 4,30

if (IQ * IR)   5, 5, 2

IF (JTEST**4 - JTEST/2)   10,15,15


C  LUNCH is INTEGER*4 and read under A1 format
   IF ( LUNCH(1) - 'P' ) 60, 70, 60
C  IF ( LUNCH(1) - 1HP ) 60, 70, 60  is equivalent
60 CONTINUE !no chance of a peanut butter sandwich
   ...
70 CONTINUE ! investigate further: LUNCH(2)='E' ?
   ...
```

IF...THEN (block IF)

Conditionally executes a block of statements. The block can include ELSE blocks, based on the value of a logical expression.

Form

```
IF (expr) THEN
    [if block of statements]
 [ELSE IF (expr) THEN
    else if block of statements]
 [ELSE IF (expr) THEN
    else if block of statements]
 [ELSE
    else block of statements]
END IF
```

where:

expr	is a logical expression.
if block of statements	is a block of statements that will execute if the IF(expr)THEN is true. Control will then pass through END IF out of the block IF.
else if block of statements	is a block of statements that will execute if the IF(expr)THEN is false and the preceding ELSE IF(expr) is true. After F77 executes the first ELSE IF block, it transfers control to END IF and then out of the block IF.
else block of statements	is a block of statements that will execute if all exprs preceding the ELSE statement are false. After an ELSE block executes, control passes through END IF and out of the block IF.

What It Does

IF(expr)THEN introduces a block IF: a group of statements that will execute conditionally. It can include zero or more ELSE IF statements and one ELSE statement; these in turn execute conditionally. As shown in the form, only IF(expr)THEN and END IF are mandatory; the IF BLOCK, ELSE IFs, ELSE IF BLOCKS, ELSE, and ELSE BLOCKS are optional.

You must close every block IF with an END IF; if the number of IF(expr)THEN statements and END IF statements doesn't match, the compiler signals an error.

You can transfer control via GO TO *to* an IF(expr)THEN statement, but you cannot transfer control into the block IF. You *can* transfer out of a block IF with GO TO.

A block IF can contain multiple ELSE IF(expr)THEN statements and one ELSE statement; it need not include ELSE. The ELSE IFs, if you include them, must precede the ELSE.

Block IF Execution

F77 evaluates the logical expression in the IF(expr)THEN. If the expression is true (.TRUE.), F77 executes the IF block statements sequentially. Control then passes to the END IF, which passes control to the next executable statement (unless an IF block statement directs control elsewhere). (If you put any real components in expr, remember that real values are usually close approximations, as detailed in logical IF, next.)

If the expression in the IF(expr)THEN is false (.FALSE.), control passes to the next ELSE IF(expr)THEN or ELSE statement. If there are no ELSE IF(expr)THEN or ELSE statements, or their blocks are empty, control passes to END IF and the next executable statement.

If an ELSE IF(expr)THEN statement gets control, F77 evaluates the expr as for the IF(expr)THEN; if true, the ELSE IF block is executed and control goes to the next executable statement after the END IF (unless directed elsewhere in the ELSE IF BLOCK). If the ELSE IF expression is false, control passes to the next ELSE IF, ELSE, or END IF statement. For example:

```
IF (2 .EQ. 4) THEN      ! The expr is false.
     I = J        !  Skip.
     X = Y        !  Skip.
ELSE IF (2 .EQ. 1) THEN     ! Check; false.
     I = K        !  Skip this...
     X = Z        !  and this.
ELSE IF (2 .EQ. 2) THEN     ! Check; true.
     I = L        !  Do this...
     X = V        !  and this.
END IF       ! Proceed.
```

If all preceding exprs are false, the ELSE block of statements executes. Control then goes to END IF and the next executable statement (unless an ELSE block statement directs it elsewhere).

In summary, you can use virtually any number of ELSE IFs in a block IF, before an ELSE. F77 will process them sequentially and will execute the block of the first IF or ELSE IF whose expression evaluates to true, then proceed to END IF. If no ELSE IF evaluates to true, F77 will execute the ELSE block statements (if present) and proceed to END IF.

IF...THEN (continued)

Nested Block IFs

You can nest block IFs (enter a block IF while in a block IF); but if you do, make sure that you close each block IF with an END IF. If the number of END IFs doesn't equal the number of IF...THENs, the compiler signals an error.

Each IF block has its own level, and F77 executes statements within that level before proceeding to another block.

Block IF Examples

```
C   Example computes regular pay and overtime
C      for an hourly employee.

    IF (HOURS .GT. 40.0) THEN
       REGPAY = RATE * 40.0
       OVTPAY = 1.5 * RATE * (HOURS - 40.0)
    ELSE
       REGPAY = RATE * HOURS
       OVTPAY = 0.0
    END IF
    PAY = REGPAY + OVTPAY
    ...
```

If HOURS is greater than 40.0, the IF block statements are executed and control passes through the END IF to the PAY assignment statement. If HOURS is not greater than 40.0, the IF block statements are skipped, the ELSE statements executed, and control passes on through the END IF.

In the next example, a nested block IF is used to test and solve quadratic equations.

```
C   Attempt to obtain discriminant.

    READ *, A, B, C
    IF (ABS(A) .GT. 0.) THEN
        DISCRIM = B**2 - 4.0*A*C
        IF (DISCRIM .GE. 0.0) THEN
            ROOT1 = (-B + SQRT(DISCRIM)) / (2.0*A)
            ROOT2 = (-B - SQRT(DISCRIM)) / (2.0*A)
            PRINT *, 'Root1 is ',ROOT1
            PRINT *, 'Root2 is ',ROOT2
        ELSE
            PRINT *, 'Roots are complex.'
        END IF
    ELSE
        PRINT *, 'Equation is not quadratic.'
    END IF
```

IF (logical)

Conditionally executes a statement based on the value of a logical expression.

Form

IF (expr) st

where:

expr — is a logical expression.

st — is any executable statement except a block IF, DO, DO WHILE, ELSE, ELSE IF, END, END DO, END IF, or another logical IF statement.

What It Does

F77 evaluates the logical expression expr. If expr evaluates to true, F77 executes statement st. Control then passes to the first executable statement following statement st unless st directs control elsewhere. If expr does not evaluate to true, then F77 passes control to the next executable statement.

When you test real values in a logical IF expression, remember that real values are often approximate. For example IF(A.EQ.0.0) may not be true when you expect it to be true, because real values are often approximate. In such situations, you can test values for equality within a reasonable tolerance, using the ABS function. For example, instead of IF(A.EQ.B), you could use IF (ABS(A-B)) .LT. 1.0E-6), which will be true whenever A and B are equal within a reasonable tolerance: 1.0E-6 = .000001.

IF (logical) Examples

```
IF (X .GT. 0) GO TO 25

IF (INDEX .GT. 1)  INDEX = 1

IF (.NOT. I)   PRINT *, 'I is false.'

IF ( ARY(J,I) .GE. AARY(J,I) )  CALL FOO

IF ( CH(1:1) .GT. ZH(1:1) )  CX = CH


C  LUNCH is INTEGER*4 and read under A1 format
   IF ( LUNCH(1) .EQ. 'P' ) GO TO 70
C  IF ( LUNCH(1) .EQ. 1HP ) GO TO 70 is equivalent
   CONTINUE !no chance of a peanut butter sandwich
   ...
70 CONTINUE ! investigate further: LUNCH(2)='E' ?
   ...
```

IMPLICIT

Establishes default data types based on symbolic names.

Form

IMPLICIT typ *[*len]* (a *[,a]* ...)
*[,typ [*len] (a [,a] ...)]* ...

where:

typ gives the data type and is INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, BYTE or CHARACTER.

len is an optional byte length specifier for the data type: for INTEGER, *len* can be 2 or 4; for REAL it can be 4 (default) or 8, for COMPLEX, it can be 8 (default) or 16; for LOGICAL, it can be 1 (AOS/VS), 2, or 4 (default); or for CHARACTER it can be any unsigned integer from 1 through 32767. If you omit *len*, all data types will be implied standard length; you can override this in a type statement.

a is a single letter or forward range of letters separated by a minus (-) sign. Symbolic names that begin with the specified letter(s) will have the type (and len, if given) that precedes this a, unless you override this in a type statement.

What It Does

IMPLICIT sets up type and length rules for entities within a program unit. After you specify a letter and data type with IMPLICIT, each entity whose name begins with that letter will be given that data type — unless you specify otherwise in a type statement.

You cannot re-assign an implicit type/character association after you have initially set it. For example,

```
IMPLICIT INTEGER*2 (X-Z)
IMPLICIT INTEGER*4 (Y)    ! Error -
! Y- variables already assigned as INTEGER*2.
```

IMPLICIT can establish the data type of any or all variables, arrays, or functions, except F77 intrinsic functions.

IMPLICIT (continued)

IMPLICIT is a specification statement and must *precede* other specification statements within a program unit with the single exception of the PARAMETER specification statement. The IMPLICIT statement does not apply to the symbolic names appearing in any preceding PARAMETER statements; it applies only to the succeeding symbolic names within its program unit.

The default name rule is

```
IMPLICIT REAL(A-H, O-Z), INTEGER(I-N)   ! all F77s
IMPLICIT INTEGER*4(?)                   ! AOS/VS
                                        ! F77
IMPLICIT INTEGER*2(?)                   ! other F77s
```

IMPLICIT Examples

```
IMPLICIT CHARACTER*16 (C), LOGICAL(L)
...
CLIST = 'ascii chars'  ! C- is char*16.
LSTAT = .TRUE.         ! L- is logical.
RVAR = 10.3            ! R- remains real.
IVAR = 4321            ! I- remains integer.
```

IMPLICIT NONE

Allows no default data types based on symbolic names.

What It Does

IMPLICIT NONE requires you to declare the types of all variables and arrays. It effectively means "explicit all". That is, IMPLICIT NONE has the compiler flag any variable or array whose type you have not explicitly declared. For example:

```
IMPLICIT NONE     ! All variables
                  !   must be declared.

REAL VAR1

VAR1 = 2.0 + 3.0  ! No compiler error

ISUM = 2 + 3      ! Compiler displays an
                  !    error for this
                  !    statement because
                  !    ISUM wasn't
                  !    declared.

PRINT *, VAR1
PRINT *, ISUM

END
```

A program unit may contain at most one IMPLICIT NONE statement.

%INCLUDE (compiler directive)

Includes a separate source file in this program.

Forms

column1

%INCLUDE 'pathname'
%INCLUDE "pathname"

where:

column1 indicates column 1; no tabs or spaces are allowed before the directive.

pathname indicates a path to an F77 source file that you want included. If this source file is in the working (current) directory, you can use its filename.

What It Does

%INCLUDE directs the compiler to include an F77 source file in the current program. %INCLUDE can occur anywhere a statement can appear, but cannot be continued to a second line. Note that %INCLUDE *must* start at column 1; inserting one or more tabs or spaces before it will cause an error.

When the compiler encounters an %INCLUDE directive, it compiles the statements of the included file, then resumes compilation with the statement following %INCLUDE. The source file to be included can in turn contain an %INCLUDE directive, to a file that can in *its* turn contain %INCLUDE, and so on, up to a nesting limit of seven %INCLUDEs.

%INCLUDE Examples

```
C23456789...
%INCLUDE 'TABLEDEFS.F77'

%INCLUDE 'F77DIR:ERR.F77.IN'

":UTIL:DATA_STANDARDS.F77"
```

INCLUDE

Includes a separate source file in this program.

Forms

column7

INCLUDE 'pathname'
INCLUDE "pathname"

where:

column7 indicates column 7 or beyond; columns 1–6 should be blank. A label is possible (but worthless) in columns 1–5.

pathname indicates a path to an F77 source file that you want included. If this source file is in the working (current) directory, you can use its filename.

What It Does

INCLUDE directs the compiler to include an F77 source file in the current program. INCLUDE can occur anywhere a statement can appear, but cannot be continued to a second line. Note that INCLUDE *must* start at column 7 or beyond.

When the compiler encounters an INCLUDE directive, it compiles the statements of the included file, then resumes compilation with the statement following INCLUDE. The source file to be included can in turn contain an INCLUDE directive, to a file that can in *its* turn contain INCLUDE, and so on, up to a nesting limit of seven INCLUDEs.

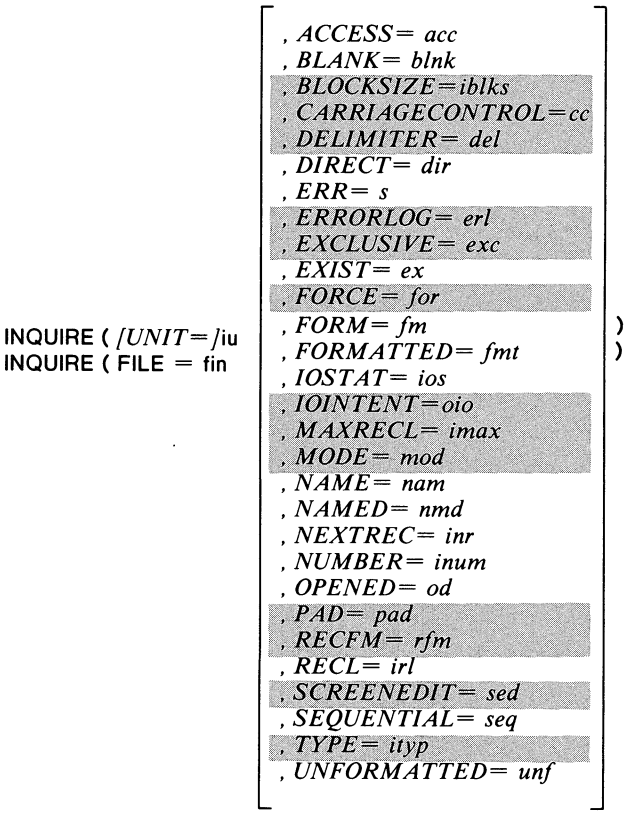
INCLUDE Examples

```
C23456789...
      INCLUDE 'TABLEDEFS.F77'

      INCLUDE 'F77DIR:ERR.F77.IN'

      INCLUDE ":UTIL:DATA_STANDARDS.F77"
```

INQUIRE

Checks the properties or existence of a unit or file.

Forms (mutually exclusive)



where:

UNIT= iu — is a unit specifier, mandatory for an INQUIRE by file. iu must be an integer expression that evaluates to an integer from 0 to 255 for AOS/VS or from 0 through 63 for AOS, F7716, MP/AOS, and MP/AOS-SU. You can omit *UNIT*= if the unit specifier is the first argument in the INQUIRE argument list.

FILE= fin — is a character expression that gives the filename, mandatory for an INQUIRE by file. If the file will not be in the directory where you will execute the program, provide a pathname to it; e.g., FILE=':UDD:JACK:MYDIR:MYFILE'.

ACCESS= — and other property items are detailed under "INQUIRE Property Items" below.

What It Does

INQUIRE allows you to check the properties of files and units; you can then take appropriate action. The unit or file need not be connected or even exist.

If a property item doesn't pertain (perhaps because there is no unit/file connection), the item's return variable becomes undefined.

When you inquire by unit, you may want to include OPENED= because other items may return undefined if the unit isn't open.

When you inquire by file, you may want to include EXIST= and/or OPENED= because some items may return undefined if the file doesn't exist or isn't open.

INQUIRE Property Items

For most INQUIRE items, you must reserve a character entity (variable, array, or substring) to receive information. If you want to pass the information to another statement, the entity must be long enough to hold the returned string (it returns left justified).

For those items whose argument names begin with i, you must use an integer entity to return information. Other items require a logical entity.

AOS/VS programmers may use a logical*1/byte entity as an INQUIRE property whenever character or integer entities are allowed. An exception is IOSTAT, since all I/O error codes exceed 127.

ACCESS= *acc*
acc is a character entity; it will receive the value SEQUENTIAL if the unit or file is connected for sequential access, or DIRECT if it is connected for direct access. If the unit/file is not connected, *acc* will be undefined.

BLANK=*blnk*
blnk is a character entity; it will receive the value NULL if the BLANK property is NULL for this connection, ZERO if the property is ZERO.

BLOCKSIZE= *iblks*
iblks is an integer entity; it will receive the integer value of the block size if it was set by the BLOCKSIZE= specifier during the OPEN statement. If you didn't specify the block size, *iblks* will return 0; if there is no connection, *iblks* will be undefined. MP/AOS and MP/AOS-SU F77 do not support BLOCKSIZE.

CARRIAGECONTROL= *cc*
cc is a character entity. It will receive the carriage control connection property: LIST if the unit/file is connected for LIST carriage control (default), NONE if connected for NONE, and FORTRAN if connected for FORTRAN carriage control.

DELIMITER= del
del is a character entity; it will receive the value EXCLUDE if the connection property is EXCLUDE, or INCLUDE if it is INCLUDE. DELIMITER is relevant only for data-sensitive files that are connected; otherwise *del* will be undefined.

DIRECT= dir
dir is a character entity; it will receive the value YES if direct access to the unit is allowed, NO if direct access isn't allowed, or UNKNOWN if F77 can't tell whether it is allowed. If there is no connection, *dir* will be undefined.

MP/AOS and MP/AOS-SU do not keep track of as much information about a file as the other operating systems do. If an MP/AOS or MP/AOS-SU F77 program inquires into an existing but unconnected file, then the returned value of *dir* is UNKNOWN.

EXIST= ex
ex is a logical entity; it will receive the value .TRUE. if the INQUIRE is by unit and the unit number exists or if the INQUIRE is by file and the file exists. Otherwise *ex* will return .FALSE. .

ERR= s
ERR= directs control to the statement labelled *s* if an error occurs; *s* must be an executable statement in the current program unit. IOSTAT's variable, if you include IOSTAT=, will contain an error code.

ERRORLOG= erl
erl is a character entity; it will receive the value YES if the unit/file will receive runtime error messages of any kind and NO if it will not. If the unit/file is not connected, *erl* will be undefined.

EXCLUSIVE= exc
exc is a character entity; it will receive the value YES if the unit/file is connected for exclusive access and NO if it is not connected for exclusive access. If the unit/file is not connected, *exc* will be undefined.

FORCE= for
for is a character entity; it will receive the value YES if F77 and system file buffering is minimized and NO if the buffering is not minimized. If the unit/file is not connected, *for* will be undefined.

FORM= fm
fm is a character entity; it will receive the value FORMATTED if the unit/file is connected for formatted I/O or the value UNFORMATTED if the unit/file is connected for unformatted I/O. If the unit/file is not connected, *fm* will be undefined.

FORMATTED= fmt
fmt is a character entity; it will receive the value YES if formatted I/O to the unit is allowed, NO if formatted I/O isn't allowed, and UNKNOWN if F77 can't tell whether formatted I/O is allowed. If there is no connection, *fmt* will be undefined.

IOINTENT= oio
oio is a character entity; it will receive the value OUTPUT if the unit is connected for OUTPUT only, INPUT if it is connected for INPUT only, or BOTH if it is connected for BOTH input and output. If the unit/file is not connected, *oio* will be undefined.

IOSTAT= ios
ios is an integer entity; it will return a status indicator. It will contain 0 if the INQUIRE executed normally or an F77 or system error code if INQUIRE encountered an error.

MAXRECL= imax
imax is an integer entity; it will receive the integer maximum record length, as defaulted or set by MAXRECL= during the OPEN statement. If the file does not exist or the unit/file is not connected, *imax* will be undefined.

MP/AOS and MP/AOS-SU do not keep track of as much information about a file as the other operating systems do. If an MP/AOS or MP/AOS-SU F77 program inquires into an existing but unconnected file, then the returned value of *imax* is 0.

MODE= mod
mod is a character entity; it will receive the value BINARY if the unit/file is connected for binary transfer, or TEXT if the unit/file is connected for text transfer (default). If the unit/file is not connected, *mod* will be undefined. MP/AOS and MP/AOS-SU F77 do not support MODE.

NAME= fn
fn is a character entity; it will receive the value of the full *pathname* (which includes the filename) of the file connected to this unit, or of the file itself. A typical pathname includes the directory names for the root directory, UDD directory, and user control point directory; e.g., :UDD:SALLY:filename. The pathname will include subordinate directory names if the file is in a subordinate directory. If the unit is not connected, *fn* will be undefined.

NAMED= nmd
nmd is a logical entity; it will receive the value .TRUE. if the file has a name, .FALSE. if it does not have a name. The only files that do not have a name are scratch files and files that the program has opened and has deleted while open. If the unit/file is not connected, *nmd* will be undefined.

INQUIRE (continued)

NEXTREC= inr
inr is an integer entity; it will receive the number of the next record in the file. If the file is connected but no records have been read or written since the OPEN statement, *inr* will return the value 1. If there is no connection or the connection is SEQUENTIAL (default), *inr* will be undefined.

NUMBER= inum
inum is an integer entity; it will return the unit number connected to the file. If the unit/file is not connected, *inum* will be undefined.

OPENED= od
od is a logical entity; it will receive the value .TRUE. if the file is connected to a unit or the unit is connected to a file. If the unit/file is not connected, *od* will return the value .FALSE..

PAD= pad
pad is a character entity; it will receive YES if padding was specified on the OPEN statement or NO if it was not specified. If the records are not data sensitive or variable length, or if the unit/file is not connected, *pad* will be undefined.

RECFM= rfm
rfm is a character entity; it will receive the value VARIABLE, DATASENSITIVE, FIXED, DYNAMIC, or UNKNOWN, depending on the record organization specified in RECFM= on the OPEN statement. This property may differ from the organization given when the file was created. If the unit is not connected, *rfm* will be undefined. If the file exists but is not connected, *rfm* will return the organization with which the file was created.

MP/AOS and MP/AOS-SU do not keep track of as much information about a file as the other operating systems do. If an MP/AOS or MP/AOS-SU F77 program inquires into an existing but unconnected file, then the returned value of *rfm* is UNKNOWN.

RECL= irl
irl is an integer entity; it will receive the byte length of each record in the file as given on the OPEN statement. If the unit is not connected, is connected for sequential access, or the file does not exist, *irl* will be undefined.

MP/AOS and MP/AOS-SU do not keep track of as much information about a file as the other operating systems do. If an MP/AOS or MP/AOS-SU F77 program inquires into an existing but unconnected file, then the returned value of *irl* is 0.

SCREENEDIT= sed
(AOS/VS only) *sed* is a character entity; it will receive the value YES if an on-line user may give SCREENEDIT control characters (such as CTRL-F for forward cursor movement) while preparing a string of ASCII characters as input to a READ statement. If SCREENEDIT is NO (default), then an on-line user cannot use SCREENEDIT control characters while keying in data.

SEQUENTIAL= seq
seq is a character entity; it will receive the value YES if sequential access to the unit is allowed, NO if sequential access isn't allowed, and UNKNOWN if F77 can't tell whether sequential access is allowed. If there is no connection, *seq* will be undefined.

MP/AOS and MP/AOS-SU do not keep track of as much information about a file as the other operating systems do. If an MP/AOS or MP/AOS-SU F77 program inquires into an existing but unconnected file, then the returned value of *seq* is UNKNOWN.

TYPE= ityp
ityp is an integer entity; it will receive the operating system file type, given or selected by default during the OPEN statement, as described in your operating system reference manual.

UNFORMATTED= unf
unf is a character entity; it will receive the value YES if unformatted I/O to the unit is allowed, NO if unformatted I/O isn't allowed, and UNKNOWN if F77 can't tell whether unformatted I/O is allowed. If there is no connection, *unf* will be undefined.

INQUIRE Examples

```
C   Example 1, INQUIRE by file.
    logical OD, EX
    ...
    inquire (file='AFILE', opened= OD, number= IU)
    if (OD) print *, 'AFILE open on unit ',IU
    ...


C   Example 2, INQUIRE by unit.
    character*30 FN
    logical OD
    ...
    inquire (3, name= FN, opened= OD)
    if (OD) print *, 'File ',FN,' open on unit 3.'
```

INTEGER

Declares one or more variables or arrays of type integer.

Form

INTEGER *[*len]* v *[(d)] [*len] [/ clist/]*
[,v [(d)] [/ clist /]] ...

where:

len is an optional byte length specifier for the variable or array elements. *len* can be 2 or 4. You can specify a *len* for each entity *after* its name; e.g., INTEGER FOO*2, FUM*4, GARRY(100)*2.

You can specify a default integer length at compilation time (F77/INTEGER=2 MYPROG). For AOS, F7716, MP/AOS, and MP/AOS-SU, the F77 compiler macros already include an /INTEGER=2 switch. Any length you specify in an INTEGER statement overrides the /INTEGER switch if there is a conflict.

v is the name you want for the variable, array, function, or dummy argument. This can be any valid symbolic name; the INTEGER statement overrides the name rule and IMPLICIT rules (if any).

d applies only to an array and is a dimension declarator; it specifies the dimensions for the array. The declarator can be an integer, integer constant, or dummy expression. For a lower array bound other than 1, the declarator must specify both upper and lower bounds; e.g., INTEGER JJ(2:9). Declarators are further described under DIMENSION and in Chapter 2.

clist is a constant list that includes one or more values for the entity. The entity will be initialized to this (these) value(s) when the program is built. If you initialize an array via *clist*, the *clist* must include exactly the number of values to fill the entire array. DG's F77 preserves the current values of all entities initialized via *clist* when it executes a subprogram's RETURN or END statement.

What It Does

The INTEGER statement declares one or more integer entities with the names, lengths, and (for arrays) the numbers of elements given. If you omit both an INTEGER and IMPLICIT statement, the name rule implies type integer for entities whose names begin with the letters I through N and ?. The range of numbers for a 4-byte integer is from at least -2,147,483,647 through 2,147,483,647; the range for a 2-byte integer is -32,768 to 32,767. Although integer*1 entities don't formally exist, logical*1/byte entities behave much the way you might expect integer*1 entities to behave. The range of numbers for a logical*1/byte entity is -128 to 127. Only AOS/VS F77 has the logical*1/byte data type.

As a type statement, INTEGER overrides both the name rule and an IMPLICIT statement. It is a nonexecutable statement; if you label it, no other statement can refer to the label. All type statements are specification statements and as such must precede statement function definitions and executable statements in a program unit.

You can establish the data type of a variable or array, and dimension an array, only once in a program unit.

INTEGER Examples

```
      integer  AMY

      INTEGER*2  ALLOW   / 999 /

      INTEGER*2 ABLE, AXY*4

      INTEGER  IARRY(20), JRY(10,20)

      INTEGER*2 HI_VALUE_2  / '99' /

      INTEGER*4 HI_VALUE_4A / '9999' /
      INTEGER   HI_VALUE_4B / '9999' /
C     HI_VALUE_4A and HI_VALUE_4B are the same
C         length and contain the same number
C         (as long as the /INTEGER=2 compiler
C         switch isn't set).
      INTEGER*4 HI_VALUE_4C / 4H9999 /  ! is
C         identical in length and content to
C         HI_VALUE_4A and HI_VALUE_4B.
      LOGICAL*1 HI_VALUE_1A / '9' /
      LOGICAL*1 HI_VALUE_1B / 'Z' /
      LOGICAL*1 HI_VALUE_1C / 127 /
      LOGICAL*1 HI_VALUE_1D / .TRUE. /
```

INTRINSIC

Identifies an intrinsic function name for use as an actual argument.

Form

INTRINSIC ifunc *[,ifunc]* ...

where:

ifunc is one of the legal F77 intrinsic function names given in Chapter 7, Tables 7-1, 7-2, 7-4, 7-5, and 7-6.

What It Does

INTRINSIC identifies an F77 intrinsic function name as an intrinsic function name, so that you can use it as an actual argument.

Normally, if you use an intrinsic function name as an actual argument, F77 will treat the name as a variable. But if you declare certain intrinsic functions INTRINSIC in a program unit, you *can* pass them as actual arguments.

Although you can declare them INTRINSIC, you cannot pass the names of intrinsic functions for type conversion, maximum/minimum value (e.g., AMIN), and lexical comparison functions. You *can* pass the names of *all* trigonometric functions (ACOS, etc.), functions like ABS, EXP, LOG, and SQRT, and word/bit functions (IAND, etc.).

INTRINSIC is a specification statement and as such must precede statement function definitions and executable statements in a program unit. Do not use a function name more than once in an INTRINSIC statement in any program unit.

INTRINSIC Example

```
C   Calling program.
    INTRINSIC SIN
    ...
    X = RFUNC(SIN)
    ...


C   Subprogram.
    FUNCTION  RFUNC(SIN2)
    ...
    PRINT *,  SIN2(Y)    ! Use function.
    ...
```

%LIST (compiler directive)

Stops or resumes output to the listing file.

Form

column 1

%LIST (OFF)

%LIST *[(ON)]*

where:

column 1 indicates column 1; no spaces or tabs are allowed before the directive.

What It Does

The %LIST(OFF) directive stops output to the listing file; the %LIST(ON) directive continues output to this file.

If you specify a listing file in the compilation line (via the /L switch, described in Chapter 9), the compiler will generate a program listing with line numbers and error notes; the listing can also include other information.

To stop compiler output to the listing file, insert the %LIST(OFF) directive in your source; to resume listing, use %LIST(ON). The effect of each directive will persist until the next %LIST directive or end of file, whichever comes first. A file included with an %INCLUDE directive can set its own listing options to override those of the original source; the original source list setting will return after the %INCLUDEd file.

A %LIST directive *must start at column 1* and cannot be continued onto a second line.

The %LIST directives appear in the listing file.

%LIST Example

```
        PROGRAM MYPROG
          ...
C23456789...
%LIST (OFF)
%INCLUDE 'ERR.F77.IN'
%INCLUDE 'MYDEFS.QQ'
%LIST (ON)
          ...
```

LOGICAL

Declares one or more variables or arrays of type logical.

Form

LOGICAL *[*len] v [(d)] [*len] [/ clist /] [, v [(d)] [/ clist /]] [...]*

where:

len is an optional byte length specifier for the variable or array elements. *len* can be 1 (AOS/VS only) or 2 or 4 (standard). BYTE has the same effect as LOGICAL*1. You can specify a *len* for each entity *after* its name; e.g., LOGICAL LFOO*2, LARY(10)*4.

You can specify a default *len* at compilation time (F77/LOGICAL=2 MYPROG). For AOS, F7716, MP/AOS, and MP/AOS-SU, the F77 compiler macros already include a /LOGICAL=2 switch. Any length you specify in a LOGICAL statement overrides the /LOGICAL switch if there is a conflict.

v is the name you want for the variable, array, function, or dummy argument. This can be any valid symbolic name; the LOGICAL statement overrides any IMPLICIT rules.

d applies only to an array and is a dimension declarator; it specifies the dimensions for the array. The declarator can be an integer, integer constant, or dummy expression. For a lower array bound other than 1, the declarator must specify both upper and lower bounds; e.g., LOGICAL LG(2:9). Declarators are further described under DIMENSION and in Chapter 2.

clist is a constant list that includes one or more values for the entity. The entity will be initialized to this (these) value(s) when the program is built. If you initialize an array via *clist*, the *clist* must include exactly the number of values to fill the entire array. DG's F77 preserves the current values of all entities initialized via *clist* when it executes a subprogram's RETURN or END statement.

What It Does

The LOGICAL statement declares one or more logical entities with the names, lengths, and (for arrays) the numbers of elements given.

As a type statement, LOGICAL overrides an IMPLICIT statement, if any. It is a nonexecutable statement; if you label it, no other statement can refer to the label. All type statements are specification statements and as such must precede statement function definitions and executable statements in a program unit.

You can establish the data type of a variable or array, and dimension an array, only once in a program unit.

Logical*1/byte entities can, along with logical*2 and logical*4 entities, contain the values .TRUE. and .FALSE.. Logical*1/byte entities can also contain one ASCII character or an integer number between -128 and +127. Only AOS/VS F77 has the logical*1/byte data type.

LOGICAL Examples

```
LOGICAL  L

LOGICAL   L1   /.True./

LOGICAL*2   L2, L5*4

LOGICAL L5(20,20)

LOGICAL  LX

LOGICAL*1 BYTE1 / .TRUE. /
LOGICAL*1 BYTE2 / 'M' /
LOGICAL*1 BYTE3 / 100 /
...
LX = K .GT. 4
IF (LX)  PRINT *, 'LX is true.'
```

OPEN

Connects a unit to an external file and establishes connection properties.

Form

```
OPEN ( [UNIT=]iu [ ,ACCESS= acc
                   ,BLANK= blnk
                   ,BLOCKSIZE= iblks
                   ,CARRIAGECONTROL=cc
                   ,DELIMITER= del
                   ,FILE= fin
                   ,ERR= s
                   ,ERRORLOG= erl
                   ,EXCLUSIVE=exc
                   ,FORCE= for
                   ,FORM= fm
                   ,IOINTENT= oio
                   ,IOSTAT=ios
                   ,MAXRECL= imax
                   ,MODE= mod
                   ,PAD= pad
                   ,POSITION= pos
                   ,RECFM= rfm
                   ,RECL= irl
                   ,SCREENEDIT= sed
                   ,STATUS= sta
                   ,TYPE= ityp ] )
```

where:

UNIT= iu — is an integer expression that specifies the unit you want to open. You can omit UNIT= if this specifier is the first argument. iu must evaluate to an integer from 0 through 255 for AOS/VS or from 0 through 63 for AOS, F7716, MP/AOS, and MP/AOS-SU.

ACCESS= — and other property specifiers are described alphabetically below under "OPEN Property Specifiers".

What It Does

OPEN connects a unit with a file and establishes connection properties for the connection.

After you open a unit/file connection, you can reopen it with the same unit/file specifiers to change the connection properties. The properties whose values you can change on a reopen are ACCESS, BLANK, CARRIAGECONTROL, DELIMITER, ERRORLOG, FORCE, FORM, MAXRECL, MODE, PAD, RECFM, RECL, and SCREENEDIT. If you specify unneeded properties on any open, F77 discards them; but if you specify conflicting properties, F77 will report a compilation or runtime error. Conflicting properties are described later.

AOS/VS programmers may use a logical*1/byte entity as an OPEN specifier whenever character or integer entities are allowed. An exception is IOSTAT, since all I/O error codes exceed 127.

You can include a filename (which can be a pathname including the filename) in FILE=; if you omit FILE=, F77 may use a predetermined disk file as described in STATUS=. If you omit the error handling specifiers IOSTAT= and ERR= the program will terminate if an error occurs on the OPEN statement.

Each unit/file connection is global while it exists; it applies throughout *all* program units. Any program unit can issue OPEN, which will be a reopen if any unit has already opened the connection — unless the OPEN statement deletes and recreates the pertinent file.

If a unit number is in use and you specify a filename that differs from the one connected to the unit, F77 closes the original connection and opens the file specified on this unit number. Units 5, 6, 9, 10, 11, and 12 are preconnected; opening any of these with an explicit filename will sever the preconnection and establish the new connection while the program runs. Exception: MP/AOS and MP/AOS-SU F77 do not have preconnected units 9 and 12.

OPEN Property Specifiers

Most *arguments* to specifiers are type character; those that begin with *i* are integer.

ACCESS= acc
To specify the type of access, provide a character expression of 'DIRECT' or 'SEQUENTIAL' for *acc*. For ACCESS='DIRECT', you must include a length in RECL=. If you omit ACCESS=, F77 connects the file for sequential access.

BLANK=blnk
To have F77 interpret embedded blanks as 0s when it reads numeric values from this file, specify BLANK='ZERO'. By default, F77 interprets blanks embedded in numeric values as nulls, ignoring them. Thus the default is BLANK='NULL'. You can also control blank interpretation in READ statements with the BZ/BN edit descriptors.

BLOCKSIZE= iblks
To specify a nonstandard device block size, provide a positive integer expression of the new size in *iblks*. The default block size is the system default size. An *iblks* value of 0 selects the default size. MP/AOS and MP/AOS-SU F77 do not support BLOCKSIZE.

CARRIAGECONTROL= cc
To specify carriage control for output of formatted records, provide a character expression of 'FORTRAN', 'NONE', or 'LIST' for *cc*. The default is CARRIAGECONTROL='LIST'.

For traditional FORTRAN carriage control, in which the first character within each record is used for vertical printing control, specify CARRIAGECONTROL='FORTRAN'. The first character will be interpreted as follows on output:

First Character in Record	Result when Record Is Printed
□	Prints record on next line (NEW LINE, record).
0	Prints record on line after next (NEW LINE, NEW LINE, record).
1	Prints record on top of next page (form feed, record).
+	Prints record on same line, starting with column 1, thus overprinting the current record (carriage return, record).
#	Prints record starting at current column position (useful to print multiple records on one line).
other character	Prints record on next line (NEW LINE, record).

On output to a data-sensitive file opened with FORTRAN carriage control, F77 does *not* terminate the file with a NEW LINE character. Such files will print correctly, but you cannot edit the last line successfully with a line-oriented text editor (the SPEED editor works well). One way to terminate the file with a NEW LINE character is with a statement pair such as

```
    WRITE (FIL_NUM, 999)
999 FORMAT (' ')  ! results in a <NL>
```

For LIST carriage control, F77 inserts a NEW LINE (<NL>) character after each record, which will print each record on a new line. You can suppress the NEW LINE with the $ edit descriptor.

For CARRIAGECONTROL='NONE', F77 inserts no carriage control characters.

Regardless of CARRIAGECONTROL, a file will print directly only if it was created and/or opened with data-sensitive record organization (the default for formatted records if you omit RECFM= and omit ACCESS='DIRECT').

If a file does not have data-sensitive organization, you can use the F77PRINT postprocessor to produce a printable version of it. F77PRINT can produce a printable version of *any* F77 output file.

DELIMITER= del
To have the data-sensitive delimiter (usually NEW LINE) read in as a character on input, specify DELIMITER='INCLUDE'. Delimiters will then be stored as characters. The default is DELIMITER='EXCLUDE', which strips the delimiter. DELIMITER has no effect on output or with files that have other than data-sensitive organization.

ERR= s
To specify a statement that will execute on an error, use that statement's number as *s*. *s* must be an executable statement in the current program unit. On an OPEN error, statement *s* will get control and IOSTAT's variable, if you include IOSTAT=, will contain an error code.

ERRORLOG= erl
erl is a character expression whose value (after F77 has ignored any trailing blanks and has translated any lowercase letters to uppercase) is 'YES' or 'NO'.

If *erl* is 'YES', then the unit is added to the program's set of error log units. This means that F77, in response to any runtime error, will write a report about the error to this set of units/files.

If *erl* is 'NO', the unit never receives an error message when a runtime error occurs. The default value of *erl* is 'NO'.

The default preconnection for unit 6, @OUTPUT, includes ERRORLOG='YES'.

F77 sends runtime error messages to all units opened with ERRORLOG='YES'. If there is no such unit, then F77 attempts to send any such message to @OUTPUT.

The current value of *erl* may change if a program unit reopens the unit.

EXCLUSIVE= exc
To connect the file for exclusive access, specify EXCLUSIVE='YES'. (If you specify exclusive access, no other connection can be made to the file and you can write to it at will.) If you specify EXCLUSIVE='YES' and the file is already connected, you'll receive an error. The default is EXCLUSIVE='NO'.

OPEN (continued)

FILE= fin
To specify the filename to connect, use a character expression with the file or pathname for *fin*. Valid system filenames include alphabetic A–Z, upper- and/or lowercase, numbers 0–9, $, period (.), question mark (?), and underscores (_); they can include up to 32 characters (15 for MP/AOS and MP/AOS-SU). If the file will not be in the directory where you will execute the program, specify a pathname through the directory tree to it; e.g., FILE=':UDD:MYDIR:MYFILE'. If you omit FILE=, F77 will use a predetermined file as described under STATUS=.

FORCE= for
To minimize F77 and system file buffering during I/O, use FORCE='YES'. For faster file I/O, which occurs with buffering, use FORCE='NO' (the default).

FORM= fm
To specify unformatted records, use FORM='UNFORMATTED'; or to specify formatted records, use FORM='FORMATTED'. If you said ACCESS='SEQUENTIAL' or defaulted ACCESS=, the default FORM is formatted. If you said ACCESS='DIRECT', the default is unformatted. FORM allows you to specify nondefault combinations of access and format properties.

IOINTENT= oio
To specify input only for this connection, use IOINTENT='INPUT'. To specify output only, use IOINTENT='OUTPUT'. The default is IOINTENT='BOTH'.

IOSTAT= ios
To specify an error status indicator, use an integer entity for *ios*. After the OPEN statement, the value of the *ios* entity will be 0 if the OPEN executed normally. If an error occurred, *ios* will contain an F77 or system error code.

If you include IOSTAT= and omit ERR=, program execution will continue at the next statement if an error occurs. If you omit both IOSTAT= and ERR=, program execution will terminate on an OPEN error.

MAXRECL= imax
To specify a nondefault maximum record length for this connection, use an integer expression for *imax*. *imax* will then become the maximum record length allowed for this connection. MAXRECL pertains only to variable, data-sensitive, and fixed files. Use RECL= for a file connected for direct access. Furthermore, MAXRECL specifies the exact length of all the records when a file with RECFM='FIXED' is created.

If any record exceeds the maximum length on a WRITE statement, F77 will generate a runtime error. If, on a READ statement, a variable-length record exceeds *imax*, F77 also will generate a runtime error. If, on a read, a *data-sensitive* record exceeds *imax*, F77 reports no error but reads the record up to *imax*; the rest of the record will be available on the next read.

For AOS/VS variable records, the default maximum record length is 9,995 bytes; it is 136 bytes for other records. The AOS/VS system limit for variable record length is 9,995 bytes. For data-sensitive records the limit is 32,766 bytes. For fixed-length records, it is 32,767 bytes (formatted) or 65,534 bytes (unformatted). For dynamic records, there is no inherent length limit.

For AOS, F7716, MP/AOS, and MP/AOS-SU, the default maximum record length is 136 bytes. The AOS/family system limit is 9,995 bytes for variable records and 65,534 bytes for other records.

The system file organization — variable length, data sensitive, fixed, or dynamic — is given or defaulted as described under RECFM=, below.

In contrast to the other operating systems, MP/AOS and MP/AOS-SU do *not* keep track of a file's record length. You need to include MAXRECL with every OPEN statement (MP/AOS, MP/AOS-SU programs only). The specified value of MAXRECL becomes the length of all records in the file.

MODE= mod
To have all characters transferred without interpretation or parity checks, specify MODE='BINARY'. This allows F77 to read character sequences like CTRL-C CTRL-B or others from the terminal without interrupting program execution. BINARY mode lasts only while this connection is open. The default is MODE='TEXT', in which control character sequences *are* interpreted. MP/AOS and MP/AOS-SU F77 do not support MODE.

PAD= pad
To have short input records padded to MAXRECL, specify PAD='YES'. The pad character(s) will be blanks for formatted records and nulls for unformatted records. Padding will prevent "Attempt to read past end of record" errors when the input format specifies more characters than remain in the record. But it will not prevent such errors if the MAXRECL length is less than the number of characters required by the input format.

PAD is meaningful only for variable or data-sensitive file organizations; F77 automatically pads records in fixed files to the length you give in RECL=. The default is PAD='NO'. On list-directed reads, *pad* is ignored and F77 treats the read as if PAD='NO'.

POSITION= pos
To position the file before the first record, specify POSITION='START'. For current position, specify POSITION='CURRENT'. To position after the last record (e.g., to append), specify POSITION='END'. After the first OPEN statement for a file, the CURRENT position is START. The only legal value on a reopen is POSITION='CURRENT'. POSITION has no meaning for files that cannot be positioned, like the terminal. The default is POSITION='CURRENT'.

RECFM= rfm
To specify a nondefault system record organization if this open creates the file, use 'VARIABLE', 'FIXED', 'DYNAMIC', or 'DATASENSITIVE' ('DS'), for *rfm*. DATASENSITIVE is the default unless ACCESS='DIRECT', in which case the default is 'FIXED.' Only data-sensitive files will print directly.

In contrast to the other operating systems, MP/AOS and MP/AOS-SU do *not* keep track of a file's record format. Unless the value of the RECFM specifier is the default, you need to include it with every OPEN statement in your MP/AOS and MP/AOS-SU F77 programs.

RECL= irl
To specify the record byte length for direct access, use a positive integer expression for *irl*. On output, F77 will pad records if needed to *irl*; it will use blanks for formatted records or nulls for unformatted records. You can specify the same *irl* length, or a different one, whenever you open this file. There is no default; to connect a file for direct access, you must include RECL=. To connect it for sequential access, you must omit RECL=.

In contrast to the other operating systems, MP/AOS and MP/AOS-SU do *not* keep track of a file's record length. You need to include RECL with every OPEN statement in which you specify ACCESS='DIRECT' (MP/AOS, MP/AOS-SU programs only).

SCREENEDIT= sed
sed is a character expression that, after F77 has ignored any trailing blanks, is 'YES' or 'NO'. If SCREENEDIT='YES', F77 will allow an on-line user to use SCREENEDIT control characters (such as CTRL-F for forward cursor movement) while preparing a string of ASCII characters as input to a READ statement. If SCREENEDIT='NO' (default), then an on-line user cannot take advantage of these SCREENEDIT control characters while keying in data.

STATUS= sta
To specify how the system will establish a unit/file connection, use 'NEW', 'FRESH', 'OLD', 'SCRATCH,' or 'UNKNOWN' for *sta*.

- For STATUS='NEW' also give the name in FILE=. F77 will attempt to create the file and will return an error if it exists.
- For STATUS='FRESH', if you give the name in FILE=, F77 will attempt to delete, then will create the file. If you omit FILE=, F77 will construct a filename based on a question mark and your process identification (pid) number.
- For STATUS='OLD', also give the name in FILE=. F77 will attempt to open the file and will return an error if it doesn't exist.
- For STATUS='SCRATCH', omit FILE=. F77 will create (or delete and recreate, if it exists) a scratch disk file in the working directory. The scratch filename is inaccessible from F77 programs. On the CLOSE statement, or when the program ends, F77 will delete the scratch file.
- For STATUS='UNKNOWN', if you include FILE=, F77 will open the file if it exists or create and open it if it doesn't exist. If you omit FILE=, F77 will use the filename ?pid.F77.UNIT.unit (?pid.F77U.unit under MP/AOS and MP/AOS-SU), where pid is your integer process ID and unit is the integer unit number. (UNKNOWN differs from FRESH in that it does not delete the file if it exists.)

On an initial open, if you omit the STATUS= specifier, F77 makes the connection with STATUS='UNKNOWN'. On a reopen, you can omit STATUS=.

TYPE= ityp
To specify a nondefault operating system file type, use an integer expression for *ityp*. The integer expression gives the file type, described in your operating system reference manual.

Table 8-1 shows the illegal combinations and values for the OPEN property specifiers.

Table 8-1. Illegal OPEN Property Specifiers

Property	Other Property, Value, Description
UNIT= <0	always an error.
UNIT= >255 (AOS/VS)	always an error.
UNIT= >63 (AOS, F7716, MP/AOS, MP/AOS-SU)	always an error.
ACCESS='SEQUENTIAL' (or omitted)	with RECL=, an error.
ACCESS='SEQUENTIAL' (or omitted) and RECFM='FIXED.'	without MAXRECL=, an error (unless the fixed-length file already exists).
ACCESS='DIRECT'	without RECL=, an error; with RECFM= anything but 'FIXED', an error.
CARRIAGECONTROL= *cc* or DELIMITER= *del*	with FORM='UNFORMATTED', specifier is ignored, no error.
FORM='FORMATTED'	with RECFM='DYNAMIC', an error.
MAXRECL= <1 or MAXRECL= >65534	always an error.
MAXRECL= >9995	with variable records, an error.
POSITION='START' or POSITION='END'	on reopen, an error.
RECFM='DYNAMIC'	with FORM='FORMATTED', an error.
RECFM='DATASENSITIVE'	with FORM='UNFORMATTED', an error.
RECL= *irl*	with ACCESS='SEQUENTIAL', an error.
RECL= *irl* (omitted)	with ACCESS='DIRECT', an error.
STATUS='SCRATCH'	with FILE=, an error.
STATUS='NEW' or STATUS='OLD'	without FILE=, or on reopen, an error.
STATUS='UNKNOWN' or STATUS='SCRATCH'	On reopen, an error.

F77 Preconnections

F77 has the following preconnections. All preconnection *properties* are detailed in Table 5-3.

MP/AOS and MP/AOS-SU F77 do not have preconnected units 9 and 12. By default, their preconnected units 5 and 11 use channel ?INCH (process input), while their preconnected units 6 and 10 use channel ?OUCH (process output).

Programs that use these preconnections are not directly transportable to (non-DG) processors that have different preconnections.

Unit	Filename and Description
5	@INPUT. For interactive use, this is the terminal keyboard. On READ statements, you can use either the unit number 5 or an asterisk (*) to specify this file. It is connected with SCREENEDIT='YES' and FORCE='YES', except for MP/AOS and MP/AOS-SU; they have SCREENEDIT='NO' and FORCE='YES.'
6	@OUTPUT. For interactive use, this is the terminal screen or printer. On WRITE statements, you can use either the unit number 6 or an asterisk (*) to specify this file. It is connected with RECFM='DS', CARRIAGECONTROL='FORTRAN', and FORCE='YES'.
9	@DATA. This is a file whose name you can set with the system CLI DATAFILE command. It is connected with RECFM='DS', CARRIAGECONTROL='LIST', and FORCE='NO'. MP/AOS and MP/AOS-SU do not have this preconnection.
10	@OUTPUT. It is the same file as 6 but is connected with CARRIAGECONTROL='LIST', ERRORLOG='YES', and FORCE='YES'. You cannot use an asterisk to specify it.
11	@INPUT. It is the same file as 5 but you cannot use an asterisk to specify it. It is connected with SCREENEDIT='YES' and FORCE='YES', except for MP/AOS and MP/AOS-SU; they have SCREENEDIT='NO' and FORCE='YES.'
12	@LIST. This is a file whose name you can set with the system CLI LISTFILE command. It is connected with RECFM='DS', CARRIAGECONTROL='FORTRAN', and FORCE='NO'. MP/AOS and MP/AOS-SU do not have this preconnection.

OPEN Examples

```
      OPEN (1, IOSTAT=IOS)

      OPEN (15, ERR=99, FILE= ':UDD:AL:IDATA')

      OPEN (0, ERR= 999, STATUS='SCRATCH')

      open (2,  status='FRESH',
     +          carriagecontrol= 'FORTRAN',
     +          file= 'M_DATA')

      OPEN (3, FILE='@LPT')    ! Line printer
                               !    queuename.

      open (4, access='DIRECT', recl=80,
     +         iostat= IOS, file='D_RECS',
     +         status= 'NEW')
```

OPEN (continued)

The following OPEN example reads records from one disk file and writes them to another.

```
      DIMENSION ARY (1000)
      ...
      OPEN (1, STATUS= 'OLD', FILE= 'SOURCE',
     +        PAD= 'YES')
      OPEN (2, FILE='DEST')

      ICOUNT = 1
30    READ   (1, '(F10.0)', IOSTAT=IOS, END= 50)
     +                      ARY(ICOUNT)

      WRITE   (2, '(1X, G13.4)', IOSTAT= IOS)
     +                           ARY(ICOUNT)
      ICOUNT = ICOUNT + 1
      GO TO  30
50    PRINT *,  ICOUNT,
     +          ' records written to file DEST.'
      CLOSE (1)
      CLOSE (2)
      END
```

The next example copies to a fresh file all records that begin with B.

```
      PARAMETER (INPUT= 15)  ! Unit 15 for input
                             !    (source) file.
      CHARACTER*80 KEY       !  For current
                             !    record.
      ...
      OPEN (INPUT, PAD= 'YES', FILE='NAMES',
     +             STATUS='OLD')

      OPEN (INPUT+1, FILE='BNAMES',
     +               STATUS='FRESH')

      LCNT =  0
10    READ (1, '(A)', END=110,
     +                 RETURNRECL=IRCL)  KEY
      ...
      IF  (KEY(1:1) .EQ. 'B')  THEN
C          Here, could manipulate the record...
C          but simply write it to output file.
           WRITE (2, '(A)')    KEY(1:IRCL)
           LCNT = LCNT + 1
      END IF
      GO TO 10
110   PRINT *, LCNT, ' records beginning with ',
     +               'B written to file BNAMES.'
      ...
```

Note how the use of RETURNRECL and IRCL prevent F77 from placing padded (to 80 characters) records in the output file.

PARAMETER

Names a constant.

Form

PARAMETER (p = expr *[,p = expr]* ...)

where:

p is the symbolic name for the constant. It must match the general class of the constant value (integer, real, etc.).

expr is a numeric, logical, character, or Hollerith expression whose value is a constant expression. All operators are allowed.

What It Does

PARAMETER associates a symbolic name with a constant value of expr. After you use it to assign a value to name p, you cannot assign any value to that name; name p will retain its PARAMETER value throughout the program unit.

PARAMETER is a specification statement and must precede statement function definitions and executable statements in a program unit. It must assign a value to name p before the program unit refers to name p — for example, in a DATA statement.

The constant value of expr must match the general class of name p. If expr yields a numeric value, p must be a numeric type; if expr yields a logical value, p must be declared or implied type logical; if expr yields a character value, p must be declared or implied type character.

You can use name p in an expression within subsequent DATA or PARAMETER statements.

You can define a character entity's length as an asterisk if you later define that character name as a constant with PARAMETER.

You cannot refer to a substring of a constant named with PARAMETER.

PARAMETER Examples

```
      PARAMETER   (PI = 3.14159)     ! Param PI.


C     The next two statements are equivalent.
C        In each of them the fourth byte
C        of BIOLOGY_PREFIX is blank (<040>).
      PARAMETER (BIOLOGY_PREFIX = 'BIO')
C     PARAMETER (BIOLOGY_PREFIX = 3HBIO)


C     Another example.
      PARAMETER   (IP = 10000)
      DIMENSION IAR(IP)
      ...
      DO 8 I = 1, IP
         IAR(I) = 0
8     CONTINUE


C     Example with CHARACTER*(*).
      CHARACTER*(*)  PART     ! Use * for length.
      CHARACTER*10  P_NUM
      PARAMETER  (PART='093-162-')
      ...
      P_NUM = PART   // '03'
C     P_NUM contains 930-162-03.
```

PAUSE

Suspends program execution until user or operator intervenes on terminal.

Form

PAUSE *[n]*

where:

n is a string of digits or a character constant.

What It Does

PAUSE suspends the program, waiting for user or operator intervention. F77 displays PAUSE and the text message *n* on the original terminal if available; it may display other messages as described below.

There are two reactions to the PAUSE statement, depending on whether the runtime operating system is either AOS/VS or AOS, or else MP/AOS or MP/AOS-SU.

AOS/VS and AOS Runtime PAUSE Result

If the program was originally executed in interactive mode (from a terminal), the F77 processor suspends the program and attempts to send the PAUSE message to the terminal from which the program was executed.

At this point you, or anyone at the terminal, can

- continue the program by pressing any data-sensitive delimiter, like NEW LINE (↓); or
- terminate the program by typing CTRL-C CTRL-B.

The program remains suspended until it receives one of these sequences.

If the F77 processor cannot send the PAUSE message to its original terminal, it attempts to walk up the system process tree. (This would happen if the program had been executed in batch mode, and thus *didn't have* a terminal.)

If a father process has a terminal connected, F77 sends the message *FORTRAN PAUSE*, then the message *n* if any, then instructions. The person at this terminal can either unblock or terminate the program per instructions; the program will wait indefinitely. If a father process has a terminal connected, but messages to this terminal are disabled, F77 issues a nonfatal error message to the process' @OUTPUT file and continues up the process tree. If a father process has *no* terminal connected, F77 simply continues up the process tree.

PAUSE (continued)

If F77 reaches the operator process (OP, PID 2), it checks if the system operator is off duty. If the system operator is off duty, F77 displays *FORTRAN PAUSE* and message *n* (if any); then it displays *program continuing* and continues program execution. If the system operator is on duty, or if F77 can't tell whether an operator is on duty, it displays *FORTRAN PAUSE*, message *n* (if any), and instructions. It will wait indefinitely to be unblocked or terminated if the operator is not off duty.

MP/AOS and MP/AOS-SU Runtime PAUSE Result

Here, PAUSE functions differently. PAUSE attempts to open the files connected to channels ?INCH (process input) and ?OUCH (process output) when the program began to execute. If both these files can be opened and are terminal files (@TTIx, @TTOy), then the PAUSE message goes to ?OUCH and the program waits for a response from ?INCH before continuing. One of the following messages goes to ?OUCH:

**** Fortran Pause ****
Type NEW-LINE to continue.

**** Fortran Pause *** Message is:*
<string of digits or character constant in PAUSE n statement>
Type NEW-LINE to continue.

If F77 cannot open both channels or if both files are not terminal files, then an appropriate error message is sent to all ERRORLOG files and execution continues. The error message includes any string of digits or character constant in the PAUSE statement.

PRINT

Transfers formatted values to unit 6, preconnected as the terminal screen or printer.

Form

PRINT f *[,iolist]*

where:

f — is a format identifier. f can be the label of a FORMAT statement, an integer variable assigned the label of a FORMAT statement, a character array name, a character expression (entity name or constant like '(1X,I8)'), or an asterisk to specify list-directed formatting. Character entities and expressions as formats are described near the beginning of Chapter 6; list-directed formatting is detailed near the end of Chapter 6.

iolist — is an I/O list of entities whose values will be written to unit 6. It can include variables, array elements, implied DO lists, and/or array names. It can also include expressions; e.g., 73 or K+5 or 'ABC'. And, if the entity is a function, the function executes; e.g., PRINT *, MYFUNC(K,L).

What It Does

PRINT transfers records to unit 6, which is preconnected to file @OUTPUT. For interactive (as opposed to batch) use, file @OUTPUT is the terminal screen or printer.

PRINT is a shorthand WRITE statement, but cannot write to any unit other than 6. It does not allow a control information list (cilist) for unit specifier, error control, and so on.

After PRINT executes, the file is positioned after the last record transferred.

PRINT Examples

```
C   List-directed PRINTs.
    PRINT *,  'IF00 is ', IF00

    PRINT *,  B,C,D

    PRINT *, (RARY(J), J = 50, 100)  ! Implied DO.

C   Edit-directed PRINTs.
    PRINT   '(1X, F9.2)',    B,C,D

    PRINT 96,   B,C,D
96  FORMAT   ( ' Values are ' , 3F9.2 )
```

Other PRINT examples appear under FORMAT and OPEN.

PROGRAM

Names a main program.

Form

PROGRAM pgm

where:

pgm is the symbolic name for the main program. This name must differ from all subprogram and common block names within this program unit. A valid symbolic name is 1 to 32 characters and includes letters, numbers, question marks, and underscores; the first character must be a letter or the question mark. MP/AOS and MP/AOS-SU names follow the same rules except that they are unique only up to the first ten characters.

What It Does

PROGRAM gives the name you specify to the program. If you omit a PROGRAM statement, the default main program name is .MAIN. If a program unit has a PROGRAM statement, PROGRAM must be the first statement in the unit. Either the default or PROGRAM-assigned name provide a reference for the high-level debugger, SWAT, and for the operating system's debugger.

Only one program unit of the unit(s) you link to form the application program can be a main program. A main program can include only one PROGRAM statement and cannot include statements that identify it as a different unit; e.g., FUNCTION, SUBROUTINE, or BLOCK DATA.

Note that a name given with PROGRAM, FUNCTION, SUBROUTINE, or BLOCK DATA does not necessarily relate to the program unit's *filename*. You determine each filename when you create the file, via a text editor or otherwise; throughout program development, the compiler and Link programs maintain the major part of this name (described in Chapter 9).

PROGRAM Example

```
PROGRAM  SUPERVISOR
COMMON  / CBLK / RARRY(200), ACM, BCM
EXTERNAL  SUB, SUB1, FUNC, FUNC1
...
END
```

READ

Transfers values from a file to iolist entities.

Forms

READ f *[,iolist]*

READ (cilist) *[iolist]*

where:

f is a format identifier. f can be the label of a FORMAT statement, an integer variable assigned the label of a FORMAT statement, a character array name, a character expression (entity name or constant like '(1X,I8)'), or an asterisk to specify list-directed formatting. Character entities and expressions as formats are described near the beginning of Chapter 6; edit descriptors you can use with them are sketched under FORMAT. List-directed formatting is detailed near the end of Chapter 6.

iolist is an I/O list of entities that will receive the values read. It can include variables, array elements, substrings, implied DO lists, and/or array names.

cilist is a control information list. It must include a unit specifier and, for formatted records, a format identifier, f. Other specifiers are optional. The READ cilist specifiers are named here and detailed later:

[UNIT=] iu
[FMT=] f
END = s
ERR = s
IOSTAT = ios
REC = irn
RETURNRECL = iret

What It Does

The first (short) form of a READ statement specifies unit 5 (preconnected to file @INPUT). This is a formatted, sequential read.

The second form of a READ can read from any legal, specified unit. For a formatted READ, include FMT=f; the data types of the iolist entities must match the type of the values read. For an unformatted READ, omit FMT=f; F77 will read values into the iolist entities regardless of their data types.

Each READ from an internal file is sequential. The statement must include an *edit-directed* format identifier. Unformatted READs or READs that specify list-directed formatting (FMT=*) are not allowed for internal files.

For any READ, if the iolist includes an unsubscripted array name, F77 will read until it has read values for all array elements in the traditional column/row order.

F77 always attempts to read characters from the record into all the iolist entities. If the record does not contain enough characters to satisfy the iolist entities, F77 will report an *End of record* runtime error, depending on the PAD= property and number of characters required by the iolist. (Preconnected units are opened with PAD='YES'.) F77 will read in a new record if

1. it encounters a slash in the format specification; or
2. the format specification has too few edit descriptors, causing format reversion.

Both of these conditions are described in Chapter 6, in the section "Multiple Record Formatting."

After a READ executes successfully, F77 has read at least one record and the iolist entities are defined with characters read. The file is positioned at the beginning of the next record.

If an error occurs, the file position and all iolist entities are undefined.

READ cilist Specifiers

[UNIT=] iu
iu specifies the unit. For an external file, iu is an integer expression that evaluates to a number from 0 through 255 for AOS/VS or from 0 through 63 for AOS, F7716, MP/AOS, and MP/AOS-SU (but you can use * to specify unit 5). For an internal file, iu is the name of the character entity that is the internal file. You can omit UNIT= if the unit specifier is the first argument in the cilist.

[FMT=] f
f is the format identifier, described above. If you omitted UNIT= from the unit specifier, and if the format identifier will be the second argument in the cilist, you can omit FMT=. For example, the following statements are equivalent:

```
READ (UNIT=14, FMT ='(I8)')   J, K
READ (14, '(I8)')    J, K
```

END= s
s is the label of an executable statement that will receive control if a READ statement encounters an end of file or a CTRL-D sequence from @INPUT. *s* must be in this program unit. When *s* gets control, IOSTAT's variable will contain a negative value. On an end of file, a READ statement terminates and the statement labelled *s* gets control. If you omit both END= and IOSTAT=, the *program* terminates on an end of file.

ERR= s
s is the label of an executable statement that will receive control if an error occurs. Statement *s* must be in this program unit. When *s* gets control, IOSTAT's variable will contain the error code.

IOSTAT= ios
ios will return a status indicator. *ios* is an integer entity that will return 0 if READ executed normally. *ios* will return a negative value on an end of file or an F77 or system error code if an error occurred.

If you include IOSTAT= and omit ERR= and END=, program execution continues at the next executable statement on an error or end of file.

REC = irn
irn gives the record number, for direct access only. *irn* is a positive integer expression that specifies the number of the record you want for a direct-access read. The unit must have been OPENed with ACCESS='DIRECT'. If you specify a nonexistent record number, the iolist entities will be undefined. If you include REC=, you cannot also include END= in the cilist. For an internal file, REC= is not allowed.

RETURNRECL= iret
iret is an integer entity that will return the byte length of the record read.

READ Examples

```
C   Formatted, sequential READs, list and
C      edit directed.

    READ *,  AVAR

    READ   (*, '(A)')    CHR

    READ   (1, '(I8)')   J

    READ   (2, '(2 F9.2)', IOSTAT=IOS)  B, C

    READ   (2, 50, ERR =1000)    X, Y, Z
50  FORMAT (3E11.4)

    READ *,  (ARY(J), J=10,1,-1)   ! Implied DO.

C   Formatted, direct-access READ:
    READ  (2, 90, REC=J, ERR=99)   B1, B2
90  FORMAT  (2 E11.4)

C   Unformatted READ:
    READ   (3, IOSTAT= IOS)   B3, B4
```

An example that reads formatted real records from a file connected for sequential access is

```
    ...
    DIMENSION  ARY2(1000)
    OPEN  (2,  FILE='PFILE', PAD= 'YES')

    READ  (2, 50, END=150 )   ARY2
50  FORMAT (F13.0)

150 WRITE  (*, 200) (J, ARY2(J), J=1,100)
200 FORMAT (1X, 'Value ',I4, ' is ',G11.4)
    CLOSE (2)
    ...
```

There are other READ examples under FORMAT and OPEN.

REAL

Declares one or more variables or arrays of type real.

Form

REAL *[*len] v [(d)] [*len] [/ clist/] [, v [(d)] [*len] [/ clist /]] ...*

where:

len is an optional byte length specifier for the variable or array elements. It can be 4 (default) or 8. You can also specify a *len* for each entity *after* its name; e.g.,
REAL NUM,NUM1*8,NARRY(100)*4.

v is the name you want for the variable, array, function, or dummy argument. This can be any valid symbolic name; the REAL statement overrides the name rule and IMPLICIT rules (if any).

d applies only to an array and is a dimension declarator; it specifies the dimensions for the array. The declarator can be an integer, integer constant, or dummy expression. For a multiply dimensioned array, include an expression for each dimension. For a lower array bound other than 1, you must specify both upper and lower bounds; e.g., REAL XX(2:9). Declarators are further described under DIMENSION and in Chapter 2.

clist is a constant list that includes one or more values for one or more v entities. The entity(ies) will be initialized to the clist value(s) when the program is built. A clist value can be a character constant or a Hollerith constant. If you initialize an array via clist, the clist must include precisely enough values to fill the entire array. DG's F77 preserves the current values of all entities initialized via *clist* when it executes a subprogram's RETURN or END statement.

What It Does

The REAL statement declares one or more real entities with the names, lengths, and (for arrays) the numbers of elements given. If you omit both a REAL and IMPLICIT statement, the name rule implies type real for entities whose names begin with letters A through H and O through Z.

A real number can include an exponent field and can range from 5.4*10**-79 to 7.2*10**75. A REAL*4 number has significance to about 6.7 decimal digits; a REAL*8 (same as double-precision) number has significance to about 16.4 decimal digits.

As a type statement, REAL overrides both the name rule and an IMPLICIT statement. It is a nonexecutable statement; if you label it, no other statement can refer to the label. All type statements are specification statements and must precede statement function definitions and executable statements in a program unit.

You can establish the data type of a variable or array, and dimension an array, only once in a program unit.

REAL Examples

```
      REAL  K

      REAL  XX  / 888.77 /

      REAL*8  K1

      REAL  LARRY (200)

      REAL  R4 (20), R5(20)

      REAL  BIGGIE(20,20,20)

      REAL R1 / 'DG' / ! last two
C           bytes are blank (<040>)

      REAL R2 / 2HDG / ! last two
C           bytes are blank (<040>)

      REAL*8 MY_CAR  / '40 Ford' /
C           last byte is blank (<040>)
```

RETURN

Returns control from a subprogram to the calling program unit.

Form

RETURN *[expr]*

where:

expr is an integer expression that indicates an alternate statement in the calling program to receive control. *expr* applies to subroutines only; you cannot specify alternate returns in a function subprogram.

What It Does

The RETURN statement returns control from a subprogram to the calling program unit.

A subprogram can have one, more than one, or no RETURN statement. Execution of an END statement has the same effect as RETURN.

From a function, control returns to the calling unit with the value of the function. From a subroutine, control returns to the statement after the CALL statement in the calling program unit unless you specified an alternate return.

Execution of any RETURN (or END) statement in a subprogram terminates the association between the subprogram's dummy arguments and the current actual arguments. All entities within the subprogram become undefined, except for

- entities saved by SAVE statements;
- entities initialized by DATA or type statement *clists*;
- entities declared in named or blank common.

Alternate RETURN Statement (Subroutines Only)

To specify an alternate return, the calling unit must give, as an actual argument, the label of an executable statement preceded by an asterisk. The corresponding dummy argument in the subroutine must be an asterisk. The *expr* in the subroutine's RETURN statement must be an integer expression that gives the position of the asterisk among other asterisks. For example:

```
C   Calling unit.
    CALL SUB (FOO, *90)
40  xxx
    ...
90  xxx
    ...
    END


C   Subroutine.
    SUBROUTINE SUB (FOO1, *)
    ...
    RETURN   ! Returns to caller at 40.
    ...
    RETURN 1 ! Returns to caller at 90.
    ...
    END
```

If *expr* has a value of less than 1 or greater than the total number of asterisks in the dummy argument list, F77 returns control as for a normal RETURN.

RETURN Examples

```
RETURN

RETURN 1
```

REWIND

Positions an open file before its first record.

Forms

REWIND iu

REWIND (*[UNIT=]* iu *[,IOSTAT=ios] [,ERR=s]*)

where:

UNIT= iu — iu is an expression that evaluates to the integer number of the unit you want to rewind. You can omit *UNIT=* if the unit specifier is the first argument.

IOSTAT= ios — *ios* is an integer variable or array element that returns a status indicator: 0 for normal, an error code if an error occurred. If you include IOSTAT= without ERR=, execution will continue at the next statement on an error.

ERR=s — *s* is a statement label to which control will go on an error. This must be an executable statement within the current program unit. If you omit both IOSTAT= and ERR=, the program will terminate on an error.

What It Does

REWIND is an auxiliary I/O statement that positions an OPEN file before its first record, at the START position.

REWIND works with all files that are connected for sequential access and that can be positioned. This includes disk files but not devices like the terminal.

If the file is already positioned at its start point, REWIND has no effect. If the unit is not connected, F77 signals an error.

REWIND Examples

```
REWIND 3

REWIND (14, IOSTAT=IOS)
```

SAVE

Preserves the values assigned by a subprogram to its variables and arrays.

Form

SAVE *[v [,v] ...]*

where:

v is any variable name, array name, or common block name surrounded by slashes. *v* cannot be a dummy argument name, subprogram or statement function name, or name of an entity *within* a common block. You cannot save a *v* name more than once in a program unit.

What It Does

SAVE can preserve the values of all subprogram variables and arrays when the subroutine returns.

Values saved are accessible only to the subprogram that saved them, unless these values are in common (named or blank).

If you include arguments, F77 preserves the values of the entity names given. If you omit arguments, it preserves the values of *all* entities defined by the subprogram. You can use more than one SAVE statement in a subprogram.

One or more SAVE statements in the main program saves main program entities in static storage instead of in the F77 runtime stack.

Instead of the SAVE statement, you can use the compiler switch /SAVEVARS; for example, F77/SAVEVARS MYPROG. This switch has exactly the same effect as a global SAVE statement in each of the program units in the compilation line.

Whether or not you use SAVE or the /SAVEVARS switch, DG's F77 preserves the values of the following items on return from any subprogram:

- entities saved by SAVE statements;
- entities initialized by DATA or type statement clists;
- entities declared in named or blank common.

SAVE Examples

```
SAVE

SAVE  IARRAY, J
```

STOP

Terminates program execution.

Form

STOP *[n]*

where:

n is a string of digits or a character constant.

What It Does

STOP unconditionally terminates execution of the program. F77 writes STOP and the text message *n* (if any) on the appropriate terminal or file before stopping the program.

If the program was executed in interactive mode (from a terminal), F77 displays the STOP message on its original terminal. If you executed the program with the CLI /S switch (XEQ/S MYPROG), the message is returned to the CLI variable STRING.

If the program was executed in batch mode (QBATCH command), F77 writes the message into the batch output listing.

STOP Examples

```
STOP 'LABEL 70'

STOP "Error on write to output file"

IF (IOS .NE. 0)  STOP "Write error."
```

SUBROUTINE

Begins a subroutine subprogram.

Form

SUBROUTINE sub *[([d [,d] ...])]*

where:

sub is the symbolic name of the subroutine. A subroutine has no data type; thus, the name rule doesn't apply. A valid symbolic name is 1 to 32 characters and includes letters, numbers, question marks, and underscores; the first character must be a letter or the question mark. MP/AOS and MP/AOS-SU names follow the same rules except that they are unique only up to the first ten characters.

d is a dummy variable name, dummy array name, or dummy procedure name. *d* can be an asterisk (*) if the subroutine will use alternate returns.

What It Does

The SUBROUTINE statement starts and names a subroutine; it must be the first statement of a subroutine. It is a nonexecutable statement and the compiler doesn't need a label on it.

On the subroutine CALL in the calling program unit, F77 transfers control to the subroutine, using any actual arguments for the subroutine's dummy arguments. The subroutine executes its statements, which may or may not modify the arguments. On return, control returns to the statement following the CALL statement unless the subprogram specifies an alternate return.

Each dummy argument may be a variable name, an array name, a dummy subprogram name, or an asterisk to receive an alternate return specifier. A dummy argument can hold the value of an expression, an array name, an external subprogram name, or dummy procedure name. The dummy argument names (if any) in a SUBROUTINE statement must correspond in order, number, and type with the actual arguments of the subroutine CALL in the calling program unit. But you can use any type of dummy argument when the actual argument is a subprogram name.

The dummy argument names used must not appear in a DATA, EQUIVALENCE, PARAMETER, INTRINSIC, SAVE, or COMMON statement within the subroutine, except as a common block name.

SUBROUTINE (continued)

A subprogram cannot use an assumed-size array (described next) in an iolist.

A subroutine can make reference to or CALL other subprograms or itself (recursion).

Actual-Dummy Argument Summary

Generally, you can use any expression as an actual argument if the corresponding dummy argument is the appropriate data type. Other rules sketched in this section are detailed in Chapter 7, in the section "Arguments to Function and Subroutine Subprograms."

Variables

To pass a variable, use its name as an actual argument in the calling unit and include a dummy argument of the same data type in the subprogram's dummy argument list. To pass an array element, use the *array name and element subscript* as an actual argument.

Arrays

To pass an array to a subroutine, use its name as an actual argument and use a dummy argument of the same data type in the subprogram. The subprogram must dimension the array. It can do so in one of three ways:

1. Via one or more integer constant expressions; if so, the total number of elements must not exceed the total number declared originally in the calling unit.
2. Via one or more integer expressions containing variables that are members of a common block or one dummy argument. An array whose size is specified this way is called an "adjustable array."
3. Via an asterisk assumed-size array declarator. For this, the caller specifies the array name as an actual argument, and the subprogram specifies a dummy argument of the proper data type to receive the name. Then the subprogram uses an asterisk as the high bound of the last dimension expression. For example:

```
C   Calling unit.
    DIMENSION  ARY(200)
    PRINT *,  MYFUNC(ARY)
    ...
    END


C   Subprogram.
    SUBROUTINE  MYSUB(ARY2)
    DIMENSION  ARY2(*)
    ...
    END
```

An assumed-size declarator can specify only the *last* dimension of a multiply dimensioned array. The subprogram cannot use an assumed-size array in an iolist.

Character Values

The data types of both actual and dummy arguments must be character. The subprogram cannot have access to more characters than were declared for the entity in the calling unit. If the dummy argument is an array, the subprogram can declare an element length that differs from the caller's. However, the subprogram cannot have access to a character beyond the last character reserved by the caller for the array.

For either a character variable or array, you can use an asterisk length specifier when you have the subprogram declare the character entity. An asterisk specifier directs F77 to use the length declared by the calling unit. For example:

```
C   Calling unit.
    PROGRAM  MAIN
    CHARACTER*10  CF
    ...
    CALL  SUB1(CF)
    ...

C   Subroutine.
    SUBROUTINE  SUB1(CF2)
    CHARACTER*(*)  CF2
C   When CALLed by program unit MAIN,
C   CF2 is 10 characters long.
    ...
```

As an extension, DG's F77 allows you to concatenate strings passed via asterisk specifiers.

Subprogram Arguments

You can use certain intrinsic functions as actual arguments if you declare them in an INTRINSIC statement; these are described under INTRINSIC.

You can use subprogram names as actual arguments if you declare each subprogram name to be passed as EXTERNAL in the calling program. This lets F77 know that the name is a subprogram — not a variable — name.

Alternate Return Specifiers

These are described under the RETURN statement.

SUBROUTINE Examples

```
SUBROUTINE  NAME9

SUBROUTINE  ELECT(X, YARD, *, J)
```

The following example uses a subroutine that reverses N elements of a 200-element array.

```
C   Calling program.
    PROGRAM  TESTIT
    DIMENSION  ARY(200)
    PRINT *,  'Type number of elements and'
    PRINT *,  '  values for them.'
    READ (*,*)  NUM,  (ARY(I), I = 1, NUM)
    CALL  REVERS(ARY, NUM)
    PRINT *,  'Here it is: ', (ARY(I), I = 1, NUM)
    END


C   Subroutine.
    SUBROUTINE  REVERS (ARY2, NUM2)
    DIMENSION  ARY2(NUM2) ! Adjustable
                          !   declarator.
    MIDDLE = NUM2 / 2     ! Get middle element.
    DO 90  I = 1, MIDDLE  ! Reverse.

C       Use a temporary variable to reverse
C          array elements.

        TEMP = ARY2(I)
        ARY2(I) = ARY2 ( NUM2 + 1 - I )
        ARY2 ( NUM2 + 1 - I ) = TEMP
90  CONTINUE
    RETURN
    END
```

WRITE

Transfers the values of iolist entities to a file.

Form

WRITE (cilist) *[iolist]*

where:

cilist is a control information list. It must include a unit specifier and, for formatted records, a format identifier/specifier. All other specifiers are optional. All cilist specifiers, detailed later, are

[UNIT=] iu
[FMT=] f
ERR = s
IOSTAT = ios
REC = irn
RETURNRECL = iret

iolist is an I/O list of entities whose values will be written to the unit. It can include anything allowed on the right side of an = sign in an assignment statement, plus implied DO lists. Examples are 73, K+5, and 'ABC'.

What It Does

WRITE transfers values from the iolist entities to an external or internal file. WRITE is the primary way to place values in records (although PRINT can transfer values to unit 6).

Unless you specified unformatted records and/or direct access on an OPEN statement, F77 expects a formatted sequential access WRITE.

For a formatted WRITE statement, include *[FMT=]* f; the values in each iolist entity must match the format used. The first character in each record may be used for carriage control as described under OPEN, CARRIAGECONTROL= property. F77 transfers all iolist values to the unit as one record, unless it encounters a slash in the format specification or format reversion occurs; these conditions are described in Chapter 6, in sections "Multiple Record Formatting" and "List-Directed Formatting."

For an unformatted WRITE statement, omit *[FMT=]* f; F77 will write values from the iolist entities as they are stored. In an unformatted WRITE, F77 always attempts to transfer iolist values to the unit as one record.

WRITE (continued)

Each WRITE to an internal file is sequential, and the statement must include an *edit-directed* format identifier. Unformatted WRITE statements or WRITE statements that specify list-directed formatting (FMT=*) are not allowed to internal files.

For any WRITE, if the iolist includes an unsubscripted array name, F77 will write until it has written values from all array elements in the traditional column/row order.

On a sequential WRITE, the end of file is automatically defined after the record just written. If the file you are writing is open on another unit or by another process, the results of the WRITE are undefined and the system may return an error from it. Either error condition can occur on the first write to the file or on a write after a read or file position operation. Thus, if other processes might open a file you will want to write, you may want to open it for EXCLUSIVE access (EXCLUSIVE='YES' on OPEN).

To end the file, you can either stop writing records to the unit or, if the file is connected for sequential access, write an end of file with ENDFILE. To append to the file, you can either open the file with POSITION='END', or you can read until you encounter the end of file (using READ's END=). Then you can overwrite the last record via a BACKSPACE and WRITE sequence.

After a WRITE executes successfully, F77 has written at least one record that consists of values from all iolist entities. The file is positioned at the beginning of the next record.

If an error occurs, file position is undefined.

Fixed-Length Records

Internal files, files you write via direct access, and files opened with RECFM='FIXED' all have fixed-length records. When you write to a fixed-length record, and the iolist values require *fewer* bytes than the record length, F77 pads the values to the record length. The pad character used is blank for a formatted record or null for an unformatted record. If the iolist entities require *more* bytes than exist in the record, F77 signals an error and the record is undefined.

When you write to an internal file, the record length is the number of bytes declared for the character variable or substring or each array element that is the internal file.

WRITE cilist Specifiers

[UNIT=] iu
iu specifies the unit. For an external file, iu must be an integer expression that evaluates to a number from 0 through 255 for AOS/VS or from 0 through 63 for AOS, F7716, MP/AOS, and MP/AOS-SU (but you can use * to specify unit 6). For an internal file, iu is the name of the character entity that is the internal file. You can omit UNIT= if the unit specifier is the first argument in the cilist.

[FMT =] f
f is the format identifier. f can be the label of a FORMAT statement, an integer variable assigned the label of a FORMAT statement, a character array name, a character expression (entity name or constant like '(1X,I8)'), or an asterisk to specify list-directed formatting. Character entities and expressions as formats are described near the beginning of Chapter 6; edit descriptors you can use in them are sketched under FORMAT. List-directed formatting is detailed near the end of Chapter 6.

If you omitted UNIT= from the unit specifier, and the format identifier will be the second argument in the cilist, you can omit FMT=. For example, the following statements are equivalent:

```
WRITE (UNIT=15, FMT = '(1X,I8)' )   J, K
WRITE (     15,       '(1X,I8)' )   J, K
```

ERR= s
s is the label of an executable statement that will receive control if an error occurs. Statement *s* must be in this program unit. When *s* gets control, IOSTAT's variable will contain the error code.

IOSTAT= ios
ios will return a status indicator. *ios* is an integer entity that will return 0 if the WRITE statement executed normally. *ios* will return an F77 or system error code if an error occurred.

If you include IOSTAT= and omit ERR=, program execution will continue at the next executable statement on an error.

REC = irn
irn gives the record number, for direct access only. *irn* is a positive integer expression that specifies the number of the record you want for a direct-access WRITE. The unit must have been opened with ACCESS='DIRECT'. If you specify a nonexistent record number, the iolist entities will be undefined. For an internal file, REC= is not allowed.

RETURNRECL= iret
iret is an integer entity that will receive the byte length of the record written.

WRITE Examples

Some examples of WRITE statements alone are

```
C   List-directed WRITEs:

    WRITE  (*, *)  RVAR
    WRITE  (10, *)  'Value ', RVAR
    WRITE  (*, *)  ' Value of CXX is: ', CXX
    WRITE  (1, *, IOSTAT=IOS)  X,Y,Z

C   Edit directed WRITEs:

    WRITE (4, '(1X, 3F11.2)', ERR=99)   B, B1, B2

    WRITE  (3, 80, IOSTAT=IOS)   BA, BB, BC
80  FORMAT (1X, 3F11.2)

    WRITE  (*,90)   J,K,L
90  FORMAT (1X, 3I8)

    CHARACTER*10 CX /'(1X, 3 I8)'/   Char. entity.
    ...
    WRITE  (8, CX)   J, K, L

    WRITE (1,95) (IRY(I), I=80,1,-1) ! Implied DO.
95  FORMAT (I10)

C   Direct access WRITE:
    WRITE  (1, 25, IOSTAT=IOS,REC=J)   X,Y
25  FORMAT (1X, 2F10.2)

C   Unformatted WRITE:
    WRITE  (2, IOSTAT=IOS)    X,Y,Z
```

The following example writes formatted records to a file connected for sequential access.

```
    ...
    dimension ARRAY(100)
    ...
C   Here, program assigns real values to ARRAY.
    ...
    open (3,iostat=IOS,file='PFIL')

    write (3, '(F14.2)', iostat=IOS)   ARRAY

    close (3)
    ...
```

There are other WRITE examples under the OPEN statement.

End of Chapter

Chapter 9
Building F77 Programs

This chapter summarizes the information you need to write, compile, and link F77 program units into an executable application program.

The major sections proceed:

- F77 Compiler and Library Files
- Program Development Overview
- Using the LISTFILE and DATAFILE Commands
- Compiler Switches
- Compiler Error Handling and Error Messages
- Link Switches

NOTE: If you are using F7716, then replace all occurrences of F77.CLI, F77LINK.CLI, and SWAT in this chapter by F7716.CLI, F77LINK16.CLI and SWAT16, respectively. See the last section of this chapter for more details.

- Runtime Errors
- F7716 Examples

F77 Compiler and Library Files

The term "FORTRAN 77 software" includes many Data-General-supplied files. For example, the AOS F77 compiler includes files F77.PR, F77.OL, and F77PASS2.PR. The AOS runtime library includes files F77IO.LB and F77MATH1.LB. See your installation's latest F77 Release Notice for the names and content summaries of all the software. Generally, your system manager places the FORTRAN 77 software and Release Notice in directory :UTIL:F77.

The compiler files are used for compiling source programs; the resulting object (.OB) files and the library files are used as input to Link (or Bind under MP/AOS and MP/AOS-SU) as it creates a program (.PR) file. But you generally do not refer to the files directly. Instead, you use system Command Line Interpreter (CLI) macros— also supplied with F77 — to compile and link programs. The compilation macro is named F77.CLI; the Link (or Bind) macro is named F77LINK.CLI.

The F77 *error parameter* file, which you can include in your source programs to process runtime errors, is ERR.F77.IN.

LINK.PR and BIND.PR

Macro F77LINK.CLI comes with all versions of F77. It invokes program LINK.PR on AOS/VS and AOS systems, and program BIND.PR on MP/AOS and MP/AOS-SU systems. Both programs perform the same function of creating an executable (.PR) file from object (.OB) and library (.LB) files. We use "Link" in this chapter as an inclusive term to mean program LINK.PR under AOS/VS and AOS, or else program BIND.PR under MP/AOS and MP/AOS-SU.

Program Development Overview

Here is a quick sketch of the way you use the system to write and build F77 programs.

1. Log on the operating system with your password; the system CLI prompt,), appears on your terminal. The CLI allows you to execute other programs.
2. Use an interactive text editor — SED or SPEED or SLATE — to type in one or more source programs. You can put more than one program unit (for example, the main program and subprograms it will use) in one source file. Or you can put each program unit in its own source file. A source file can have any valid system filename. By convention, F77 source filenames end with .F77 but this is not required.
3. Compile your source program(s) with the CLI macro command:

   F77*[switch]*... filename ↵

   Switches modify compiler action and are described after the next section. Compiler error messages that you might receive are described after the switch section. Correct the compilation errors (if any) using the error message text as a guide. Compilation produces an object file (filename.OB) from each source program file. If an object file with the same name already exists in the working directory, the compiler deletes this file before creating the new object file.

4. Link the desired object files into an executable program file with the CLI macro command:

   F77LINK/*switch*/... filename ... ⤸

   Link produces *one* executable program file named filename.PR. If a file named filename.PR already exists in the working directory, Link deletes the old file before creating the new one. The main program filename must precede all other filenames in the F77LINK command line; the filenames of pertinent subprograms and libraries (if any) must follow it. For example,

   F77LINK/DEBUG MYPROG SUB_1 SUB_2 & MY_LIB.LB ⤸

   creates fresh program file MYPROG.PR from MYPROG.OB, SUB_1.OB, SUB_2.OB, and any referenced subprograms in library MY_LIB.LB.

5. Execute the program file:

   X filename ⤸

   If the program runs properly, you're done; if not, diagnose its problem(s) with runtime results, runtime error messages, and/or the SWAT™ high-level debugger (SWAT filename ⤸). Fix the program source(s) and run through the compile and Link steps again.

MP/AOS and MP/AOS-SU do not have the SWAT debugger. And, even though their equivalent of Link is Bind, you still use the macros F77 and F77LINK under MP/AOS and MP/AOS-SU.

If you include the suffix .F77 in a source filename, you can either omit or include .F77 in the compile line. The compiler looks for the given filename plus .F77; if not found, it looks for the given filename. The Link utility looks for the given filename plus .OB; if not found, it looks for the given filename. The dialog on the terminal might go like this during development of a hypothetical program called MPROG:

```
) CREATE/I MPROG ⤸
))      PROGRAM MPROG ⤸
        ...
))      END ⤸
))) ⤸
) F77 MPROG ⤸                    Compile source.
  ...                            Messages.
) F77LINK MPROG ⤸                Link source.
) X MPROG ⤸                      Execute program.
  ...                            Program and
  ...                            runtime messages.
)
```

The CLI prompt,), returns to the terminal after each step.

For different kinds of suffixes on MPROG, the object and program filenames would be

Source Filename	Object Filename	Program Filename
MPROG.F77	MPROG.OB	MPROG.PR
MPROG	MPROG.OB	MPROG.PR
MPROG.F77.F77	MPROG.F77.OB	MPROG.F77.PR

Using the LISTFILE and DATAFILE Commands

This section does not apply to MP/AOS and MP/AOS-SU.

In addition to the @INPUT and @OUTPUT preconnections, F77 has preconnections to unit 9 (@DATA) and unit 12 (@LIST). You can set @DATA and @LIST to any file you want via the CLI commands DATAFILE and LISTFILE. Both preconnections open the associated files for both input and output, with data-sensitive records, with padding. @DATA is opened with LIST carriage control; @LIST is opened with FORTRAN carriage control. Traditionally, but not necessarily, unit 9 connects to an input file and unit 12 connects to an output file.

Having selected filenames via DATAFILE and LISTFILE, you can use these files for many different things: compiler listing output, input or output files, and so on.

For example, you can create two fresh files and set DATAFILE and LISTFILE to their names (MYPROG.DATA and MYPROG.OUT) with the following commands:

```
) CREATE/2=IGNORE/DATASENS MYPROG.DATA ↵
) DATAFILE MYPROG.DATA ↵
) CREATE/2=IGNORE/DATASENS MYPROG.OUT ↵
) LISTFILE MYPROG.OUT ↵
)
```

Then run the program that uses the preconnections. Of course it need not open the preconnected units:

```
C     F77 Program.
      ...
      READ  (*, *)      B4, B5, B6
C     Place B4, B5, B6 in MYPROG.DATA
      WRITE (9, 200)    B4, B5, B6  ! List c.c.
200   FORMAT  (3F9.2)
C     Place B4, B5, B6 in MYPROG.OUT
      WRITE (12, 300)   B4, B5, B6  ! Fortran c.c.
300   FORMAT  (1X, 3F10.2)
      ...
```

If a program uses preconnected units 9 or 12 without opening them, you must select a filename with DATAFILE or LISTFILE before running the program. If not, F77 will report a runtime *File does not exist* error.

Compiler Switches

A *switch*, applied to the compiler macro name, modifies compiler action; e.g., to specify short (2-byte) integers or a listing file on disk. Apply one or more compiler switches via the form shown in step 3 in the earlier section "Program Development Overview."

F77*[switch]*... filename

where *switch* is one of the following switches (described alphabetically). The CLI allows abbreviations of the switch names, which means that you need type only the shortest unique abbreviation of the switch name — a great timesaver. For example, either of the following commands generates an assembly language listing in the listing file.

```
F77/CODE/L=MYPROG.LS MYPROG
F77/CO/L=MYPROG.LS MYPROG
```

The command

F77/C/L=MYPROG.LS MYPROG

is illegal because "C" is not unique — F77 has no idea of whether you mean "CODE" or "CARDFORMAT".

Table 9-1 contains F77 switches and the versions of F77 that apply to the switches. The descriptions of the switches follow the table.

Table 9-1. F77 Switches and Their Versions of F77

F77 Switch Name \ Version of F77	AOS/VS	AOS	F7716	MP/AOS	MP/AOS-SU
/CARDFORMAT	√	√	√	√	√
/CCOMPILE	√	√	√	√	√
/CODE	√	√	√	√	√
/DEBUG	√	√	√		
/DOTRIP	√	√	√	√	√
/E	√	√	√	√	√
/ERRORCOUNT	√	√	√	√	√
/HOLLERITH	√	√	√	√	√
/INTEGER	√	√	√	√	√
/L	√	√	√	√	√
/LINEID	√				
/LOGICAL	√	√	√	√	√
/N	√	√	√	√	√
/NOFNS		√	√	√	√
/NOLEF		√	√	√	√
/NOMAP	√				
/NOWARNINGS	√	√	√	√	√
/O	√	√	√	√	√
/OPTIMIZE	√	√	√	√	√
/PROCID	√				
/SAVEVARS	√	√	√	√	√
/STATISTICS	√	√	√	√	√
/STRINGS	√	√	√	√	√
/SUB	√	√	√	√	√
/TMPDIR	√				
/XREF	√				

/CARDFORMAT
By default, the compiler scans an entire line of text. But if you include this switch, it will read and interpret only up to the first 72 characters, allowing you to have any text you want (like sequence numbers) in columns 73–80. If there are less than 72 characters, the compiler will pad the record to 72 characters (this is significant for character strings that are continued to the next line).

To insert the text that you want the compiler to ignore in columns 73–80, be sure to type spaces (not tab over) to column 73 before starting the number. Tabs will not produce the desired results. Each character, including a tab, counts as one character as you move to column 72.

/CCOMPILE*[=letters]*
This switch allows conditional compilation according to the specified letters and the contents of column 1 of the source program file.

By default a "C", "c", "*", or "!" in column 1 indicates a comment line; any other nondigit character results in a compiler error. If you want to compile statements with a letter different from "C" or "c" in column 1, then use the /CCOMPILE switch and specify the letter.

For example, suppose you want to insert extra PRINT statements as you develop program PROG_140.F77, yet you don't want the statements to execute after you've tested and debugged the program. You want the statements to remain in PROG_140.F77 for future testing and debugging. One way to have them remain inactively is to place an "X" in column 1 of the extra PRINT statements. Then, give the following CLI commands (without the COMMENT statements).

```
) COMMENT COMPILE STATEMENTS WITH ↓
) COMMENT    AN "X" IN COLUMN 1. ↓
) F77/CCOMPILE=X PROG_140 ↓
) F77LINK PROG_140 ↓
) XEQ PROG_140 ↓
) COMMENT OUTPUT FROM EXTRA PRINT * ↓
) COMMENT STATEMENTS GOES HERE. ↓
) COMMENT CHANGE PROG_140.F77 AS NEEDED, ↓
) COMMENT    BUT DON'T CHANGE ↓
) COMMENT    COLUMN 1 = "X" STATEMENTS. ↓
) COMMENT RECOMPILE, BUT NOT THE ↓
) COMMENT COLUMN 1 = "X" STATEMENTS. ↓
) F77/CCOMPILE PROG_140 ↓
) F77LINK PROG_140 ↓
) XEQ PROG_140 ↓
) COMMENT NO OUTPUT FROM EXTRA PRINT * ↓
) COMMENT STATEMENTS OCCURS. ↓
```

If you give the command

```
F77 PROG_140
```

without the /CCOMPILE switch, the compiler reports an error for the statements with an "X" in column 1.

The following pseudo code describes the compiler's decision whether to compile a statement and its actions.

```
If "C" or "c" or "*" or "!" is in column 1 then
    it is a comment line
else if there is a letter in column 1 then
    if /CCOMPILE switch is supplied then
        if letter is in /CCOMPILE=<letters> then
            compile the statement
        else
            the statement is a comment
        end if
    else
        error: illegal character in column 1
    end if
end if
```

In other words, you can place any letter different from "C" and "c" in column 1 and use the /CCOMPILE switch. If this switch specifies letters, the letter in column 1 must be among the letters following /CCOMPILE= (the statement is compiled) or else the statement becomes a comment. A non-"C" and non-"c" letter in column 1 without using the /CCOMPILE switch results in an error.

There is still no way to compile lines that begin with "C" or "c" in column 1. The compiler converts any lowercase letters in column 1 or in /CCOMPILE=s letters to uppercase before it compares the letters. Thus /CCOMPILE=ABDXY, /CCOMPILE=abdxy, and /CCOMPILE=AbDxY have the same effect; "E" and "e" in column 1 produce identical results (here, to specify comment statements).

For another example, suppose some statements in MYPROG_150.F77 begin with X, Y, or Q in column 1 and you only want the ones beginning with X or Y to compile. Just give the command

```
) F77/CCOMPILE=XY MYPROG_150 ↓
```

The statements beginning with X or Y are compiled; the statements beginning with Q are not compiled — they become comments.

Experienced DG FORTRAN IV and FORTRAN 5 programmers can interpret this explanation of the /CCOMPILE switch in the following two statements. "Placing an X in column 1 and using the /X compiler switch as before is equivalent to placing an X in column 1 and using the /CCOMPILE=X F77 switch now. Placing an X in column 1 and not using the /X compiler switch

resulted in a comment statement; placing an X in column 1 and not using the /CCOMPILE F77 switch results in an error now."

/CODE
This switch instructs the compiler to generate an assembly language listing of the program and write it to the file you specify with /L. This switch has no effect unless you specify a listing file with /L.

/DEBUG
This switch instructs the compiler to generate symbols and code for the high-level debugger (SWAT) and place these symbols in the object file. You must also include this switch in the Link macro command line.

See the explanation of the /OPTIMIZE switch for further information about the /DEBUG switch.

MP/AOS and MP/AOS-SU do not support the SWAT debugger.

/DOTRIP = 1 or 0
By default, the compiler will set up code so that DO loops that need not execute will not execute. It always provides for these 0-trip DO loops unless you specify /DOTRIP=1. If you need to have each DO loop in the pertinent program(s) execute at least once, insert /DOTRIP=1. Some programs — written according to the 1966 FORTRAN standard — expect the results produced by /DOTRIP=1. If /DOTRIP=1, the program must not modify the incrementation or terminal parameter in the range of any DO loop.

/E=name
By default, the compiler writes error messages to file @OUTPUT and to the listing file, if any. File @OUTPUT, for interactive use, is the terminal screen or printer. If you include the /E= switch, the compiler will write error messages to the file specified and to any listing file — and *not* to @OUTPUT. If the error file does not exist, the compiler will create it; if it does exist, the compiler will append to it. For example,

```
) F77/E=MYPROG.ERRS MYPROG ↵
```

/ERRORCOUNT=n
The optional /ERRORCOUNT=n compiler switch causes the compiler to report and count both warning and error messages; when this count exceeds the value of n, the compiler terminates. n is a positive integer between 1 and 32767.

If AOS/VS F77 receives a value of n larger than 32767, it displays a warning message and changes n to 32767.

If AOS, F7716, MP/AOS, or MP/AOS-SU F77 receives a value of n larger than 32767, it changes n to 32767 without displaying a warning message.

The default value of this switch is /ERRORCOUNT=100.

/HOLLERITH=ANSI or NON_DG or OLD_DG
This optional switch affects the storage of character and Hollerith constants both in expressions and as arguments to subprograms. The possible values for the /HOLLERITH switch are

- ANSI
- NON_DG
- OLD_DG

ANSI is the default value. That is, F77 MY_PROGRAM and F77/HOLLERITH=ANSI MY_PROGRAM give identical results.

Table 9-2 summarizes the effect of the /HOLLERITH switch settings.

Table 9-2. The Values of the /HOLLERITH Switch

<table>
<tr><th>/HOLLERITH= Switch Setting
Effect on Character and Hollerith Constants Used in</th><th>None or ANSI</th><th>NON_DG</th><th>OLD_DG</th></tr>
<tr><td>Expressions</td><td colspan="2">Padding with blanks to an appropriate length (*)</td><td>Padding with nulls to an appropriate length (*)</td></tr>
<tr><td>DATA Statements (non-CHARACTER type)</td><td colspan="3">Padding with blanks to an appropriate length (*)</td></tr>
<tr><td rowspan="2">Arguments to Subprograms</td><td>Character constants: addition of one null(**); corresponding dummy argument must be CHARACTER type (byte pointer is passed)</td><td>Character constants: padding with blanks to 16 bytes; corresponding dummy argument must be non-CHARACTER type (word pointer is passed)</td><td>Character constants: padding with nulls to 16 bytes (***); corresponding dummy argument must be non-CHARACTER type (word pointer is passed)</td></tr>
<tr><td colspan="2">Hollerith constants: padding with blanks to 16 bytes; corresponding dummy argument must be non-CHARACTER type (word pointer is passed)</td><td>Hollerith constants: padding with nulls to 16 bytes (***); corresponding dummy argument must be non-CHARACTER type (word pointer is passed)</td></tr>
</table>

* The "appropriate length" depends on the size of related operands or variables. For example,

```
REAL*8 MY_NAME
MY_NAME = 'TOM'
```

results in <124><117><115><040><040><040><040><040> being stored in variable MY_NAME (as long as the /HOLLERITH=OLD_DG switch isn't used).

** The compiler stores a null byte at the end of the character constant (for possible use as input to a system call).

*** The compiler stores a null byte at the end of the constant (for possible use as input to a system call), but this addition doesn't change the length of the constant.

NOTE: The phrases "padding with blanks to 16 bytes" and "padding with nulls to 16 bytes" mean the padding occurs only when the constant is less than 16 bytes.

For example, consider this program segment.

```
      ...
   10 READ (9, ...) ..., NAME, ...
C     Select McDonald, Morgan, Mraz, etc.
      IF ( NAME(1) .EQ. 'M' ) GO TO 20
C     Process non-M's
      ...
      GO TO 10   ! Get the next record
C
   20 CONTINUE   ! by processing the M's
      ...
```

Assume that NAME is an INTEGER array and that you don't use the /HOLLERITH=OLD_DG compiler switch. Then, the complete program compares NAME(1) to 'M□□□' if NAME is INTEGER*4 or else to 'M□' if NAME is INTEGER*2. Note that if the ASCII character 'L' is read into NAME(1) under the edit descriptor A1, then NAME(1) will contain either 'L□□□' = <114><040><040><040> (NAME is INTEGER*4) or 'L□' = <114><040> (NAME is INTEGER*2).

You may replace the IF statement of this example by

```
IF ( NAME(1) .EQ. 1HM ) GO TO 20
```

and the results — compiler and runtime — will be identical.

Whenever the compiler extends a character constant or a Hollerith constant, it issues a warning message. This message tells you that the compiler has added pad characters that you didn't explicitly specify. If, in this example, NAME were INTEGER*4, then the compiler would generate <115><040><040><040> from either 'M' or 1HM. Then it would report a warning for the IF statement.

The default pad character is a blank (<040>). You may instead select a null (<000>) as the pad character with the /HOLLERITH=OLD_DG compiler switch.

/INTEGER=2 or 4
The standard F77 integer length is 4 bytes (32 bits). If, on an AOS/VS system, you insert /INTEGER=2, the default integer length *for this compilation* will be 2 bytes.

On 16-bit systems, 4-byte integers take longer to assign and manipulate than 2-byte integers. Therefore the F77 compiler macro supplied with AOS, F7716, MP/AOS, and MP/AOS-SU F77 contains an /INTEGER=2 switch. This makes 2-byte integers the default on all programs compiled using the supplied macro. However, you can use /INTEGER=4 with the supplied macro if desired.

For either system, any type declaration within the program unit(s) overrides the /INTEGER= switch. Regardless of the switch, if a program unit says INTEGER*2 IFOO, F77 will give IFOO 2 bytes of storage. If a program says INTEGER*4 IFOO, F77 will give IFOO 4 bytes of storage.

To conform to standard, integers must have the same length as real and logical entities. This length is 4 bytes.

Be careful with the /INTEGER (and /LOGICAL) switches: arguments passed to subprograms must agree in type. Integer (and logical) entities of different lengths *are different types*. For example, a 2-byte integer in a calling program unit will *not* agree with a 4-byte integer dummy argument in a subprogram.

/L[=*name*]
By default, the compiler generates no listing file. If you include this switch, it will send a listing, with line-numbered statements, to the file you specify. The source file(s) can control output to this file with the directives %LIST(OFF) and %LIST(ON). The listing will include a storage map for all entity names and error messages. To the listing, you can add assembly language code (/CODE), compilation statistics (/STATISTICS), and a cross-reference (/XREF).

If you omit =*name*, the listing will be appended to your current LIST file, as set with the CLI command LISTFILE. If you include *name* and the filename specified doesn't exist, the compiler will create it; if the file does exist, the compiler will append to it. For example:

```
) F77/L=MYPROG.LIST/XREF MYPROG ↓
```

/LINEID
For AOS/VS only, this switch tells the compiler to insert line-number information in the object file. If a runtime error occurs, F77 will report the number of the line that caused the error. This number will correspond to the listing file line number within the program unit. For AOS/VS, the /LINEID switch automatically sets the /PROCID switch to identify each program unit within the compile line. The line number reporting feature, used in conjunction with a listing file, can ease debugging. For a multiple unit example, the command line might be

```
) F77/LINEID/L=MPROG.LS  MPROG SUBPROG ↓
```

/LOGICAL=2 or 4
As with integers, 4-byte logical entities are the default for AOS/VS. But you *may* want to specify 2 bytes for large logical arrays. You can do this in type statements (e.g., LOGICAL*2 LARRAY(1000)) or, globally, with this switch.

For AOS, F7716, MP/AOS, and MP/AOS-SU, the F77 compiler macro supplied with the systems contains a /LOGICAL=2 switch. This produces 2-byte logical entities in all programs compiled using the supplied macro. However, you can specify /LOGICAL=4 to the supplied macro if desired. For all systems, a type declaration in a program overrides this switch if there is a conflict.

As described under /INTEGER, entities of different lengths have differing data types — a fact to bear in mind before you use this switch.

Although 1-byte logical entities exist, the /LOGICAL=1 F77 switch does *not* exist. The only way to declare a logical*1/byte entity is with the LOGICAL*1 or BYTE statement. Only AOS/VS F77 has the logical*1/byte data type.

/N

By default, if the program does not contain serious errors, the compiler produces an object file named filename.OB. If you include this switch, the compiler will scan the source(s) but not produce an .OB file. The /N switch can save time for first compilations, which may contain syntax or other errors and thus produce a useless object file.

/NOFNS

For AOS, F7716, MP/AOS, and MP/AOS-SU, this switch suppresses the generation of FNS (floating no-operation) instructions following FFAS instructions.

FFAS instructions in some machines don't work properly with the integer*2 number -32768 (high order bit on, other bits off). To correct this, the compiler by default generates an FNS instruction after each FFAS instruction.

Not specifying /NOFNS always results in a program that correctly handles the integer*2 number -32768. However, if you're sure that your machine's FFAS instruction handles this number correctly or that your program never encounters this number, then specify /NOFNS. The result is a program without any FFAS/FNS instruction pairs that executes faster. If you specify /NOFNS and your machine's FFAS instruction doesn't handle this number correctly, the results are indeterminate.

/NOLEF

For AOS, F7716, MP/AOS, and MP/AOS-SU, this instructs the compiler to avoid the use of LEF instructions in the code generated for each program unit compiled. Use this switch if you intend to disable LEF mode from within your program.

Programs compiled by AOS/VS F77 never use LEF instructions.

/NOMAP

For AOS/VS only, if you specify /L, F77 includes a storage map for all program entities. Use this switch under AOS/VS to omit the map from the listing file.

/NOWARNINGS

The optional /NOWARNINGS compiler switch suppresses reporting and counting of warning messages. Use this switch with caution because changing a program in one place may unexpectedly cause problems in another place.

/O=name

By default, the compiler names the object file for the first source file in the command line, with the .OB extension. Use this switch if you want the .OB file to have a different name; for example, to preserve an existing filename.OB. (If an old version of filename.OB exists in the working directory, the compiler deletes it before creating the new filename.OB.) The object file will have the name specified with /O=, and you must specify this name, not the original source filename, to the Link macro. For example:

```
) F77/O=MYPROG1 MYPROG ↵
) F77LINK MYPROG1 ↵
```

These command lines create MYPROG1.OB from MYPROG.F77, and then MYPROG1.PR from MYPROG1.OB.

/OPTIMIZE= *NONE* | *FULL [-FLOAT]* | *DEBUG[-FLOAT]*

For AOS/VS only, this switch controls the level of optimization that the compiler performs on your program. The allowable combinations have the following effects:

1. /OPTIMIZE=NONE — Perform no optimizations. This is the same as not specifying the /OPTIMIZE switch.

2. /OPTIMIZE=FULL — Perform all possible optimizations. One of them is collecting constants so that expressions such as (2.0*3.0-46) are evaluated during compilation instead of at runtime. Other optimizations include trigonometric identities, converting division by a constant into multiplication by its reciprocal, improved addressing code for array subscripts in loops, and removing invariant computations from DO loops.

   Invariant computations are expressions that do not depend on the value of the index variable. In this way, F77 needs to compute each expression only once before the program enters the loop.

   Using /OPTIMIZE=FULL with the /DEBUG switch allows the program to execute under the SWAT debugger. Some features of SWAT may not work at various places or times during the execution of the program. The debugger displays a message whenever a feature does not work.

If you specify /OPTIMIZE without a value and do not specify /DEBUG, F77 assumes you mean /OPTIMIZE=FULL. If you specify /OPTIMIZE without a value and do specify /DEBUG, F77 assumes you mean /OPTIMIZE=DEBUG. /OPTIMIZE=DEBUG is explained below.

3. /OPTIMIZE=FULL-FLOAT — Perform all optimizations except floating-point ones. Consequently, the generated code performs operations precisely as you specified them. For example, RESULT/5.0 would result in division by 5.0 and not in multiplication by its reciprocal, 0.2.
4. /OPTIMIZE=DEBUG — Perform all optimizations that do not interfere in any way with the execution of the program under the SWAT debugger, and perform any floating-point optimizations.
5. /OPTIMIZE=DEBUG-FLOAT — Perform all optimizations that do not interfere in any way with the execution of the program under the SWAT debugger, and do not perform any floating point optimizations.

Earlier versions of AOS/VS F77 included the /OPTIMIZE switch whose optional values were =0, =1, =2, and =3. (AOS, F7716, MP/AOS, and MP/AOS-SU F77 currently use these values.) You may still use these values. Their effects are as follows.

Earlier Switch	Current Effect
None	/OPTIMIZE=NONE
/OPTIMIZE=0	/OPTIMIZE=NONE
/OPTIMIZE=1	/OPTIMIZE=DEBUG
/OPTIMIZE=2	/OPTIMIZE=FULL
/OPTIMIZE=3	/OPTIMIZE=FULL
/OPTIMIZE	/OPTIMIZE=FULL

F77 accepts the values DEBUG, FULL, NONE, and FLOAT with minimal uniqueness.

An example of the AOS/VS F77 /OPTIMIZE switch is

```
F77/OPTIMIZE=FULL-FLOAT MYPROG.F77 ↵
```

/OPTIMIZE*[=1][=2][=3]*
For AOS, F7716, MP/AOS, and MP/AOS-SU, this switch controls the level of optimization. The levels range from 1 (lowest) to 3 (highest). If you omit this switch, the compiler performs no optimizations. If you include the switch without an argument, the compiler executes level-3 (highest) optimization. If you include /DEBUG, we suggest that you omit /OPTIMIZE. If you do include both switches, the CONTINUE AT command will not be available from SWAT. Levels 1, 2, and 3 produce the following optimizations:

1. The compiler eliminates unreachable code, like the code produced for an unlabelled statement after an unconditional GO TO. And it optimizes logical expressions so that only the parts needed to determine the value are executed.
2. Includes level-1 optimization; the compiler also eliminates redundant computations.
3. Includes level-1 and -2 optimizations; the compiler also removes invariant computations from DO loops. Invariant computations are expressions that do not depend on the value of the index variable. In this way, F77 needs to compute each expression only once before the program enters the loop. Note that movement of code out of loops may mean that statements that would not otherwise execute *will* execute. For example:

```
      J1 = 1
      J2 = 2
      A = 0.0
      DO 10 I = J2, J1
      Y = X/A
10    CONTINUE
```

This loop would not normally execute. But at level-3 optimization, invariant expressions are moved out of loops, so the expression X/A, being invariant, would be evaluated prior to execution of the loop. This would cause a division by 0.

/PROCID
For AOS/VS only, this instructs the compiler to insert the names of procedures in the object file so that, at runtime, F77 can report the offending program unit if it encounters an error. A *procedure* is any intrinsic function, statement function, function subprogram, or subroutine subprogram. The /LINEID switch includes the functionality of the /PROCID switch. If you omit both the /PROCID and /LINEID switches, any runtime error messages will include only the message text.

/SAVEVARS
This has the effect of a SAVE statement in each program unit compiled. It preserves the values of all variables and arrays local to each subprogram unit after the unit issues a RETURN statement. Named and blank common blocks, the values passed to dummy arguments, and values assigned with a DATA statement are always preserved whether or not you include this switch.

Another effect of the /SAVEVARS switch is to initialize certain common blocks, local variables, and local arrays to zero (numeric), null (character), and .FALSE. (logical). These entities:

- are not specified in DATA statements
- are specified in the program unit's SAVE statement(s)
- contain their zero/null/.FALSE. values when the program unit — or subprogram — first executes

/STATISTICS
This instructs the compiler to write compiler statistics (number of lines compiled, etc.) to file @OUTPUT. Statistics are also included in the listing file if you specify /L.

/STRINGS=ANSI or DG
By default, the compiler interprets text within angle brackets as control characters, as described in Chapter 2. For example, when it sees '<BEL>' or '<7>', it inserts a character that will sound the terminal tone. But if you say F77/STRINGS=ANSI program-name ⤶, the compiler will accept the angle brackets and characters within them literally. This switch allows you to have angle brackets appear as a literal printed character.

/SUB
This switch instructs the compiler to generate code for subscript and character substring checking. Examples of out-of-range references are ARAY(I) where ARAY was dimensioned with 100 elements and I was assigned the value 101, and CFOO(1:I) where CFOO was declared with 10 characters.

Out-of-range subscript/substring references can produce invalid results. However, subscript/substring checking takes time, which means that you might want to use this switch only during the program development/debugging phase.

TMPDIR=prefix
For AOS/VS only, this switch inserts the specified prefix in front of the temporary filenames that the compiler uses. Normally, F77 creates temporary files in the current directory. Use this switch when you want these files in a directory other than the current directory. Be sure you have write access to the other directory. You must follow a directory prefix with a colon. For example, if you want F77 to place its temporary files in directory :UDD3:TEMP1 while compiling source file MYPROG.F77, give the command

```
F77/TMPDIR=:UDD3:TEMP1: MYPROG ⤶
```

/XREF
For AOS/VS only, this switch instructs the compiler to append an alphabetical cross-reference of all program entities, with line numbers, to the listing file. The /XREF switch has no effect unless you also include /L. The alphabetical cross-reference can help verify variable and array names. However, named constants (which PARAMETER statements declare) do not appear in the cross-reference listing unless they are used elsewhere in the program.

Switch Pointers and Cautions

Certain switches are primarily useful during program development; they are less desirable for final programs because they slow down a program or produce a larger program. These development-oriented switches are: /DEBUG and /SUB for AOS and F7716, /SUB for MP/AOS and MP/AOS-SU, and /DEBUG, /LINEID, /PROCID, and /SUB for AOS/VS.

The F77 compiler switches affect all program units in the compile line. So, for a multiple-unit application program, if you want to use different switches on different units, write each unit as a different file and compile the units separately. For example:

```
F77/INTEGER=2/L=MAIN.LS/XREF MAIN ⤶
...
F77/SUB/LOGICAL=2 SUB1
...
F77/SAVEVARS SUB2 ⤶
...
```

For the /INTEGER and /LOGICAL switches, see the caution under /INTEGER.

Compile Line Examples

```
) F77 MYPROG ⤶
) F77/OP=3 APROG ⤶
```

Next, assume three different source programs written as separate files:

```
) LISTFILE MPROG.LS ⤶
) F77/DEBUG/L/XREF/NOMAP/LINEID MPROG ⤶
...
) F77/DEBUG/LINEID SUB1 ⤶
...
) F77/DEBUG/STRINGS=ANSI SUB2 ⤶
```

Commands in the above example set the LISTFILE to MPROG.LS, then compile three separate files, generating a compilation listing from MPROG only, and sending this listing to listfile MPROG.LS.

```
) F77/CARDF/LINEID/L=XX C_PROG ⤶
...
```

The above example compiles source file C_PROG in card format, with LINEIDs and listing file XX.

Compiler Error Handling and Messages

During compilation, the compiler writes a description of every error it encounters to file @OUTPUT or to the file given with /E=; it also writes this description to the listing file, if any. These error messages are quite explicit; where possible they will include the line number and entity name of the offending line. An example of a compiler error message is

ERROR 299 SEVERITY 3 BEGINNING ON LINE 86
The statement label "100" is multiply declared.

When the F77 compiler process terminates, either normally or because of an error, it sends a message to the CLI. The termination message will be null if the compiler detected no errors; otherwise the message will be one of the following, depending on the severity of the error:

F77 COMPILATION WARNING 1
F77 COMPILATION ERROR 2
F77 COMPILATION ERROR 3
F77 COMPILATION ABORT 4

The *SEVERITY* number in the error message and in the termination message indicates the *severity level* of the compilation error(s). The severity levels vary from 1 (mild) to 4 (abort), as follows:

Severity Level	Description
1	Level-1 errors provoke a warning message but produce executable code. The compiler continues.
2	Level-2 errors are errors that the compiler can resolve by making an assumption. Compilation continues and code is generated, but the generated code may not execute the way you want it to.
3	Level-3 errors are serious enough to prevent the compiler from proceeding to the code generation phase. The compiler scans the entire source program and returns control to the CLI.
4	Level-4 errors are serious enough to prevent the compiler from continuing. It halts and returns control to the CLI.

A sample compile with error messages might look like this:

```
) F77 MYPROG ↵
```

Source file: MYPROG.F77
Compiled on 19-Sep-84 at 10:33:18 by F77 Rev x.xx
Options: F77

ERROR 255 SEVERITY 2 BEGINNING ON LINE 45
Missing string delimiter in character-string constant.

ERROR 299 SEVERITY 3 BEGINNING ON LINE 86
The statement label "100" is multiply declared.

ERROR 217 SEVERITY 4 BEGINNING ON LINE 86
Errors of severity 3 prevent code generation.

F77 COMPILATION ABORT 4
)

A list of all possible F77 compiler errors is *not* in this manual. Instead, as you have seen, the compiler outputs the full text of an error whenever it detects one. Similarly, F77 runtime errors are output from ERR.F77.IN.

Link Switches

Macro F77LINK directs Link to construct an executable program file. This section explains the switches most pertinent to F77 that you can give to F77LINK and their effects. In addition, Link has an array of its own switches, described in your system's Link user's manual. An example of a Link switch you can give to F77LINK is /SUPST; it suppresses creation of the symbol table file. You can apply any legal combination of Link *command* switches to the F77LINK macro name, or any legal combination of Link *argument* switches to the filename arguments.

You must spell out all F77LINK switches completely; you cannot abbreviate their names. Use them in the form

F77LINK/*switch*/... program-object-filename ...

An introductory example on an AOS/VS system is

```
F77LINK/FMTMM/TRUNCATE MY_PROG SUB_1 &
SUB_2 MY_LIB.LB ↵
```

Here, macro F77LINK directs Link to create MY_PROG.PR based on object files MY_PROG.OB, SUB_1.OB, SUB_2.OB, subprograms in library file MY_LIB.LB, and library files that Data General has supplied. The common language runtime library, LANG_RT.LB, is among these library files.

Table 9-3 contains F77LINK switches and the versions of F77 that apply to the switches. The descriptions of the switches follow the table.

Table 9-3. F77LINK Switches and Their Versions of F77

F77LINK Switch Name \ Version of F77	AOS/VS	AOS	F7716	MP/AOS	MP/AOS-SU
/AOS			√		
/CHANNELS		√	√	√	√
/CIS		√	√		
/DEBUG	√	√	√		
/EVENTMARKS				√	√
/FINT		√	√		
/FMTMM	√	√	√	√	√
/FPF		√	√		
/IOCONFLICT	√	√	√		
/PRECONNECTIONS	√	√	√	√	√
/RESOURCEMARKS				√	√
/ROUND	√				
/SEMAPHORES				√	√
/TASKS	√	√	√	√	√
/TRUNCATE	√				

/AOS[=*revision-number*]
For F7716 only, directs Link to search URT.LB, the AOS runtime library. If you include *revision-number,* as in

```
) F77LINK16/AOS=4.10 MY_PROG ↵
```

Link searches the version of URT.LB whose revision level equals *revision-number.*

/CHANNELS=n
For AOS, F7716, MP/AOS, and MP/AOS-SU only, this switch has a default value of 8. If your program has more than 8 units open (including preconnected ones), increase the value of n.

/CIS
For AOS and F7716 only, directs Link to include modules from the F77 runtime library which will directly execute the CMP and CMV instructions. These instructions are part of the standard Commercial Instruction Set for ECLIPSE C-series, M-series, and MV/Family computers; and part of the optional Character Instruction Set available on some ECLIPSE S-series computers.

If your processor has the CMP and CMV instructions, specifying the /CIS switch will make F77 programs which use character data run faster.

/DEBUG
Directs Link to include the SWAT high-level language debugger and needed symbols in the program file, filename.PR. Link then also creates related files filename.DL and filename.DS for the SWAT debugger. /DEBUG is meaningful only if one or more of the object files in the command line have been compiled with the /DEBUG compiler switch.

MP/AOS and MP/AOS-SU do not support the SWAT debugger.

/EVENTMARKS=ev
For MP/AOS and MP/AOS-SU multitasked programs only, ev is the number of eventmarks allocated for your program. See your documentation changes file (EWICS_DOC) or Software Release Notice for more information.

/FINT
For AOS and F7716 only, directs Link to include modules from the F77 runtime library which will directly execute the FINT instruction. By default, F77 simulates FINT in software.

If your processor has the FINT instruction, specifying the /FINT switch will make some F77 programs run faster. These programs include floating-point arithmetic.

/FMTMM
/FMTMM is the ForMaT MisMatch switch. Without it you may not, for example, output an integer entity with edit descriptor F4.1. With this switch you may, for example, output an integer entity with the F4.1 edit descriptor; an integer entity whose value is 18 is output here as the four characters "18.0". See Table 6-3 in Chapter 6 for allowable entity/edit descriptor mismatches and the operations F77 performs when you give the /FMTMM switch.

/FPF
For AOS and F7716 only, directs Link to include, from the F77 runtime libraries, modules that will use the Floating-Point Function instructions in the M600 instruction set. Use this switch only if you will run the program on an M600.

If your processor has the Floating-Point Function instructions, specifying the /FPF switch will make some F77 programs run faster. These programs include trigonometric functions, complex numbers, exponentiation, logarithms, and square roots.

/IOCONFLICT
You may need this switch for multitasked programs. You may not use it under MP/AOS and MP/AOS-SU. See your Environment Manual for its explanation.

/PRECONNECTIONS= * or NONE or pathname
You may accept default DG unit/file preconnections (see Table 5-3) or choose other ones. Choosing other ones may make it easier to import F77 programs from non-DG systems or, as another possibility, to have your programs use less memory.

By default (or by the inclusion of /PRECONNECTIONS=F77DGPCT), macro F77LINK directs Link to include in the program file (.PR file) the unit/file preconnections of Table 5-3. These default preconnections are in the DG-supplied files F77DGPCT.SR and F77DGPCT.OB.

/PRECONNECTIONS=*
The /PRECONNECTIONS=* F77LINK switch results in the inclusion in the program file of the DG-supplied file F7756PCT.OB. F7756PCT.OB is a subset of F77DGPCT.OB and includes preconnections only for units 5 and 6.

/PRECONNECTIONS=NONE
The /PRECONNECTIONS=NONE F77LINK switch results in the inclusion, in the program file, of the DG-supplied file F77NOPCT.OB. F77NOPCT.OB contains no preconnections. Consequently, use of the * unit specifier will generate a *Unit is closed* runtime error.

/PRECONNECTIONS=pathname
Data General used macro F77BUILD_PCT.CLI (which is supplied to you) to create each of the .OB files described under the previous three switches. You may use this macro, a changed copy of source program file F77DGPCT.SR, and the /PRECONNECTIONS=pathname F77LINK switch to create your own preconnections. See the section "Defining Your Own Preconnections" below for details.

Any preconnection you create means one less runtime call to the general open routine. Thus, if you have all of a program's units and files preconnected, then Link won't place the general open routine into the program file.

/RESOURCEMARKS=rm
For MP/AOS and MP/AOS-SU multitasked programs only, rm is the number of resourcemarks allocated for your program. See your documentation changes file (EWICS_DOC) or Software Release Notice for more information.

/ROUND
For AOS/VS only, will direct the ECLIPSE MV/Family hardware floating-point unit to use rounding for floating-point values. /ROUND is the default, /TRUNCATE (next) is an option.

/SEMAPHORES=sm
For MP/AOS and MP/AOS-SU multitasked programs only, sm is the number of semaphores allocated for your program. See your documentation changes file (EWICS_DOC) or Software Release Notice for more information.

/TASKS=n
You need this switch for multitasked programs. See your Environment Manual for its explanation.

/TRUNCATE
Will direct the ECLIPSE MV/Family floating-point unit to truncate floating-point values.

Link Command Line Examples

```
) F77LINK MYPROG ⤸
) F77LINK/DEBUG MYPROG ⤸
) F77LINK MXPROG SUB1 SUB2 SUB3 SUB4 ⤸
```

Defining Your Own Preconnections

Using this switch is the last step in defining your own preconnections between F77 units and files. The complete set of steps is

1. Begin by deciding on a name for your new preconnection file. We'll use F77NEWPCT.SR in these steps, but you can choose any other unused name for your new preconnection file. Then, copy the default preconnection source file to this new file. Use the CLI commands

   ```
   DELETE/2=IGNORE F77NEWPCT.SR ; &
   COPY F77NEWPCT.SR F77DGPCT.SR
   ```

   Doing this also lets you have a convenient file to restart from if necessary.

2. Print file F77NEWPCT.SR. Examine it, and then decide what preconnections you want to delete, change, or add.

3. Use the six sets of statements of the form

   ```
   RESET
   specifier1 = value1
   specifier2 = value2

   ...

   specifiern = valuen
   PRECONNECTION unit [filename]
   ```

   as models as you delete, change, or add to these sets of statements in file F77NEWPCT.SR. Be sure to specify unit with a decimal point; otherwise, the macroassembler (MASM) will assume an octal number.

   Omit *filename* with caution. If your program tries to use a preconnected unit that doesn't have a filename, then F77 attempts to open a file with a system-dependent name. For example, the entry PRECONNECTION 23 will result (if your PID is 018 on an AOS/VS system) in F77's runtime attempt to open file ?018.F77.UNIT.23.

   Construct *filename* with caution. MASM treats the text string you supply for the actual value of *filename* as part of a macro definition. This means that the ^ (caret/uparrow) and _ symbols (whose respective octal values are 136 and 137) are processed according to certain rules. You can find these rules in both the *AOS Macroassembler Reference Manual* and the *AOS/VS Macroassembler Reference Manual.* For example:

filename Text String	**MASM-Produced Text String**	**Comment**
FOO1	FOO1	OK (no ^ or _ symbol)
FOO_2	FOO2	_ disappears
FOO^3	---	error
FOO___4	FOO_4	first _ disappears
^FOO5	---	error
_^FOO6	^FOO6	_ disappears
FOO7:DIR___1	FOO7:DIR_1	first _ disappears

4. Invoke macro F77BUILD_PCT.CLI by typing

   ```
   F77BUILD_PCT F77NEWPCT ⤸
   ```

   This macro directs MASM to read files F77NEWPCT.SR, F77_PCT1.SR, and F77_PCT2.SR. The last two files, supplied with F77, contain additional macros and some table initialization code, respectively. MASM creates an object file with the preconnections, F77NEWPCT.OB, and a listing file, F77NEWPCT.LS.

5. Include the preconnections in your program file. Assume that you have already given the command

   ```
   F77 MY_PROGRAM ⤸
   ```

   Then type

   ```
   F77LINK/PRECONNECTIONS=F77NEWPCT
     [other switches] MY_PROGRAM
   ```

   [other switches] represents any other switches you want to add to F77LINK.

Runtime Errors

F77 has its own error handling for most runtime errors that can occur: for example, integer overflow, mismatched iolist data types and edit descriptors, or a nonexistent file. An F77 runtime error passes control to F77, which outputs the appropriate error message. If the error is fatal, F77 halts the program and returns control to the CLI. If the error is nonfatal, F77 returns control to the program, usually to the next executable statement.

If a statement provides for error handling — the way OPEN, READ, and WRITE do — you can check the error return variable and then process errors within the program. If you omit the error-handling specifiers, like IOSTAT= and ERR=, F77 treats any error from the statement as a fatal error.

For an example of runtime error handling, the following program segment tries to open a unit, then tests its error variable for the value of "Invalid unit number".

```
...
300  OPEN  (IUNIT, IOSTAT=IOS, FILE='FOO')
     IF (IOS .EQ. 11264)  THEN
         PRINT *,  'Invalid unit number,'
         PRINT *,  '   decrementing by 1.'
         IUNIT = IUNIT - 1
         GO TO 300
     ELSE IF (IOS .NE. 0)  THEN
         PRINT *,  'OPEN error other than'
         PRINT *,  '   Invalid unit, is ', IOS
         STOP
     END IF
     ...
```

Number 11264 is the integer error code for "Invalid unit number". All F77 runtime errors, their mnemonics, and their meanings are described in file ERR.F77.IN. This file was supplied with the F77 software; you can print it to discover the integer codes, their mnemonics, meanings, and text messages involved.

You may wish to use the error mnemonics (as opposed to the integer value of the error) to process runtime errors. Then, you must include file ERR.F77.IN in the source program.

For example, the mnemonic for "Invalid unit number" is IF77ERUNIT. To use the mnemonic, you'd substitute it for the numeric code (being sure to include the error file). With the mnemonic, the example above might be

```
%LIST (OFF)
%INCLUDE 'ERR.F77.IN'
C or INCLUDE 'ERR.F77.IN'
%LIST
     ...
300  OPEN (IUNIT, IOSTAT=IOS, FILE='FOO')
     IF (IOS .EQ. IF77ERUNIT)  THEN
         PRINT *, 'Invalid unit number,'
         PRINT *, '   decrementing by 1.'
         IUNIT = IUNIT - 1
         GO TO 300
       ELSE IF (IOS .NE. 0) THEN
         PRINT *, 'OPEN error other than'
         PRINT *, '   Invalid unit, is ', IOS
         STOP
     END IF
     ...
```

The error mnemonics will not change in future revisions of F77, whereas the error numbers might possibly change; this is an advantage to using the mnemonics.

F7716 Examples

The purpose of F7716 software is to let you create and execute F77 programs on an AOS/VS system such that the execution results are the same as if you had created and executed the programs on an AOS system. Then, you transfer the program files from an AOS/VS system to an AOS system and know what the results will be. Developing large F77 programs is usually faster on an AOS/VS system.

For an example of the possible cases of using F7716, consider program DEMO.F77.

```
PROGRAM DEMO
SUM = 2.5 + 3.0
PRINT *, 'SUM IS ', SUM
STOP
END
```

Construction on AOS/VS for Execution Under AOS/VS as a 32-bit Process

```
) F77 DEMO ⤸
) F77LINK DEMO ⤸
) XEQ DEMO ⤸
```

SUM IS 5.5
STOP

Construction on AOS/VS for Execution Under AOS/VS as a 16-bit Process

```
) F7716 DEMO ⤸
) F77LINK16 DEMO ⤸
) XEQ DEMO ⤸
```

SUM IS 5.5
STOP

Construction on AOS/VS for Execution Under AOS (as a 16-bit Process)

```
) F7716 DEMO ⤸
) COMMENT The /AOS switch has Link ⤸
) COMMENT scan URT.LB (AOS System Library). ⤸
) F77LINK16/AOS DEMO ⤸
) COMMENT DEMO.PR will not execute under ⤸
) COMMENT  AOS/VS. ⤸
) COMMENT Transfer it to an AOS system. If you ⤸
) COMMENT want to execute DEMO.PR under SWAT ⤸
) COMMENT  there, transfer DEMO.DS, DEMO.DL, ⤸
) COMMENT  and ⤸
) COMMENT  DEMO.ST as well. Be sure to transfer ⤸
) COMMENT  DEMO.OL, if it exists, in any case. ⤸
) COMMENT  Log on to the AOS system, and then: ⤸
) XEQ DEMO
```

SUM IS 5.5
STOP

End of Chapter

Index

Within this index, "f" or "ff" after a page number means "and the following page" (or "pages"). In addition, primary page references for each topic are listed first.

C

D

L

M

DataGeneral

users group

Installation Membership Form

Name _____ Position _____ Date _____

Company, Organization or School _____

Address _____ City _____ State _____ Zip _____

Telephone: Area Code _____ No. _____ Ext. _____

CUT ALONG DOTTED LINE

1. Account Category

- ☐ OEM
- ☐ End User
- ☐ System House
- ☐ Government

2. Hardware

	Qty. Installed	Qty. On Order
M/600		
MV/Series ECLIPSE®		
Commercial ECLIPSE		
Scientific ECLIPSE		
Array Processors		
CS Series		
NOVA® 4 Family		
Other NOVAs		
microNOVA® Family		
MPT Family		

Other _____
(Specify) _____

3. Software

- ☐ AOS
- ☐ AOS/VS
- ☐ AOS/RT32
- ☐ MP/OS
- ☐ MP/AOS
- ☐ RDOS
- ☐ DOS
- ☐ RTOS
- ☐ Other

Specify _____

4. Languages

- ☐ ALGOL
- ☐ DG/L
- ☐ COBOL
- ☐ Interactive COBOL
- ☐ PASCAL
- ☐ Business BASIC
- ☐ BASIC
- ☐ Assembler
- ☐ FORTRAN 77
- ☐ FORTRAN 5
- ☐ RPG II
- ☐ PL/1
- ☐ APL
- ☐ Other

Specify _____

5. Mode of Operation

- ☐ Batch (Central)
- ☐ Batch (Via RJE)
- ☐ On-Line Interactive

6. Communication

- ☐ HASP
- ☐ HASP II
- ☐ RJE80
- ☐ RCX 70
- ☐ RSTCP
- ☐ 4025
- ☐ X.25
- ☐ SAM
- ☐ CAM
- ☐ XODIAC™
- ☐ DG/SNA
- ☐ 3270
- ☐ Other

Specify _____

7. Application Description

○ _____

8. Purchase

From whom was your machine(s) purchased?

- ☐ **Data General Corp.**
- ☐ Other

Specify _____

9. Users Group

Are you interested in joining a special interest or regional Data General Users Group?

○ _____

DataGeneral

Data General Corporation, Westboro, Massachusetts 01580, (617) 366-8911

FOLD

TAPE

FOLD

TAPE

FOLD

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 26 SOUTHBORO, MA. 01772

Postage will be paid by addressee:

Data General

ATTN: Users Group Coordinator (C-228)
4400 Computer Drive
Westboro, MA 01581

DataGeneral

TP_____

TIPS ORDER FORM
Technical Information & Publications Service

BILL TO:
COMPANY NAME_____
ADDRESS _____
CITY _____
STATE_____ ZIP _____
ATTN: _____

SHIP TO: (if different)
COMPANY NAME_____
ADDRESS _____
CITY _____
STATE_____ ZIP _____
ATTN: _____

QTY	MODEL #	DESCRIPTION	UNIT PRICE	LINE DISC	TOTAL PRICE

(Additional items can be included on second order form)

[Minimum order is $50.00]

Tax Exempt #_____
or Sales Tax (if applicable)

TOTAL	
Sales Tax	
Shipping	
TOTAL	

METHOD OF PAYMENT

☐ Check or money order enclosed
For orders less than $100.00

☐ Charge my ☐ Visa ☐ MasterCard
Acc't No._____ Expiration Date_____

☐ Purchase Order Number:_____

SHIP VIA

☐ DGC will select best way (U.P.S or Postal)

☐ Other:
☐ U.P.S. Blue Label
☐ Air Freight
☐ Other _____
_____

NOTE: ORDERS LESS THAN $100, INCLUDE $5.00 FOR SHIPPING AND HANDLING.

Person to contact about this order _____ Phone _____ Extension _____

Mail Orders to:

Data General Corporation
Attn: Educational Services/TIPS F019
4400 Computer Drive
Westboro, MA 01580
Tel. (617) 366-8911 ext. 4032

_____
Buyer's Authorized Signature Date
(agrees to terms & conditions on reverse side)

_____
Title

_____
DGC Sales Representative (If Known) Badge #

DISCOUNTS APPLY TO MAIL ORDERS ONLY

012-1780

educational services

CUT ALONG DOTTED LINE

DATA GENERAL CORPORATION
TECHNICAL INFORMATION AND PUBLICATIONS SERVICE
TERMS AND CONDITIONS

Data General Corporation ("DGC") provides its Technical Information and Publications Service (TIPS) solely in accordance with the following terms and conditions and more specifically to the Customer signing the Educational Services TIPS Order Form shown on the reverse hereof which is accepted by DGC.

1. **PRICES**
   Prices for DGC publications will be as stated in the Educational Services Literature Catalog in effect at the time DGC accepts Buyer's order or as specified on an authorized DGC quotation in force at the time of receipt by DGC of the Order Form shown on the reverse hereof. Prices are exclusive of all excise, sales, use or similar taxes and, therefore are subject to an increase equal in amount to any tax DGC may be required to collect or pay on the sale, license or delivery of the materials provided hereunder.

2. **PAYMENT**
   Terms are net cash on or prior to delivery except where satisfactory open account credit is established, in which case terms are net thirty (30) days from date of invoice.

3. **SHIPMENT**
   Shipment will be made F.O.B. Point of Origin. DGC normally ships either by UPS or U.S. Mail or other appropriate method depending upon weight, unless Customer designates a specific method and/or carrier on the Order Form. In any case, DGC assumes no liability with regard to loss, damage or delay during shipment.

4. **TERM**
   Upon execution by Buyer and acceptance by DGC, this agreement shall continue to remain in effect until terminated by either party upon thirty (30) days prior written notice. It is the intent of the parties to leave this Agreement in effect so that all subsequent orders for DGC publications will be governed by the terms and conditions of this Agreement.

5. **CUSTOMER CERTIFICATION**
   Customer hereby certifies that it is the owner or lessee of the DGC equipment and/or licensee/sub-licensee of the software which is the subject matter of the publication(s) ordered hereunder.

6. **DATA AND PROPRIETARY RIGHTS**
   Portions of the publications and materials supplied under this Agreement are proprietary and will be so marked. Customer shall abide by such markings. DGC retains for itself exclusively all proprietary rights (including manufacturing rights) in and to all designs, engineering details and other data pertaining to the products described in such publication. Licensed software materials are provided pursuant to the terms and conditions of the Program License Agreement (PLA) between the Customer and DGC and such PLA is made a part of and incorporated into this Agreement by reference. A copyright notice on any data by itself does not constitute or evidence a publication or public disclosure.

7. **DISCLAIMER OF WARRANTY**
   DGC MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY AND FITNESS FOR PARTICULAR PURPOSE ON ANY OF THE PUBLICATIONS SUPPLIED HEREUNDER.

8. **LIMITATIONS OF LIABILITY**
   IN NO EVENT SHALL DGC BE LIABLE FOR (I) ANY COSTS, DAMAGES OR EXPENSES ARISING OUT OF OR IN CONNECTION WITH ANY CLAIM BY ANY PERSON THAT USE OF THE PUBLICATION OF INFORMATION CONTAINED THEREIN INFRINGES ANY COPYRIGHT OR TRADE SECRET RIGHT OR (II) ANY INCIDENTIAL, SPECIAL, DIRECT OR CONSEQUENTIAL DAMAGES WHATSOEVER, INCLUDING BUT NOT LIMITED TO LOSS OF DATA, PROGRAMS OR LOST PROFITS.

9. **GENERAL**
   A valid contract binding upon DGC will come into being only at the time of DGC's acceptance of the referenced Educational Services Order Form. Such contract is governed by the laws of the Commonwealth of Massachusetts. Such contract is not assignable. These terms and conditions constitute the entire agreement between the parties with respect to the subject matter hereof and supersedes all prior oral or written communications, agreements and understandings. These terms and conditions shall prevail notwithstanding any different, conflicting or additional terms and conditions which may appear on any order submitted by Customer.

DISCOUNT SCHEDULES

DISCOUNTS APPLY TO MAIL ORDERS ONLY.

LINE ITEM DISCOUNT

5-14 manuals of the same part number - 20%
15 or more manuals of the same part number - 30%

DISCOUNTS APPLY TO PRICES SHOWN IN THE CURRENT TIPS CATALOG ONLY.

DataGeneral

TIPS ORDERING PROCEDURE:

Technical literature may be ordered through the Customer Education Service's Technical Information and Publications Service (TIPS).

1. Turn to the TIPS Order Form.
2. Fill in the requested information. If you need more space to list the items you are ordering, use an additional form. Transfer the subtotal from any additional sheet to the space marked "subtotal" on the form.
3. Do not forget to include your MAIL ORDER ONLY discount. (See discount schedules on the back of the TIPS Order Form.)
4. Total your order. (MINIMUM ORDER/CHARGE after discounts of $50.00.)

   If your order totals less than 100.00, enclose a certified check or money order for the total (include sales tax, or your tax exempt number, if applicable) plus $5.00 for shipping and handling.
5. Please indicate on the Order Form if you have any special shipping requirements. Unless specified, orders are normally shipped U.P.S.
6. Read carefully the terms and conditions of the TIPS program on the reverse side of the Order Form.
7. Sign on the line provided on the form and enclose with payment. Mail to:

   TIPS
   Educational Services – M.S. F019
   Data General Corporation
   4400 Computer Drive
   Westboro, MA 01580
8. We'll take care of the rest!

educational services

Data General Corporation, Westboro, MA 01580

093-000162-03