UNISYS

OS 1100 ASCII FORTRAN Programming Reference Manual

February 1991

Printed in U S America

UP-8244.4-B

Priced Item

UNISYS

OS 1100 ASCII FORTRAN Programming Reference Manual

Release Level 11R2 February 1991

Printed in U S America
Priced Item UP-8244.4-B

Contents

Section 9. Writing a FORTRAN Program

Section 10. Debugging a Program

Figures

Tables

Examples

About This Manual

Purpose

This manual is for users of ASCII FORTRAN level 11R2. ASCII FORTRAN level 11R2 contains all the features of the FORTRAN standard, X3.9-1978 (also known as FORTRAN 77).

Organization

This manual is organized as follows:

Section 1.

Introduces you to the FORTRAN programming language.

Section 2.

Introduces you to the language characteristics of FORTRAN.

Section 3.

Discusses assignment statements.

Section 4.

Discusses control statements.

Section 5.

Discusses input and output statements.

Section 6.

Discusses specification and data assignment statements.

Section 7.

Discusses function and subroutine procedures.

Section 8.

Discusses program control statements.

Section 9.

Guides you in writing an ASCII FORTRAN program.

Section 10.

Discusses the debug features of ASCII FORTRAN.

Appendix A.

Discusses the differences between FORTRAN processors.

Appendix B.

Provides tables of ASCII symbols and codes.

Appendix C.

Points out the most commonly encountered programming errors to the novice programmer.

Appendix D.

Lists the diagnostic messages that may be issued during the compilation of ASCII FORTRAN programs.

Appendix E.

Shows if a specific data type can be converted to another data type.

When conversion is possible, a brief description of the conversion method is given.

Appendix F.

Lists, in table form, nonexecutable and executable ASCII FORTRAN statements.

Appendix G.

Discusses ASCII FORTRAN run-time input/output using the processor common I/O modules.

Appendix H.

Discusses large programs and the multibanking features of ASCII FORTRAN.

Appendix I.

Lists, in table form, the diagnostic messages associated with the checkout mode of the ASCII FORTRAN compiler.

Appendix J.

Compares ASCII FORTRAN level 8R1 to level 9R1 and higher.

Appendix K.

Discusses interlanguage communication.

Appendix L.

Discusses the ASCII FORTRAN interface with the sort/merge package.

Appendix M.

Discusses the use of virtual space with your ASCII FORTRAN program.

Related Product Information

The following related documents apply to this guide when you use the OS 1100 software. Use the version that corresponds to the level of software in use at your site. Consult the *1100 and 2200 Series Systems Customer Product Information Catalog and Price List* (3938 8641) for specific levels and document numbers.

***OS 2200 Exec System Software Executive Requests Programming Reference Manual* (7830 7899)**

***OS 1100 System Service Routines Library (SYSLIB) Programming Reference Manual* (7833 1733)**

***OS 2200 Sort/Merge Programming Guide, Level 17R1* (7831 0687)**

***OS 1100 Collector Programming Reference Manual, Level 33R1* (7830 9887)**

***OS 1100 Exec System Software Common Banks Programming Guide* (7830 7386)**

***OS 1100 Conversational Time Sharing (CTS) Programming Reference Manual* (UP-7940)**

***OS 1100 Assembly Instruction Mnemonics (AIM) Programming Reference Manual* (UP-9047)**

***OS 2200 Processor Common Input/Output System (PCIOS) Administration and Programming Reference Manual, Level 6R2* (7831 0588)**

***OS 1100 UDS Shared File System (UDS SFS 1100) Administration and Support Reference Manual, Level 2R2A* (7831 0786)**

***OS 1100 Procedure Definition Processor (PDP) Operations Reference Manual* (UP-10070)**

Notation Conventions

Throughout this manual examples illustrate syntax or other material covered in each section. The comments associated with these examples are shown as FORTRAN comment statements, with C or * in the first position of each line. Therefore, the examples in this manual are self-documenting.

Example:

```
COMMON A,B,C(5,5) /COM1/D,M,V,S
C   Variables A, B and array C are placed in blank common.
C   Variables D, M, V, and S are placed in
C   the common block named COM1.
```

The notation conventions used in the format of statements or clauses and in other portions of this manual are as follows:

Convention	Use
A	Capital letters represent entries you must code exactly as shown.
a	Lowercase italic letters represent data you must supply.
[ ]	Items within brackets represent optional entries that you can use or omit.
{ }	Items within braces represent choices from which you select one.
. . .	Ellipses in statement formats indicate entries you can repeat as necessary.
. . .	Ellipses in program examples indicate missing FORTRAN statements deleted because they are not relevant to the example.
△	A delta indicates a blank character.
bolding	Bolding denotes features of ASCII FORTRAN that are extensions to the FORTRAN 77 standard, ANSI X3.9-1978.

Most of the features that are bolded in this manual also cause a message indicating nonstandard usage to be printed at compilation time if the T option is used (see 9.5.1 and 9.8):

```
   CHARACTER*4 A,B*8
   B = A&'END'
*NON-STD USAGE 3151 at line 2 '&' used as concatenation operator
```

Section 1 Introduction

1.1. Reason for FORTRAN

FORTRAN (from FORmula TRANslation) is a programming language designed for extensive use in mathematical, scientific, and technological areas. The advantages of FORTRAN are minimum programming time and cost, and maximum interchangeability of FORTRAN programs on different FORTRAN processors.

FORTRAN statements resemble English statements and the equations of elementary algebra. Therefore, FORTRAN statements are self-documenting since the intended operation is apparent from the statement itself. For example, to find the average of two numbers, you write a statement such as:

```
AVRGE = (A+B) / 2.0
```

Since the FORTRAN programming language resembles the language ordinarily used for the solution of problems, relatively little time is required to learn the language. As a result, programming effort can be devoted to the logic of the problem without being troubled by the intricacies of computer operation. The self-documenting feature of FORTRAN reduces debugging time and enables other programmers to readily grasp the logic of a program so that it can be modified or adapted to other purposes with minimal effort.

OS 1100 ASCII FORTRAN handles the full American Standard Code for Information Interchange (ASCII) character set. Throughout this manual, the language is referred to as ASCII FORTRAN.

ASCII FORTRAN is written in accordance with the specifications of the American National Standards Institute, Inc. (ANSI), in ANSI X3.9-1978 (also known as FORTRAN 77). ASCII FORTRAN is a superset of the FORTRAN 77 standard, containing additions and enhancements (extensions) to the standard. Extensions to the standard are represented by bold text in this manual.

The ASCII FORTRAN compiler is highly compatible with OS 1100 FORTRAN V. (See Appendix A for a comparison of these two compilers.)

1.2. Evolution of FORTRAN

The fundamental unit of information handled by a data processing system is the bit (binary digit). Basic machine language words are composed of a sequence of bits. A word is interpreted by the computer as an instruction or as data. The meanings of the bits in instruction words (machine language) are highly specific to each computer. Each bit has two mutually exclusive states, represented by 0 and 1.

The next logical step from machine language is commonly known as assembly language. Assembly language requires a language translation program (an assembler). Using assembly language, you can use symbolic references to storage and specific mnemonic codes instead of numeric codes to designate the operation to be performed.

FORTRAN is one of many high-level languages that has evolved from assembly language. It is considered a high-level language because the translation of a single FORTRAN statement can result in many machine language instructions. This conversion is performed by a program called the FORTRAN processor. The design of a FORTRAN processor is machine-oriented and is not part of the FORTRAN language. Offsetting this processor complexity is the decreased programming time required for reading, writing, debugging, and maintaining FORTRAN programs.

1.3. FORTRAN System

The FORTRAN system is composed of three main entities:

1. the program,
2. the language processor, and
3. the execution-time system.

The word “program” means the source program and the object program . The source program is what you write to solve a particular problem. For the computer to understand the source program, it is translated by the FORTRAN processor into machine language. This output of the FORTRAN processor is called an object program.

1.3.1. FORTRAN Processor

The main function of the ASCII FORTRAN processor is to prepare a machine language object program from the source program code. Generally, the processor is referred to as a compiler.

The source program is composed of the FORTRAN statements, which represent logical steps for solving a particular problem. The organization of a source program is discussed in detail in Section 9.

The compiler makes use of the overall logic structure of the program and generates machine instructions for each FORTRAN statement contained in the source program. To produce a machine-acceptable object program, it also assigns storage locations for variables and constants, and creates references to external programs and variables, when required.

The ASCII FORTRAN compiler is a multiphase, modular compiler. The phases are separated, on the basis of general operations, into a set of reentrant instruction banks and a single, nonreentrant data bank. The separation into banks means that only those portions of the compiler necessary to process your program are loaded. Because the instruction banks are reentrant, multiple users can simultaneously execute instructions from a single bank of the compiler. Thus, the compiler makes efficient use of resources, especially in an active multiprogramming environment.

The compiler reads the source program from a series of statement lines present in a runstream or saved in a program file. Comment lines can be used freely within a source program without affecting the compilation.

The main output of a compilation consists of a relocatable binary program. This object program contains the following in binary coded form:

- the names of all common blocks, external functions, and external subroutines referenced in the source program, and
- the machine code for the source program.

ASCII FORTRAN generates reentrant I-bank code (that is, the code doesn't modify itself).

Optionally, the compiler can be directed to generate code in main storage and immediately execute it. This mode of operation is the FORTRAN checkout mode, during which additional diagnostic facilities are provided. Throughput is increased during this mode when the programs processed are executed only once without modification or when the execution time is relatively short. No relocatable binary program is produced in this case (see 9.5 and 10.3).

A separate restart processor, FTNR, is available in conjunction with FORTRAN checkout mode. FTNR lets you reenter execution of a previously saved program (see the checkout debug mode SAVE command, 10.3.3.12), without recompiling the source program. For more information, see 10.3.6.

Much of the compiler's effort is devoted to the detection of source language errors. When any errors are detected or remarks are to be made, the compiler prints diagnostic messages adjacent to the FORTRAN statements containing these errors. The compiler listing produced is controlled by several options that determine the amount of information printed. The compiler listing is described in 9.4.2.

By using processor options, you can direct the compiler to devote additional time to optimizing the FORTRAN statements before generating the relocatable binary output . This is done at the expense of compilation speed, but the resulting output usually executes significantly faster than without this additional optimization. The various types of optimization are explained in 9.6.

The compilation process (when not in checkout mode) is illustrated in Figure 1-1.

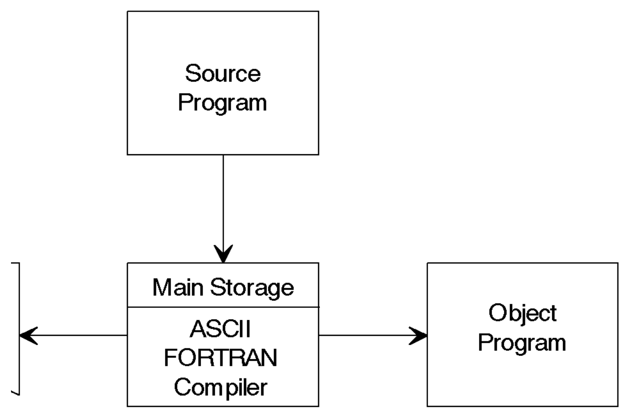


Figure 1-1. Compilation Process

1.3.2. FORTRAN Execution Time System

The FORTRAN execution-time system consists of the following system software:

- the compiler and library elements
- the program
- the computer itself
- any peripheral devices attached to the computer

The system software is known as the operating system. It may contain programs that control the following:

- scheduling
- file management
- input/output
- compilation
- debugging
- storage assignment
- linking

- loading
- assembly
- other necessary functions

1.4. Sample Program

A simple executable program is shown in Figure 1-2. The concepts and terminology used in the description of the program are described in detail in other sections of the manual. Organization of a FORTRAN program is discussed in 9.2. This sample program consists of two program units:

1. the main program, and
2. the external function AMEAN.

This program calculates the average of a series of numbers. The subprogram is generalized in order to calculate the average of a variable number of values. The subprogram results are printed in the main program together with explanatory text.

This is not the only program that could have been written for the problem, nor is it the shortest in terms of lines required. It does introduce the general framework of the FORTRAN system and some of the nomenclature. The number in parentheses preceding each statement is for reference purposes; it is not part of the program. The delta (Δ) symbol indicates a significant blank.

1.4.1. Explanation of Main Program Coding

Line 4 is a comment line indicated by the character C in column 1. This comment consists of all blank characters producing a blank line (except that the C is printed). Comment lines are not required, but they aid the reader in understanding the program.

Line 5 of the program is also a comment line. This line (including the C) is printed when the program is compiled, but doesn't affect execution of the program. Comment lines document the program.

Line 8 indicates that the symbolic name AMEAN is the name of an external function.

Line 9 causes the ASCII FORTRAN compiler to set aside five locations for array A. A is a single-precision real number array.

Line 10 sets initial values for A. A is a symbolic name for the data values used to compute the mean. To reduce execution time, the DATA statement assigns these initial values at compilation time.

Line 11 contains a FORMAT statement that is used to print the character constant 'AVERAGE IS:' followed by five print positions for the value returned from the function AMEAN. This value is printed with two digits to the right of the decimal point.

Line 12 assigns the average of array A to variable AVE. The average is obtained from function AMEAN.

Line 14 is a PRINT statement specifying that the value stored in AVE is printed according to the FORMAT statement prefixed by the identifying number 1. The sample output is:

```
AVERAGE Δ IS: Δ3.00
```

Line 16 indicates the end of the main program.

```
(1) @RUN SAMPL,999999,SMW
(2) @ASG,CP MY*FILE1
(3) @FTN,SI MY*FILE1.MAIN,TPF$.MAIN
(4) C
(5) C COMPUTE AVG OF NUMBERS TO BE INPUT FROM DATA
(6) C
(7) C INITIALIZE
(8)          EXTERNAL AMEAN
(9)          REAL A(5)
(10)         DATA A/1.,2.,3.,4.,5./
(11) 1       FORMAT('ΔAVERAGEΔIS:',F5.2)
(12)         AVE = AMEAN (A,5)
(13) C PRINT AVERAGE AND TEXT
(14)         PRINT 1,AVE
(15) C INDICATE END OF MAIN PROGRAM
(16)         END
(17) C FOLLOWING FUNCTION IS CALLED BY MAIN PROGRAM
(18)         FUNCTION AMEAN(DATA,N)
(19)         DIMENSION DATA (N)
(20)         SUM = 0
(21)         DO 1 I = 1,N
(22) 1            SUM = SUM+DATA(I)
(23)         AMEAN = SUM/N
(24) C INDICATE END OF FUNCTION
(25)         RETURN
(26)         END
(27) @XQT
(28) @FIN
```

Figure 1-2. Sample Program

1.4.2. Explanation of Subprogram Coding

Line 18 identifies an external function subprogram named AMEAN with inputs of DATA and N. This procedure computes the average of N numbers and then returns the result to the referencing program.

Line 19 is a specification statement. It declares the input variable DATA to be an array. The dimension N indicates that the array length is variable and that the actual length is determined by the main program.

Line 20 is an arithmetic assignment statement that initializes the variable SUM to 0.

Line 21 contains a DO statement that controls the repeated execution of the statement on line 22. In this program, the statement labeled 1 (line 22) executes five times since N is given the value 5 by the main program. The variable I initially has the value of 1. This increments by 1 each time lines 21 and 22 are executed.

Line 22 is an arithmetic assignment statement that updates the running total. It obtains the current value of SUM, adds to it the Ith element in array DATA, and assigns this sum as the new value for SUM. Line 22 is repeated five times in this sample. Each time the statement executes, a new value is obtained from the input array DATA, starting at DATA(1) and ending with DATA(5).

Line 23 computes the average as the sum divided by the number of elements, and assigns the result as the value of function AMEAN.

Line 25 returns control from function AMEAN to the statement in the main program that referenced or called this function. In this sample, the return is to line 12.

Line 26 indicates the end of the function subprogram.

Every statement (columns 7 through 72) except an assignment statement starts with a keyword. (Comment lines aren't considered statements.) The keyword is an English word that describes the purpose of the statement. Every statement in ASCII FORTRAN, except statement functions and certain assignment statements, begins with a keyword. Keywords aren't reserved words in ASCII FORTRAN. They can be used anywhere in the program as symbolic names.

1.4.3. Explanation of System Command Coding

Line 1 of Figure 1-2 is a RUN statement. It must be the first Executive control command in a run. It identifies the run to the system and supplies accounting information.

Line 2 is an Executive control statement. It is used to assign an external file to the run under the name MY*FILE1.

Line 3 is the ASCII FORTRAN compiler call. The compiler call specifies the location of source input, the type of listing desired, and the placement of relocatable binary code after compilation. (See 9.5.)

Line 27 is an execute command. It causes the program to be collected (when not previously specified in a @MAP control statement) with the ASCII FORTRAN relocatable library elements in SYS$LIB$*FTN. If these elements are not in SYS$LIB$*FTN, a specific collection (@MAP statement) must be done before the program can execute (@XQT statement).

Line 28 signals the termination of the run.

Section 2
Language Characteristics

2.1. General

The ASCII FORTRAN programming language is made up of a group of statements. In forming these statements, one complies with a set of syntax rules.

This section is a summary of various aspects of the ASCII FORTRAN language used to form source program statements.

2.2. Character Set

The ASCII FORTRAN character set consists of 26 letters (uppercase and **lowercase**), 10 digits, and 16 special characters from the ASCII character set. (You won't receive a nonstandard usage message if you use lowercase letters and other nonstandard characters.)

The compiler reads all input in the 9-bit character form of the American Standard Code for Information Interchange (ASCII). All character literal data is retained in this form by the compiler. The 9-bit representation of ASCII consists of one zero bit followed by the 8-bit ASCII code.

The compiler doesn't always distinguish between uppercase and lowercase alphabetic characters. In literal strings (items enclosed in apostrophes), a distinction is made between lowercase and uppercase characters. Thus the literal string 'abcd' is distinct from 'ABCD' in internal representation. In other instances, the compiler considers lowercase alphabetic characters identical to uppercase alphabetic characters. Thus, the symbolic name xxx is identical to the symbolic name XXX.

The full ASCII character set appears in Appendix B. The ASCII FORTRAN character set appears in Table 2-1.

Table 2-1. ASCII FORTRAN Character Set

Character Group	Members
Uppercase Alphabetics:	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Lowercase Alphabetics:	**a b c d e f g h i j k l m n o p q r s t u v w x y z**
Digits:	0 1 2 3 4 5 6 7 8 9
Special Characters:	blank (represented by △ in this manual)
	= (equals)
	+ (plus)
	- (minus)
	* (asterisk)
	/ (slash)
	((left parenthesis)
	) (right parenthesis)
	, (comma)
	. (decimal point)
	: (colon)
	’ (apostrophe)
	< (less than)
	> (greater than)
	$ (currency symbol)
	& (ampersand)

2.3. Constants

A constant is a quantity that does not change value during the execution of a program. The value of a constant is determined from its name and initial value.

A constant can be contrasted with a variable (see 2.4.3), which is given a name, but the value is allowed to vary during the program's execution.

Sometimes constants and single variables are referred to as scalars because of their single nature. Arrays are not scalar.

The manner in which a constant is written (i.e., its form) specifies its value and its data type.

The following constants are used in ASCII FORTRAN:

- Integer constant
- Real constant (single and double precision)
- Complex constant (single and **double** precision)
- Logical constant
- Character constant
- **Hollerith** constant
- **Octal** and **Fieldata** constants

2.3.1. Integer Constants

An integer constant is a string of 1 to 11 digits, possibly prefixed by a plus or minus sign, that represents a whole decimal number (a number without a fractional part).

The absolute value of an integer constant must be less than or equal to:

$2^{35} - 1 = 34,359,738,367$

The following are valid integer constants:

```
0
+753
-999999
2501
```

An integer is represented internally as a fixed-point number, occupying one word of storage. It must not contain a decimal or comma. It can assume a positive, negative, or zero value.

2.3.2. Real Constants

FORTRAN distinguishes between integer and real constants by the presence or absence of a decimal point and/or exponent. If a constant contains a decimal point or an exponent, or both, it is called a real constant.

Real constants can be one of two types:

- single-precision real (see 2.3.2.1)
- double-precision real (see 2.3.2.2)

A real constant is approximate if it contains a fractional part because the value is stored in binary floating-point form. Use of double-precision real constants improves the accuracy of approximation.

2.3.2.1. Single-Precision Real

A single-precision real constant can be expressed in one of two forms:

1. As a string of one to eight significant decimal digits having a maximum absolute value of (2^{27}-1) with one decimal point preceding, imbedded in, or following the digits. The constant can be signed or unsigned.
2. As the constant form described in 1 above followed by a decimal exponent. The power of ten is expressed by appending the letter E followed by a signed or unsigned integer to the real constant. A decimal point doesn't need to be included in the real part of this form.

A real constant is approximate if it contains a fractional part, since the value is stored in binary floating-point form. The following are valid real constants:

```
0.0
.0
1.
-15.07
2.0E2 (means 200.0)
2E2 (means 200.0)
0.00095
4.0E2 (means 400.0)
4.0E+2 (means 400.0)
4.0E-2 (means 0.04)
```

The approximate magnitude of a real constant must be between 10^{-38} and 10^{38}. A real constant occupies one word (four bytes) of storage:

S	Exponent	Fraction
1	2 9	10 36

2.3.2.2. Double-Precision Real

A double-precision real constant can have up to 18 significant digits with a maximum absolute value of (2^{60}-1)and can have an approximate magnitude in the range of 10^{-308} to 10^{308}.

Double-precision real constants are specified with a decimal point and 1 to 18 significant digits. A double-precision constant must contain an exponent that consists of the letter D followed by a signed or unsigned integer. The D has the same meaning for double precision as the E has for single precision.

The following are acceptable double-precision constants:

0.0D0 (means 0)
1.0D0 (means 1.0)
1D0 (means 1.0)
16.9D+1 (means 169.0)
+8.897D+10
-1750.D+19
123.4567891D0
.1234567891D3 (The values of these last two constants are equal.)

A double-precision real constant occupies two words (eight bytes) of storage:

S	Exponent	Fraction
1	2 12	13 36

Fraction (continued)
1   36

2.3.3. Complex Constants (Single and Double Precision)

A complex constant is an ordered pair of real or integer constants, the first representingthe real part of the complex number and the second the imaginary part.

The form of a complex constant is:

```
(r,i)
```

where *r* is the real part and *i* is the imaginary part. Both parts can be either single precision (integer or real) **or double precision**. Both parts of a complex constant must have the same precision.

A single-precision complex constant requires two consecutive words (eight bytes) of storage, with the real part occurring first. When specified with two integer constants, a single-precision complex constant is represented internally (in storage) as two single-precision real numbers.

Double-precision complex constants are formed by a pair of double-precision real constants and require four words (16 bytes) of storage.

The following are valid complex constants:

(0.0,1.0)
(2,3)
(3426.78,293.6)
(4.12E2,6.5)
(4.12E-2,6.5E+3)
(4.12E-10,6.5E-3)
(4.12D-10,6.5D-3) (double precision)

2.3.4. Logical Constants

A logical constant specifies a logical value that is either true or false. The only valid logical constants are:

.TRUE.
.FALSE.

A logical constant occupies one word (four bytes) of storage.

2.3.5. Character Constants

A character constant is a string of characters that can include any of the ASCII characters. The string is enclosed in apostrophes (single quotes). Two apostrophes in succession represent a single apostrophe in the text.

Valid character constants are:

'ABCD'
'123ABC$! !'
'THAT''S' (represented internally as: THAT'S)
'xyz'

Each character in a character constant requires one 9-bit byte (one-fourth of a word) of storage. All character data is represented in the full ASCII form. The 9-bit representation of ASCII consists of one 0-bit followed by the 8-bit ASCII code.

The maximum size of a character constant is 511 characters.

When you split a character constant between two or more lines in the source program, the constant extends up to and including column 72 of the first line and begins again in column 7 of the following continuation line.

2.3.6. Hollerith Constants

A Hollerith constant is a string of ASCII characters that is preceded with *n*H, where *n* is the number of characters in the string.

Valid Hollerith constants are:

```
3HWOW
5HTHAT'S
4HA1+$
27HUPPER and lowercase letters
```

Storage requirements for Hollerith constants are the same as for character constants. (See 2.3.5.)

2.3.7. Octal and Fieldata Constants

Octal numbers (radix 8) and Fieldata characters may be used as constants only in the initialization lists of specification statements and as input to namelists.

An octal constant is expressed by the letter O followed by 1 to 12 octal digits for constants used in initialization lists of specification statements, and 1 to 24 octal digits for namelist input.

A Fieldata constant is written as a character or Hollerith constant immediately followed by the letter F.

2.4. Symbolic Names

A symbolic name is a series of characters assigned by the programmer to refer to a programmer-defined entity such as a variable, array, program unit, or labeled common block.

A symbolic name consists of one to six of the following characters:

- uppercase and **lowercase** letters
- numerics (0, 1, 2, . . . , 9)
- **currency symbol ($)**

The first character of each symbolic name must be alphabetic.

Symbolic names that contain lowercase alphabetics are identical to symbolic names that contain uppercase alphabetics. Thus, the name xxx is identical to the name XXX.

You are entirely free in the choice of words for symbolic names. Keywords such as GO TO, READ, FORMAT, etc., aren't reserved words and can be used as symbolic names or as parts of symbolic names. However, when GO TO, READ, FORMAT, etc., are used as statement keywords, they aren't considered symbolic names. The same applies to sequences of characters that are format edit descriptors such as I3 (see 5.3.1). The use of keywords as variables, function names, or subroutine names is not recommended, since such usage makes programs difficult to read.

You shouldn't use the currency symbol ($) in function and subroutine names, because this may conflict with entry names in the ASCII FORTRAN library and the OS 1100 Operating System relocatable library.

2.4.1. Uniqueness of Symbolic Names

A symbolic name, perhaps qualified by a subscript, identifies a member of one (and only one) of the following entities within a program unit (unless otherwise noted in the additional rules that follow):

1. An array and its elements
2. A scalar variable
3. A statement function
4. An intrinsic function
5. An external function
6. **An internal function**
7. An external subroutine
8. **An internal subroutine**
9. An external procedure that can't be classified as either a subroutine or a function in the program unit in question.
10. A common block
11. A constant defined in a PARAMETER statement
12. **A NAMELIST name**
13. **A program bank name**
14. A main program name
15. A BLOCK DATA subprogram

Some additional rules for using symbolic names are:

- A common block **or bank name** in a program unit can also be the name of any local entity other than a symbolic name of a constant, intrinsic function, or local variable that is also an external function in a function subprogram.
- A main program name is global to the executable program and must not be the same as the name of an external subprogram in the same executable program.
- A FUNCTION subprogram name should also be a variable name in the FUNCTION subprogram.
- Once a symbolic name is used as **a program bank name**, a FUNCTION, SUBROUTINE, or BLOCK DATA subprogram name, a common block name, or a main program name in any unit of an executable program, no other program unit of that executable program can use that name to identify another member of any of these classes.
- The source input to the compiler can contain more than one program unit (see 9.2). Each compilation produces one relocatable element regardless of the number of program units. There are **internal and** external program units. The only entities shared between external program units are arguments passed between them and entities defined in common blocks. Local names in one external program unit have no relationship to the local names in another external unit. **Internal subprograms can have their own local entities and, in addition, can directly access the global entities of their external program unit (see 7.9.)**

  For example, suppose the source input consists of a main program and an external subroutine. The symbolic name A can be used as a local scalar variable in the main program and also as a local array in the subroutine. The two As have no relationship to each other; each A has its own storage sequence.

2.4.2. Data Types of Symbolic Names

A symbolic name representing a variable, an array, or a function (except intrinsic functions) must have only a single data type in a program unit. This data type is associated with the symbolic name throughout the program unit.

The allowed data types for symbolic names are:

1. Integer
2. Real (single precision or double precision)
3. Complex (single precision or **double precision**)
4. Logical
5. Character

The data type of a function determines the type of the data it supplies to the expression in which it appears.

The data type of an array element name is the same as that of its array name.

The data type of a symbolic name can be explicitly declared or implied by its first letter. There are three ways to indicate the type of a symbolic name:

1. Implied declaration using the name rule
2. Implied declaration using the IMPLICIT statement
3. Explicit declaration

2.4.2.1. Implied Declaration Using the Name Rule

The name rule is a traditional FORTRAN convention whereby the data type of a variable, array, or function (except an intrinsic function) is implied by the first character of the symbolic name:

- The data type is integer when the first character of the symbolic name is I, J, K, L, M, or N.
- The data type is single-precision real when the first character of the symbolic name is any other alphabetic character.

The following variables are integer types:

```
I15
J24K
IKE
LOOP
KIM
MASS
L3
J123T
```

The following variables are single-precision real types:

```
TEMP
XX
F55
P3ZK
COUNT
RESULT
ALPHA
XX
```

Double-precision real, complex (single precision and **double precision**, logical, and character types can’t be declared by the name rule.

2.4.2.2. Implied Declaration Using the IMPLICIT Statement

You can override the data type established by the name rule by using the IMPLICIT statement to designate the data type and length that is associated with the specified initial letters of a symbolic name.

For example, the declaration:

```
   IMPLICIT INTEGER (P-R), REAL (M)
```

changes the implied declaration of the name rule so that symbolic names that begin with P, Q, or R are now also integer type, while symbolic names that begin with M are now real type.

You can use the IMPLICIT statement to specify all types of symbolic names (integer, real, character, complex, and logical) and to indicate length specifications for character data type.

See 6.3.1 for details on the IMPLICIT statement.

2.4.2.3. Explicit Declaration

You can explicitly declare the data type of a specific symbolic name by using a type statement. Declaration by a type statement overrides the name rule and type specifications of IMPLICIT statements.

Explicit declaration differs from the first two ways of specifying data type. An explicit specification statement declares the type of a particular symbolic name by its unique name rather than by the particular letter that begins a name. A symbolic name can thus be assigned a specific data type (see 6.3.2). Character data types can be assigned a specific length.

2.4.3. Variables

The symbolic name used to identify a single storage sequence is called a variable. The name remains fixed throughout the execution of a program, while its contents (its value) may vary.

The size of the storage sequence and the type of value placed in that sequence are determined by the data type declared implicitly or explicitly for the variable.

The allowed data types for variables are:

- Integer
- Real (single or double precision)
- Complex (single or **double** precision)
- Logical
- Character

The storage sequence represented by the variable is undefined prior to assignment of its first value. A variable may be defined through one of the following means:

- a DATA statement,

- an input/output statement,
- an assignment statement,
- its use as an argument in a subprogram reference,
- its use in a DO statement, or
- its association in a COMMON or EQUIVALENCE statement.

A variable is commonly described by the type of data it represents. Thus, an integer variable represents integer data, a real variable represents real data, etc.

A variable occupies the same number of storage units as a constant of the same type.

Once the type of a variable is defined, you can assign it different values of this type in the program. This changeability distinguishes it from a constant.

In the following example, the symbol = indicates that the value of the evaluated right-hand expression is to be assigned to the variable on the left:

```
SUM = TOTAL1 + TOTAL2
```

SUM, TOTAL1, and TOTAL2 are variables. The sum of the values represented by TOTAL1 and TOTAL2 is stored in the storage sequence represented by SUM at execution time. As with all expressions, TOTAL1 and TOTAL2 must have previously been defined to obtain a predictable value for SUM.

A choice of appropriate variable names aids documentation of a program and makes the program more readable.

2.4.4. Arrays and Subscripts

An array is an ordered set of homogeneous objects identified by a symbolic name (the array name). The members of the set represented by the array are referred to as array elements.

The array name together with its position in the array uniquely identifies an array element. An element's position in the array is indicated by a parenthesized expression, known as a subscript, following the array name. The presence of the subscript ensures proper identification of the array element. For example:

```
A(1)
```

identifies the first element in array A (if the lower dimension bound for A is 1; see 2.4.4.1). Each appearance of an array name must include its qualifying subscripts except in the following cases:

- In a DIMENSION statement (the dimension declarator resembles an array's subscripts, but they aren't the same)
- In a COMMON statement
- In a type statement

- In an EQUIVALENCE statement
- In a DATA statement
- In a list of arguments for a reference to a subprogram
- In a list of dummy arguments
- In a list of an input/output statement, when the array isn't an assumed-size array
- As a unit identifier for an internal file in an input/output statement, when the array isn't an assumed-size array
- As the format identifier in an input/output statement, when the array isn't an assumed-size array
- In a SAVE statement

For the preceding cases, the array name without the qualifying subscripts identifies the entire sequence of elements of the array except for the EQUIVALENCE statement (see 6.4).

As with variables, an array and its elements have no defined value until you assign them a value in some way.

2.4.4.1. Array Declaration

As with a variable, an array can have the following data types:

- Integer
- Real (single or double precision)
- Complex (single or **double** precision)
- Logical
- Character

The type of an array is also that of its data elements. The data type of an array is declared implicitly or explicitly (see 2.4.2).

Arrays are introduced in DIMENSION, COMMON, or type statements. They are introduced via array declarators of the form:

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

where:

a

is a symbolic name for the array.

d

is a dimension declarator, which has the following form:

```
[l:]u
```

where:

l

is the lower-dimension bound and is assumed to be 1 if omitted.

u

is the upper-dimension bound.

The number of dimensions of an array is the number of dimension declarators in the array declaration. The minimum number of dimensions is one, and the maximum is seven.

The lower- and upper-dimension bounds are arithmetic expressions in which all constants, symbolic names of constants, and variables must be type integer. A variable or symbolic name of a constant appearing in a dimension-bound expression that is not of default integer type must be specified as integer by an IMPLICIT statement or a type statement prior to its use in the dimension-bound expression. The upper-dimension bound of the last dimension can be an asterisk (*) in an assumed-size array (see 2.4.4.2). A dimension-bound expression must not contain a function or array element reference. Integer variables may appear in dimension-bound expressions only for adjustable arrays.

The value of either dimension bound is positive, negative, or zero. The value of the upper-dimension bound must be greater than or equal to the value of the lower-dimension bound. When only the upper-dimension bound is specified, the value of the lower-dimension bound is 1. An upper-dimension bound of an asterisk is always greater than or equal to the lower-dimension bound.

The following are valid array declarations:

```
DIMENSION ARRAY1(2,3), VARIED(-10:L)
DOUBLE PRECISION ARRAY2(15), LIST2(K:2*K, 100)
COMPLEX RTAB(0:10, 0:20, *), USIZ(*)
COMMON WANTED(3,4,5)
```

2.4.4.2. Constant, Adjustable, and Assumed-Size Arrays

Arrays dimensioned with only integer constant expressions are known as constant dimensioned arrays. An adjustable array has one or more integer variables specified in its dimension declarator. The integer variables must be either dummy arguments or common block variables that must be defined when the reference to the subprogram containing the adjustable array is executed. An assumed-size array is a constant or adjustable array except that the upper-dimension bound of the last dimension is an asterisk.

An adjustable array or assumed-size array can appear only in functions or subroutines, and the array name must appear in the dummy argument list. An array name in a dummy argument list is called a dummy array. The actual argument corresponding to the dummy array name may be an array name, array element, or array element substring. The number and values of the dimensions need not be the same in both the calling and called routines. Storage for the dummy array isn’t allocated in the subprogram and the dummy array can't assume more storage than what its corresponding actual argument is allocated.

The following are examples of the three types of arrays:

```
SUBROUTINE SUB1(X,Y,Z,I)
DIMENSION X(1900:1978)                 @ constant dimensioned array
REAL Y(I:2*I)                          @ adjustable dimensioned array
CHARACTER*8 Z(*)                       @ assumed-size array
```

2.4.4.3. Actual and Dummy Arrays

An actual array is an array whose array name does not appear in a dummy argument list. An actual array has a determinable size and is allocated storage. An actual array is introduced in a DIMENSION, COMMON, or type statement and may have constant array declarators only.

A dummy array is an array whose array name appears in a dummy argument list. The values of a dummy array are the values of the corresponding actual argument. A dummy array is a constant, adjustable, or assumed-size array. Dummy arrays are dimensioned only in DIMENSION and type statements appearing in functions and subroutines.

2.4.4.4. Array Element Reference

An array element is identified by the array name followed by a subscript representing its position within the array. The subscript consists of one to seven subscript expressions separated by commas, forming a list which is enclosed in parentheses. The number of subscript expressions must correspond to the number of dimensions specified when the array dimensions were declared.

Each subscript expression must be an arithmetic expression that yields an integer, **real, or double-precision real** value. When the subscript expression is evaluated **and converted to integer if necessary,** the value is negative, positive, or zero. The value should be within the range of its corresponding array dimension. **The integer result of a real subscript expression may not be what you expect because of the approximate nature of real numbers (see 2.3.2.1)**. When the upper-dimension bound is an asterisk, the value of the corresponding subscript expression must be such that the subscript value doesn't exceed the size of the actual array. Subscript expressions are unrestricted in form and can include array element or function references.

For example, the expression B(3,2) refers to the element in the third row, second column of array B. The expression C(3/3,SQRT(4.),4) refers to the element in the second row, fourth column, of the first plane of array C.

2.4.4.5. Location of Elements Within an Array

The elements of each array are stored in column-major order. This means the array element with lowest subscripts is stored in the lowest storage position and those with higher subscripts (leftmost subscripts increasing most rapidly) are stored in subsequent storage positions. For example, if array B is declared:

```
DIMENSION B(3,3)
```

the array elements of B are stored in ascending storage positions in the following order:

```
 B(1,1)
 B(2,1)
 B(3,1)
 B(1,2)
 B(2,2)
 B(3,2)
 B(1,3)
 B(2,3)
 B(3,3)
```

2.4.5. Character Substrings

A character substring is a contiguous portion of a character variable or array element and is of type character. A character substring is identified by a substring name and can be assigned values and referenced.

The forms of a substring name are:

$v([e_1] : [e_2])$

$a(s[,s] \ldots)([e_1] : [e_2])$

where:

v

is a scalar character variable name.

a(*s*[,*s*] . . .)

is a character array element name.

e_1 and e_2

are integer expressions called substring expressions.

The value e_1 specifies the leftmost character position of the substring, and the value e_2 specifies the rightmost character position.

For example, A(2:4) specifies characters in positions two through four of the scalar character variable A. B(4,3)(1:6) specifies characters in positions one through six of character array element B(4,3).

The values of e_1 and e_2 must satisfy the following relational expression:

$1 \leq e_1 \leq e_2 \leq len$

where *len* is the length of the scalar character variable or array element. When e_1 is omitted, a value of one is implied. When e_2 is omitted, a value of *len* is implied. Both e_1 and e_2 can be omitted. For example, the form *v*(:) is equivalent to *v*, and the form *a*(*s*[,*s*] . . .)(:) is equivalent to *a*(*s*[,*s*] . . .).

The length of a character substring is $e_2 - e_1 + 1$. **Therefore, the expression $v(e_1:e_2)$ is equivalent to the expression SUBSTR($v, e_1, e_2 - e_1 + 1$). See 7.7.2.3 for a description of the SUBSTR pseudo-function. The SUBSTR pseudo-function isn't allowed in certain places where a substring is allowed (for example, DATA and EQUIVALENCE statements)**.

Example:

```
      CHARACTER*4 C1, C2(2,2)
C
C     The following statements are equivalent.
C
      C2(1,1)(I:J) = C1(3:4)
      SUBSTR(C2(1,1),I,J-I+1) = SUBSTR(C1,3,2)
```

2.5. Expressions

An expression is formed from operands, operators, and parentheses in which each primary is a scalar value.

Five kinds of FORTRAN expressions result from combinations of operands and operators:

1. Arithmetic
2. Character
3. Logical
4. Relational
5. **Typeless**

The value of an arithmetic expression is always integer, real (single or double precision), or complex (single **or double** precision). A character expression yields an ASCII character string of length one or greater. The value of a logical or relational expression is .TRUE. or .FALSE., **while that of a typeless expression is a 36-bit string.**

A constant expression is an arithmetic constant expression (see 2.5.1.3), a character constant expression (see 2.5.2.3), a logical constant expression (see 2.5.4.3), or a typeless constant expression (see 2.5.5.2).

2.5.1. Arithmetic Expressions

An arithmetic expression is used to express a numeric computation. Evaluation of an arithmetic expression produces a numeric value.

A simple arithmetic expression is one of the following:

- An unsigned arithmetic constant
- A symbolic name of an arithmetic constant
- An arithmetic variable reference
- An arithmetic array element reference
- An arithmetic function reference

More complicated arithmetic expressions may be formed by using one or more arithmetic operands together with arithmetic operators and parentheses. Arithmetic operands must identify values of type integer, real (single or double precision), complex (single or **double** precision), and **typeless**.

2.5.1.1. Arithmetic Operators

Table 2-2 lists the arithmetic operations permitted in ASCII FORTRAN.

Table 2-2. Arithmetic Operators and Operations

Operator	Operation	Example
+	Binary addition or unary plus	A + B, + A
-	Binary subtraction or unary minus	A - B, - A
*	Multiplication	A * B
/	Division	A / B
**	Exponentiation	A ** B

2.5.1.2. Arithmetic Operands and Forming Arithmetic Expressions

The arithmetic operands are:

1. Primary
2. Factor
3. Term
4. Arithmetic expression

Table 2-3 defines the permissible forms of the arithmetic operands.

Table 2-3. Forms of Arithmetic Operands

Arithmetic Operand	Permissible Forms	Examples
Primary	Unsigned constant	5
	Symbolic name of a constant	PI
	Variable reference	RESULT
	Array element reference	V(1)
	Function reference	SQRT(X)
	Arithmetic expression in parentheses	(A + B)
Factor	Primary	A
	Primary ** factor	A**B
Term	Factor	A
	Term / Factor	A**B/A
	Term * Factor	A**B*A**B
Arithmetic expression	Term	A*B
	+ Term	+A**B
	- Term	-A*B
	Arithmetic expression + term	A*B+V(1)
	Arithmetic expression - term	+A/B-SQRT(X)

Thus, an arithmetic expression is formed from a sequence of one or more terms separated by either the addition operator or the subtraction operator. The first term in an arithmetic expression may be preceded by a unary plus or unary minus. The last two forms of an arithmetic expression indicate that in interpreting an arithmetic expression containing two or more addition or subtraction operators, the terms are combined from left to right.

Note these forms do not permit expressions containing two consecutive arithmetic operators, such as A**-B or A+-B. However, by using parentheses such as A**(-B) or A+(-B), the desired operations can be formed.

2.5.1.3. Arithmetic Constant Expressions

An arithmetic constant expression is an arithmetic expression in which each primary is an arithmetic constant, the symbolic name of an arithmetic constant, or an arithmetic constant expression enclosed in parentheses. The exponentiation operator is not permitted unless the exponent is of type integer. Variable, array element, and function references are not allowed. **ASCII FORTRAN allows the following nonstandard primaries:**

- **exponentiation of constant operands where the exponent is of any legal arithmetic type**
- **intrinsic function references that have all constant arguments**

2.5.1.4. Integer Constant Expressions

An integer constant expression is an arithmetic constant expression in which each constant or symbolic name of a constant is of type integer. Variable, array element, and function references are not allowed.

2.5.1.5. Data Types Permitted for Arithmetic Expressions

You may use different data types for the operands of an arithmetic expression. The type of expression *a op b* is determined as shown in Table 2-4. It shows the permitted combinations of data types and gives the data type and length of the result.

Table 2-4. Type and Length of Result for Arithmetic Expressions

Type of Left Operand (a)	Type of Right Operand (b)				
	INTEGER	REAL	DOUBLE PRECISION	COMPLEX	**COMPLEX*16**
INTEGER	INTEGER	REAL	DOUBLE PRECISION	COMPLEX	**COMPLEX*16**
REAL	REAL	REAL	DOUBLE PRECISION	COMPLEX	**COMPLEX*16**
DOUBLE PRECISION	DOUBLE PRECISION	DOUBLE PRECISION	DOUBLE PRECISION	**COMPLEX*16**	**COMPLEX*16**
COMPLEX	COMPLEX	COMPLEX	**COMPLEX*16**	COMPLEX	**COMPLEX*16**
COMPLEX*16	**COMPLEX*16**	**COMPLEX*16**	**COMPLEX*16**	**COMPLEX*16**	**COMPLEX*16**

The length of the operands and results are:

- REAL and INTEGER - one word
- DOUBLE PRECISION and COMPLEX - two words
- **COMPLEX*16** - four words

The data type of a unary operation is the same as that of its operand.

2.5.1.6. Arithmetic Expression Evaluation

When an arithmetic expression includes more than one arithmetic operator, evaluation of that expression is performed based on the following hierarchy:

Precedence	Operation	Example
1	Expressions in parentheses	(*e*)
2	Function evaluation	SQRT *e*
3	Exponentiation	2**3
4	Multiplication and division	2*3 2/3
5	Addition and subtraction or their unary operations	2+3 -2

Expressions are evaluated, with one exception, in a left-to-right fashion. When a subexpression contains the form:

$a\ op_1\ b\ op_2\ c$

the part $a\ op_1\ b$ is evaluated first as long as op_1 has a greater or equal precedence with respect to op_2 as described by the table above.

The exception to the left-to-right evaluation rule is the form:

```
a ** b ** c
```

The part *b* ** *c* is evaluated first.

Examples of equivalent expressions:

```
a - b * c ** d
```

is equivalent to

```
    a - (b*(c**d) )
```

```
a - b * c * d
```

is equivalent to

```
    a - ( (b*c)*d)
```

```
a ** (b - c) * d
```

is equivalent to

```
(a**(b - c) )*d
```

2.5.2. Character Expressions

A character expression is used to express a character string. Evaluation of a character expression produces a result of type character.

A simple character expression is one of the following:

- A character constant
- A symbolic name of a character constant
- A character variable reference
- A character array element reference
- A character substring reference (see 2.4.5)
- A character function reference

More complicated character expressions may be formed by using one or more character operands together with the character concatenation operator and parentheses.

2.5.2.1. Character Operators

The binary concatenation operator // is the only character operator in ASCII FORTRAN. It is used in the following manner, where e_1 and e_2 are character expressions:

e_1 // e_2

Assume e_1 is a character expression of length m and e_2 is a character expression of length n. The result of the expression is a character expression of length $m + n$. The first m characters of the resulting expression are those of e_1. The remaining n characters are those of e_2.

For example, the expression:

```
'TWIST∆ ' // 'AND∆ TURN'
```

results in the value 'TWIST AND TURN'.

The expression:

```
'TYPE' // '123'
```

results in the value 'TYPE123'.

For syntactic compatibility with other FORTRAN processors, & is also allowed as a concatenation operator.

2.5.2.2. Character Operands and Forming Character Expressions

The character operands are:

1. Character primaries
2. Character expressions

Table 2-5 defines the permissible forms of the character operands.

Table 2-5. Forms of Character Operands

Character Operands	Permissible Forms	Examples
Character primary	Character constant	'AB'
	Symbolic name of a character constant	TITLE
	Character variable reference	DATE
	Character array element reference	CARR(I)
	Character substring reference	C(I:I+1)
	Character function reference	LOWERC(C)
	Character expression in parentheses	(C//'A')
Character expression	Character primary	'AB'
	Character expression //character primary	'AB'//'CD'

Thus, a character expression is a sequence of one or more character primaries separated by the concatenation operator. The last form of a character expression indicates that in a character expression containing two or more concatenation operators, the primaries are combined from left to right. Note that parentheses have no effect on the value of a character expression.

The FORTRAN 77 standard doesn't allow a character item with a length of * to appear as a concatenation operand in an expression in a subprogram argument or an I/O list. **However, ASCII FORTRAN doesn't have this restriction.**

2.5.2.3. Character Constant Expressions

A character constant expression is a character expression in which each primary is a character constant, the symbolic name of a character constant, or a character constant expression enclosed in parentheses. Variable, array element, substring, and function references are not allowed. **ASCII FORTRAN allows the following nonstandard primary:**

- **character intrinsic function references that have all constant arguments**

2.5.3. Relational Expressions

A relational expression is used to compare the values of two arithmetic expressions or two character expressions. A relational expression may not be used to compare the value of an arithmetic expression with the value of a character expression.

Relational expressions may appear only within logical expressions. Evaluation of a relational expression produces a result of type logical, with a value of true or false.

2.5.3.1. Relational Operators

Table 2-6 shows the relational operators which allow you to quantitatively test the relationship between two expressions.

Table 2-6. Relational Operators

Operator	Usage	Explanation
.GT.	e_1 .GT. e_2	True if e_1 is greater than e_2.
.GE.	e_1 .GE. e_2	True if e_1 is greater than or equal to e_2.
.LT.	e_1 .LT. e_2	True if e_1 is less than e_2.
.LE.	e_1 .LE. e_2	True if e_1 is less than or equal to e_2.
.EQ.	e_1 .EQ. e_2	True if e_1 is equal to e_2.

e_1 and e_2 represent either both arithmetic expressions or both character expressions.

2.5.3.2. Arithmetic Relational Expressions

The form of an arithmetic relational expression is:

```
e1 relop e2
```

where *relop* is a relational operator and e_1 and e_2 are each an integer, real, double-precision, complex, **double-precision complex**, or **typeless** expression.

A complex operand is permitted only when the relational operator is .EQ. or .NE.

If the two arithmetic expressions are of different types, the value of the relational expression

e_1 *relop* e_2

is the value of the expression

((e_1) - (e_2)) *relop* 0

where 0 (zero) is of the same type as the expression ((e_1) - (e_2)) and *relop* is the same relational operator in both expressions. **Note that the comparison of a double-precision value and a complex value is permitted with the .EQ. and .NE. operators.**

Example:

```
        LOGICAL L1, L2, L3
        INTEGER I
        REAL A
        COMPLEX C

C       This program results in variables L1 and L3
C       being output as true values and L2 as a false value.

        I = 3
        A = 20.5
        C = (20.5, 0.0)

        L1 = I .EQ. 3
        L2 = 10 .GT. A
        L3 = C .EQ. A

C       Variable A is converted to type
C       complex before the comparison is done.

        PRINT *, L1, L2, L3
        END
```

2.5.3.3. Character Relational Expressions

The form of a character relational expression is:

e_1 *relop* e_2

where *relop* is a relational operator and e_1 and e_2 are character or **typeless** expressions.

In relational expressions with character operands, character relations are performed left to right using the ASCII collating sequence. When one of the two expressions must be extended to make it equal in length to the other, it is extended on the right using blanks.

Example:

```
        LOGICAL L1, L2
        CHARACTER A*2, B*2, C*3

C       This program results in variable L1 being output
C       as a true value and L2 as a false value.

        A = 'ab'
        B = 'ac'
        C = 'abc'

        L1 = A .LT. B
        L2 = A .GT. C

C       Variable A is extended on the right with
C       one blank character to make it equal length
C       to C before the comparison is done.

        PRINT *, L1, L2
        END
```

2.5.4. Logical Expressions

A logical expression is used to express a logical computation. Evaluation of a logical expression produces a result of type logical, with a value of true or false.

A simple logical expression is one of the following:

- A logical constant
- A symbolic name of a logical constant
- A logical variable reference
- A logical array element reference
- A logical function reference
- A relational expression

More complicated logical expressions may be formed by using one or more logical operands together with logical operators and parentheses.

2.5.4.1. Logical Operators

Table 2-7 shows the logical operators permitted by ASCII FORTRAN. The operands (e_i) for logical expressions can be of logical **or typeless** (see 2.5.5) type.

Table 2-7. Logical Operators

Operator	Usage	Explanation
.NOT.	.NOT. e_1	The expression has the logically opposite value of the expression e_1.
.AND.	e_1 .AND. e_2	If both e_1 and e_2 have the value of .TRUE., the expression has the value .TRUE. ; otherwise it has the value .FALSE.
.OR.	e_1 .OR. e_2	If either, or both, e_1 or e_2 has the value .TRUE., then the expression has the value .TRUE. ; otherwise the expression has the value .FALSE.
.EQV.	e_1 .EQV. e_2	(Logical equivalence.) If both e_1 and e_2 have the value of .TRUE. or both have the value of .FALSE., then the expression has the value of .TRUE.; otherwise it has the value of .FALSE.
.NEQV.	e_1 .NEQV. e_2	(Logical nonequivalence.) If e_1 has the value of .TRUE. and e_2 has the value of .FALSE., or if e_1 has the value of .FALSE. and e_2 has the value of .TRUE., then the expression has the value of .TRUE.; otherwise the expression has the value of .FALSE.

2.5.4.2. Logical Operands and Forming Logical Expressions

The logical operands are:

1. Logical primary
2. Logical factor
3. Logical term
4. Logical disjunct
5. Logical expression

Table 2-8 defines the permissible forms of the logical operands.

Table 2-8. Forms of Logical Operands

Logical Operands	Permissible Forms	Examples
Logical primary	Logical constant	.TRUE.
	Symbolic name of a logical constant	FLAG
	Logical variable reference	COND
	Logical array element reference	L(I)
	Logical function reference	FCTN(A,B)
	Relational expression	A .GT. B
	Logical expression in parentheses	(L1 .OR. L2)
Logical factor	Logical primary	.TRUE.
	.NOT. logical primary	.NOT. (A. GT. B)
Logical term	Logical factor	.NOT. L2
	Logical term .AND. logical factor	.NOT. L1 .AND. L2
Logical disjunct	Logical term	A .GT. B
	Logical disjunct .OR. logical term	A .GT. B .OR. L2
Logical expression	Logical disjunct	A .GT. B
	Logical expression .EQV. logical disjunct	(A .GT. B) .EQV. L1
	Logical expression .NEQV. logical disjunct	L1 .NEQV. L2

Thus, a logical expression is a sequence of one or more logical disjuncts separated by either the .EQV. operator or the .NEQV. operator. The last two forms of a logical expression indicate that in interpreting a logical expression containing two or more .EQV. or .NEQV. operators, the logical disjuncts are combined from left to right.

2.5.4.3. Logical Constant Expressions

A logical constant expression is a logical expression in which each primary is a logical constant, the symbolic name of a logical constant, a relational expression in which each primary is a constant expression, or a logical constant expression enclosed in parentheses. Variable, array element, and function references are not allowed. **ASCII FORTRAN allows the following nonstandard primary:**

- **logical intrinsic function references that have constant arguments**.

2.5.4.4. Logical Expression Evaluation

If a logical expression contains more than one logical operator, evaluation of that expression is performed using the following hierarchy:

Precedence	Operator
1	Relationals
2	.NOT.
3	.AND.
4	.OR.
5	EQV., .NEQV.

When more than one relational operator exists, they are performed in left-to-right order. The same applies to multiple appearances of the same operator.

Examples:

Assume A has the value .TRUE. and B the value .FALSE. Some logical expressions and their values are:

```
Expression                    Evaluation

 A .AND. B                   .FALSE.

 A .AND. .NOT. B              A .AND. .TRUE.
                             .TRUE.

 3 .LT. 4 .OR. B             .TRUE. .OR. B
                             .TRUE.

.NOT. A .AND. B              .FALSE. .AND. B
                             .FALSE.

.NOT. 3 .LE. 4 .OR. B        .NOT. .TRUE. .OR. B
                             .FALSE. .OR. B
                             .FALSE.

A .NEQV. B                   .TRUE.
```

2.5.5. Typeless Expressions

A typeless expression is a one-word (36-bit) string that results from using a typeless function with an argument of type integer, real, character (length of 4 or less), logical, or typeless.

2.5.5.1. Typeless Functions

The following primitive typeless functions are permitted in ASCII FORTRAN, where e_i is any single-word expression:

Expression	Explanation
AND (e_1,e_2)	Bit-by-bit logical product
OR (e_1,e_2)	Bit-by-bit logical sum
XOR (e_1,e_2)	Bit-by-bit exclusive OR
BOOL (e_1)	Treat e_1 as typeless
COMPL (e_1)	Bit-by-bit complement

If (e_i) is type character with length less than 4, the argument is space-filled on the right to make it length 4. If (e_i) is type character of unknown length, the argument is either truncated or space-filled to make it length 4.

2.5.5.2. Typeless Constant Expressions

A typeless constant expression is a one-word (36-bit) string that results from using a typeless function with an argument that is one of the following:

- an arithmetic constant expression of type integer or real
- a character constant expression (length 4 or less)
- a logical constant expression
- a typeless constant expression

2.5.5.3. Typeless Expression Evaluation

When an expression consists only of typeless expressions and arithmetic operators, or of typeless expressions and relational operators, the typeless expressions (bit strings) are treated as though they are integer values.

When an expression consists only of typeless expressions and logical operators, the typeless expressions are treated as though they are logical values. These

typeless expressions must have pure logical values for the logical operators to function correctly. Logical values use the rightmost bit for the value (1=.TRUE., 0=.FALSE.); the rest of the word must be zero.

When an expression (or part of an expression) is of the form e_1 *op* e_2, where one of the e_i is a typeless quantity and the other expression is of type integer, real, character, or logical, then assuming *op* is a suitable operator for comparing the two entities, the typeless quantity is taken to be integer, real, character, or logical, respectively, without being converted to that type. Don't combine a typeless expression with a complex or double-precision expression.

The order of expression evaluation is significant when typeless expressions are involved. For example, the following expressions are not the same:

```
(integer+typeless)+real
```

(typeless is taken as integer)

```
integer+(typeless+real)
```

(typeless is taken as real)

2.5.6. Hierarchy of Operators

The following hierarchy determines the evaluation order of subexpressions involving the operators mentioned in subsections of 2.5.

Precedence	Kind	Operation
1	Parenthesized	All
2	Functions	All
3	Arithmetic	Exponentiation (**)
4	Arithmetic	Multiplication and division (* and /)
5	Arithmetic	Addition and subtraction (+ and -) and unary operation
6	Character	Concatenation (// **and &**)
7	Logical	Relational comparisons (.GT., .GE., .LT., .LE., .EQ., .NE.)
8	Logical	.NOT.
9	Logical	.AND.

continued

Precedence	Kind	Operation
10	Logical	.OR.
11	Logical	.EQV., .NEQV.

Notes:

- *Expressions are evaluated left to right, except where the hierarchy dictates sotherwise, or in the case of successive exponentiation operators (see 2.5.1.6).*
- *Logical expressions may not require that all parts be evaluated. For example, if A is a logical variable whose value is .TRUE. and LEG is a logically valued function, the expression A .OR. LEG(x) may not result in a call to the function LEG. Since A has a true value, the value of the expression is already determinable.*

Section 3
Assignment Statements

3.1. General

Assignment statements are the basic mechanism by which the result of an expression is stored (and therefore, saved) in a variable, array element, or substring for future reference.

There are four types of assignment statements:

1. Arithmetic assignment, for storing the result of an arithmetic expression, **character constant, or Hollerith constant** in a numeric (integer, real, or complex) variable, array element, or **pseudo-function.**
2. Character assignment, for storing the result of a character expression in a character variable, character array element, character substring, or **pseudo-function.**
3. Logical assignment, for storing the .TRUE. or .FALSE. result of a logical expression in a logical variable or logical array element.
4. Statement label assignment (ASSIGN statement), for storing the location of a statement label in an integer variable. The unique form of this type of assignment statement unambiguously differentiates statement labels from numeric constants.

The equal sign (=) is the usual assignment operator.

Many types of conversions can occur during evaluation of the expression before the final result is stored. Special rules for conversion, size determination, and so on, apply in many cases. These are cited in the discussion of each type of assignment statement.

3.2. Arithmetic Assignment Statement

Purpose:

The arithmetic assignment statement is used to define an item of type INTEGER, REAL, DOUBLE PRECISION, COMPLEX, **or COMPLEX*16**.

Form:

```
v [,v] . . . = e
```

where:

v

is the name of a variable, array element, **or pseudo-function reference (see 7.7.2).**

e

is an arithmetic expression, **character constant, or Hollerith constant.**

Description:

The result of the expression is stored in the target variable**s** which appear to the left of the assignment (=) operator. When the result of the expression differs in type from the target variable**s**, conversions are performed in most cases as indicated in Table 3-1.

Exercise caution when using user-defined functions as array subscripts **or pseudo-function arguments.** Their order of evaluation cannot be relied upon.

The sequence of execution is as follows:

1. *e* is completely evaluated.
2. **For each target variable**, the result of the expression is converted, when necessary, as indicated in Table 3-1. **A character constant or Hollerith constant is stored to an arithmetic data item with no conversion performed regardless of the length of the literal item or the data item in which it is stored.**
3. The subscript expressions **or pseudo-function arguments** in *v* (if any) are evaluated. Their order of evaluation should not be relied upon.
4. The result is then stored in *v*.

The number of target variables in an arithmetic assignment statement is limited to 32.

Table 3-1. Conversion for Arithmetic Assignment

Target Type Desired	**Present Type of Expression**						
	INTEGER	REAL	DOUBLE PRECISION	COMPLEX	**COMPLEX *16**	Character	Logical
INTEGER	Store.	Fix. Store.	Fix.	Ignore imaginary part. Fix. Store.	**Ignore imaginary part. Fix. Store.**	**Not possible unless it's a char. or Hollerith constant, in which case, store.**	Not Possible
REAL	Float. Store.	Store.	Truncate to single. Store.	Ignore imaginary part. Store.	**Ignore imaginary part. Truncate to single. Store.**	**Not possible unless its a char. or Hollerith constant, in which case, store.**	Not Possible
DOUBLE PRECISION	Double. Store.	Expand to double. Store.	Store.	Ignore imaginary part. Expand to double. Store.	**Ignore imaginary part. Store.**	**Not possible unless its a char. or Hollerith constant, in which case, store.**	Not Possible

continued

Key

Fix	Is the same as applying the INT intrinsic function.
Float	Is the same as applying the REAL intrinsic function .
Double	Is the same as applying the DBLE intrinsic function.

Table 3-1. Conversion for Arithmetic Assignment (cont.)

Target Type Desired	**Present Type of Expression**						
COMPLEX	Float. Store to real part. Store 0 to imaginary part.	Store to real part. Store 0 to imaginary part.	Truncate single. Store to real part. Store 0 to imaginary part.	Store.	**Truncate both parts to single. Store.**	**Not possible unless its a char. or Hollerith constant, in which case, store.**	Not Possible
COMPLEX *16	**Double. Store to real part. Store 0 to imaginary part.**	**Expand to double. Store to real part. Store 0 to imaginary part.**	**Store to real part. Store 0 imaginary part.**	**Expand parts to double. Store.**	**Store.**	**Not possible unless its a char. or Hollerith constant, in which case, store.**	Not Possible

Key

Fix	Is the same as applying the INT intrinsic function.
Float	Is the same as applying the REAL intrinsic function .
Double	Is the same as applying the DBLE intrinsic function.

Examples:

```
               A = B + C
      C            The value of the expression B + C is stored in A.

               INDX = INDX + 1
      C            The value of the expression INDX + 1 is stored in INDX.

               X = CSIN(W * PI + THETA) - ORIGIN
      C            The value of the expression is stored in the variable X.

               N = 1
      C            The constant 1 is stored in N.
               END

      C            The following program sequence may depend upon side effects
      C            because of the order of evaluation and optimization.
      C            Since side effect dependence is forbidden, the sequence
      C            may not produce the results intended.
               DIMENSION A(10,10,10)
               COMMON/WATCH/MIGHT,CHANGE
               SUBJCT = 1
               MIGHT = 2
               CHANGE = 4
               A(MIGHT,FC(SUBJCT),CHANGE) = D
                 .
                 .
                 .
               END
      C
```

```
          FUNCTION FC(A)
          COMMON/WATCH/MIGHT,CHANGE
          MIGHT = MIGHT + 1
          CHANGE = CHANGE + 1
          FC = MIGHT + CHANGE - A
            .
            .
            .
          END
C             The example above shows the introduction of side effects during
C             evaluation of the function FC due to the COMMON properties of
C             MIGHT and CHANGE.  ASCII FORTRAN presently evaluates
C             A(MIGHT,FC(SUBJECT),CHANGE) to A(3,7,4).  However, you are
C             warned not to rely on side effects because a change in
C             compiler evaluation of the code could change the result.
```

3.3. Character Assignment Statement

Purpose:

The character assignment statement is used to define an item of type character with the result of a character expression.

Form:

```
c [,c] . . . = ce
```

where:

c

is the name of a character variable, character array element, character substring, **or pseudo-function reference (see 7.7.2).**

ce

is a character expression.

Description:

The result of the character expression is stored in the target variable**s** that appear on the left of the assignment (=) operator.

When the expression involves character variables, character array elements, character substrings, **or pseudo-functions**, the target *c* must be type character; no type conversions are performed. **When the expression is a character constant or Hollerith constant, the target variables can be of any type; no conversions are performed.**

When the length of the character string result of the expression differs from the size of the target variables, the string is truncated or padded with blanks as indicated in Table 3-2.

None of the character positions being defined in *c* may be referenced in *ce*.

Table 3-2. Conversions for Character Assignment

Condition	Action
$L_1 < L_2$	The leftmost L_1 characters of the expression result are stored in the target; the remaining characters of the expression result are ignored.
$L_1 = L_2$	The expression result is stored in the target.
$L_1 > L_2$	The expression result is stored in the leftmost L_2 characters of the target; the remaining rightmost L_1 minus L_2 characters are filled with blanks.

Legend

L_1	is the length of the target in characters.
L_2	is the length of the result of the character expression in characters.

The sequence of execution is the same as for arithmetic assignment statements.

The number of target variables in a character assignment statement is limited to 32.

Examples:

```
      CHARACTER*12 CSFMSG, COMMND
      CSFMSG = '@' // COMMND
C         The character string result of the concatenation expression
C         '@' // COMMND is stored in the character variable CSFMSG.

      CHARACTER BLANK*1
        .
        .
        .
      BLANK = ' '
C         The single blank is stored in character variable BLANK.
      END

      CHARACTER*4 S2(10,10)
        .
        .
        .
      S2(I,J) = 'abcd'
C         The string 'abcd' is stored in the character array
C         element S2(I,J)
      S2(I,J)(2:3) = 'ef'
C         The string 'ef' is stored in the substring in character
C         positions 2 through 3 of array element S2(I,J) which
C         makes S2(I,J) the string 'aefd'.
```

3.4. Logical Assignment Statement

Purpose:

The logical assignment statement transfers a logical value, either .TRUE. or .FALSE., to an item of type logical.

Form:

```
l [ ,l] . . . = le
```

where:

l

> is the name of a logical variable, logical array element, **or pseudo-function reference (see 7.7.2).**

le

> is a logical expression, **character constant, or Hollerith constant.**

Description:

The result of the logical expression is stored in the target variable**s** appearing to the left of the assignment (=) operator.

No conversions are performed. The expression must be a logical expression, and the target *l* must be type logical. **When *le* is a character constant or Hollerith constant (a special case), the first word of the constant is stored directly in the target, with no conversion.**

The sequence of execution is the same as for arithmetic assignment statements.

The number of target variables in a logical assignment statement is limited to 32.

Examples:

```
         LOGICAL INDIC, FLAG
         INDIC = LEFT .LT. RIGHT
C                The logical result of the logical expression
C                LEFT .LT. RIGHT is stored in the logical variable INDIC.

         FLAG = .TRUE.
C                The value .TRUE. is stored in the logical variable FLAG.
```

3.5. Statement Label Assignment (ASSIGN Statement)

Purpose:

The ASSIGN statement transfers the location of a statement label constant to a variable for subsequent reference in a GO TO statement or an I/O statement.

Form:

```
ASSIGN n TO iv
```

where:

n

is an unsigned positive integer indicating an executable or FORMAT statement label.

iv

is an unsubscripted integer variable.

Description:

The location of the statement is stored in the target variable that follows TO.

This statement is not the same as an arithmetic assignment. The value of the statement label is not stored and is not subsequently available, such as for output, in that form. Only its location is stored.

The target variable must be of type integer. No conversion is allowed.

The location of the indicated statement label is stored in the target variable in a single operation.

The target variable can be redefined with the same or different statement label value or an integer value.

Example:

```
      ASSIGN 10 TO ISTMT
C              The location of statement 10 is stored in variable ISTMT.
```

Section 4
Control Statements

4.1. General

Execution of a program unit is normally sequential, starting with the first executable statement and continuing with each successive statement through the last executable statement of the program unit.

Control statements change this normal sequence of execution. Some of these statements specify unconditional modifications of program flow, while others change the sequence of execution, depending on the results of a test contained within the statement.

FORTRAN control statements are:

- GO TO statements (unconditional GO TO, computed GO TO, assigned GO TO)
- IF statements (logical IF, arithmetic IF)
- Blocking statements (block IF, ELSE IF, ELSE, END IF)
- DO
- CONTINUE
- PAUSE
- STOP
- END
- CALL
- RETURN

CALL and RETURN are discussed with procedures in Section 7.

Both the control statement and the statement label referred to must be in the same program unit. A unique statement label can appear on any statement. However, there are rules as to which labels can be referenced (see 9.3.1).

4.2. GO TO Statements

GO TO statements transfer control to an executable statement specified by a statement label cited in the GO TO statement. Control is transferred either unconditionally or conditionally. Under certain circumstances this transfer can be within the boundaries of a DO statement. This is discussed further in NO TAG.

The three types of GO TO statements are:

1. Unconditional GO TO
2. Computed GO TO
3. Assigned GO TO

4.2.1. Unconditional GO TO

Purpose:

The unconditional GO TO statement transfers program control to a specified statement.

Form:

```
GO TO x
```

where *x* is the number (statement label) of an executable statement within the same program unit.

Description:

Each execution of an unconditional GO TO statement causes control to transfer to the statement specified by the statement label.

Any executable statement immediately following the GO TO statement should have a statement label. Otherwise, control can never reach such a statement.

Example 1:

```
C              Causes control to be transferred to the statement
C              labeled 180.
       GO TO 180
110    . . .
180    X = X + 1
```

Example 2:

```
C              If RENT is not greater than the current HIGH,
C              new occupants are added.  The total rent which is
C              collectable is the product of the number of occupants
C              times the cost of renting.
       IF (RENT .GT. HIGH) GO TO 100
       OCCUPY = OCCUPY + NEW
```

```
100    TOTAL = OCCUPY * RENT
```

Example 3:

```
C              This sample program uses the unconditional GO TO
C              statement to either add or subtract from a summation
C              of 10 numbers.
       SUM = 0.
       DO 1 l = 1, 10
C             This is a list-directed READ.
           READ *, K,J
           IF (K .EQ. 1) GO TO 14
           IF (K .EQ. 0) GO TO 15
           GO TO 1
    14     SUM = SUM + J
           GO TO 1
    15     SUM = SUM - J
     1  CONTINUE
        PRINT *, SUM
        STOP
        END
```

4.2.2. Computed GO TO

Purpose:

A computed GO TO statement transfers control to an indexed member of a statement label list.

Form:

```
GO TO (x [ ,x] . . . ) [ , ]e
```

or:

```
GO TO ( [x] [ , [x] ] . . . ) [ , ]e
```

where:

each *x*

is the statement label of an executable statement in the same program unit containing the GO TO statement, or **a variable containing such a number (see 3.5), or is omitted.** This list can have a maximum of 128 entries.

e

is an integer expression used to index a member of the statement label list. The value of *e* should not exceed the number of *x*'s appearing in the statement.

Description:

This statement causes control to transfer to the statement numbered *x* indexed by *e*.

x is chosen from the statement label list according to the value of *e*. When *e* = 1, the first statement label is used; when *e* = 2, the second is used; etc. When *e* is less than 1 or greater than the number of elements **(including empty positions)** in the list, **or if the selected *x* is omitted**, the execution sequence continues as though a CONTINUE statement were executed.

Example 1:

```
C              Control transfers to statement 10, 15, 20, or 25
C              depending on the value of CHOICE.  For example, if the
C              value of CHOICE is 2, control transfers to statement 15.
       INTEGER CHOICE
       GO TO (10,15,20,25), CHOICE
```

Example 2:

```
C               SALARY is:
C
C               1 if income is 8,000 or less
C               2 if greater than 8,000 but less than or equal to 12,000
C               3 if greater than 12,000 but less than or equal to 20,000
C               4 if greater than 20,000 but less than or equal to 80,000
C               5 if greater than 80,000
C
C               Control transfers to the proper addition depending
C               on the value of SALARY.
       INTEGER SALARY, UPCLAS, WEALTH, RICH
       GO TO (100,200,300,400,500) , SALARY
100    LOW = LOW + 1
        .
        .
        .
200    MIDDLE = MIDDLE + 1
        .
        .
        .
300    UPCLAS = UPCLAS + 1
        .
        .
        .
400    WEALTH = WEALTH + 1
        .
        .
        .
500    RICH = RICH + 1
        .
        .
        .
```

Example 3:

```
C                 This program uses the computed GO TO statement to
C                select one of four summation types
C                (sum of the actual values, sum of base e logarithms,
C                sum of base 10 logarithms, and sum of reciprocals).
       SUM = 0.0
       READ *, N, KNODE
       DO 1 I = 1, N
           READ *, X
           GO TO (6, 7, 8, 9), KNODE
6          SUM = SUM + X
```

```
             GO TO 1
    7        SUM = SUM + ALOG(X)
             GO TO 1
    8      SUM = SUM + ALOG10(X)
           GO TO 1
    9      SUM = SUM + 1. / X
    1   CONTINUE
        PRINT *, N, KNODE, SUM
        STOP
        END
```

4.2.3. Assigned GO TO

Purpose:

An assigned GO TO statement transfers control to the statement whose label is equal to the current value of a specified variable.

Form:

```
GO TO m [[ ,] (x[ ,x] . . . )]
```

where:

m

is a scalar integer variable (not an array element). Its value must equal one of the values of *x*, unless you omit the statement label list.

x

is the statement label of an executable statement in the program unit containing the GO TO statement. This list can have a maximum of 128 entries.

Description:

At the time of execution of an assigned GO TO statement, the current value of *m* must be defined as one of the statements *x* by the previous execution of an ASSIGN statement **or by initialization in a DATA or type specification statement**. The ASSIGN statement is explained in 3.5. The DATA statement is discussed in 6.12. The value of *m* must identify a statement in the same main program, subroutine, or function as the GO TO statement. No diagnostic is issued for failure to do so.

Any executable statement immediately following this statement must have a statement label. Otherwise, control can never reach it.

Logically, the assigned GO TO statement can be used whenever a computed GO TO is used (see 4.2.2). The formats differ and the assigned GO TO requires at least one previous ASSIGN statement.

When you omit the statement label list, the assumed list contains every statement label that is associated with the variable *m* in an ASSIGN, **DATA initialization, or type specification** statement. The result of executing the GO TO statement when *m* does not identify a statement in the list depends on the degree of program optimization being performed, and is not generally predictable.

Example 1:

```
C                      Control transfers to statement in list whose
C                      value matches that of CHOICE.  For example, when the
C                      last value assigned to CHOICE was 25, control
C                      passes to statement 25.
         INTEGER CHOICE
         GO TO CHOICE, (10,15,20,25)
```

Example 2:

```
C                     This sample program uses the assigned GO TO statement
C                     to arrange a selection from one of four types of
C                     summation (sum of the actual values, sum of base e
C                     logarithms, sum of base 10 logarithms, and sum
C                     of reciprocals).
        SUM = 0.0
        READ *, N, KNODE
        IF (KNODE .EQ. 6) ASSIGN 6 to KSWTCH
        IF (KNODE .EQ. 7) ASSIGN 7 to KSWTCH
        IF (KNODE .EQ. 8) ASSIGN 8 to KSWTCH
        IF (KNODE .EQ. 9) ASSIGN 9 to KSWTCH
        DO 1 I = 1, N
            READ *, X
            GO TO KSWTCH, (6, 7, 8, 9)
6           SUM = SUM + X
            GO TO 1
7           SUM = SUM + ALOG(X)
            GO TO 1
8           SUM = SUM + ALOG10(X)
            GO TO 1
9           SUM = SUM + 1. / X
1     CONTINUE
      PRINT *, N, KNODE, SUM
      STOP
      END
```

Example 3:

```
C                   This example demonstrates an incorrect usage of J.
C                   The GO TO is illegal because the statement
C                   number assigned to J is not associated with J in
C                   an ASSIGN or by initialization in a DATA or type
C                   specification statement.  The example would be correct
C                   if the GO TO statement were: GO TO J, (10,20,30).
       INTEGER GOTO(3), GOTO1, GOTO2, GOTO3
       EQUIVALENCE (GOTO(1), GOTO1), (GOTO(2), GOTO2), (GOTO(3), GOTO3)
       ASSIGN 10 TO GOTO1
       ASSIGN 20 TO GOTO2
       ASSIGN 30 TO GOTO3
       READ *, I
       J = GOTO(I)
       GO TO J
```

4.3. IF Statements

IF statements are the decision-making elements in FORTRAN. They test relationships within the statement and can modify the normal sequence of execution based on the result of this test.

There are three types of IF statements:

1. Logical IF
2. Arithmetic IF
3. Block IF

The block IF statement is described with the rest of the blocking statements in 4.4.

4.3.1. Logical IF

Purpose:

The logical IF statement evaluates a logical expression and executes or skips a specified statement, depending on whether the value of the expression is true or false, respectively.

Form:

```
IF (l) s
```

where:

l

is any logical expression.

s

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

Description:

Statement *s* is executed if expression *l* is true. When *l* is false, the execution sequence continues as though a CONTINUE statement were executed. Although *s* itself is syntactically a complete statement, it must appear in the same line or set of continuation lines as the clause IF (*l*).

Example 1:

```
C                     When RENT is less than or equal to the desired
C                     HIGH, the number of new occupants is increased
C                     by 1 before the new rent total is computed.
         IF (RENT .LE. HIGH) OCCUPY = OCCUPY + 1
         TOTAL = OCCUPY * RENT
```

Example 2:

```
C                     The following program sums positive values of B
C                     using a logical IF statement.
      SUM = 0
      DO 2 I = 1,50
         READ *, B
         IF (B .LE. 0.) GO TO 2
         SUM = SUM + B
2     CONTINUE
      PRINT *, SUM
      STOP
      END
```

Example 3:

```
C                     This accomplishes the same thing using the greater
C                     than (.GT.) operator.
      SUM = 0
      DO 2 I = 1,50
         READ *, B
2        IF (B .GT. 0.) SUM = SUM + B
      PRINT *, SUM
      STOP
      END
```

4.3.2. Arithmetic IF

Purpose:

The arithmetic IF statement acts as a multi-destination branch depending on the condition which is satisfied.

Form:

IF (*a*) x_1, x_2, x_3

or:

IF (*a*) [x_1] [, [x_2] [, [x_3]]]

where:

a

is an arithmetic expression of any type except complex (that is, *a* is type integer, real, double precision, or **typeless**).

each *x*

is the statement label of an executable statement in the program unit containing the IF statement, **or an integer scalar variable that is assigned a statement label using the ASSIGN statement (see 3.5), type statement, or DATA statement. It can also be omitted.**

Description:

The arithmetic IF statement transfers control to the statement numbered x_1, x_2, or x_3 when the value of the arithmetic expression (a) is less than, equal to, or greater than zero, respectively.

Any two, or all three, statement labels can be the same. When all three are the same, the statement has the same effect as an unconditional GO TO.

An executable statement immediately following this statement should have a statement label. Otherwise, control can never reach it. **However, an omitted position in the list x_1, x_2, x_3, such as:**

```
IF (A) 5,,6
```

implies a reference to the succeeding statement.

When any *x* is specified as an integer scalar variable, all three positions must be present (that is, two commas must appear in the list.

Example:

```
C                          The basic use of the arithmetic IF statement is to
C                          discriminate between negative, zero, and positive
C                          values for variables or expressions.
        SUM = 0
        DO 2 I = 1,50
           READ *, B
           IF (B) 2,2,1
1          SUM = SUM + B
2       CONTINUE
        PRINT *, SUM
        STOP
        END
```

4.4. Blocking Statements

The IF-THEN-ELSE blocking structure of FORTRAN permits blocks of statements to be executed on a conditional basis. There are four blocking statements:

- Block IF
- ELSE IF
- ELSE
- END IF

With the proper use of this feature, the GO TO statement is seldom necessary. This allows more structure in FORTRAN programs, making them more reliable and understandable.

An IF-THEN-ELSE block has the following structure:

```
IF (e) THEN
    [block of executable statements]   (This is an IF-block)
[ELSE IF (e) THEN]
    [block of executable statements]   (This is an ELSE IF-block)
       .
       .
       .
    [additional ELSE IF-blocks]
       .
       .
       .
[ELSE]
    [block of executable statements] (This is an ELSE-block)
END IF
```

where *e* is a logical scalar expression and the following rules apply:

1. A block begins with a block IF statement and ends with an END IF statement.
2. Between the two statements, ELSE IF statements and one ELSE statement can appear.
3. The ELSE statement (when it appears) must follow all ELSE IF statements (if any) in the block.
4. Executable statements can appear in the blocks between the blocking statements.
5. Blocks can be nested.

Subsection 4.4.5 contains examples that show the FORTRAN IF-THEN-ELSE blocking statements. The examples use all of the blocking statements (block IF, ELSE IF, ELSE, and END IF), and also show the different types of blocks (IF-block, ELSE IF-block, ELSE-block) and IF-level nesting. Indention of lines in the source code makes programs with blocking statements more readable.

4.4.1. Block IF Statement

Purpose:

The block IF statement is used with the END IF statement and, optionally, the ELSE IF and ELSE statements, to control the execution sequence.

Form:

```
IF (e) THEN
```

where *e* is a logical expression.

4.4.1.1. IF-Level

The IF-level of a statement *s* is:

```
n1 - n2
```

where n_1 is the number of block IF statements from the beginning of the program unit up to and including *s*, and n_2 is the number of END IF statements in the program unit up to but not including *s*.

The IF-level of every statement must be 0 or positive. The IF-level of each block IF, ELSE IF, ELSE, and END IF statement must be positive. The IF-level of the END statement of each program unit (or the last statement in the program unit, unless the last statement is END IF) must be 0.

The maximum IF-level allowed at any point in a program unit is 25.

4.4.1.2. IF-Block

An IF-block consists of all the executable statements after the block IF statement up to, but not including, the next ELSE IF, ELSE, or END IF statement that has the same IF-level as the block IF statement. An IF-block can be empty.

4.4.1.3. Block IF Statement Execution

Execution of a block IF statement causes evaluation of the expression *e*. When the value of *e* is true, normal execution continues with the first statement of the IF-block. When the value of *e* is true, and the IF-block is empty, control transfers to the next END IF statement that has the same IF-level as the block IF statement. When the value of *e* is false, control transfers to the next ELSE IF, ELSE, or END IF statement that has the same IF-level as the block IF statement.

Transfer of control into an IF-block from outside the IF-block is allowed but is not recommended.

When execution of the last statement in the IF-block doesn't result in a transfer of control, control transfers to the next END IF statement that has the same IF-level as the block IF statement that immediately precedes the IF-block.

4.4.2. ELSE IF Statement

Form:

```
ELSE IF (e) THEN
```

where *e* is a logical expression.

4.4.2.1. ELSE IF-Block

An ELSE IF-block consists of all the executable statements after the ELSE IF statement up to, but not including, the next ELSE IF, ELSE, or END IF statement that has the same IF-level as the ELSE IF statement. An ELSE IF-block can be empty.

4.4.2.2. ELSE IF Statement Execution

Execution of an ELSE IF statement causes evaluation of the expression *e*. When the value of *e* is true, normal execution sequence continues with the first statement of the ELSE IF-block. When the value of *e* is true and the ELSE IF-block is empty, control transfers to the next END IF statement that has the same IF-level as the ELSE IF statement. When the value of *e* is false, control transfers to the next ELSE IF, ELSE, or END IF statement that has the same IF-level as the ELSE IF statement.

Transfer of control into an ELSE IF-block from outside the ELSE IF-block is allowed but is not recommended. The statement label, if any, of the ELSE IF statement must not be referred to by any statement **except DELETE.**

When execution of the last statement in the ELSE IF-block doesn't result in a transfer of control, control transfers to the next END IF statement that has the same IF-level as the ELSE IF statement that immediately precedes the ELSE IF-block.

4.4.3. ELSE Statement

Form:

```
ELSE
```

4.4.3.1. ELSE-Block

An ELSE-block consists of all the executable statements after the ELSE statement up to, but not including, the next END IF statement that has the same IF-level as the ELSE statement. An ELSE-block can be empty.

4.4.3.2. ELSE Statement Execution

Execution of an ELSE statement has no effect. Normal execution sequence continues.

Transfer of control into an ELSE-block from outside the ELSE-block is allowed but is not recommended. The statement label, if any, of an ELSE statement must not be referred to by any statement, **except DELETE.**

4.4.4. END IF Statement

Form:

```
END IF
```

Description:

Execution of an END IF statement has no effect. Normal execution sequence continues.

For each block IF statement, there must be a corresponding END IF statement in the same program unit. A corresponding END IF statement is the next END IF statement that has the same IF-level as the block IF statement.

4.4.5. Examples Using Blocking Statements

Example 1:

```
C                 In this example, when A and B are equal,
C                 the three statements in the IF-block (that is, the
C                 statements between the block IF and END IF statements)
C                 are executed, and then the statement after the END IF is
C                 executed.  If A and B are not equal, then the
C                 statement after the END IF is executed.
C
           IF (A .EQ. B) THEN
                  I = I + 1
                  A = B + C
                  CALL SUB1 (I,A)
           END IF
              .
              .
              .
```

The following three sequences of statements (examples 2 through 4) are equivalent. L1 and L2 are logical scalar variables in the following examples. The examples cause I and J to be set depending on the values of L1 and L2. If L1 is .TRUE., then I and J are set to 1 and 2, respectively. If L1 is .FALSE. and L2 is .TRUE., then I and J are set to 2 and 3, respectively. If L1 and L2 are both .FALSE., then I and J are set to 3 and 4, respectively.

Example 2:

```
C                 This example uses all of the blocking statements:
C                 BLOCK IF, ELSE IF, ELSE, and END IF.  The
C                 maximum IF-level is one.
C
           IF (L1) THEN
             I = 1
             J = 2
           ELSE IF (L2) THEN
                 I = 2
                 J = 3
           ELSE
             I = 3
             J = 4
           END IF
```

Example 3:

```
C                   This example uses all of the blocking statements
C                   except ELSE IF.  The maximum IF-level is two.
C
           IF (L1) THEN
                I = 1
                J = 2
           ELSE
           IF (L2) THEN
                I = 2
                J = 3
           ELSE
                I = 3
                J = 4
           END IF
       END IF
```

Example 4:

```
C                    This example uses none of the blocking statements.
       IF (.NOT. L1) GO TO 10
       I = 1
       J = 2
       GO TO 30
10     IF (.NOT. L2) GO TO 20
       I = 2
       J = 3
       GO TO 30
20     I = 3
       J = 4
30     CONTINUE
```

4.5. DO Statement

Purpose:

Use a DO statement to specify a loop, called a DO-loop. This loop controls repeated execution of a set of executable statements.

Form:

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

where:

s

is the statement label of an executable **or FORMAT** statement. The statement identified by *s*, called the terminal statement of the DO-loop, follows the DO statement in the sequence of statements within the same program unit as the DO statement. The terminal statement of a DO-loop must not be an unconditional GO TO, assigned GO TO, arithmetic IF, block IF, ELSE IF, ELSE, END IF, RETURN, STOP, END, or DO statement.

i

is the symbolic name of an integer, real, or double-precision variable, called the DO-variable.

e_1

is an integer, real, or double-precision expression indicating the initial value for the DO-variable *i*.

e_2

is an integer, real, or double-precision expression indicating the terminal test value for the DO-variable *i*.

e_3

is an integer, real, or double-precision expression indicating the increment value for the DO-variable *i*. e_3 must not be 0. When e_3 is omitted, it is assumed to have a value of 1.

4.5.1. DO-Loop Range

The range of a DO-loop consists of all of the executable statements that follow the DO statement, up to and including the terminal statement of the DO-loop. **When the terminal statement is nonexecutable (that is, FORMAT), the range of the DO-loop isn't extended to include the next executable statement.**

When a DO statement appears within an IF-block, ELSE IF-block, or ELSE-block, the range of that DO-loop is contained entirely within that IF-block, ELSE IF-block, or ELSE-block, respectively.

When a block IF statement appears within the range of a DO-loop, the corresponding END IF statement must also appear within the range of that DO-loop.

4.5.2. Nested DO-Loops

DO statements can appear within the range of DO-loops. When a DO statement is placed within the range of a DO-loop, observe the following rules:

- The range of the second DO-loop must be entirely within the range of the first. The encompassing DO-loop is called the outer DO-loop. The nested DO-loop is called the inner DO-loop.
- The range of an inner DO-loop can, however, contain the last statement in the range of the next outer DO-loop. When DO-loops are nested in this manner, only iterations of the inner DO-loop can be terminated by a transfer of control to the last statement of the loop.

Such a set of DO-loops is called a DO-nest. DO-loops and implied DO-loops can be nested to a maximum depth of 25 loops.

A nest of DO-loops is considered completely nested when no loop in the nest terminates before the last loop begins. For example, the following two nests are completely nested:

```
        DO 50 I = 1, 4
           A(I) = B(I)**2
           DO 50 J = 1, 5
50            C(1,J) = A(I)
        DO 10 I = L, M
           N = I + K
           DO 15 J = 1, 100, 2
15            TABLE(J,I) = SUM(J,N)-1
10         B(N) = A(N)
```

The following DO-loops are not completely nested:

```
        DO 100, I = 1, 10
           DO 200, J = 2, 12
             .
             .
             .
200        CONTINUE
           DO 300, K = 1, 10
             .
             .
             .
300        CONTINUE
100     CONTINUE
```

The following is not properly nested and is in error:

```
        DO 100, 1 = 1, 10
        M = I + N
        DO 200, J = 10, 1, -1
100     INDEX(I,J) = TABLE (M,J)
200     A(J) = Z(N-J)
```

4.5.3. Active and Inactive DO-Loops

A DO-loop is either active or inactive. Initially inactive, a DO-loop becomes active only when its DO statement is executed.

Once active, the DO-loop becomes inactive only when:

- its iteration count is tested (4.5.4.2) and determined to be zero;
- a RETURN statement is executed within its range;
- control transfers to a statement that is in the same program unit and is outside the range of the DO-loop;
- any STOP statement in the executable program is executed; or
- execution is terminated for any other reason (for example, error or end-of-file conditions on I/O statements).

An active DO-loop does not become inactive when a function reference or CALL statement appearing in the range of a DO-loop is executed, except when control returns by means of an alternate return specifier (that is, RETURN *i*) to a statement that is not in the range of the DO-loop.

When a DO-loop becomes inactive, the DO-variable of the DO-loop retains its last defined value.

4.5.4. DO-Loop Execution

Execution of a DO-loop involves the following steps:

- Executing the DO statement
- Loop control processing
- Execution of the range
- Terminal statement execution
- Incrementation processing

These steps are described in the paragraphs that follow.

4.5.4.1. DO Statement Execution

The following steps are performed in sequence when a DO statement is executed:

1. The initial parameter m_1, the terminal parameter m_2, and the incrementation parameter m_3 are established by evaluating e_1, e_2, and e_3, respectively. When necessary, this includes conversion to the type of the DO-variable according to the rules for arithmetic conversion. When e_3 is not specified, m_3 is assumed to be 1.
2. The DO-variable is defined by the value of the initial parameter m_1.
3. The iteration count is established and is the value of the expression:

   $$\texttt{MAX(INT((}m_2\texttt{ - }m_1\texttt{ + }m_3\texttt{) / }m_3\texttt{), 0)}$$

   The iteration count is 0 (that is, the DO-loop range is executed zero times) when:

   $$m_1 > m_2 \text{ and } m_3 > 0$$

   or:

   $$m_1 < m_2 \text{ and } m_3 < 0.$$

   Since the iteration count is calculated only once (before entry into the loop), variables appearing in the expressions e_1, e_2, and e_3 can be changed during execution of the loop with no effect on the iteration count.

   When noninteger expressions for e_1, e_2, and e_3 are used, the iteration count may be different than expected due to the approximate nature of real values and the integer conversion operation.

On completion of execution of the DO statement, loop control processing begins.

4.5.4.2. Loop Control Processing

Loop control processing determines if further execution of the range of the DO-loop is required. The iteration count is tested. If it isn't zero, execution of the first statement in the range of the DO-loop begins.

When the iteration count is zero, the DO-loop becomes inactive. When all of the DO-loops sharing the terminal statement of this DO-loop are inactive, normal execution continues with execution of the next executable statement following the terminal statement. However, when some of the DO-loops sharing the terminal statement are active, execution continues with incrementation processing (see 4.5.4.5).

4.5.4.3. DO-Loop Range Execution

Statements in the range of a DO-loop are executed until the terminal statement is reached. Except by incrementation (see 4.5.4.5), the DO-variable of the DO-loop can neither be redefined nor become undefined during execution of the range of the DO-loop. It is your responsibility to ensure that such redefinitions don't occur.

4.5.4.4. Terminal Statement Execution

Execution of the terminal statement occurs as a result of the normal execution sequence or as a result of transfer of control. Unless execution of the terminal statement results in a transfer of control, execution then continues with incrementation processing.

4.5.4.5. Incrementation Processing

The following steps are performed in sequence when incrementation processing occurs:

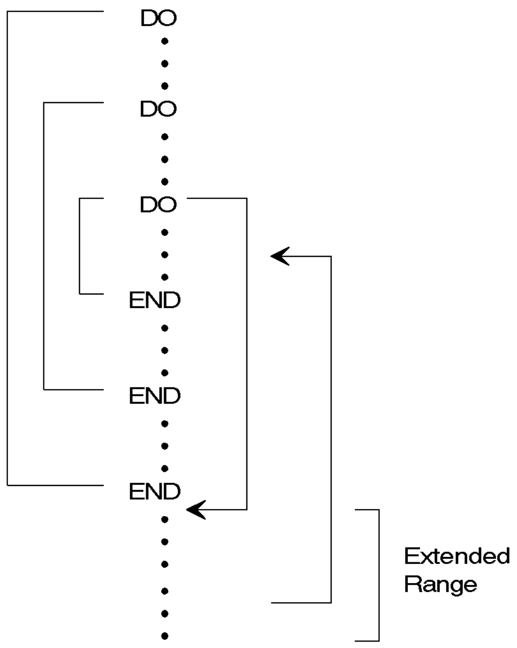
1. The DO-variable, the iteration count, and the incrementation parameter of the active DO-loop whose DO statement was most recently executed are selected for processing.
2. The value of the DO-variable is increased by the value of the incrementation parameter m_3.
3. The iteration count is decreased by one.
4. Execution continues with loop control processing (see 4.5.4.2) of the same DO-loop whose iteration count was decreased.

```
Extended Range of a DO-Loop
```

The extended range of a DO-loop is defined as those statements that are executed between the transfer out of the innermost DO-loop of a set of completely nested DO-loops and the transfer back into the range of this innermost DO-loop. In a set of completely nested DO-loops, the first DO-loop is

not in the range of any other DO-loop, and each succeeding DO-loop is in the range of every DO-loop that precedes it. The following restrictions apply:

- Transfer into the range of a DO-loop is permitted only when the transfer is from the extended range of the DO-loop.
- The extended range of a DO-loop must not contain another DO statement that has an extended range if the second DO statement is in the same program unit as the first.
- The DO-variable can't be changed in the extended range of the DO-loop.
- Transfer into extended range must be from the innermost loop. See the following illustration:



- A statement that is the end of the range of more than one DO-loop is within the innermost DO-loop. The statement label of such a terminal statement can't be used in any GO TO or arithmetic IF statements that occur anywhere but in the range of the most deeply contained DO-loop with that terminal statement.
- The use of, and return from, a subprogram (see 7.1) from within any DO-loop in a nest of DOs, or from within an extended range, is permitted. However, when the DO-variable of the loop is passed to the subprogram, the subprogram should not alter its value.

Note: The extended range of a DO-loop is not considered good programming practice. Try not to use it in new programs.

4.5.5. Availability of the DO-Variable Value

When optimization is not selected, the memory location associated with the DO-variable of a DO-loop always contains the current value of the DO-variable.

When optimization is selected, the DO-variable of a DO-loop (as well as other variables) can be maintained in a machine register. Consequently, the storage location associated with the variable may not always contain the current value of the variable. This is not noticeable to you unless the storage location associated with the variable is dumped. The variable is in storage when a reference that requires use of the variable is performed.

The following types of references require that the variable be used from storage:

- The variable appears in an input/output list within the loop other than as part of a subscript.
- The variable is used as an argument of a subprogram referenced within the loop.
- The variable is used outside the loop before being redefined and there is a branch to a statement outside the range of the DO-loop.

4.5.6. DO-Loop Examples

Example 1:

```
C        In this example, execution of the DO statement causes the
C        initialization of (1) the DO-variable I to 1, and (2) the
C        iteration count to 50.  After statement 100 is
C        executed each time through the loop, incrementation
C        processing causes (1) I to be incremented by 1, and
C        (2) the iteration count to be reduced by 1.  After
C        50 executions of the DO-loop, incrementation processing
C        causes (1) I to be set to 51, and (2) the iteration
C        count to be set to 0.  Loop control processing then
C        causes the DO-loop to become inactive, thus causing
C        execution of the third statement.  It contains the value 51.
      DO 100 I = 1,50
100      CURRENT(I) = CURRENT(I) - OUT(I)
      X = Y * Z
```

Example 2:

```
C        In this example, execution of the DO statement causes the
C        initialization of (1) the DO-variable J to 10, and (2) the
C        iteration count to 5.  Incrementation processing
C        (after statement 200) causes (1) J to be incremented by
C        -2, and (2) the iteration count to be reduced by 1.
C        After five executions of the DO-loop, incrementation
C        processing causes (1) J to be set to 0, and (2) the
C        iteration count to be set to 0.  Loop control processing
C        then causes the DO-loop to be inactive, thus causing
C        execution of the fourth statement.  J contains the value 0.
      DO 200, J = 10,1,-2
         I = J + K
200      ROW (I) = COL(I)
      X = Y * Z
```

Example 3:

```
C        After execution of these statements and at the execution
C        of the CONTINUE statement, I=11, J=10, K=6, L=5, and N=50.
     N = 0
     DO 100 I = 1,10
        J = I
        DO 100 K = 1,5
            L = K
100         N = N + 1
101  CONTINUE
```

Example 4:

```
C        After execution of these statements and at the execution of
C        the CONTINUE statement, I=11, J=10, K=5, and N=0.  The
C        value of L is not changed by these statements, since the
C        iteration count for the innermost loop is 0
C        (that is, the innermost loop is traversed zero times,
C        since the increment value is not specified and
C        is assumed to be one.  See 4.5.4.1).
     N = 0
     DO 200 I = 1,10
        J = I
        DO 200 K = 5,1
            L = K
200         N = N + 1
201  CONTINUE
```

Example 5:

```
C        The following example demonstrates some of the effects of
C        real-valued DO loops and subscripts.  Note the effects
C        of using a rounding factor.
    DIMENSION I(12)
    DATA I/1,2,3,4,5,6,7,8,9,10,11,12/
    A = 2.4
    B = 4.0
    C = 3.0
    DO 100 R = A/2.0, B * C, 1.2
100       PRINT *, I(R), I(R + .0001), R
    DO 200 R = A/2.0, B * C + .0001, 1.2           @ rounding factor
200       PRINT *, I(R), I(R + .0001), R
     END
```

```
   B*C       B*C+.0001
 Without       With
Rounding     Rounding
 Factor       Factor          R

   1            1          1.2000000
   2            2          2.4000000
   3            3          3.6000000
   4            4          4.8000000
   5            6          5.9999999
   7            7          7.1999999
   8            8          8.3999999
   9            9          9.5999998
  10           10          10.800000
   1            1          1.2000000
   2            2          2.4000000
   3            3          3.6000000
   4            4          4.8000000
   5            6          5.9999999
   7            7          7.1999999
   8            8          8.3999999
   9            9          9.5999998
  10           10          10.800000
  11           12          12.000000
```

4.6. CONTINUE Statement

Purpose:

The CONTINUE statement serves as a point of reference in a FORTRAN program and the effect of its execution is that no operational function is performed.

Form:

```
CONTINUE
```

Description:

The CONTINUE statement doesn't perform any executable function, but acts as a dummy executable statement.

You can place this statement anywhere in the source program where an executable statement can appear. It doesn't affect the program execution sequence.

A statement label is usually used with the CONTINUE statement. In this way, it provides a point to which control is transferred without implying any executable action.

CONTINUE is primarily used as the terminal statement of a DO-loop range. A transfer of control from any point within the loop to the CONTINUE statement allows the completion of an iteration of the loop without specifying an additional action. Use of the CONTINUE statement in this way facilitates program updating.

Example:

```
C                       This example illustrates the use of CONTINUE as the
C                       terminal statement for a DO-loop.
          DO 29 I = 1,10
                 .
                 .
                 .
             IF (A) 29,38,34
29
          CONTINUE
```

4.7. PAUSE Statement

Purpose:

The PAUSE statement temporarily suspends execution of a demand (interactive) program.

Form:

where:

```
PAUSE [ {n|message} ]
```

where:

n

is a string of **one to six** decimal digits. (The FORTRAN 77 standard allows a string of one to five decimal digits.)

message

is a character constant. The maximum length of *message* is 124 characters.

Description:

A PAUSE of either form temporarily halts execution of the program.

The program waits until you transmit a carriage return; the program then resumes execution, starting with the next statement after the PAUSE statement.

PAUSE *n*, PAUSE *message*, or PAUSE 00000 are displayed on the demand terminal, depending upon whether *n*, *message*, or neither was specified, respectively. Program execution is suspended until input is received from the terminal.

When the program is not in demand mode, the PAUSE *n*, the PAUSE *message*, or PAUSE 00000 is displayed on the system print file, and program execution continues.

Example:

```
C              This example causes execution to pause at statement 90
C              if A=0.  When the user causes the program to resume
C              execution, the next statement executed is 'GO TO 180'.
60       IF (A) 80,90,110
70       CONTINUE
80       STOP 'A IS NEGATIVE'
90       PAUSE 'A IS 0'
100      GO TO 180
110      A = B**2 + C**3
120      GO TO 60
```

4.8. STOP Statement

Purpose:

STOP terminates the execution of the program.

Form:

```
STOP [ {n|message} ]
```

where:

n

is a string of **one to six** decimal digits. (The FORTRAN 77 standard allows a string of one to five decimal digits.)

message

is a character constant. The maximum length of *message* is 124 characters.

Description:

Execution of the STOP statement terminates the execution of the program.

When *n* or *message* are specified, STOP *n* or STOP *message* are displayed on the system print file. Otherwise, nothing is displayed.

All open files are closed as part of program termination.

Example:

```
C                  Execution of this program ceases at statement label 80
C                  when A is negative.
60         IF (A) 80,90,110
70         CONTINUE
80         STOP 'A IS NEGATIVE'
90         PAUSE 'A IS 0'
100        GO TO 180
110        A = B**2 - C**3
120        GO TO 60
```

4.9. END Statement

Purpose:

The END statement indicates the end of a program unit. **In ASCII FORTRAN, the END statement indicates the end of compilation for a particular group of program units.**

Form:

```
END
```

Description:

The END statement terminates a group of program units, consisting of an external program unit and zero or more internal subprograms. An END statement must appear at the end of one of the following:

- an external program unit (main program, or external function or subroutine) **that has no associated internal subprograms**,
- **the last internal subprogram associated with a given external program unit, or**
- a BLOCK DATA program.

When present, the END statement must physically be the last statement in a program unit. **It directs the compiler to compile the preceding statements as one group of program units. It also indicates that the following program unit (if any) is an external program unit.**

The execution of an END statement implies a RETURN in a subprogram or a STOP in a main program.

When the source input to the compiler contains more than one program unit, each external program unit must be terminated by an END statement, a FUNCTION statement, or a SUBROUTINE statement. The FUNCTION and SUBROUTINE statements in this case signal the start of an internal subprogram as well as the end of the previous program unit (see 7.3.2 and 7.3.3).

When the END statement is missing, the next control image signals the physical end of the source input to the compiler.

Section 5
Input/Output Statements

5.1. General

Input statements obtain data for program use from input files. Output statements store results produced in the program in output files. These files can be information storage files in the computer system (internal storage) or they can be devices such as keyboards, printers, display terminals, or other peripheral devices. Therefore, you can use input/output statements to transfer data:

- from internal storage to an output device,
- from an input device to internal storage, or
- from internal storage to internal storage.

Input/output statements can be organized according to the access method or location of the file being manipulated:

1. Sequential
2. Direct
3. Internal

The sequential input/output statements are READ, WRITE, PRINT, **PUNCH**, BACKSPACE, ENDFILE, REWIND, OPEN, CLOSE, INQUIRE, **and DEFINE FILE.** Use them to access files sequentially. When you use sequential input/output statements, it is not possible to read, for example, the seventh record of a file directly; it is necessary to indicate that the preceding six records have been passed over. Once the seventh record is read, it may be impossible (as in the case of a card reader) to go back and reread the fourth record. However, this can be done where the BACKSPACE or REWIND statement is effective (as on magnetic tape).

Direct input/output statements are READ, WRITE, **FIND**, OPEN, CLOSE, INQUIRE, **and DEFINE FILE.** They refer to random access files in any order based on the record number.

The internal input/output statements are READ, WRITE, **ENCODE, and DECODE.** Use them for internal storage to internal storage transfers.

Two types of records (formatted and unformatted) can be read or written depending on the I/O statement used. The formatted type of record can be further divided into format-directed, list-directed, and name-directed forms. The format-directed form is called formatted, while the name-directed form is called namelist.

Record Forms	I/O Statements Allowed
Formatted	Sequential, direct, internal
Namelist	**Sequential**
List-directed	Sequential, **internal (ENCODE and DECODE only)**
Unformatted	Sequential, direct

The form of record that is read or written is determined from the form of the READ or WRITE statement.

Formatted records are read or written under the control of a FORMAT statement which describes the characteristics of the data transferred. See 5.3 for a description of the FORMAT statement.

Namelist records are read or written under the control of the NAMELIST statement with an implied format control. See 5.4 for details of the NAMELIST statement.

List-directed records are read or written with an implied format control. No FORMAT statement is required. See 5.5 for a description of list-directed input/output.

Unformatted records are read or written without format control.

Each execution of an input/output statement processes a new record.

All character data is assumed to be in the ASCII character code.

The default record sizes are shown in Table 5-1. To exceed the default record sizes, use an OPEN **or DEFINE FILE** statement.

Table 5-1. Default Record Sizes

Type of Record	Record Size
APRINT-APRNTA-AREADA symbionts	132 characters
AREAD symbionts	80 characters
APUNCH-APNCHA symbionts	80 characters

continued

Table 5-1. Default Record Sizes (cont.)

Type of Record	Record Size
System Data Format (SDF) files	33 words (132 characters)
ANSI files	132 characters

In addition to the information in this section, more detailed information on input/output appears in Appendix G.

5.2. Elements of Input/Output Statements

Input/output statements are composed of FORTRAN keywords (READ, WRITE, etc.) and a list that contains control information about an input or output operation. The components of this control information list are called control specifiers.

The control specifiers allowed on an input/output statement depend on the type of input/output statement used, the type of file access, and the type of record desired. The various control specifiers are:

- Unit specifier (UNIT=)
- Record specifier (REC=)
- Input/output list
- Format specifier (FMT=)
- **Namelist name specifier**
- Error specifier (ERR=)
- End-of-file specifier (END=)
- Input/output status specifier (IOSTAT=)

The complete input/output statements are described in 5.6, 5.7, 5.9, and 5.10.

5.2.1. Unit Specifier (UNIT=)

Refer to a file by its unit specifier, sometimes called the file reference number. The unit specifier in the input/output statement indicates the file the statement references. The form of a unit specifier is:

```
[ UNIT = ] u
```

where:

UNIT=

is optional. When this clause is missing, the unit identifier *u* must be the first item in a list of specifiers.

u

is the unit identifier. The unit identifier for an external file can be specified as an unsigned integer constant or an integer expression whose value is greater than or equal to zero. You can use an asterisk to designate the symbiont units 5 and 6 (the symbionts AREAD$ and APRINT$; see G.6) when using formatted sequential READ and WRITE statements. The asterisk can't be used for the unit identifier in auxiliary input/output statements.

The unit identifier for an internal file may be a:

- character variable,
- character array,
- character array element, or
- character substring.

Once a unit specifier is associated with a file and that file is opened, the unit specifier can't be associated with another file until the first file is closed by either the CLOSE statement or the CLOSE service subprogram.

There is no standard convention for numbering system files. Your site can establish its own convention for assigning file numbers to input/output media (for example, card readers or printers). However, unless changed by the site, a default unit specifier assignment is made; see Table 5-2.

Table 5-2. Default Unit Specifier Assignment

Unit Number	Unit
5	Standard Input (AREAD$)
6	Standard Print (APRINT$)
1	Standard Punch (APUNCH$)
0	**Reread**

Information on other units and the assignment of unit numbers appears in Appendix G.

The external file can't be word-addressable mass storage.

5.2.2. Record Specifier (REC=)

The direct access input/output statements READ, WRITE, and **FIND** require you to specify the relative position (index) of the record in the file.

The forms of the record specifier are:

```
REC = rn
```

or:

```
'rn
```

where *rn* is an integer expression that specifies the relative position (index) of the record that is to be read or written in a direct access file.

When you use the second form of the record specifier, the optional UNIT= clause must not appear before the unit identifier.

5.2.3. Input/Output List

An output list determines which variables (storage sequences) are written to the output record when a WRITE statement is executed. An input list determines which variables (storage sequences) are filled from the input record when a READ statement is executed. The positioning of the names in the input/output list specifies the order in which the data is transferred between the record and the variables (storage sequences).

An input/output list is composed of list items separated by commas. The following can be list items:

- Variable name
- Array name
- Array element
- Character substring name
- Expression (output list only)
- Implied-DO list

Note the following input/output list rules:

1. The name of an assumed-size dummy array must not appear as an input/output list item.
2. When a function reference is used in an output list, the function and any subprogram that it activates must not perform input/output. An attempt to perform recursive I/O results in a fatal error message.
3. A character expression involving concatenation of an operand whose length specification is an asterisk in parentheses isn't allowed in an output list according to the FORTRAN 77 standard, **although ASCII FORTRAN allows it.**

When a variable name or array element appears in the list, one item is transmitted between the storage sequence and a record. When an array name appears in the list, the entire array is transmitted in the order in which it is stored (column-major order). When the array has more than one dimension, it is stored in ascending storage sequences with the value of the first subscript increasing most rapidly and the value of the last subscript increasing least rapidly. This ordering is described in 2.4.4.4.

On output, when numeric values fail to fit in the specified output field, the field is filled with asterisks and no error or warning message results.

Parts of arrays can be read or written using an implied-DO clause in the input/output list.

The implied-DO clause enables selected array elements to be referenced for input/output operations without putting each value on a separate record (for example, when the statement is in a DO-loop) or listing each element individually.

An input/output list containing the implied-DO clause has the form:

```
(input/output list , i = e1 , e2 [ ,e3] )
```

Note that the implied-DO clause must be enclosed in parentheses.

The elements i, e_1, e_2, and e_3 are as specified for the DO statement (see 4.5). The range of an implied-DO clause is the input/output list of the implied-DO. For input lists, i can't appear as input list items within the range of the implied-DO. Note that i can't be a DO-variable in any containing implied-DO clauses or DO statements.

If the implied DO-loop is terminated early, the DO variable i becomes undefined. This occurs when an end-of-file condition or an error condition occurs during execution of a READ statement. If an error condition occurs during execution of an output statement, the DO-variable i becomes undefined.

An example of an input/output list without an implied-DO clause is:

```
VAR1, ARRAY1, ARRAY2(5)
```

This list refers to input/output of the contents of variable VAR1, array ARRAY1, and element five of array ARRAY2.

The list:

```
(ARRAY2(J), J = 1,3)
```

refers to the first, second, and third elements of ARRAY2 without having to specify each element separately.

Note that an input/output list of an implied-DO clause can be another implied-DO clause. This produces a nesting of implied-DO clauses similar to nesting DO-loops (see 4.5.2). Since each implied-DO clause must be surrounded by parentheses, the implied-DO clause at the deepest parenthesized level is the innermost loop. For example, the nested implied-DO clauses:

```
( (ARRAY4(J,K), J = 1,9,4), K = 2,3)
```

refers to elements ARRAY4(1,2), ARRAY4(5,2), ARRAY4(9,2), ARRAY4(1,3), ARRAY4(5,3), and ARRAY4(9,3) in that order. The loop controlled by DO-variable J is the innermost loop.

Implied-DO clauses may be nested to a maximum depth of 25 loops.

The list:

```
VAR1, VAR2, (VAR3, ARRAY5(J), J = 3,6)
```

refers sequentially to VAR1, VAR2, VAR3, ARRAY5(3), VAR3, ARRAY5(4), VAR3, ARRAY5(5), VAR3, and ARRAY5(6).

5.2.4. Format Specifier (FMT=)

To read or write formatted records, you must include a format specifier in the READ or WRITE statement.

The form of the format specifier is:

```
[ FMT = ] f
```

where:

FMT =

is optional. When the FMT= clause is omitted, the format identifier *f* must be the second item in the list of specifiers, and the unit identifier must be the first item without the optional UNIT= clause. When the UNIT= and FMT= clauses are present, the specifiers in the control information list may be in any order.

f

is a format identifier and can be any of the following:

- Statement label of a FORMAT statement.
- Name of an integer variable containing the statement label (assigned with the ASSIGN statement) of a FORMAT statement.
- Name of a character array (except assumed-size array) or scalar character variable containing a format specification.
- **Name of a noncharacter array (except assumed-size array) containing a format specification.**
- Character expression containing the format.
- Asterisk specifying list-directed formatting.

The format specifications are described in 5.3. When the format identifier identifies a FORMAT statement, it must be in the same program unit as the input/output statement.

5.2.5. Namelist Name Specifier

To read or write namelist records, you must specify a NAMELIST statement name in the READ or WRITE statement.

A namelist name specifier is a symbolic name (see 2.4). By referring to this name, you can write a simple input/output statement that has the same effect as an input/output statement with a long list and a reference to a complicated FORMAT statement. The NAMELIST statement is described in 5.4.

5.2.6. Error Specifier (ERR=)

An error specifier (ERR clause) is allowed in certain input/output statements. The error specifier specifies that when an error or warning occurs while executing an input/output statement, execution of the statement terminates as soon as the error or warning is detected, the file position is indeterminate, the IOSTAT variable (if present) is given a value, and execution is transferred to the specified statement label.

When an error occurs while executing an input/output statement and neither an error specifier nor an input/output status specifier (see 5.2.8) is specified, the program terminates.

When a warning occurs and no error specifier is present, execution of the input/output statement continues after the IOSTAT variable (if present) is given a value.

The error specifier has the form:

```
ERR = s
```

where *s* is the statement label (in the same program unit as the input/output statement) to which control is transferred when an error or warning condition is detected.

5.2.7. End-of-File Specifier (END=)

An end-of-file specifier (END clause) is allowed in certain input**/output** statements. The end-of-file specifier is a statement label (in the same program unit as the input**/output** statement) to which transfer is made when an end-of-file condition is encountered during execution of the input**/output** statement. When an end-of-file condition is encountered while executing an input**/output** statement and neither an end-of-file specifier nor an input/output status specifier (see 5.2.8) is specified, the program terminates.

The end-of-file specifier has the form:

```
END = sn
```

where *sn* is the statement label to which control is transferred.

5.2.8. Input/Output Status Specifier (IOSTAT=)

An input/output status specifier (IOSTAT clause) is an integer variable or integer array element that, when specified in an input/output statement, receives a value determined by the success of the execution of the statement. The value returned is one of the following:

- A zero is returned when neither an error condition nor an end-of-file condition is encountered.
- The value of the I/O status word, PTIOE, from the storage control table is returned when an error or warning condition is encountered. The value returned has the format:

substatus	*unit number*	*error clause number*

  See G.9 for a further discussion of PTIOE.

- The value -1 is returned when no error condition is encountered but an end-of-file condition is encountered.

The input/output status specifier has the form:

```
IOSTAT = ios
```

where *ios* is an integer variable or integer array element.

When the IOSTAT clause is present, control returns to the FORTRAN program after an error or warning condition is encountered regardless of the presence or absence of an ERR or END clause. When an ERR or END clause is specified, control resumes at the statement associated with the label named in the clause, but when only an IOSTAT clause is specified, execution resumes with the statement following the I/O statement containing the IOSTAT clause.

5.3. FORMAT Statement

Purpose:

The FORMAT statement specifies the external form of the values of the input/output list elements and the arrangement of the data within the transmitted records.

Form:

```
f FORMAT format-specification
```

where:

f

is a statement label.

format-specification

is a format specification having the following form:

```
( [ format-list ] )
```

where:

( )

parentheses are required **but may be empty [that is, FORMAT( )]. When empty, and an input/output list is present, the input or output is list-directed (see 5.5).** When empty and there is no input/output list, one record is skipped on input or one blank record is written on output.

format-list

describes the fields to be input or output. The *format-list* can include the following:

- Edit descriptors (see 5.3.1) and repetition factors
- Sign options
- Grouping delimiters
- Scale factors
- Carriage controls
- Line delimiters

Description:

The FORMAT statement is a nonexecutable statement and can be placed anywhere in the source program. Since statement labels are local to the routine in which they appear, FORMAT statements are local to the routine by implication.

A format list describes fields to be input or output. A field is a string of adjacent character positions. The width of a field is the number of character positions in the string. A formatted record is a character string. On output, a record is constructed by concatenating the output fields. On input, the record is divided into fields according to the format list.

For any field, the format list must define its width and the type of conversion between the internal and the external forms, or the literal characters desired, or it must indicate that the field is to be skipped.

FORMAT statements are examined by the compiler for correctness and converted to a more efficient internal form that is used during the execution of the FORTRAN program.

As stated in 5.2.4, a character array, scalar character variable name, character expression, **or noncharacter array** can be used in place of a statement label of a FORMAT statement in a formatted input/output statement. The contents are format list items that can be modified during the execution of the object program. Such formats are not converted to an internal form by the compiler and are not examined by the compiler for correctness. The rules governing the contents of such formats appear in 5.3.9.

5.3.1. Edit Descriptors for Format Specification

Use an edit descriptor to specify the characteristics (format list) of a field. An edit descriptor can specify the type of conversion and field width, the explanatory literal characters, or the number of characters to be skipped. When two or more edit descriptors appear in a format specification, separate them from each other by a comma, a colon, or a slash.

The size of the integer constants *w*, *d*, and *e* must be less than 512. The size of *p* must be less than 256.

Edit descriptors:

I*w*

Integer. The field is to occupy *w* positions, the type of list item is integer, and the value of the list item is to appear as an integer constant right-justified in the field. When the field width specified for output is not large enough to contain the entire integer including a sign if it is required, the field is asterisk-filled to indicate overflow. When the field width specified is larger than that required for the constant, it is blank-filled on the left.

In an input field, leading blanks are ignored while embedded blanks are interpreted as zeros unless the BN edit descriptor has been encountered or BLANK=NULL occurred on the OPEN statement (see 5.10.1).

I*w.d*

Integer. This code is used for output only; it is ignored on input. The *w* is the same as for I*w*. The *d* indicates that at least *d* digits are written. This includes any zeros needed to equal *d* digits. The difference between *w* and *d* is space-filled on the left of the field. When a sign is required, *w* must be greater than *d*.

J*w*

Integer zero-filled. This code is the same as I*w* for input. For output, the integer is right-adjusted as in an I*w* field; however, the rest of the field is zero-filled. When there is a sign, it appears in the leftmost position in the field.

F*w.d*

External fixed point. The field is to occupy *w* positions on output. The list item is real, one part of a complex value, or double precision. The integer portion of the corresponding value appears as a right-adjusted real constant in the leftmost *w*-*d*-1 positions in the field. This integer part can't have an exponent. A decimal point character occupies position *d* + 1 from the right-hand end of the field ($d \geq 0$). The fractional part of the number is to occupy *d* digits written to the right of the decimal point. When the number is negative, the field width must permit a minus sign to be written immediately to the left of the most significant digit of the number. When the field width specified is larger than that necessary to contain the number, it is space-filled on the left. When the width specified is not large enough to contain the number including any necessary sign, the field is asterisk-filled. When a sign is required, the minimum field width necessary for an F edit descriptor is $w = 2 + d + s$, where $s = \text{MAX}(p + i, 0)$, p is the scale factor (see 5.3.6), and i is defined by $10^{i-1} \leq |M| < 10^{i}$, where M is the quantity to be written.

On input, F*w.d* designates a field of *w* characters, of which *d* characters follow the assumed decimal point. The assumed decimal point is ignored if a decimal point appears in the input. An exponent can follow the number. It must be a signed integer constant or begin with an E or D followed by an optionally signed integer constant. Blanks embedded in the input field are treated as zeros unless the BN edit descriptor has occurred or the OPEN statement for the file contained a BLANK=NULL. Embedded and trailing blanks are then ignored. Leading blanks are ignored.

E*w.d*

Floating point. The field is to occupy *w* positions. The list item is real, one part of a complex value, or double precision. The value of the list item is to appear as a decimal number, right-justified in the field in the form:

```
- . xxxxxxseee
```

in which *xxxxxx* are the *d* most significant digits of the mantissa, *s* is a plus or minus sign, and *eee* is the corresponding 3-digit exponent. When the exponent is positive, a plus precedes *eee*. When it is negative, a minus is written. When the value of the list item is positive, the minus sign in front of the decimal point is omitted or replaced by a plus sign if the SP edit descriptor has occurred. The minimum field width necessary to contain a number of this type, including the sign of the fraction, is $w = 6 + d + sf$; with $sf = \max(p,0)$, where p is the scale factor (see 5.3.6). When the field width specified is larger than necessary to contain the number, it is blank-filled on the left. When the field width is too small to express the value to be written, the field is filled with asterisks.

On input, E*w.d* is equivalent to F*w.d*.

Ew.*d*Ee
E*w.d*De

Specific floating point. This is a special E*w.d* format used for output only. The output format is:

```
- . xxxxxxEseee        or        - . xxxxxxDseee
```

where *e* specifies the number of exponent digits. When the exponent exceeds the number of exponent digits specified, the field is filled with asterisks.

D*w.d*

Double precision floating point. This is the same as E*w.d* except that the list item is regarded as double precision. The exponent is generally a 3-digit number. Thus, $w \geq 6 + d + sf$.

*p*P

Scale factor (see 5.3.6).

BN
BZ

Blank. This descriptor controls the interpretation of embedded or trailing blanks in numeric input fields. The BN edit descriptor indicates that embedded blanks are ignored in numeric input fields. The BZ edit descriptor indicates embedded blanks are treated as zeros in numeric input fields. Embedded blanks in numeric input fields are normally treated as zeros during format control unless an OPEN statement is present for the file reference number of this formatted I/O statement. (See the BLANK= clause of the OPEN statement [5.10.1].) When a BN edit descriptor is encountered in the format list, embedded blanks are ignored in numeric input fields until the end of that format or until a BZ edit descriptor is encountered.

S
SP
SS

Sign. This edit descriptor controls the writing of optional plus signs in numeric output fields. If SP or **+S** is used, all optional plus signs are written. The field width must be large enough to handle the sign or the field is filled with asterisks. An SS, S, or **-S** turns the option off, and then only the minus signs are written.

L*w*

Logical. This edit descriptor is used only with input and output of logical variables. When L*w* is specified for output and the value of the logical list item is .TRUE., the rightmost position of the field with length *w* contains the letter T with *w*-1 blanks on the left. When the value is .FALSE., the letter F is written.

On input, the field width is scanned from left to right for optional blanks, optionally followed by a decimal point, followed by a T or F, and the value of the corresponding logical list item is set to .TRUE. or .FALSE., respectively. All other characters following the T or F in the external input field are ignored. In the absence of T or F in the input field, no value is stored and a warning message is issued.

O*w*

Octal. The field occupies *w* positions, the value of the list item is interpreted as a 12-digit (or 24-digit) octal number, and the quantity is written as an octal number that is right-justified in the field. When the field width, *w*, is less than or equal to 12, the *w* least-significant digits appear. When *w* is larger than 12, it is filled out to the left with blanks on output and binary zeros on input. When the list item is double precision, 24 digits can be read or written. When *w* is less than or equal to 12, a double-precision list item is treated as a single-precision item; only the most significant word of the list item is used. When the output list item is character, the leftmost *w* characters of the list item are placed in the field and blank-filled on the left if *w* is greater than the character item length. When the input list item is character, the leftmost portion of *w* characters is stored to the list item and blank-filled on the right if *w* is less than the length of the list item.

A[*w*]

Alphanumeric. The field occupies *w* positions. Let *s* be the length of the character list item. This is the length of the variable for type character, **four for integer, real, or logical; eight for double precision.** On output, when *w* is less than or equal to *s*, the leftmost *w* characters of the list item are placed in the field. When *w* is greater than *s*, then the *s* characters of the list item are placed right-justified in the field and the field is blank-filled on the left.

On input, when *w* is less than *s*, the *w* characters of the field are placed in the leftmost *w* characters of the list item and the rightmost *s*-*w* characters are blank-filled. When *w* is greater than or equal to *s*, the rightmost *s* characters are placed in the list item. When *w* equals zero, no characters are transferred. When *w* is missing, the number of characters transferred is the character length of a list item of type character only. The *w* is optional only for list items of type character.

R*w*

Right-justified alphanumeric. This code is similar to A*w*. However, when *w* is less than *s* on input, the next *w* characters are placed right-justified in the list item with zero fill (000, ASCII NUL). When *w* is less than *s* on output, the rightmost *w* characters in the list are transmitted.

$wHh_1...h_w$

H edit descriptor (Hollerith). The field is to occupy *w* positions. The field consists of literal characters and is filled with the *w* characters (including blanks) that follow H. The field designation is independent of the list items. The length *w* is limited only by the external medium and by the maximum of 511 significant characters for *w*.

On input, the Hollerith specification is read into the format specification itself and is available on the next write using the same nonvariable format. Variable formats can't save the Hollerith data in the format itself.

'$h_1h_2...h_w$'

Apostrophe. This is the same as the H edit descriptor; the only difference involves apostrophes within the literal string. When an apostrophe is to be included in the literal string, use two apostrophes to indicate its position.

*w*X

Skip. A field whose length is *w* is skipped. The field designation is independent of the list items. Skipping forward over a character position that has not yet been set in the record causes the character position to be set to blank. **When *w* is negative, the skipping is in a backward direction. However, a negative *w* can't go back further than the beginning of the record; the skipping stops at the beginning of the record. No blanking is done when *w* is negative.**

G*w.d*

General. When the output item (M) is real, an E, D, or F edit descriptor is used, depending on the absolute value of M. If $10^{i-1} \le |M| < 10^i$ and $d \ge i \ge 0$ (where *d* is the number of significant decimal digits) the output field is formatted by F(*w*-4).(*d*-*i*),4X. When $|M| < 0.1$ or $|M| \ge 10^{**}d$, the output field is formatted by E*w.d* or D*w.d*. When the scale factor *p* (see 5.3.6) is applied, the edit descriptor *p*PE*w.d* is used. The G edit descriptor doesn't change the value of the item. Values are strictly numeric or logical. No conversion is done for character type for output. A scale factor of zero is assumed when the F edit descriptor is used.

On input, the G edit descriptor is the same as an F edit descriptor without a scale factor. Thus, F*w.d* is used on input when *p*PG*w.d* is specified. The G edit descriptor also provides for **integer** and **logical conversion** as though it were **I*w* and L*w*,** respectively.

G*w.d*E*e*

General. This is a special G*w.d* format used for output only. The output format is:

```
±.xxxxxxEseee
```

where *e* specifies the number of exponent digits. This format is used when the absolute value of the output item requires an E edit descriptor. If the F edit descriptor is used, the output field is formatted by F(*w*-(*e*+2)).(*d*-*i*),(*e*+2)X.

T*w*

Character position. This edit descriptor causes the input or output operation to begin at the w^{th} position of the record. Therefore:

```
FORMAT(10X,F10.3)
```

is equivalent to:

```
FORMAT(T11,F10.3)
```

On output, if *w* is greater than any character position written up to that time, the T edit descriptor will blank the character positions between *w* and the highest character position written. The order of the associated list does not need to be the same as that of the record input. For example:

```
FORMAT(T50,F10.3,T5,F10.2)
```

is valid for writing the first list item in positions 50-59 and the second in positions 5-14. The first character position of a record is number one.

TL*w*

Character position left. This edit descriptor causes the character position in the record to be moved *w* positions to the left or backward from the current position. If the current position is less than or equal to *w*, the new character position is the first character position in the record.

TR*w*

Character position right. This edit descriptor causes the character position in the record to move *w* positions to the right or forward from the current position. The TR*w* can't write beyond the end of the record.

/

Slash. (See 5.3.7.1.)

:

Colon. (See 5.3.7.2.)

When numeric output values fail to fit in the specified field length, the field is filled with asterisks and no error results.

Examples of the uses of these edit descriptors follow:

```
C         Integer codes:
    K=76
    M=3333
    WRITE (6,10) K
10  FORMAT(1X,I10)
    WRITE(6,20) K
20  FORMAT(1X,I10.7)
    WRITE(6,30) K
30  FORMAT(1X,J10)
    WRITE (6,40) M
40  FORMAT (1X,I3)
C         Format 10 writes integer K as ΔΔΔΔΔΔΔΔ76
C         Format 20 writes K as ΔΔΔ0000076
C         Format 30 writes K as 0000000076
C         Format 40 writes M as *** since M doesn't
C         fit in the 3-digit field.

C         Real value codes:
    R=76.8
    WRITE(6,40) R
40  FORMAT(1X,F7.2)
    WRITE(6,44) R
44  FORMAT(1X,E10.2)
    WRITE(6,46) R
46  FORMAT(1X,D11.2)
    WRITE(6,48) R
48  FORMAT(1X,SP,G10.4)
C         Format 40 writes R as ΔΔ76.80.
C         Format 44 writes R as ΔΔΔ.77+002.
C         Format 46 writes R as ΔΔΔΔ.77+002.
C         Format 48 writes R as +76.80ΔΔΔΔ
C         Alphanumeric edit descriptor:
    CHARACTER*4 X,Y
    CHARACTER*8 A,B
50  FORMAT(A3,A5,R2,R6)
    READ(5,50)A,X,Y,B
55  FORMAT(1X,A6,A3,R5,R3)
    WRITE(6,55)X,X,Y,Y
    END
C          Using the string 'ABCDEFGHIJKLMNOP' as
C          input, the READ statement produces
C          the following values:
C          A='ABCΔΔΔΔΔ'
C          B='■ ■KLMNOP'
C          X='EFGH'
C          Y='■ ■IJ'
C          The WRITE statement produces the record:
C          ΔΔEFGHEFGΔ ■ ■IJ■IJ
C          Note that in these examples, Δ represents the
C          nonprinting graphic for code 040 (the ASCII space
C          code) and ■ represents the nonprinting graphic
C          for code 000 (the ASCII NUL).
```

5.3.2. Repeating Edit Descriptors

When two or more successive edit descriptors (except *w*H, *w*X, T*w*, TL*w*, TR*w*, /, S, SP, SS, BN, BZ, *p*P, :, and apostrophe) are identical in every respect, a shorthand notation can be used. Write the edit descriptor only once and prefix it with an unsigned integer constant less than 512 that indicates the number of repetitions of the field. For example, you can write:

```
FORMAT(I5,I5,I5)
```

as:

```
FORMAT(3I5).
```

5.3.3. Repeating Groups of Edit Descriptors

When two or more successive groups of edit descriptors occur and each element of one group is identical to the corresponding element of the other groups, then a further shorthand notation can be used. Specify the group only once; enclose it in parentheses, and prefix the resulting parenthesized group with an unsigned integer constant less than 512 that indicates the desired number of repetitions of the group. When none is specified, a repeat count of one is assumed. For example, you can write:

```
FORMAT(I5,F9.4,E8.2,I5,F9.4,E8.2)
```

as:

```
FORMAT(2(I5,F9.4,E8.2) ).
```

Only four-deep nesting of parentheses is permitted (see 5.3.8). For example, the following statement is legal:

```
FORMAT('0',5(F9.3,5(E12.6,2(I8,2(I10,I5) ) ) ) )
```

However, if it has another nested group, that group is executed as if it has a group count of one.

When the input/output list is longer than the number of edit descriptors in the format specification, the format scan returns to the first level of parentheses (unless the format contains an indefinite repetition group). In the example, the scan returns to F9.3 and continues until the input/output list is exhausted. Reversion of format control has no effect on scale factor, sign control, or blank control (see 5.3.8).

5.3.4. Carriage Control

When an output record is printed, the first character of the record controls line spacing and is not printed. Printing begins in the first character position of a line with what is normally the second character of the record. You normally provide an appropriate first character with one of the following:

- H edit descriptor
- apostrophe edit descriptor
- *w*X edit descriptor (providing a blank)

The effect of these carriage control characters is as follows:

Character	Effect
blank	Single space before printing
0 (zero)	Double space before printing
1	Skip to top of next page
+	Suppress spacing

When an H, apostrophe, or *w*X edit descriptor are not provided, the first character of whatever field comes first is used for carriage control. If it is an illegal character, the default blank character is used.

When an empty format is used without an input/output list, one record is skipped on input or one blank record is written on output.

Carriage control characters are only effective for print files (output symbiont units). See 5.10.1 for information on using the OPEN statement TYPE clause to change the file format type to symbiont.

The effect of the carriage control characters depends on the output device that you use.

5.3.5. Complex Variables

One complex variable requires two D, E, F, or G edit descriptors. They need not be the same.

When read or written under control of a FORMAT statement, a complex value appears as a pair of real values. They are not enclosed in parentheses or separated by a comma unless the format specification provides them explicitly.

5.3.6. Scale Factor

Input and output using the F, E, G, and D edit descriptors can be modified by a power of 10 by using a scale factor of the form *p*P, where *p* is a signed integer.

The separating comma can be omitted between a scale factor and an F, E, D, or G edit descriptor that immediately follows.

When format control is initiated, a scale factor of zero (that is, 0P) is established. Once established, a scale factor applies to all subsequent F, E, G, and D edit descriptors in its FORMAT statement or until another scale factor is encountered in the statement. The scale factor doesn't affect any of the other edit descriptors.

For input, a scale factor, P, has the following effect:

- The scale factor has no effect on input with F, E, G, and D editing that contains an exponent in the external field.
- For F, E, G, and D editing with no exponent in the external field, the internal value is the external value divided by 10^p.

For output, a scale factor, P, has the following effect:

- For F editing, the external value is the internal value multiplied by $10^{p.}$
- For values using E or D editing, the fraction is multiplied by 10^p and the exponent is decreased by *p*. In other words, the field changes in form but not in value on printed output.

  If $p <= 0$ and $p > -d$, the output field to the right of the decimal point contains *d* digits, of which $|p|$ are leading zeros and $d- |p|$ are significant digits.

  If $p > 0$ and $p < d+2$, the output field contains exactly *p* significant digits to the left of the decimal point and $d-p+1$ significant digits to the right of the decimal point.

  For other values of *p*, the output field is asterisk-filled.
- For G editing, the effect of the scale factor depends on the magnitude of the value. When its size requires E editing, the scale factor effect is that of E editing. When F editing is necessary, the scale factor has no effect.

For instance, assume that variable A has the value -12764.316. If A is printed according to the specification E13.6, then the field printed is -.127643+005. If A is printed according to the specification 2PE14.6, then the field printed is -12.76432+003. If A is printed according to the specification -3PF7.2, then the field printed is -12.76.

5.3.7. Control of Record Handling and List Fulfillment

The colon and slash delimiters let you halt unnecessary output and begin processing a new record, respectively.

5.3.7.1. Using a Slash (/) to Control Multiple-Line Formats

A slash following an edit descriptor terminates the record being input or output. Interpretation of a new record's format continues with the edit descriptor following the slash for the next list item. When *n* slashes are written in sequence before the first edit descriptor in the format, *n* blank records are written or *n* records are skipped before reading a record. When *n* slashes are written after the last edit descriptor in the format, *n* blank records are written or *n* records are skipped on input. When edit descriptors are used before and after *n* sequential slashes, *n*-1 blank records are written or *n*-1 records are skipped on input.

Separating commas can be omitted immediately before or after a slash.

The following examples produce three blank lines when the output goes to the printer:

```
(///I2)
(I2///)
(1X///I2)
(I2///1X)
(I2////F8.2)
```

The third and fourth examples place one blank in a record or line, which is then printed to produce three blank lines.

When the file is a direct-access file, the record number is increased by the number of records skipped on input or by the number of blank records written on output.

Slashes can also be used in list-directed input/output (see 5.5).

When a single slash is used as the format specification; that is, (/), without an input/output list, two records are skipped on input or two blank records are written on output. An empty format specification; that is, ( ), without an input/output list, skips one record on input or writes one blank record on output.

5.3.7.2. Using a Colon to Control List Fulfillment

A colon in the format list indicates that when a record is being read or written and the colon is encountered, the operation is ended if the last item in the list has already been processed. A single colon can appear at the beginning or end of the FORMAT statement or between pairs of format list items.

You can omit separating commas immediately before or after a colon.

An example of the use of the colon for output is:

```
1  FORMAT ('ΔAΔ=',F10.3:, ΔBΔ=',F10.3:'ΔCΔ=',F10.3)
   IF (I.EQ.1) PRINT 1,A
   IF (I.EQ.2) PRINT 1,A,B
   IF (I.GT.2) PRINT 1,A,B,C
```

Using the values I=2, A=17., and B=3.1416, the output is:

```
△A△=△△△17.000△B△=△△△△3.142
```

Using commas in place of the colons, the output prints C= before discovering the list is empty, resulting in:

```
△A△=△△△17.000△B△=△△△△3.142△C△=
```

An example of the use of the colon on input is:

```
2  FORMAT (F10.3 / )
1  FORMAT (F10.3, : / )
   READ (5,2) E
   READ (5,1) E1
```

If the input is:

```
△△△△△△△1.2
△△△△△104.2
△△△△△△△△△2
△△△△△△△111
```

the first READ statement sets E to 1.2 and the second READ statement sets E1 to .002. If the first READ statement uses a colon before the slash, E1 is set to 104.2.

5.3.8. Relationships of a Format Specification to an I/O List

During the execution of an input/output statement, the format specification is scanned from left to right. Edit descriptors of the form *w*H, *w*X, *p*P, BN, BZ, S, SP, SS, TL*w*, TR*w*, T*w*, slash, and apostrophe are interpreted, and the appropriate action is taken without reference to the input/output list.

When another edit descriptor occurs, either there is at least one list item remaining to be transmitted or there is not. If one remains, the next list item is converted according to the format specification and transmitted. The format scan then continues. If no values remain, the transmission is terminated. (See 5.3.7.2 for the effect of a colon in the input/output list.)

There can be up to five levels of parentheses in a FORMAT statement. That is, there may be up to five left parentheses followed by matching right parentheses. The outermost pair is called the zero-level parentheses.

If format control encounters the rightmost parenthesis of a complete format specification and another list item is not specified, format control terminates. However, if another list item is specified, the file is positioned at the beginning of the next record and format control then reverts to the beginning of the format specification terminated by the last preceding right parenthesis. If there is no such preceding right parentheses, format control reverts to the first left parenthesis of the format specification. If such reversion occurs, the reused portion of the format specification must contain at least one repeatable edit descriptor. If format control reverts to a parenthesis that is preceded by a repeat specification, the repeat specification is reused. Reversion of format control, of itself, has no effect on the scale factor (see 5.3.6), the S, SP, or SS edit descriptor sign control, or the BN or BZ edit descriptor blank control (see 5.3.1).

In the following example:

```
FORMAT('0',5(F9.3,5(E12.6,2(I8,2(I10,I5)))))
```

the second (and every following) record is started with an F9.3 edit descriptor. If there are no nested parentheses, scan control reverts to the beginning of the format specifications.

You cannot specify more than 80 characters for one record of symbiont input or more than 132 characters for a symbiont output record unless an OPEN statement specified a different record size for the symbiont. If the scale factor is modified by a *p*P edit descriptor, it is not reset to zero when control reverts to the appropriate left parenthesis. Each *p*P edit descriptor remains in effect until the end of the FORMAT input/output list or until another *p*P edit descriptor is encountered (see 5.3.6).

If the format specification is empty and an I/O list is present, list-directed I/O is used. If the I/O list is not present and the format specification is empty, one record is skipped on input or one blank record is written on output.

The IOFLG$ subprogram prevents checking for a type mismatch between an I/O list item and a format edit descriptor during the execution of a formatted I/O statement. For more information, see 7.7.3.23.

5.3.9. Variable Formats

Any of the input/output statements that specify formats can contain an array or scalar character variable name or a character expression in place of the reference to a format.

Assume the symbolic name V is an array. The contents of the array can be initialized and modified like any array. Array V must contain legal format specifications in ASCII form at the time it is used as a format. The format specifications in V must have the form:

```
(format-list)
```

The form of this specification is as described in 5.3 except the word FORMAT is omitted. Blanks can appear before the first left parenthesis (zero-level), indicating the beginning of the format. Characters following the right parenthesis, which indicates the end of the format (zero-level right parenthesis), are ignored. For example:

```
     CHARACTER*4 V
     DIMENSION V(5),K(8)
     DATA (V(I),I=1,5)/4H(1H1,2H,8,1HI,2H10,1H) /
     READ(5,60)K
     WRITE(6,V)K
60   FORMAT(8I10)
     END
```

generates (1H1,8△△I△△△10△△)△△△. The blanks (△) are ignored when the array is used as a format.

The use of V as a format is equivalent to using:

```
FORMAT(1H1,8I10)
```

H edit descriptors (Hollerith specifications) inside variable formats must be of the form *n*H. The variable format is converted by the library to a coded format and stored in a library storage area. The variable format is not changed by the library. The variable format can be changed by the program. **When the H edit descriptor is used in a variable format on a READ statement, the H edit descriptor in the variable format doesn't contain the Hollerith data read in; that is, the variable format isn't changed.**

A character expression in the control information list as the format could be:

```
WRITE (6,'(1X,8I10)') K
```

When you specify an array element for the variable format, only the contents of that element are used for the format.

5.3.10. Representation of Input/Output Data

The format of data in an input field can be the same as that generated by the same format edit descriptor for output.

- Unless positioning is achieved by the X, T, or TR edit descriptors, column 1 is the first column read. There is no carriage control.
- Plus signs can be omitted or can be indicated by +. Minus signs are indicated by -.
- Embedded or trailing blanks in a numeric input field are equivalent to zeros unless the BN edit descriptor is used or BLANK=NULL on the OPEN statement is used (see 5.10.1).
- Only the high-order 8 digits are retained for input into real variables; the high-order 18 digits are retained for D input into double-precision variables.
- For E and D editing, the exponent can be indicated by an E or a D (interchangeable), or, if E or D is omitted, by + or - (not a blank). Thus, E2, E+2, E+02, +2, and D2 are all permissible, equivalent exponents. Lowercase e and d are recognized as E and D.
- For E and D, if the exponent is omitted, it is taken to be either zero or the value of the P specification (see 5.3.6).
- On input with E, D, F, and G editing, a decimal point appearing in the input field overrides the assumed decimal point position specified by the edit descriptor.
  - The exponent is right-justified in the subfield defined by the E or D, or +, or -, at the right-hand limit of the field. The subfield can be empty. The fraction is right-justified in the remaining field, to the left of the E, D, +, or -.
  - When the decimal point is omitted, it is assumed to be located between the *d* and *d*+1 positions (from the edit descriptor) to the left of the exponent. When no exponent is present, the decimal point is assumed to be located between the *d* and *d*+1 rightmost positions in the field.

5.4. Namelist Input/Output

The nonexecutable NAMELIST statement and the associated forms of the input/output statements provide a simplified means of transmitting an annotated list of data to and from files with implied format control.

The NAMELIST statement specifies the items to be transferred. In the associated input/output statement, the list of items to be transferred is void. See 5.6.1.3 and 5.6.2.3 for details on input/output statements that use the namelist specification. Therefore, by referring to a single name *n,* it is possible to form a simple input/output statement that has the same effect as a similar statement with a long list and a reference to a complicated FORMAT statement.

On input, the list of items in the namelist specifies those items that can have their values defined in the records to be read. Not all items of the namelist need be used in the input records nor must the input fields be in the same order as the list items. If a list item is an array name, data can be assigned to the entire array, a group of contiguous elements, or any individual elements of the array as specified by the input records. Within input records, the data to be read is identified within the input record itself. Thus, to read the number 1.5 into variable A, the external input field contains:

```
A=1.5
```

where the field is delineated by commas. The acceptable forms of the data and the format in which the data is to be stored are dictated by the type of the list item (see 5.4.2).

On output, each list item of the designated namelist is formatted in a standard fashion for output in the order specified by the list (see 5.4.3). Namelist output can be used as a convenient source language diagnostic tool, especially when it is coupled with a parameterized DELETE statement.

When a whole array is read or written, its size must be less than 262,143 words and it must contain fewer than 262,143 elements.

5.4.1. NAMELIST Statement

Purpose:

The NAMELIST statement provides a simplified means of identifying a list of data for transferral to or from files.

Form:

```
NAMELIST /n/x[ ,x] ... [/n/x[ ,x] ... ] ...
```

where:

each *n*

is a namelist name which must satisfy the rules for a symbolic name. Each must be unique within its program unit.

each *x*

is a simple variable, a subscripted variable, or an array name.

Description:

Within a NAMELIST statement, a namelist name is enclosed in slashes. The list of variables associated with a namelist name ends when a new namelist name enclosed in slashes is encountered or with the end of the NAMELIST statement.

A namelist name can be defined only once in a program unit by its appearance in a NAMELIST statement. Within the unit in which it is defined, a namelist name can appear only in input/output statements and in the defining NAMELIST statement.

A variable name, array element name, or array can be associated with one or more namelist names. Array names must be previously declared. The subscripts of an array element must consist of constants or parameter constants. Dummy arguments are not permitted in a namelist list. Don't use a $ in the variable name, array element name, or array name.

For a description of local-global rules for names used in NAMELIST statements in internal subprograms, see 7.9.

See 5.4.2 for details on input associated with a NAMELIST statement and 5.4.3 for information on the form of output resulting from the use of a namelist WRITE statement.

Example:

```
      DIMENSION A(10),I(5,5),L(10)
      NAMELIST/NAM1/A,B,I,J,L(3)/NAM2/A,C,J,K,I(2,3)
C             Arrays A and I, variables B and J, and subscripted
C             variable L(3) are associated with namelist name NAM1,
C             and array A and variables C, J, K, and I(2,3) are
C             associated with namelist name NAM2.
```

5.4.2. Namelist Input

The READ (*u,n*) statement (see 5.6.1.3) causes the records that contain the input data for the variables and arrays that belong to the namelist name *n* to be read from input unit *u*.

For READ (*u,n*), the first character of the first record of a group of data records that is read is ignored, and the second character must be the currency symbol ($) or ampersand (&), immediately followed by the namelist name and a blank. When the name is not *n*, the input medium is searched for $*n*. The remainder of the first record and following records can contain any

combination of the legal data items which are separated by commas (a comma after the last data group is ignored). The last input record is terminated by the characters $END or &END.

The forms the input data items can take are:

- *variable-name* = *constant* (*Variable-name* is a simple variable name.)
- *subscripted-variable* = *constant* (The array element appears in the NAMELIST statement. The subscripts in the input record must be constants.)
- *subscripted-variable* = *set-of-constants* (The set is represented by constants separated by commas, where *k**constant* represents *k* constants. The name of the array of which the subscripted variable is an element must be in the namelist list. The number of constants must not exceed the space available in the array, which is from the element specified to the end of the array, inclusive.)
- *array-name* = *set-of-constants* (The set is represented by constants separated by commas, where *k**constant* represents *k* constants. The number of constants must be less than or equal to the number of elements in the array.)

Constants used in the data items can take any of the following forms:

- Integer constants (see 2.3.1).
- Real constants (see 2.3.2).
- Double-precision constants (see 2.3.2.2). The D is optional in the exponent.
- Complex constants (see 2.3.3).
- Logical constants (see 2.3.4). These can be written as .TRUE., .FALSE., T, or F. The value stored is 0 for false and 1 for true.
- Character and Hollerith constants (see 2.3.5). Character input must not span records. All of the *n*H edit descriptor must appear on one line as well as the full apostrophe edit descriptor. An individual Hollerith input value cannot be transferred to a data area that spans from one virtual bank to another (for example, the value 8Habcdefgh cannot be read into two consecutive elements of an integer array if the two elements happen to be allocated in two different virtual banks).
- Octal constants (see 2.3.7). If the type of the namelist variable is double precision, 24 digits can be used.

Logical and complex constants can associate only with logical and complex variables, respectively. Hollerith constants can associate only with integer or real variables. Character constants can associate with character variables only. The other types of constants can associate with integer, real, or double-precision variables, and they are converted in accordance with the type of the variable.

The data items that appear in the input records need not appear in the same order as the corresponding variable or array names in the NAMELIST statement list. All variable or array names of the namelist need not have a corresponding data item in the input records. When none appears, the contents of the variable or array are unchanged. Names that are equivalenced to these names can't be used in the input records unless they also are part of the namelist.

You must not embed blanks in a constant or a repeat constant field, but you can use them freely elsewhere in a data record. The last item in each record that contains data items must be a constant, optionally followed by a comma. The comma is optional in the record that contains or precedes the $END sentinel.

For example, the NAMELIST statement:

```
DIMENSION A(10), K(5,5), L(10)
NAMELIST/NAM1/L(3),A, K, J
```

could have the following input records:

```
Record                  Position and Contents

First data record       △$NAM1△ K (2,3)=5,7,9,
Second data record           0,10,L(3)=4.3,A=4,3,
Third data record          8*4.3,J=4.2,
   .
   .
   .
Last data record        $END
```

When the preceding data is used with a READ statement, the following action takes place:

1. The first data image is read and examined to verify that its name is the same as the namelist name in the READ statement. (Note that the name must begin in column two of the first record.) The integer constant 5 is placed in K(2,3), 7 in K(3,3), and 9 in K(4,3).
2. When the second data image is read, integer 0 is placed in K(5,3), and 10 in K(1,4); the real constant 4.3 is truncated to integer 4 and placed in L(3); integers 4 and 3 are converted to real and placed in A(1) and A(2), respectively.
3. When the third data image is read, the real constant 4.3 is stored in A(3), A(4), . . . ,A(10) and the real constant 4.2 is truncated to integer 4 and placed in J.

5.4.3. Namelist Output

The WRITE (*u,n*) statement (see 5.6.2.3) causes all names of variables and arrays (as well as their values) that belong to the namelist name *n* to be written to the output unit *u*.

In the WRITE (*u,n*) statement, all variables and arrays (as well as their values) belonging to the namelist name are written out according to their types. The output data is written such that:

- **The name of a variable and its value are written on one line or record.**
- **The name of an array is written, with the values of the elements of the array written in a convenient number of columns, in the order of the array in main storage, that is, in column-major order.**
- **Whole arrays start at the beginning of the output buffer (that is, column 1).**
- **The data fields are large enough to contain all the significant digits.**
- **The output can be read by an input statement referring to the namelist name.**

For example, if the values of the variables A through H and arrays L and M have the values 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0; L(1) = 5 and M = 1,2,3,4, the statements:

```
DIMENSION L(4), M(2,2)
NAMELIST /OUT1/ A,B,C,D,E,F,G,H,L(1),M
WRITE(8,OUT1)
```

cause the following five records to be written (where a record represents one printed line):

```
  $OUT1
A  =  .10000000+001,B =  .20000000+001,C =  .30000000+001,D =  .40000000+001,E =  .50000000+001,
F  =  .60000000+001,G =  .70000000+001,H =  .80000000+001,L(1)  =                  5,
M  =              1,               2,              3,                4
$END
```

5.5. List-Directed Input/Output

Use list-directed input/output statements to read and write free-field records with implied format control. A list-directed write outputs records consisting of sequences of values composed of characters that can be represented internally.

A list-directed read inputs as many records as necessary to provide a value for each item in the input list, unless a slash appears in the input. When a slash appears in the input, the slash serves as an end-of-record-transmission signal. Any list items remaining after the slash is encountered undergo no change in value (see 5.5.1).

The output from a list-directed write (except for a write to a print or punch unit) can be used as input to a list-directed read unless the input variable is character.

5.5.1. List-Directed Input

The list-directed input consists of a sequence of values separated by commas, blanks, slashes, or end of the line. Numeric, logical, and character data can be read. Hollerith and octal data can't be used.

When a value is repeated for several list items, it can be written as:

```
r*c
```

where *r* is an integer constant between 1 and 511, inclusive, and *c* is a value that can be blank. Blanks can't be embedded in either *r* or *c*, except in character constants and complex constants. Examples of this form are:

```
5*4.0

5*Δ
```

Each constant must be of the same type as its corresponding list item. An error occurs if the types don't match. Blanks are never used as zeros and embedded blanks aren't permitted except in character constants and complex constants. For example, 5.Δ0,2.0 represents three values (5.0, 0.0, and 2.0) and 5.0,Δ2.0 represents two values. The end of a line can be considered a blank except when it appears in a character constant.

Specify a null value by:

- No characters between two commas (,,)
- No characters before the first comma in the first record of the list-directed input
- Only blanks between two commas (,ΔΔ,)
- Only blanks before the first comma in the list-directed input record
- A blank following the asterisk in the r*c form (r*Δ)
- A comma following the asterisk in the r*c form (r*,)

A null value has no effect on the corresponding list item. If the list item has no value, it still has no value. A null value can't be used as the real or imaginary part of a complex constant, but it can represent an entire complex constant. The end of a line before or after a comma does not generate a null item.

A slash causes the end of the list-directed read after the assignment of the previous value. When there are more items in the list, the effect is as though null values are supplied for them.

The parts of a complex constant must be separated by a comma and enclosed in parentheses. The comma can be preceded or followed by spaces or the end of a line. For example:

```
(25.2,
 2.0)
```

and:

```
(25.2, ΔΔΔ2.0)
```

are legal forms of a complex constant.

Hollerith constants are not allowed in free-field input records, but character constants are legal if the type of the list item is character. Each apostrophe within a character constant must be represented by two apostrophes with no intervening spaces or an end-of-line. Character constants can continue from one line to the next without a blank inserted for the end of the line. Character constants can contain blanks, commas, or slashes.

5.5.2. List-Directed Output

A list-directed output record consists of a sequence of values that have the same form as constants, except as noted. The values are separated by one or more blanks. New lines are started as necessary. The constants are formatted with set lengths. The end of a line and blanks don't occur within a constant except for complex and character constants.

Logical constants are written as T or F.

Real and double-precision values are written using an E or F editing format, depending on the value of the constant.

Complex constants are enclosed in parentheses with a comma separating the real and imaginary parts.

Hollerith constants aren't written for list-directed output records. Character constants written for list-directed output records aren't delimited by apostrophes or blanks nor are double apostrophes written for internal apostrophes.

Null values, commas, slashes, and the repeat form of *r*c* aren't produced in list-directed output records.

An example using list-directed input/output is:

```
DIMENSION L(3)
COMPLEX C
DOUBLE PRECISION D
READ (3,*)D,R,C,(L(J),J=1,3)
WRITE (8,*)D,R,C,(L(J),J=1,3)
```

These statements can read the following data image:

```
3.0D1,4.0,(3.0,2.0),5,6,7
```

Setting D=3.0D1, R=4.0, C=(3.0,2.0), L(1)=5, L(2)=6, and L(3)=7. The output record is:

```
Δ30.0000000000000000ΔΔΔΔΔΔ4.0000000ΔΔΔΔΔ(Δ3.0000000ΔΔΔΔ,Δ2.00000000ΔΔΔΔ)ΔΔΔΔΔΔΔΔΔΔΔΔ5ΔΔΔΔΔΔΔΔΔΔΔΔ6ΔΔΔΔΔ
ΔΔΔΔΔΔ7
```

5.6. Sequential Access Input/Output Statements

There are five sequential access input/output statements:

1. READ (reads records from a sequential file)

2. WRITE (writes records to a sequential file)
3. BACKSPACE (positions file for input/output by backspacing one record)
4. ENDFILE (defines the end of a sequential file)
5. REWIND (positions a unit at its beginning)

The OPEN statement **and the DEFINE FILE statement** are used in conjunction with these statements. They let you describe the format of a file.

Using these statements, you can process a file in sequential order. Each record is manipulated or passed over, as desired, one after another. Unlike direct access input/output statements that refer to random access files, sequential access input/output statements can't directly address a specific record.

5.6.1. Input Statements

READ statements obtain values for data elements from files. The form of each READ statement depends on the type of record (formatted, unformatted, **namelist**, or list-directed) read.

5.6.1.1. Formatted READ

Purpose:

A formatted READ statement reads values into items specified in an input list according to a specified format.

Form:

```
READ ([UNIT=] u, [FMT=] f [,ERR=s] [,END=sn] [,IOSTAT=ios]) [iolist]
```

or:

```
READ f [,iolist]
```

where:

UNIT= *u*

is a unit specifier (see 5.2.1); *u* is a unit identifier that must be present in the first form. The UNIT= clause is optional.

FMT=*f*

is a format specifier (see 5.2.4); *f* is a format identifier that must be present. The FMT= clause is optional.

ERR= *s*

is an error specifier (see 5.2.6); *s* is a statement label.

END= *sn*

is an end-of-file specifier (see 5.2.7); *sn* is a statement label.

IOSTAT= *ios*

is an input/output status specifier (see 5.2.8); *ios* is an integer variable or integer array element.

iolist

is an input list (see 5.2.3).

Description:

When the formatted READ statement is executed, one or more formatted records are read from the file specified by *u*. The information in the records is scanned and converted as specified by the format identifier *f*. The resulting values are assigned to the variables specified in the *iolist*. If you use the UNIT= and FMT= clauses, the specifiers can appear in any order. If you use the UNIT= clause, you must use the FMT= clause.

The second form, which doesn't have a unit specifier, implies that the symbiont file AREAD$ is used. This is equivalent to specifying unit number 5 (the AREAD$ symbiont; see G.6) in the first form.

Examples:

```
      READ 10, IVAR1, ARRAY2
10    FORMAT(I8,10E9.2)
C           This formatted READ statement reads (from the
C           program input unit) an integer value into variable
C           IVAR1 and 10 real values into array ARRAY2.


      READ (3, 20,ERR = 140) IVAR1, ARRAY2
20    FORMAT(I8, 10E9.2)
C           This formatted READ statement obtains values from input
C           unit number 3 and assigns them to members of the input
C           list according to the format specified in statement 20.
C           When an error occurs during processing of this statement,
C           program control transfers to statement 140.

      READ (FMT = 30, UNIT = 3, IOSTAT = IVAL) VAR1, ARRAY3
30    FORMAT(F8.2,8E9.2)
C           This formatted READ statement obtains values
C           from input unit 3 and assigns them to members of the input
C           list according to the format specified in statement 30.
C           The order of the clauses is optional when the UNIT =
C           and FMT= clauses are used.  When an error or end-of-file
C           condition occurs during the processing of this statement,
C           a nonzero value (see 5.2.8) is stored in IVAL and
C           control resumes with the first executable statement
C           following the READ statement.  When the statement is
C           successful, a 0 is stored in IVAL.
```

5.6.1.2. Unformatted READ

Purpose:

The unformatted READ statement reads values into items specified in an input list, with no conversion or reformatting of the input values.

Form:

```
READ ([ UNIT=] u [,ERR=s] [,END=sn] [,IOSTAT=ios]) [iolist]
```

where:

UNIT= *u*

is a unit specifier (see 5.2.1); *u* is a unit identifier that must be present. The UNIT= clause is optional.

ERR= *s*

is an error specifier (see 5.2.6); *s* is a statement label.

END= *sn*

is an end-of-file specifier (see 5.2.7); *sn* is a statement label.

IOSTAT= *ios*

is an input/output status specifier (see 5.2.8); *ios* is an integer variable or integer array element.

iolist

is an input list (see 5.2.3).

Description:

When the unformatted READ statement is executed, the next record is read from the file specified by *u*. The record values are assigned to the variables specified by the input list with no conversion. The number of values required by the input list must be less than or equal to the number of values in the unformatted record. If you use the UNIT= clause, the specifiers in the control information list can appear in any order.

Examples:

```
      READ (3) VAR2, VAR3, ARRAY3
C             This statement obtains values from input unit 3 and,
C             without changing their forms, assigns values to variables
C             VAR2 and VAR3 and to the elements of array ARRAY3.

      READ (3, ERR = 150, END = 280) VAR2, VAR3, ARRAY3
C             This statement is executed in the same way as the first
C             example.  However, when an error is detected during
C             statement execution, control transfers to the
C             statement labeled 150 rather than stopping execution.
C             When the end-of-file is reached, program control
C             passes to the statement labeled 280.
```

```
      READ (END = 280, ERR = 150, UNIT = 3) VAR2, VAR3, ARRAY3
C           This statement is equivalent to the previous statement.
C           The optional UNIT= clause is used so that the parameters
C           within the control information list can appear in any
C           order.
```

5.6.1.3. Namelist READ

Purpose:

The namelist READ statement provides a simplified means of obtaining an annotated list of data from an input device.

Form:

```
READ (u, n [,ERR=s] [,END=sn] [,IOSTAT=ios ])
```

or:

```
READ n
```

where:

u

is a unit identifier that must be present in the first form (see 5.2.1).

n

is a namelist name specifier that must be present (see 5.2.5).

ERR= *s*

is an error specifier (see 5.2.6); *s* is a statement label.

END= *sn*

is an end-of-file specifier (see 5.2.7); *sn* is a statement label.

IOSTAT= *ios*

is an input/output status specifier (see 5.2.8); *ios* is an integer variable or integer array element.

Description:

The namelist READ statement allows reading of specially formatted records without specifying an *iolist* on the READ statement. See 5.4 for further details.

The second form doesn't have a unit specifier. It implies that the symbiont file AREAD$ is to be used. This is equivalent to specifying unit identifier 5 in the first form (see G.6).

Example:

```
      DIMENSION ARR4(10), ARR5(4,5)
      NAMELIST/SPECN1/VAR3, VAR4, ARR4, ARR5, VAR5, VAR6
      READ (5, SPECN1, ERR = 140)
C           Data for the variables specified in the namelist called
C           SPECN1 are read in.
```

5.6.1.4. List-Directed READ

Purpose:

The list-directed READ statement reads in specially formatted records without specification of an associated FORMAT statement.

Form:

READ ([UNIT=] *u*, [FMT=] * [,ERR=*s*] [,END=*sn*] [,IOSTAT=*ios*])[*iolist*]

or:

READ * [, *iolist*]

where:

UNIT= *u*

is a unit specifier (see 5.2.1); *u* is a unit identifier that must be present in the first form. The UNIT= clause is optional.

FMT= *

is a format specifier (see 5.2.4); * indicates a list-directed statement and must be present. The FMT= clause is optional.

ERR= *s*

is an error specifier (see 5.2.6); *s* is a statement label.

END= *sn*

is an end-of-file specifier (see 5.2.7); *sn* is a statement label.

IOSTAT= *ios*

is an input/output status specifier (see 5.2.8); *ios* is an integer variable or integer array element.

iolist

is an input list (see 5.2.3).

Description:

When the list-directed READ statement is executed, data is read from the file specified by *u*. These data values are assigned to the variables specified by *iolist*.

The second form implies the symbiont file AREAD$ is used.

A record containing list-directed input data consists of constants, null values, and value separators.

When the UNIT= and FMT= clauses are present, the specifiers can appear in any order in the control information list. When you use the UNIT= clause, you must use the FMT= clause.

The ERR, END, and IOSTAT clauses can appear in any order.

See 5.5 for additional details.

Examples:

```
      READ *, VAR3, VAR4, ARR5, ARR6
C             Reads in formatted values for the specified variables
C             and arrays.

      READ (7, *, END = 260, ERR = 140) VAR1, VAR2
C             Reads formatted values from records of unit 7.  When an
C             error is detected during execution of this READ
C             statement, control passes to statement 140.  When
C             fewer records exist in the input file than required
C             to fill the elements in the list, control transfers
C             to the statement labeled 260.

      READ (ERR = 140, FMT = *, END = 260, UNIT = 7) VAR1, VAR2
C             This statement is the same as the previous READ except
C             the UNIT= and FMT= clauses allow the control information
C             list items to appear in any order.
```

5.6.1.5. Reread

A unit specifier can be assigned to reread the last formatted symbiont or SDF sequential record read. Thus, it is possible to read the last record any number of times. Input items read with edit descriptors according to their types can, for instance, be reread with an A edit descriptor as alphanumeric information. Subsequent formatted READ statements change the record to be reread. The last record read is unavailable when OPEN or DEFINE FILE statements are executed that define a record size larger than previously defined or defaulted record sizes.

The unit number zero (0) is a default reread unit. Other units can be declared as reread units by generating a new file reference table (see G.6) or through the OPEN statement (see 5.10.1). However, only a unit that is closed or never opened can be opened for rereading with an OPEN statement. The format of the reread statement is:

```
READ ([UNIT= ] u, [FMT= ] f [,ERR= s] [, IOSTAT= ios]) [iolist]
```

When rereading a record, the BLANK=NULL clause provided by the OPEN statement is ignored. All blanks other than leading blanks are treated as zeros unless the BN edit descriptor is used.

Through the use of the reread capability, more than one attempt to read a record can be made. If an error results from a READ with one format, an ERR clause can direct control to another READ that can reread the record with a different format.

If, for example, 0 is the reread unit number and X and Y are formats, the following sequence can make three attempts to read a record:

```
     READ (5,X,ERR=10) A, B, C
      .
      .
      .
10   READ (0,Y,ERR=20) D, E, F
      .
      .
      .
20   READ (0,*,IOSTAT=IVAL) G, H, I
     IF (IVAL.NE.0) GO TO 50
      .
      .
      .
50   STOP 777
```

If the record cannot be read with format X, format Y is tried. If that attempt results in an error, the list-directed read is used.

An end-of-file specifier is ignored on a reread statement. The reread statement must not request more than one record.

5.6.2. Output Statements

For each type of READ statement, there is a corresponding WRITE statement. The various WRITE statement forms depend on the type of record (formatted, unformatted, **namelist,** or list-directed) written. WRITE statements transfer data from variables into files. PRINT is a form of WRITE statement.

5.6.2.1. Formatted WRITE

Purpose:

The formatted WRITE statement writes specified data into a specific file according to a given format.

Form:

```
WRITE([UNIT=]u,[FMT=]f[,ERR=s][,END=sn][,IOSTAT=ios])[iolist]
```

or:

```
PRINT f [, iolist]
```

or:

```
PUNCH f [, iolist]
```

where:

UNIT= *u*

is a unit specifier (see 5.2.1); *u* is a unit identifier that must be present in the first form. The UNIT= clause is optional.

FMT= *f*

is a format specifier (see 5.2.4); *f* is a format identifier that must be present. The FMT= clause is optional.

ERR= *s*

is an error specifier (see 5.2.6); *s* is a statement label.

END= *sn*

is an end-of-file specifier (see 5.2.7); *sn* is a statement label.

IOSTAT= *ios*

is an input/output status specifier (see 5.2.8); *ios* is an integer variable or integer array element.

iolist

is an output list (see 5.2.3).

Description:

The formatted WRITE statement writes one or more formatted records to the file specified by *u*. The variables specified by the *iolist* are edited according to the format specifier *f* and written to file *u*. If the UNIT= and FMT= clauses are present, the specifiers in the control information list can be in any order. When you use the UNIT= clause, the FMT= clause must appear.

The PRINT form implies that the symbiont file APRINT$ is used as the output file. This is equivalent to the WRITE form with a file specifier of 6 (the APRINT$ symbiont; see G.6).

The PUNCH form implies that symbiont file APUNCH$ is used as the output file. This is equivalent to the WRITE form with a file specifier of 1 (the APUNCH$ symbiont, see G.6).

Examples:

```
      WRITE(8, 40, ERR = 160) VAR1, VAR2, ARR1
C           The contents of variables VAR1 and VAR2 and of array
C           ARR1 are written into file unit 8 using the
C           format specified in the statement labeled 40.  When an
C           error occurs during processing of this statement,
C           control transfers to the statement labeled 160.

      WRITE (FMT = 40, UNIT = 8, ERR = 160) VAR1, VAR2, ARR1
C            This is the same as the first WRITE except that with the
C            presence of the UNIT= and FMT= clauses, the clauses in the
C            control information list can be in any order.

      PRINT 40, VAR1, VAR2, ARR1
C            Prints the same values on system printer.

     PUNCH 40, VAR1, VAR2, ARR1
C            Punches the same values on cards.
```

5.6.2.2. Unformatted WRITE

Purpose:

The unformatted WRITE statement writes a record consisting of the specified data to a given file.

Form:

```
WRITE ([UNIT=] u [,ERR=s] [,END=sn] [,IOSTAT=ios]) [iolist]
```

where:

UNIT= *u*

is a unit specifier (see 5.2.1); *u* is a unit identifier that must be present. The UNIT= clause is optional.

ERR= *s*

is an error specifier (see 5.2.6); *s* is a statement label.

END= *sn*

is an end-of-file specifier (see 5.2.7); *sn* is a statement label.

IOSTAT= *ios*

is an input/output status specifier (see 5.2.8); *ios* is an integer variable or integer array element.

iolist

is an output list (see 5.2.3).

Description:

The unformatted WRITE statement writes one record to the file specified by *u*. The record contains the sequence of values specified by the *iolist*.

Examples:

```
      WRITE (8) ARR3, VAR1, VAR2
C             Writes a record to file 8 without reformatting of
C             data in the output list.

      WRITE (ERR = 180,UNIT = 8) ARR3, VAR1, VAR2
C             Same as the previous example except that control passes
C             to the statement labeled 180 when an error is
C             encountered while executing this statement.  With the
C             presence of the UNIT= clause, the clauses in the
C             control information list can be in any order.
```

5.6.2.3. Namelist WRITE

Purpose:

The namelist WRITE statement lets you write a particular list of variables and arrays without including the output list in the WRITE statement.

Form:

```
WRITE (u, n [, ERR= s] [,IOSTAT= ios ])
```

or:

```
PRINT n
```

or:

```
PUNCH n
```

where:

u

is a unit specifier that must be present in the first form (see 5.2.1).

n

is a namelist name specifier that must be present (see 5.2.5).

ERR= *s*

is an error specifier (see 5.2.6); *s* is a statement label.

IOSTAT= *ios*

is an input/output status specifier (see 5.2.8); *ios* is an integer variable or integer array element.

Description:

One or more records are written to the file specified by *u*. The records consist of data values specified in the NAMELIST set named *n*.

The PRINT statement implies that the symbiont file APRINT$ is used. This is equivalent to specifying unit 6 (the APRINT$ symbiont; see G.6) in the first form.

Refer to 5.4 for additional details on NAMELIST.

Examples:

```
      NAMELIST /SPECN2/ VAR1, ARR2, VAR3, ARR4, VAR5
      WRITE (8, SPECN2)
C             Writes each name specified in namelist SPECN2
C             on a record with its associated values.

      WRITE (8, SPECN2, ERR = 180)
C             Same as the previous example except that program
C             control transfers to the statement labeled 180
C             when an error occurs during statement execution.
```

5.6.2.4. List-Directed WRITE

Purpose:

The list-directed WRITE statements write out specially formatted records without specifying an associated FORMAT statement.

Form:

```
WRITE ([UNIT=]u,[FMT=] * [,ERR=s] [,END=sn] [,IOSTAT=ios])[iolist]
```

or:

```
PRINT *[, iolist]
```

or:

```
PUNCH *[, iolist]
```

where:

UNIT= *u*

is a unit specifier (see 5.2.1); *u* is a unit identifier that must be present in the first form. The UNIT= clause is optional.

FMT= *

is a format specifier (see 5.2.4); * indicates the statement is list-directed and must be present. The FMT= clause is optional.

ERR= *s*

is an error specifier (see 5.2.6); *s* is a statement label.

END= *sn*

is an end-of-file specifier (see 5.2.7); *sn* is a statement label.

IOSTAT= *ios*

is an input/output status specifier (see 5.2.8); *ios* is an integer variable or integer array element.

iolist

is an output list (see 5.2.3).

Description:

When a list-directed WRITE statement is executed, data is written to the file specified by *u* (or to the assumed print or **punch** file).

The PRINT form implies that the symbiont file APRINT$ is used. This is equivalent to specifying unit number 6 (the APRINT$ symbiont; see G.6) in the WRITE form.

The PUNCH form implies that the symbiont file APUNCH$ is the output file, which is equivalent to specifying unit number 1 (the APUNCH$ symbiont, see G.6) in the WRITE form.

If the UNIT= and the FMT= clauses are present, the specifiers in the control information list can be in any order.

When you use the UNIT= clause, the FMT= clause must be present.

See 5.5 for more information on list-directed input/output statements.

Examples:

```
      WRITE (8, *, ERR = 180) VAR3, VAR4, ARR5, ARR6
C           Writes out formatted data values to output file 8.
C           When an error occurs during processing, program control
C           transfers to the statement labeled 180.

      WRITE (ERR = 180, UNIT = 8, FMT = *) VAR3, VAR4, ARR5, ARR6
C           This is the same as the previous WRITE except that
C           the UNIT= and FMT= clauses are present and the clauses
C           in the control information list can be in any order.

      PRINT *, VAR3, VAR4, ARR5, ARR6
C            Writes formatted data values from the output list to
C            the output print file APRINT$.

      PUNCH *, VAR3, VAR4, ARR5, ARR6
C            Punches these same values on cards using the output
C            punch file APUNCH$.
```

5.6.3. BACKSPACE Statement

Purpose:

The BACKSPACE statement repositions the file pointer for a file stored on a magnetic tape, disk, or drum unit by backspacing one record. System files (for example, AREAD$, APRINT$, and APUNCH$) and files not created by PCIOS can't be backspaced (see G.2.3). Backspacing over a list-directed record is also prohibited.

Form:

```
BACKSPACE ( [ UNIT= ] u [ ,ERR= s] [ ,IOSTAT= ios ] )
```

or:

```
BACKSPACE u
```

where:

UNIT= *u*

is a unit specifier (see 5.2.1); *u* is a unit identifier that must be present. The UNIT= clause is optional.

ERR= *s*

is an error specifier (see 5.2.6); *s* is a statement label.

IOSTAT= *ios*

is an input/output status specifier (see 5.2.8); *ios* is an integer variable or integer array element.

Description:

The file specified by *u* is backspaced one record. If the file identified by *u* is already at its initial point, the BACKSPACE statement has no effect.

An end-of-file record counts as one record.

When the UNIT= clause is present, the specifiers in the control information list can be in any order.

Example:

```
45     FORMAT ( ... )
          .
          .
          .
       READ (IOSTAT=IVAL,FMT=45,UNIT=MTAPE) VAR1, VAR2
          .
          .
          .
       READ (MTAPE, 45) VAR1, VAR2
          .
```

```
            .
            .
         BACKSPACE MTAPE
            .
            .
            .
         READ (FMT=45,UNIT=MTAPE) VAR1, VAR2
C             The first READ refers to the first record on MTAPE,
C             the second READ refers to the second record, and the
C             third READ also refers to the second record on MTAPE
C             because the file pointer is backspaced one record.
```

5.6.4. ENDFILE Statement

Purpose:

The ENDFILE statement marks the end of a file.

Form:

```
ENDFILE ([ UNIT=] u [,ERR=s] [,IOSTAT= ios])
```

or:

```
ENDFILE u
```

where:

UNIT= *u*

is a unit specifier indicating the file to be demarcated (see 5.2.1); *u* is a unit identifier that must be present. The UNIT= clause is optional.

ERR= *s*

is an error specifier (see 5.2.6); *s* is a statement label.

IOSTAT= *ios*

is an input/output status specifier (see 5.2.8); *ios* is an integer variable or integer array element.

Description:

An end-of-file record is written to the file specified by *u*.

System files (for example, AREAD$ and APRINT$) can't have an end-of-file record written on them using this statement.

When the UNIT= clause is present, the specifiers in the control information list can appear in any order.

Example:

```
      ENDFILE MTAPE
C           This statement writes an end-of-file record
C           on MTAPE to mark the end of a file.
```

5.6.5. REWIND Statement

Purpose:

The REWIND statement repositions the file pointer for a file to its initial point.

Form:

```
REWIND ( [ UNIT= ] u [ ,ERR= s ] [ ,IOSTAT= ios ] )
```

or:

```
REWIND u
```

where:

UNIT= *u*

is a unit specifier (see 5.2.1); *u* is a unit identifier that must be present. The UNIT= clause is optional.

ERR= *s*

is an error specifier (see 5.2.5); *s* is a statement label.

IOSTAT= *ios*

is an input/output status specifier (see 5.2.8); *ios* is an integer variable or integer array element.

Description:

This statement positions the file pointer for *u* at the file's initial point (load point). Tapes are rewound to the load point and mass storage files are positioned at logical address 0.

System files (for example, AREAD$, APRINT$, and APUNCH$) cannot be rewound.

When the UNIT= clause is present, the specifiers in the control information list can appear in any order.

Example:

```
55     FORMAT ( ... )
        .
        .
        .
       DO 65 I = 1, 20
         READ (MTAPE,55)
        .
        .
        .
 65    CONTINUE
       REWIND MTAPE
 C          During execution of the DO loop, 20 records are read
 C          from MTAPE.  MTAPE is therefore positioned at the
 C          start of its twenty-first record when the REWIND
 C          statement is encountered.  MTAPE is then repositioned
 C          to its load point.
```

5.6.6. Sequential-Access DEFINE FILE Statement

Purpose:

The sequential-access DEFINE FILE statement describes the characteristics of a file that is used during a sequential-access input/output operation. The three basic types that can be defined are: symbiont, SDF, or ANSI. (See 5.7.1 for use of the DEFINE FILE statement with direct access input/output operations.)

The DEFINE FILE statement must precede all executable statements in the program unit that refer to the file being described. When a DEFINE FILE or OPEN statement is not used to describe a unit specifier, the unit defaults to SDF type unless designated as a symbiont file type in the file reference table (see G.6). The first DEFINE FILE statement encountered for a unit is the one used during execution. Subsequent descriptions for SDF and ANSI have a validity check done on the requested type, and then are ignored. Subsequent descriptions for symbiont units have a similar validity check, and the record size can be changed. All of the capabilities provided by a DEFINE FILE statement are included in the capabilities provided by the OPEN statement.

You must use a DEFINE FILE statement or an OPEN statement when:

- **ANSI interchange tape files are to be processed.**
- **You want sizes other than default block size, record size, or segment size for SDF files.**
- **You want sizes other than the normal default record sizes for symbiont files.**
- **A define-file-block is to contain the file description (see G.10).**

Form:

For symbiont files:

```
DEFINE FILE u (st, ,rs)
```

where:

u

is a unit identifier (see 5.2.1); *u* must be an unsigned integer constant or symbolic name of an integer constant.

st

specifies the symbiont type and can be one of the following:

APRINT for ASCII print symbiont
APUNCH for ASCII punch symbiont
AREAD for ASCII read symbiont
APRNTA for ASCII print alternate symbiont
APNCHA for ASCII punch alternate symbiont
AREADA for ASCII read alternate symbiont

rs

is an integer constant or symbolic name of an integer constant specifying the record size in characters. The ranges allowed for *rs* are:

Symbiont	Range
APRINT, APRNTA	$0 < rs \leq 252$
AREADA	$0 < rs \leq 160$
AREAD	$0 < rs \leq 131{,}071$
APUNCH, APNCHA	$0 < rs \leq 80$

For SDF files:

```
DEFINE FILE u (SDF, , [rs] , [bs] , [ss] )
```

where:

u

is a unit identifier (see 5.2.1); *u* must be an unsigned integer constant or symbolic name of an integer constant.

rs

is an integer constant or symbolic name of an integer constant specifying the record size in words (see G.7 for the default record size). Record size (*rs*) must be specified when records (other than unformatted records) larger than the default size (33 words) are written to an SDF file. Record size (*rs*) specification need not reflect unformatted records as they are read and written in segments.

bs

is an integer constant or symbolic name of an integer constant specifying the block size in words. The default block size is 224 words (see G.2.1.3).

ss

is an integer constant or symbolic name of an integer constant in the range $0 < ss \le 2047$ specifying the record segment size in words (see G.8 for the default segment size). Record segment size (*ss*) is the size at which all records are segmented when they are written to the SDF file. It is also the size at which unformatted records are processed as they are processed in segments.

When you give the record size and choose a proper segment size and block size, the amount of input/output overhead can be minimized (see G.2.1.2 and G.2.1.3).

For ANSI tape files:

```
DEFINE FILE u (ANSI , [rf] , [rs] , [bs] , [bo] )
```

where:

u

is a unit identifier (see 5.2.1); *u* must be an unsigned integer constant or symbolic name of an integer constant.

rf

specifies the record format as shown below. The default record format is U.

rf Code	Records Represented	Record Size (*rs*) Considerations
U	Undefined	Indicate maximum record size.
F	Fixed, unblocked	Indicate exact record size.
FB	Fixed, blocked	Indicate exact record size.
V	Variable, unblocked	Indicate maximum record size, excluding four characters of record control.
VB	Variable, blocked	Indicate maximum record size, excluding four characters of record control.
VS	Variable, unblocked, segmented	Indicate maximum record size excluding the five-character record control segment.
VBS	Variable, blocked, segmented	Indicate maximum record size excluding the five-character record control segment.

The VBS and VS forms specify the ANSI S format. The V and VB forms specify the ANSI D format. The F and FB specify the ANSI F format. The U specifies the ANSI U format.

rs

is an integer constant or symbolic name of an integer constant specifying the record size in characters. The default record size is 132 characters.

bs

is an integer constant or symbolic name of an integer constant specifying the block size in characters. The default block size is 132 characters. The block size (*bs*) for ANSI files depends on the record size (*rs*), the buffer offset (*bo*), and the record format (*rf*) as follows:

- **For U or F record format:**

  ```
  bs = rs + bo
  ```

- **For V record format:**

  ```
  bs = rs + 4 + bo
  ```

- **For FB record format:**

  ```
  bs = n(rs) + bo
  ```

  where *n* is the number of records per block.

- **For a VB record format:**

  ```
  bs = rs + 4 + bo
  ```

- **For VS or VBS record format:**

  The block size can be less than, equal to, or greater than the record size, but it must be large enough to contain any buffer offset and the five characters of control information appended to each record segment.

bo

is an integer constant or symbolic name of an integer constant in the range $0 \leq bo \leq 99$, which is the number of characters specifying the buffer offset. The default for buffer offset is 0. It must be specified if an ANSI input file contains buffer offset information in the front of the data blocks. This parameter isn't required for ASCII FORTRAN-produced ANSI tapes since no buffer offset information is appended to the data blocks.

Description:

As illustrated in the form section, this statement can have one of three general forms, depending on the type of file format.

When you omit an optional parameter, you can omit its preceding comma only when it is a trailing comma (no other parameters follow it). When an optional parameter is omitted, default parameters are used.

Use the DEFINE FILE statement for symbiont files to define a unit that is not already defined as a symbiont file in the file reference table. It can also specify a record size other than the default record size of 132 characters for print symbiont files and 80 characters for a punch or read file. When the unit is already defined as a symbiont file in the file reference table, then the symbiont type on the DEFINE FILE statement must match its definition. In this case, you are only specifying the maximum record size for the symbiont file. When the symbiont file is a print file and the record size is larger than 132 characters, an Executive Request is done to the print symbiont to expand the image size for the file. Therefore, to specify a 160-character print record, the DEFINE FILE statement is used as follows:

```
DEFINE FILE u (APRINT , , 160)
```

or:

```
DEFINE FILE u (APRNTA , , 160)
```

On program termination, the primary print symbiont is reset to the system default record size.

For the punch symbionts, a record size of over 80 characters isn't allowed. When a record size of over 80 characters is specified, the record size for the punch file defaults to 80 characters.

A table of default and maximum values for symbiont files appears in G.10.

Examples:

```
      DEFINE FILE 3 (ANSI,FB,50,205,5)
C            This statement defines file 3 in ANSI format
C            with fixed, blocked records of 50 character length.
C            The block size is 205 characters with a buffer offset
C            of 5 characters.

      DEFINE FILE 9 (SDF,,45,336)
C            The SDF file 9 has records 45 words long
C            and a block size of 336 words.
```

5.7. Direct-Access Input/Output Statements

The direct-access statements let you read and write records randomly from any defined location within a direct file.

There are three direct-access input/output statements:

1. READ (reads records from a direct file)
2. WRITE (writes records to a direct file)

3. **FIND (allows prepositioning of a direct file during program execution; this allows overlapping of record searching with other activities).**

Use the OPEN **and DEFINE FILE** statements in conjunction with these statements.

Using these statements, you can go directly to any point in a direct file, process a record, and go directly to any other point without having to process all the records in between. This is done using the relative record number. The relative record number is unique and is the record's relative position within the file.

Direct files can contain formatted records, unformatted records, **or both.** When formatted records are used, a FORMAT statement must specify the form in which data is transmitted.

5.7.1. Direct-Access DEFINE FILE Statement

Purpose:

The direct-access DEFINE FILE statement describes the characteristics of a direct file that is used during a direct-access input/output operation. This statement must logically precede any READ, WRITE, or FIND statements pertaining to the file it describes.All of the capabilities provided by a DEFINE FILE statement are included in the capabilities provided by the OPEN statement.

Form:

```
DEFINE FILE u (m, s, b, v ) [ ,u(m, s, b, v) [ , ...u (m, s, b, v) ] ]
```

where:

u

is a unit identifier (see 5.2.1); *u* must be an unsigned integer constant or symbolic name of an integer constant.

m

represents an integer constant or symbolic name of an integer constant that specifies the maximum number of records the file will contain.

s

represents an integer constant or symbolic name of an integer constant that specifies the maximum record size for a file. The record size is interpreted as characters or words depending on *b*.

b

is a letter that indicates whether the file is to contain formatted or unformatted records, or both. *b* can be one of the following:

L

File contains formatted or unformatted records or both. The maximum record size is measured in characters.

M

Same as L, except the file is skeletonized when it is created or extended.

E

File contains formatted records. The maximum record size is measured in characters.

F

Same as E, except the file is skeletonized when it is created or extended.

U

File contains unformatted records. The maximum record size is measured in number of words.

V

Same as U, except the file is skeletonized when it is created or extended.

v

represents an unsubscripted integer variable called an associated variable. It receives a value after execution of its associated DEFINE FILE, READ, WRITE, or FIND statement as described in the following paragraphs. The associated variable can't be used as an argument in a function or subroutine reference.

Description:

When you have already assigned file *u* and it doesn't have enough space to contain *m* records, use the OPEN statement to extend the file. When the file is not assigned, then a dynamic assignment via @ASG,T (using *m* and *s* to calculate the file length) is performed. The maximum number of records specification is only relevant when a direct-access file is being defined for the first time.

When the record size, *s*, is in characters, the record is actually stored in mass storage using word-sized locations. The number of words necessary to contain the record plus a constant and the record number are used to calculate the location in mass storage.

At the conclusion of each READ and WRITE, *v* is set to a value that points to the record that immediately follows the last record transmitted. At the conclusion of a DEFINE FILE, *v* is set to 1. At the conclusion of a FIND, *v* is set to a value that points to the record found. When the associated variable is used in a READ or WRITE statement to specify the record number, sequential processing is automatically secured. The associated variable can't appear in the

input/output list of a READ or WRITE statement for a file associated with the DEFINE FILE statement.

When more than one DEFINE FILE statement exists for a given file *u*, the first one encountered for the file is the one used during program execution. Subsequent descriptions of the file are ignored other than performing a validity check on the type (SDF, ANSI, symbiont).

Examples:

```
      DEFINE FILE 3(50,120,L,I), 4(120,50,L,J)
C           This DEFINE FILE statement describes two files as numbers
C           3 and 4.  The data in the first file consists of 50
C           records, each with a maximum length of 120 characters.
C           The L specifies that the file can contain formatted
C           and/or unformatted records.  I is the associated
C           variable that points to the next record.

C           The data in the second file consists of 120 records,
C           each with maximum length of 50 characters.  The L
C           specifies that the file contains formatted or
C           unformatted records or both; J is the associated
C           variable that points to the next record.

      DEFINE FILE 3(50,30,U,I),4(120,13,U,J)
C           Same as the previous statement, except that the data is
C           transmitted without format control.

      DEFINE FILE 3(50,120,E,I),4(120,50,E,J)
C            Same as the first statement, except that a FORMAT
C            statement is required and the data is transmitted
C            under format control.
```

5.7.2. Direct-Access READ Statement

Purpose:

The direct-access READ statement transfers records from a file to internal storage. The file read must be previously defined with an OPEN **or DEFINE FILE** as a direct-access file.

Form:

```
READ ([UNIT=] u [,[FMT=]f] ,REC=r [,ERR=s] [,IOSTAT=ios]) [iolist]
```

or:

```
READ (u'r [ , f] [ ,ERR = s] [ ,IOSTAT = ios] ) [iolist]
```

where:

UNIT= *u*

is the same unit specifier (see 5.2.1) as for an associated OPEN **or DEFINE FILE** statement; *u* is a unit identifier that must be present. The unit identifier must be followed by an apostrophe for the second form. The UNIT= clause is optional for the first form and must not be present for the second form.

FMT= *f*

is a format specifier (see 5.2.4); *f* is a format identifier. The FMT= clause is optional for the first form and must not be present for the second form.

REC= *r*

is a record specifier (see 5.2.2); *r* is an integer expression that represents the relative position of a record within the file. The REC= clause must be present for the first form and must not be present for the second form.

ERR= *s*

is an error specifier that transfers control to statement label *s* when an error is detected in the execution of the READ statement (see 5.2.6).

IOSTAT= *ios*

is an input/output status specifier (see 5.2.8); *ios* is an integer variable or integer array element.

iolist

is an ordered input list of variables that receive the record read (see 5.2.3).

Description:

A record is read from position *r* of file *u* (which is described previously by an OPEN **or DEFINE FILE** statement).

When the UNIT= clause is present for unformatted I/O statements, the specifiers can appear in any order. When the UNIT= clause is present for formatted I/O statements, the FMT= clause must be used.

The relative record number of the first record in a direct-access file is 1.

Example:

```
      OPEN (UNIT=3, RECL=120, RCDS=50, RFORM='E', ASSOC=I)
        .
        .
        .
      READ(FMT=222,REC=43,UNIT=3,ERR=140)VAR1,ARR1

or:

      READ (3'43,222,ERR = 140) VAR1, ARR1
C           These direct-access reads transfer the contents of the
C           43rd record on unit 3 to VAR1 and array ARR1 according
C           to the format specified at the statement labeled 222.
```

5.7.3. Direct-Access WRITE Statement

Purpose:

The direct-access WRITE statement transfers records to mass storage from internal storage. The file written must be previously defined with an OPEN **or DEFINE FILE** statement as a direct-access file.

Form:

```
WRITE ([UNIT=] u [,[FMT=]f] ,REC=r [,ERR=s][,IOSTAT=ios]) [iolist]
```

or:

```
WRITE (u'r [ ,f] [ ,ERR = s] [,IOSTAT = ios] ) [iolist]
```

where:

UNIT= *u*

is the same unit specifier (see 5.2.1) as for an associated OPEN **or DEFINE FILE** statement; *u* is a unit identifier that must be present. The unit identifier must be followed by an apostrophe for the second form. The UNIT= clause is optional for the first form and must not be present for the second form.

FMT= *f*

is a format specifier (see 5.2.4); *f* is a format identifier. The FMT= clause is optional for the first form and must not be present for the second form.

REC= *r*

is a record specifier (see 5.2.2); is an integer expression that represents the relative position of a record within the file. The REC= clause must be present for the first form and must not be present for the second form.

ERR= *s*

is an error specifier that transfers control to statement label *s* when an error is detected in the execution of the WRITE statement (see 5.2.6).

IOSTAT= *ios*

is an input/output status specifier (see 5.2.8); *ios* is an integer variable or integer array element.

iolist

is an ordered list of variables to be written (see 5.2.3).

Description:

A record is written to position *r* of file *u*. File *u* must be previously described by an OPEN **or DEFINE FILE** statement. When the *iolist* of an unformatted direct-access

WRITE doesn't fill the record, the remainder of the record is undefined. When the *iolist* and the format of a formatted direct-access WRITE don't fill a record, blank characters are added to fill the record.

When the UNIT= clause is present for unformatted I/O statements, the specifiers can appear in any order. When the UNIT= clause is present for formatted I/O statements, you must use the FMT= clause.

Example:

```
      OPEN (UNIT=9, RECL=20, RCDS=75, RFORM='U', ASSOC=I)
        .
        .
        .
      WRITE(REC=53,ERR=140,UNIT=9)VAR1,ARR2
or:

      WRITE (9'53,ERR = 140) VAR1, ARR2
C           These WRITE statements direct the contents of variable
C           VAR1 and array ARR2 into record 53
C           of unit 9 without format control.
```

5.7.4. FIND Statement

Purpose:

The FIND statement prepositions a direct-access file to a given relative position.

Form:

```
FIND ( [ UNIT=] u, REC= r [ ,ERR= s ] [ ,IOSTAT= ios ] )
```

or:

```
FIND (u'r [ ,ERR = s ] [ ,IOSTAT = ios] )
```

where:

UNIT= *u*

> **is a unit specifier (see 5.2.1); *u* is a unit identifier that must be present. The unit identifier must be followed by an apostrophe for the second form. The UNIT= clause is optional for the first form and must not be present for the second form.**

REC= *r*

> **is a record specifier that must be present (see 5.2.2); *r* is an integer expression specifying the position of a record in the file. The REC= clause must be present for the first form and must not be present for the second form.**

ERR= ***s***

is an error specifier (see 5.2.6); ***s*** **is a statement label.**

IOSTAT= ***ios***

is an input/output status specifier (see 5.2.8); ***ios*** **is an integer variable or integer array element.**

Description:

The FIND statement positions the direct-access file to the record requested but returns control without completing any input/output operation. This can locate the next input record while the present record is being processed, thereby increasing the execution speed of the object program.

Normally, records are written immediately following the previous record. They are written anywhere in a sector without regard to sector boundaries. As a result, it is advantageous to use the FIND statement before doing any output if a particular alignment or positioning is required.

When the UNIT= clause appears, the specifiers can appear in any order for the first form.

Example:

```
      FIND (9'54)
C           Finds 54th record of output file 9.

      FIND (REC=44,UNIT=8)
C           Locates 44th record of input file 8.
```

5.8. Input/Output Contingencies

When an error occurs during ASCII FORTRAN I/O processing, the standard action is to print an error message, possibly call the ASCII FORTRAN interactive postmortem dump (see 10.4), and terminate the program. To prevent program termination and to allow the use of files that contain multiple logical subfiles (for example, multiple file reels), the ERR, END, and IOSTAT contingency clauses can be used in certain input/output statements.

When the ERR or END clause is used, the error message is not printed and control passes to the statement specified by the particular contingency clause. In addition, the error status is encoded in an input/output status word that can be referenced using the functions IOC, IOS, and IOU.

When the IOSTAT clause is present, control returns to the executable statement following the I/O statement containing the IOSTAT clause (without printing the error message) unless an END or ERR clause is present. The IOSTAT clause also returns the input/output status.

5.8.1. Input/Output Contingency Clauses

The contingency clauses are written as optional parts of a particular input/output statement. The order of the clauses (if any are used) is immaterial. Use them to transfer control to the statement indicated when the specified contingency condition is encountered while executing the input/output statement. The forms of the contingency clauses are:

ERR = *s*	(described in 5.2.6)
END = *sn*	(described in 5.2.7)
IOSTAT = *ios*	(described in 5.2.8)

When transfer is made to the statement specified by the ERR clause, the input/output status word is set to indicate the cause of the error or warning, the unit specifier for the file in error, and, in certain cases, a substatus. See Appendix G for a detailed description of the input/output status word.

The three fields (cause, unit, substatus) of the I/O status word are all coded integers and can be tested or retrieved using the functions:

Function	Field Reference
IOC( )	cause
IOU( )	unit
IOS( )	substatus

The particular field referred to is set to 0 following the reference. The I/O status word is set when an I/O error occurs. It remains set until the next I/O error occurs or until it is referred to through the functions IOC, IOU, and IOS, in which case only the field referred to is set to 0.

5.8.2. Input/Output Error Messages

There are two types of input/output errors:

1. Those causing a OS 1100 Executive contingency interrupt
2. Those detected by the ASCII FORTRAN input/output modules

When a contingency interrupt occurs, control is given to the ASCII FORTRAN contingency routine. This routine prints the contingency error message, may call the ASCII FORTRAN interactive postmortem dump (see 10.4), closes all open files, and terminates the program. The ERR clause is not valid for this type of error.

When the error is detected by the I/O handler and no ERR or IOSTAT clause is specified, an error message is printed, all open files are closed, the FORTRAN postmortem dump routine (FTNPMD) can be called, and the program terminates. When the IOSTAT clause is present, the message isn't printed and the program continues with the executable statement following the I/O statement containing the IOSTAT clause. When the ERR clause is specified, the message isn't printed but the I/O status word is set and transfer is made to the statement specified by the ERR clause.

Some error conditions are detected that aren't considered serious enough to cause program termination. This less serious class of error conditions is referred to as warnings. When neither an ERR nor an IOSTAT clause is present when one of these conditions is detected, the I/O status word is set, a warning message is printed, and execution of the current statement continues. The presence of only an ERR clause sets the I/O status word, and transfer is made to the statement specified by the ERR clause without completing execution of the current statement.

The warning and error messages for I/O are itemized in Appendix G.

5.9. Internal Files

Internal files provide a means of transferring and converting data from internal storage to internal storage. The internal file READ and WRITE **and the ENCODE and DECODE** statements are similar to sequential-access formatted READ and WRITE statements for external files. However, rather than transferring data between a file and main storage, data is transferred between storage sequences in main storage. Therefore, it is possible to move information from a storage block to a data list while manipulating it with format specifications without a physical peripheral device.

5.9.1. Internal File Formatted READ

Purpose:

The internal file formatted READ statement reads values into items specified in an input list according to a specified format.

Form:

```
READ ([UNIT=] u [FMT=]f [,ERR=s] [,END=sn] [,IOSTAT=ios]) [iolist]
```

where:

UNIT=

is a unit specifier (see 5.2.1); *u* is a character variable, character array, character array element, or character substring that must be present. The UNIT= clause is optional.

FMT=*f*

is a format specifier (see 5.2.4); *f* is a format identifier that must be present. The FMT= clause is optional.

ERR= *s*

is an error specifier (see 5.2.6); *s* is a statement label.

END= *sn*

is an end-of-file specifier (see 5.2.7); *sn* is a statement label.

IOSTAT= *ios*

is an input/output status specifier (see 5.2.8.); *ios* is an integer variable or integer array element.

iolist

is an input list (see 5.2.3).

Description:

The internal file formatted READ statement reads one or more formatted records from the internal file specified. The file is a character variable, character array, character array element, or character substring.

When the file is a character variable, character array element, or a character substring, it consists of a single record whose length is the length of the character variable, array element, or substring.

When the file is a character array, each array element is a record of the file. The order of the records in the file is the order of the array elements in the array. The length of the record is the length of the array element.

The information in the record is scanned and converted as specified by the format identifier *f*. The resulting values are assigned to variables specified in *iolist*. An empty FORMAT [FORMAT( )] and an asterisk for *f* aren't allowed for an internal file READ statement that contains an *iolist*.

When the optional UNIT= and FMT= clauses appear, the control information list specifiers can appear in any order. When the UNIT= clause is present, the FMT= clause must appear.

An internal file is always positioned at the beginning of the first record before the READ statement.

The end-of-file condition is raised when an attempt is made to read beyond the end of the internal file.

5.9.2. DECODE Statement

Purpose:

The DECODE statement converts characters in internal records starting at *b* to data items according to the format *f* and stores them in the I/O list items in *iolist*.

Form:

```
DECODE ( [c, ] f,b [ ,t] [ ,ERR = s] ) iolist
```

where:

c

is an unsigned integer constant, symbolic name of an integer constant, integer variable, or integer array element whose value specifies the number of characters per internal record to be scanned.

f

is the statement label of a FORMAT statement or the name of an integer variable containing the statement label of a FORMAT statement or the name of an array that contains format specifications.

b

is the name of an array (except assumed-size array), array element, or variable from which data is transferred.

t

is the name of an integer variable or array element which, upon completion of the operation, contains the number of characters actually scanned during execution of the DECODE statement.

ERR= *s*

is an error specifier that transfers control to statement label *s* when an error is detected in the execution of the DECODE statement (see 5.2.6).

iolist

is as specified for a READ statement and contains the items whose values are set by decoding the information in *b*.

Description:

You can unconditionally omit *t* and ERR=*s*. *c*, with its following comma, can be omitted if *t* is also omitted. When *c* is omitted, the internal record size is assumed to be 132 characters.

When the DECODE statement is executed, the characters in the internal records in *b* are converted to data items, according to the FORMAT specified by *f*, and stored into the elements specified in *iolist*.

If the format is empty, the characters in the internal records are converted according to the type of items in the list. The characters of the internal records to be processed are contained in consecutive character positions of *b* with the first character of the first record in the first character position (leftmost quarter) of *b*. When a new record is to be processed, the first character of the record is contained in the first character position following the previous record.

Thus, the first character of the first record is contained in the first character position of *b,* the first character of the second record is contained in the (*c* + first) character position, etc. If *c* is omitted, the second record begins in character position 133.

If the interaction of *iolist* with *f* requires more characters to be read from a record than the count specified by *c,* or 132 if *c* is omitted, a warning message is given. A list-directed DECODE can read more than one record.

After execution of the DECODE statement, *t,* if specified, contains the total number of characters scanned for the records processed. The count doesn't include a count of the trailing characters that are skipped over when terminating the processing of a record.

Example:

```
      CHARACTER*7 P(6)
      P(1)='00001.2'
      P(2)='00023.0'
      P(3)='00075.6'
2     FORMAT (3G7.2)
      DECODE (23,2,P,L) U,V,W
C            The DECODE statement generates the following
C            internal values:
C                  U = 1.2
C                  V = 23.0
C                  W = 75.6
C                  L = 21
```

5.9.3. Internal File Formatted WRITE

Purpose:

The internal file formatted WRITE statement writes specified data into a specific internal file according to a given format.

Form:

```
WRITE ([UNIT=] u, [FMT=] f [,ERR= s] [,IOSTAT= ios]) [iolist]
```

where:

UNIT= *u*

is a unit specifier (see 5.2.1); *u* is a character variable, character array, character array element, or character substring. The UNIT= clause is optional.

FMT= *f*

is a format specifier (see 5.2.4); *f* is a format identifier that must be present. The FMT= clause is optional.

ERR= *s*

is an error specifier (see 5.2.6); *s* is a statement label.

IOSTAT= *ios*

is an input/output status specifier (see 5.2.8); *ios* is an integer variable or integer array element.

iolist

is an output list (see 5.2.3).

Description:

The internal file formatted WRITE statement writes one or more formatted records to the internal file specified. The file can be a character variable, character array, character array element, or character substring.

When the file is a character variable, array element, or substring, it consists of a single record whose length is the length of the character variable, array element, or substring.

When the file is a character array, each array element is a record of the file. The order of the records in the file is the order of the array elements in the array. The length of the record in a character array is the length of the character array element.

The information in the list items is converted and stored to the internal file as specified by the format specifier. An empty format [FORMAT( )] isn't allowed. Don't use the asterisk for *f*.

When the optional UNIT= and FMT= clauses appear in the control information list, the specifiers can appear in any order. When you use the UNIT= clause, the FMT= clause must also appear.

An internal file is always positioned at the beginning of the first record before the WRITE.

5.9.4. ENCODE Statement

Purpose:

The ENCODE statement converts data items in *iolist* to character form according to the format *f* and places them in records in main storage starting at *b*.

Form:

```
ENCODE ([c,] f, b [,t] [,ERR=s]) iolist
```

where:

c

is an unsigned integer constant, symbolic name of an integer constant, integer variable, or integer array element whose value specifies the number of characters per internal record.

f

is the statement label of a FORMAT statement or the name of an integer variable containing the statement label of a FORMAT statement or the name of an array that contains format specifiers.

b

is the name of an array (except assumed-size array), array element, or a variable to which data is transferred.

t

is the name of an integer variable or array element that, upon completion of the operation, contains the number of characters (excluding trailing blanks) actually generated by execution of the ENCODE statement.

ERR=*s*

is an error specifier that transfers control to statement label *s* when an error is detected in the execution of the ENCODE statement. (See 5.2.6.)

iolist

is as specified for a WRITE statement and contains the items whose values are to be encoded and stored in *b*.

Description:

You can unconditionally omit *t* and ERR=*s*. *c*, with the following comma, can be omitted if *t* is also omitted.

When you omit *c*, the internal record size in the block area is determined according to *f* and *iolist* with a maximum of 132 characters. However, the next internal record in the block area starts with the first character of the next word and the remainder of the last word is blank-filled. When you use an empty FORMAT statement, the internal record size is 132 characters when *c* is omitted.

When the ENCODE statement is executed, the data items specified by *iolist* are converted to character strings according to the FORMAT specified by *f*. When an empty FORMAT is given, the data items in the list are converted according to their type. The characters generated are placed in consecutive character positions of *b* with the first generated character going into the first character position (leftmost quarter) of *b*. When a new record begins and *c* is specified, the first character generated is placed in the first character position following the previous record. Thus, the first character of the first record is placed in the first character position of *b*; the first character of the second record is placed in the (*c* + first character) position, and so on.

When the interaction of *iolist* with *f* causes more characters to be generated than specified by *c* or 132 when *c* is not specified, the extra characters are lost and a warning message is given. They are not placed into the character positions following the current record. When the number of characters

generated is less than that required to fill the record, the remainder of the record is filled with blanks. A list-directed ENCODE may write more than one record.

On completion of execution of the ENCODE statement, *t*, if specified, contains a count of the total number of characters generated for the records. The count doesn't include any trailing blanks generated to fill out the records.

Note: ***It is your responsibility to ensure that the area to which data is transferred (b) is large enough. If it isn't, data transfer may not be complete or other areas can be overwritten unintentionally. The size c relates to the internal record size; the array b can receive multiple records.***

Example:

```
      DIMENSION P(6)
      R = 1.2
      S = 23.0
      T = 75.6
1     FORMAT (3G7.2)
      ENCODE (23,1,P,J) R,S,T
      END
C            This produces an array P which appears in storage
C            as follows:
C
C                1.2ΔΔΔΔ23.ΔΔΔΔ76.ΔΔΔΔΔΔ
C
C            Effectively, the numbers are rounded.  The last
C            two positions are blank-filled because the values
C            stored are less than the record size.  J is
C            equal to 21.
```

5.10. Auxiliary Input/Output Statements

There are three auxiliary input/output statements:

1. OPEN (defines the characteristics of a particular external file).
2. CLOSE (terminates the association of a particular external file to a particular unit and closes the file **or changes the status of a unit from reread to closed).**
3. INQUIRE (returns information about the properties of a particular external file or the association to a particular unit).

You can use the auxiliary input/output statements with either sequential or direct-access files. However, auxiliary input/output statements must not specify an internal file.

5.10.1. OPEN

Purpose:

An OPEN statement defines the characteristics of a particular file that can be used during a sequential- or direct-access input/output operation or that can be referenced by an auxiliary input/output statement. The basic file types which can be defined with an OPEN statement are symbiont, ANSI, and sequential or direct SDF. In addition, the execution of the OPEN statement can:

- associate an existing file with a unit identifier;
- create a file and associate it with a unit identifier;
- change certain characteristics of an association between a file and a unit identifier;
- create a file that is initially associated with a symbiont unit or alternate symbiont unit;
- open the unit if the unit has not previously opened with its associated file;
- **assign a unit number for rereading the last formatted record read;**
- **override and modify the symbiont code for a unit in the FRT; or**
- **declare that the direct-access file is a shared file, using the RFORM clause.**

An OPEN statement can open a unit in any program unit of an executable program. Once opened, it can be referenced in any program unit of the executable program.

A file is considered to exist when it meets one of the following conditions:

- The file is associated with an opened unit.
- The file is assigned to the run. A file that is opened with a status of NEW and is preassigned to a particular size is considered to exist prior to execution of the OPEN statement.
- The file is already cataloged and can be assigned to the run by an @ASG,A control image (@ASG,AY for INQUIRE). When the @ASG assigns the file, the file is freed with a @FREE,R control image.

Creation of a file causes a file to exist that did not previously exist. Deleting a file terminates the existence of the file.

In the form that follows, the unit identifier *u* is required. The remaining clauses are optional, except the record length (RECL=*rl*), which you must provide when the file is opened for direct access. The optional clauses are unordered. The unit specified must exist. Restrictions involving the combinations of clauses allowed are specified under the discussions of the individual clauses.

Form:

```
OPEN([UNIT=]u [,IOSTAT=ios] [,ERR=s] [,FILE=fname] [,STATUS=sta]

     [,ACCESS=acc] [,FORM=fm] [,BLANK=blnk] [,RECL=rl]

     [,RFORM=rfm] [,MRECL=mrcl] [,TYPE=type] [,BLOCK=bsize]

     [,OFF=boff] [,SEG=ssize] [,RCDS=mnum] [,ASSOC=var] [,REREAD=rrd]

     [,PREP=psize] [,BUFR=bufsize])
```

where the clauses are:

UNIT=*u*

u is an integer expression specifying a unit identifier (see 5.2.1). When UNIT= is present, this clause need not be the first. When UNIT= is absent, *u* must precede all other optional clauses. Don't use an asterisk for *u*.

IOSTAT=*ios*

is an input/output status specifier (see 5.2.8); *ios* is an integer variable or integer array element.

ERR=*s*

is an error specifier (see 5.2.6); *s* is a statement label.

FILE=*fname*

fname is a character expression whose value is the name of the external file to be associated with the specified unit. Trailing blanks are ignored. Leading blanks are not allowed. The file name must be of the form:

```
[[qualifier]*]file-name[.].
```

Keys and F-cycles aren't allowed.

When this clause is present, a @USE *unit-num*,*qualifier*file name.* control statement is executed if the unit number doesn't match the file name. This clause can't be present if a status (*sta*) of SCRATCH is specified. When this clause is omitted and the unit isn't opened, the unit identifier is used as the file name.

STATUS=*sta*

sta is a character expression whose value is OLD, NEW, SCRATCH, **EXTEND**, or UNKNOWN. Trailing blanks are ignored. When UNKNOWN, OLD, NEW, or **EXTEND** are specified, a FILE= clause is optional. When you omit this clause, the default value is UNKNOWN.

When you specify OLD, the file must exist.

When you specify NEW and the file isn't preassigned to desired size, a file is created (using an @ASG,CP statement) according to rules 3 and 4 given under the following discussion on UNKNOWN. **New ASCII alternate symbiont files aren't assigned by ASCII FORTRAN. These are assigned by the Executive system in batch mode only. NEW can't be specified for a shared direct-access file.**

When you specify SCRATCH, a file is created (using an @ASG,T statement) with a file name equal to the unit identifier. This file is deleted at the execution of a CLOSE statement referring to the same unit or at the termination of the executable program. **SCRATCH can't be specified for a shared direct-access file.**

When you specify EXTEND, the file must exist.

For direct-access files, EXTEND allows writing records up to a new maximum relative record count specified by the RCDS clause.

For sequential-access files, EXTEND positions the file after the current last record of the file.

The following rules apply:

- **When necessary, reassign the file with a larger track size prior to execution.**
- **A file created on tape can be extended only while residing on tape.**
- **A file created on mass storage can be extended while residing on mass storage or tape; once the file is extended on tape it can't be extended later when residing on mass storage.**
- **A tape file must be extended with the same block size used to create the file.**
- **A mass storage file that is created with a block size other than a multiple of 28 words must be extended with the same block size used to create the file.**
- **A mass storage file created with a block size equal to a multiple of 28 words can be extended with any block size which is a multiple of 28 words.**
- **For direct-access files, the record format used when creating the file must also be used when extending the file. Thus, if a file is skeletonized when initially created, skeletonization must also be specified when the file is extended.**
- **When the file is extended in the same executing program that created it, the file must be closed prior to executing the OPEN statement that extends it.**
- **EXTEND has no effect on symbiont files.**

When UNKNOWN is specified:

1. A check is made to see if the file is assigned to the run. When it is assigned, processing continues.
2. When the file is not assigned, an assignment using an @ASG,A statement is performed. When the file is cataloged and not rolled out, processing continues.
3. When the file is not cataloged and the access mode is direct, an assignment using an @ASG,T *file name*, *///fsize* statement (ASG,CP if its status is NEW) is performed before processing continues. The *fsize* parameter is determined by:

$$\frac{\texttt{mnum*(}\mathit{rl}\texttt{+}\mathit{nbr\text{-}control\text{-}words}\texttt{)+}\mathit{label\text{-}size}\texttt{+}\mathit{eof\text{-}word}}{\mathit{track\text{-}size}}$$

where:

mnum	**see RCDS=*mnum* clause.**
rl	record length in words, see RECL=*rl* clause.
nbr-control-words	(*rl*+2046)/2047.
label-size	112 words.

eof-word	1 word.
track-size	1792 words/track.

When *mnum* is not provided, *fsize* is given a value of 128 tracks.

4. When the file isn't cataloged and the access mode is sequential, an assignment using an @ASG,T *file-name* statement (@ASG,CP if the status is NEW) is performed before processing continues.

Shared direct-access files are not assigned by the @ASG command. See Appendix G.

This discussion of UNKNOWN doesn't apply to ASCII symbiont files. See Appendix G for a discussion on the handling of ASCII symbiont files.

Execution of a DEFINE FILE statement is equivalent to executing an OPEN statement with a status of UNKNOWN with corresponding optional clauses specified.

ACCESS=*acc*

This clause specifies the access method used for the file. *acc* is a character expression whose value is one of the following:

```
SEQUENTIAL    or    SEQ

DIRECT        or    DIR
```

Trailing blanks are ignored.

When you omit this clause, the default value is SEQUENTIAL.

FORM=*fm*

fm is a character expression whose value is FORMATTED or UNFORMATTED. Trailing blanks are ignored. It specifies that the file is opened for formatted or unformatted input/output, respectively.

For SDF direct files, a value of FORMATTED implies a record format of E while a value of UNFORMATTED implies a record format of U (for files containing formatted and unformatted records, see the following discussion on *rfm*).

When you omit this clause **and the RFORM clause is also absent**, a value of UNFORMATTED is assumed if the file is opened for direct access **(a value of U is assumed for a record format; see the following discussion on *rfm*).** A value of FORMATTED is the default value if the file is opened for sequential access with a file type other than ANSI. For ANSI files, the default value is UNKNOWN.

This clause can't appear when the RFORM clause is present and an SDF direct file is opened.

BLANK=*blnk*

blnk is a character expression whose value is NULL or ZERO. Trailing blanks are ignored.

When you specify NULL, all blank characters in numeric formatted input fields on the specified unit are ignored except that a field of all blanks has a value of zero.

When you specify ZERO, all blanks other than leading blanks are treated as zeros.

This clause is permitted only for a file connected for formatted input/output.

When you omit this clause, the default value is NULL. When the OPEN statement is not present for a particular unit specifier, the default value is ZERO. (See BN and BZ edit descriptors in 5.3.1.)

RECL=*rl*

rl is an integer expression in the range 0 to 262,143 that specifies the length of each record in the file opened for direct access. When the file is opened for formatted input/output, the length is the number of characters. When the file is opened for unformatted input/output, the length is measured in words.

This clause must appear only when a file is opened for direct access; it cannot appear when a file is opened for sequential access.

RFORM=*rfm*

***rfm* is a character expression whose value specifies the record format for sequential ANSI files or SDF direct files. Trailing blanks are ignored. Allowable values for sequential ANSI files are:**

U

Undefined. The record is written with no control information appended to it. The records can be variable in length. Each data block contains only one record. When a record size is provided, it indicates the maximum record size. This is the default value for ANSI files.

F

Fixed, unblocked. The record is written with no control information appended to it. All records are of the same length. Each data block contains only one record. When a record size is provided, it must indicate the exact record size.

FB

Fixed, blocked. Basically the same as fixed, unblocked; however, each block contains one or more records. Each data block has the same number of records, except possibly the last block. The last block is truncated if it does not contain the full number of records. When a record size is provided, it indicates the exact record size.

V

Variable, unblocked. Appended to each record are four ASCII characters of control information containing the total number of characters in the record (including the four control characters). The records can be variable in length. The data block contains only one record. If a maximum record size is specified with the MRECL clause, the size need not include the four characters for control information.

VB

Variable, blocked. The same as variable, unblocked except that the block contains as many complete records as possible.

VS

Variable, unblocked, segmented. The record is segmented at the block boundary. Appended to each record segment are five characters of control information. The first character is the record segment indicator and the remaining four are the segment length. The data block contains only one segment of the record. If a maximum record size is specified with the MRECL clause, the size need not include the five characters for control information.

VBS

Variable, blocked, segmented. Similar to variable, unblocked, segmented; however, each block can contain more than one record segment but never more than one segment of the same record.

The VBS and VS forms specify the ANSI S format. The V and VB forms specify the ANSI D format. The F and FB specify the ANSI F format. The U specifies the ANSI U format.

To read a prelevel 9R1 ANSI file (unpacked character data), append '66' to the appropriate record format. For instance, to read a prelevel 9R1 ANSI file that is fixed and blocked, the resulting clause is RFORM = 'FB66'.

Allowable values for SDF direct files are:

L

The file contains formatted or unformatted records or both. The maximum record size is measured in characters.

LS

Same as L, except the file is a shared file.

M

Same as L, except the file is skeletonized when it is created or extended. Skeletonization consists of writing a dummy record for every record in a direct-access file at the time the file is created. A dummy record consists of a record control word with the record type field set to deleted.

MS

Same as M, except the file is a shared file.

E

The file contains formatted records. The maximum record size is measured in characters.

ES

Same as E, except the file is a shared file.

F

Same as E, except the file is skeletonized when it is created or extended.

FS

Same as F, except the file is a shared file.

U

The file contains unformatted records. The maximum record size is measured in number of words. This is the default value when both the FORM and RFORM clauses are absent.

US

Same as U, except the file is a shared file.

V

Same as U, except the file is skeletonized when it is created or extended.

VS

Same as V, except the file is a shared file.

See Appendix G for a discussion on skeletonization.

This clause is not allowed when the FORM clause is present and an SDF direct file is opened.

MRECL=*mrcl*

***mrcl* is an integer expression in the range of 0 to 262,143 that specifies the maximum record size for the file connected for sequential access. The TYPE clause determines the unit of specification. The defaults are:**

APRINT, APRNTA, AREADA	**132 characters**
APUNCH, APNCHA, AREAD	**80 characters**
SDF sequential	**33 words**
ANSI	**132 characters**

The range for symbionts is:

AREADA	**$0 < mrcl \leq 160$**
APRINT, APRNTA	**$0 < mrcl \leq 252$**
AREAD	**$0 < mrcl \leq 131{,}071$**
APUNCH, APNCHA	**$0 < mrcl \leq 80$**

For SDF, maximum record size must be specified when records (other than unformatted records) larger than the default size are to be written. Maximum record size need not reflect unformatted records, as they are read and written in segments.

For ANSI, the maximum record size must be specified when records larger than the default size are read or written, regardless of whether they are formatted or unformatted. The maximum size allowed for an ANSI file is 9,995 characters if the record format is V or VB; otherwise, the maximum size is 99,999 characters.

TYPE=*type*

***type* is a character expression that specifies the file format for files opened for sequential access. The possible values are:**

SDF
ANSI
APRINT
APRNTA
APUNCH
APNCHA
AREAD
AREADA

The default value for this clause is SDF.

For more information about SDF processing, see Appendix G.

ANSI files involve the use of labeled and unlabeled tapes. See Appendix G for more information about ANSI file processing.

This clause can override and modify the symbiont code field specified when the file reference table was built, provided the unit was never opened before or had a closed file status prior to the open. See G.4 for processing of alternate symbiont files.

BLOCK=*bsize*

***bsize* is an integer expression in the range of 0 to 262,143 that specifies the block size for ANSI and SDF sequential files. Refer to Appendix G for more information about block size.**

This clause is optional for SDF sequential files. When omitted, the default value is 224 words.

This clause is optional for ANSI files. When omitted, the default value is 132 characters.

The block size (*bsize*) for ANSI files depends on the maximum record size (*mrcl*), the buffer offset (*boff*), and the record format (*rfm*) as follows:

- **For U or F record format:**

```
bsize = mrcl + boff
```

- **For V record format:**

```
bsize = mrcl + 4 + boff
```

- **For FB record format:**

```
bsize = n (mrcl) + boff
```

- **For VB record format:**

```
bsize ≥ n (mrcl + 4) + boff
```

- **For VS or VBS record format:**

  The block size can be less than, equal to, or greater than the record size, but it must be large enough to contain any buffer offset and the five characters of control information appended to each record segment.

OFF=*boff*

boff is an integer expression in the range 0 to 99 characters that specifies the ANSI buffer offset. The default value for this field is zero.

The variable ***boff*** must be specified if an ANSI input file contains buffer offset information in the front of the data blocks. This clause isn't needed for ASCII FORTRAN-produced ANSI tapes since no buffer offset information is appended to the data blocks. Refer to Appendix G for more information about ANSI files.

SEG=*ssize*

ssize is an integer expression that specifies the record segment size in words for sequential SDF files. The range is 1 ≤ ***ssize*** ≤ 2047. Record segment size is the size that all records are segmented at when they are written to the SDF file. It is also the size at which unformatted records are processed as they are processed in segments. The default value for this clause is 111 words. Refer to Appendix G for more information about record segment size.

RCDS=*mnum*

mnum is an integer expression that specifies the maximum number of records that a direct-access file contains. This clause is only applicable when a direct-access file is opened for the first time or when a direct-access file is opened with STATUS of EXTEND. When absent, the default value for ***mnum*** is:

$$\frac{(\textit{size} * \textit{track-size}) - (\textit{label-size} + \textit{eof-word})}{\textit{rl} + \textit{nbr-control-words}}$$

where:

size

is the preassigned size of the file or 128 tracks (default).

track-size, label-size, eof-word, nbr-control-words

see the description under the STATUS=*sta* clause.

rl

is the record length in words; see the RECL=*rl* clause.

ASSOC=*var*

***var* is an unsubscripted integer variable that is used as an associated variable. The associated variable receives the next relative record number for SDF direct files after the execution of a READ or WRITE statement involving the file, and the specified record number after the execution of a FIND statement. The associated variable is initialized to a value of 1 by the OPEN statement. The associated variable can't be used as an argument in a function or subroutine reference.**

This clause can only appear when the file is opened for direct access.

REREAD=*rrd*

***rrd* is a character expression whose value is ON. Trailing blanks are ignored.**

The only other clauses that can be present when the REREAD clause is present are: ERR, UNIT, and IOSTAT. The OPEN statement may not try to open a unit for REREAD while that unit is open as a normal I/O file. An implicit CLOSE is not performed.

The reread unit rereads the last formatted SDF sequential or symbiont record read (see 5.6.1.5).

PREP=*psize*

***psize* is an integer expression in the range of 0 to 262,143 that specifies the physical prep factor in words of the device on which an SDF direct-access file is stored.**

This clause is optional for SDF direct files. When omitted, the default value is 112 words.

The PREP clause is ignored for shared direct-access files.

BUFR=*bufsize*

***bufsize* is an integer expression in the range 0 to 262,143 that specifies the buffer size in words used for SDF direct files.**

This clause is optional for SDF direct files. When omitted, the default buffer size is the minimum allowable buffer size given by the following formula:

```
minimum bufsize = rl + nbr-control-words + 2*psize + 1
```

where:

rl

is the record length in words; see the RECL = *rl* clause

nbr-control-words

(*rl* + 2046)/2047

psize

is the physical prepping factor; see the PREP=*psize* clause. The 2*psize portion of this formula is ignored for shared direct-access files.

Description:

Once an SDF or ANSI file is initially created, use of the ACCESS, FORM, RECL, **RFORM, MRECL, TYPE, and OFF** clauses on subsequent opens doesn’t update the file header.

Specifiers that can be character expressions can be uppercase, **lowercase, or a mixture of both.**

Multiple OPEN statements without an intervening clause involving the same file and unit number are permitted. Only the BLANK **and MRECL (symbionts only)** clauses can specify a value different from the one in effect.

If the file to be associated with the unit is not the same as the file that is currently associated with the unit, an implicit CLOSE statement is executed for the unit. The default value for the STATUS= clause of the CLOSE statement is used when doing the implicit CLOSE. The unit is then associated with the new file and the new file is opened.

If a file is open, execution of an OPEN statement involving that file and a unit other than the unit currently associated with that file is not permitted. **An implicit CLOSE is not executed if the unit is currently a reread unit. An explicit CLOSE must be executed before the unit can be associated with a file.**

Example 1:

```
INTEGER OUTKOM
OPEN (10, IOSTAT=OUTKOM, ERR=50, STATUS='OLD')
```

This OPEN statement opens a file for sequential access.

If a file is not associated with 10 via a @USE control statement, 10 is used as the file name.

The status of OLD requires that the file must already exist; otherwise, an error condition results.

When an error condition is encountered during execution of the OPEN statement, the variable OUTKOM receives the contents of the I/O status word, PTIOE, contained in the storage control table, and control transfers to the statement with label 50.

The remaining applicable clauses and their default values are:

```
FORM = 'FORMATTED'

BLANK = 'NULL'

MRECL = 33

TYPE = 'SDF'

BLOCK = 224

SEG = 111
```

Example 2:

```
CHARACTER*20 FSTAT
FSTAT = 'UNKNOWN'
OPEN (12, FILE='DATA1', STATUS=FSTAT, ACCESS='DIRECT', RECL=25,
RCDS=1000,
1     ASSOC = NREC)
```

This OPEN statement opens a file for direct access.

A @USE 12,DATA1 control statement is executed. Since the status is UNKNOWN, an attempt is made to assign the file when it is not yet assigned. When the file doesn't exist, a temporary file is assigned.

Since neither the FORM **nor the RFORM** clause is present, the default record format applied specifies that the file contains unformatted records. The file can contain up to 1000 records of 25 words each.

The NEXTREC clause of the INQUIRE statement can be used to determine the next relative record number.

When an error condition is encountered while executing the OPEN statement, an error message is printed and the program terminates. Control doesn't return to you, as both the ERR and IOSTAT clauses are absent.

5.10.2. CLOSE

Purpose:

A CLOSE statement terminates the association of a particular external file with the specified unit and closes the file **or changes the status of a unit from reread to closed.**

A CLOSE statement for a particular unit can occur in any program unit of an executable program and need not occur in the same program unit as the OPEN statement referring to the unit.

An OPEN of a new file followed by a CLOSE doesn't create an empty file with a file header. The file header is not written to the file until the first WRITE.

In the form that follows, the unit identifier *u* must appear. The remaining clauses are optional and (if present) are unordered.

Form:

```
CLOSE ([UNIT=]u [,IOSTAT=ios] [,ERR=s] [,STATUS=sta] [,REREAD=rrd])
```

where:

UNIT=*u*

u is an integer expression specifying a unit identifier (see 5.2.1). When UNIT= is present, this clause need not be the first. When the UNIT= clause is absent, *u* must precede all other optional clauses. Don't use an asterisk for *u*.

IOSTAT=*ios*

is an input/output status specifier (see 5.2.8); *ios* is an integer variable or integer array element.

ERR=*s*

is an error specifier (see 5.2.6); *s* is a statement label.

STATUS=*sta*

sta is a character expression whose value is one of the following:

KEEP
DELETE
REWIND
FREE
COMMIT

Trailing blanks are ignored. These values determine the disposition of the file being closed.

When KEEP is specified, the file continues to exist after execution of the CLOSE statement. KEEP must not be specified for a file whose OPEN status is specified as SCRATCH. If the file is a tape file, the tape remains positioned after the EOF mark.

When DELETE is specified, the file ceases to exist after execution of the CLOSE statement. When the file is assigned with the C or U options, a @FREE,I control statement is executed; otherwise, a @FREE,D control statement is executed. **DELETE should not be specified for a shared direct-access file.**

When you specify REWIND, the file is rewound and then treated as if a KEEP is specified.

When you specify FREE, a @FREE control statement is executed. FREE should not be specified for a shared direct-access file.

When you specify COMMIT, the shared direct-access file is updated at the time of the CLOSE rather than at program termination. This allows you to re-open a shared file as a sequential file.

The default value is KEEP, unless the file's OPEN status is specified as SCRATCH, in which case the default value is DELETE.

[REREAD *rrd*]

rrd is a character expression whose value is OFF. Trailing blanks are ignored.

The STATUS clause can't be present when the REREAD clause is present. The REREAD clause must not be present when the unit is open as a normal I/O file. This clause removes the assignment of the unit as a reread unit. After the CLOSE with REREAD, the unit can be associated with a normal I/O file in the same program unit (see 5.6.1.5).

Description:

Specifiers that can be character expressions can be uppercase, **lowercase, or a combination of both.**

A CLOSE statement that specifies a unit that doesn't exist or isn't open has no effect. Execution continues with the next statement.

After a unit is no longer associated with a file due to the execution of a CLOSE statement, it can be reassociated in the same executable program to the same file or a different file, **or it can be used as a reread unit.**

After a file is closed, it can be associated, in the same executable program, to the same unit or a different unit and opened again.

A CLOSE statement frees the file control buffers for the file closed. Thus, no position information on the file is available.

Example:

```
CLOSE (12,IOSTAT=MSTAT,ERR=100,STATUS='DELETE')
```

This CLOSE statement closes the file associated with unit 12.

Since DELETE is specified, the file no longer exists after execution of the CLOSE statement.

When an error condition is encountered while executing the CLOSE statement, the variable MSTAT receives the contents of the I/O status word, PTIOE, contained in the storage control table, and control transfers to the statement with label 100.

After successful completion of this CLOSE, unit 12 isn't associated with any file.

5.10.3. INQUIRE

Purpose:

An INQUIRE statement returns information about the properties of a particular file or the association to a particular unit. There are two forms of the INQUIRE statement:

1. INQUIRE by file - The list of optional clauses must contain exactly one FILE clause and can't contain a UNIT clause **or a REREAD clause.**
2. INQUIRE by unit - The list of optional clauses must contain exactly one external UNIT clause and can't contain a FILE clause. The unit specified need not exist or be associated with a file.

The INQUIRE statement can be executed before, while, or after a file is opened. All values that the INQUIRE statement assigns are those that are current at the time the statement is executed.

The optional clauses of the INQUIRE statement that follow may be used in any order and can be combined with other clauses as specified in the following discussion.

Form:

```
INQUIRE  ([UNIT=]u [,FILE=fin] [,IOSTAT=ios] [,ERR=s] [,EXIST=ex]
         [,OPENED=od] [,NUMBER=num] [,NAMED=nmd] [,NAME=fn]
         [,ACCESS=acc] [,SEQUENTIAL=seq] [,DIRECT=dir] [,FORM=fm]
         [,FORMATTED=fmt] [,UNFORMATTED=unf] [,RECL=rcl]
         [,NEXTREC=nr] [,BLANK=blnk] [,RFORM=rfm] [,MRECL=mrcl]
         [,TYPE=typ] [,BLOCK=bsize] [,OFF=boff] [,SEG=ssize] [,REREAD=rrd]
         [,PREP=psize] [,BUFR=bufsize])
```

Note: *You may inquire by file or unit but not both. Therefore, either a UNIT or FILE clause must be present.*

where:

UNIT=*u*

u is an integer expression specifying a unit identifier (see 5.2.1). When UNIT= is present, this clause need not be the first. When UNIT= is absent, *u* must precede all other optional clauses. Don't use an asterisk for *u*.

Note: *This clause must not be present when an INQUIRE by file is performed.*

The unit specified need not exist or be associated with a file.

FILE=*fin*

fin is a character expression whose value when any trailing blanks are removed is the name of the external file inquired about. Leading blanks are not allowed. The file name must be of the form:

```
[[qualifier]*]file-name[.]
```

Keys and F-cycles can't be part of *fin* and aren't returned by the INQUIRE statement on *fin*.

Note: *This clause must not appear when an INQUIRE by unit is performed.*

The named file need not exist or be opened.

IOSTAT=*ios*

is an input/output status specifier (see 5.2.8); *ios* is an integer variable or integer array element.

ERR=*s*

is an error specifier (see 5.2.6); *s* is a statement label.

EXIST=*ex*

ex is a logical variable or logical array element. If a file with the specified name exists, the INQUIRE by file statement assigns the value .TRUE. to *ex;* otherwise, *ex* is assigned the value .FALSE.

See 5.10.1 for a discussion of file existence.

If the specified unit exists, the INQUIRE by unit statement assigns the value .TRUE. to *ex;* otherwise, *ex* is assigned the value .FALSE.

A unit is said to exist if:

0 ≤ unit number ≤ FRT length

where FRT is the file reference table (see G.5).

OPENED=*od*

od is a logical variable or logical array element. If the file or unit specified is open, the INQUIRE by file statement or INQUIRE by unit statement assigns the value .TRUE. to *od;* otherwise, *od* is assigned the value .FALSE.

NUMBER=*num*

num is an integer variable or integer array element that is to receive the value of the external unit identifier if the unit is open. For an INQUIRE by file, the unit in question is the unit associated with the file. If no unit is associated with the file, *num* is undefined.

NAMED=*nmd*

nmd is a logical variable or logical array element that is assigned a .TRUE. or .FALSE. value, which indicates the existence of an external file name based on the following conditions:

- For an INQUIRE by file, *nmd* is assigned a value of .TRUE. if the file exists; it need not be opened.
- For an INQUIRE by unit, *nmd* is assigned a value of .TRUE. if the unit is open and not associated with a symbiont. A value of .FALSE. is returned if the unit is open and associated with a symbiont.

NAME=*fn*

fn is a character variable or character array element that is assigned the value of the external name of the file when the file has an external name; otherwise, it is undefined.

The value returned by this optional clause has the form:

*qualifier*file-name.*

ACCESS=*acc*

acc is a character variable or character array element that is assigned the value SEQUENTIAL if the file is open for sequential access, and DIRECT if the file is open for direct access. If the file is not open, *acc* is undefined.

SEQUENTIAL=*seq*

seq is a character variable or character array element that is assigned the value YES if the file is open for sequential access and a value of NO if the file is not open for sequential access. A value of UNKNOWN is returned when the access method for the file can't be determined.

When the file is not open (INQUIRE by unit) or the file doesn't exist (INQUIRE by file), *seq* is undefined.

DIRECT=*dir*

dir is a character variable or character array element that is assigned the value YES if the file is open for direct access and a value of NO if the file is not open for direct access. A value of UNKNOWN is returned when the access method for the file can't be determined.

When the file is not open (INQUIRE by unit) or the file doesn't exist (INQUIRE by file), *dir* is undefined.

FORM=*fm*

fm is a character variable or character array element that is assigned the value FORMATTED if the file is open for formatted I/O, UNFORMATTED if the file is open for unformatted I/O, **and BOTH if the I/O type is mixed**. When the file is not open, *fm* is undefined. List-directed and namelist I/O are considered formatted.

FORMATTED=*fmt*

fmt is a character variable or character array element that is assigned the value YES when the I/O type for the file is formatted, and NO when the I/O type for the file is not formatted. A value of UNKNOWN is returned when the I/O type of the file can't be determined.

When the file is not open (INQUIRE by unit) or the file doesn't exist (INQUIRE by file), *fmt* is undefined.

UNFORMATTED=*unf*

unf is a character variable or character array element that is assigned the value YES when the I/O type for the file is unformatted, and NO when the I/O type for the file is not unformatted. A value of UNKNOWN is returned when the I/O type of the file can't be determined.

When the file is not open (INQUIRE by unit) or the file doesn't exist (INQUIRE by file), *unf* is undefined.

RECL=*rcl*

rcl is an integer variable or integer array element that is assigned the value of the record length of the file if it is open for SDF direct access.

If the file is open for formatted input/output or can contain both formatted and unformatted records, the length is in characters. The length is measured in words when the file is open for unformatted input/output.

When the file is not open or the file is not open for direct access, *rcl* is undefined.

NEXTREC=*nr*

nr is an integer variable or integer array element that receives the next relative record number for the file if it is open for SDF direct access. If the file is open but no records are read or written after the opening, *nr* is assigned the value 1.

When the file is not open for direct access or the position of the file is indeterminate due to a previous error condition, *nr* is undefined.

BLANK=*blnk*

blnk is a character variable or character array element. A value of NULL is returned if the file is open for formatted input/output and null blank control is in effect. A value of ZERO is returned if the file is open for formatted input/output and if zero blank control is in effect.

If the file is not open or if it is not open for formatted input/output, *blnk* is undefined.

RFORM=*rfm*

***rfm* is a character variable or character array element that receives a literal value representing the record format for ANSI files and SDF direct files.**

If the file is open as an ANSI file, one of the following values is returned:

Value Returned	Record Format
U	**Undefined**
F	**Fixed, unblocked**
FB	**Fixed, blocked**
V	**Variable, unblocked**
VB	**Variable, blocked**
VS	**Variable, unblocked, segmented**
VBS	**Variable, blocked, segmented**

If the file is open as an SDF direct file, one of the following values is returned:

Value Returned	Record Format
L	Formatted or unformatted records or both. The maximum record size is measured in characters.
LS	Same as L, except the file is a shared file.
M	Same as L, except the file is skeletonized.
MS	Same as M, except the file is a shared file.
E	Formatted records. The maximum record size is measured in characters.
ES	Same as E, except the file is a shared file.
F	Same as E, except the file is skeletonized.
FS	Same as F, except the file is a shared file.
U	Unformatted records. The maximum record size is measured in words.
US	Same as U, except the file is a shared file.
V	Same as U, except the file is skeletonized.
VS	Same as V, except the file is a shared file.

The *rfm* variable is undefined for all other cases.

MRECL=*mrcl*

mrcl is an integer variable or integer array element that is assigned the value of the maximum record length of the file if it is open for sequential access.

When the file is not open or the file is not open for sequential access, *mrcl* is undefined.

The applicable units of measure are characters or words, as follows:

symbionts

characters

SDF sequential

words

ANSI

characters

TYPE=*typ*

typ is a character variable or character array element that is assigned one of the following file format values if the file is open for sequential access:

SDF
ANSI
AREAD
AREADA
APUNCH
APNCHA
APRINT
APRNTA

If the file is open for SDF direct or not open, *typ* is undefined.

BLOCK=

***bsize* is an integer variable or integer array element that receives the block size if the file is open with a file format of ANSI or sequential SDF.**

For SDF sequential files, the unit of measure is words.

For ANSI files, the unit of measure is characters.

For any other file format or if the file is not open, *bsize* is undefined.

OFF=*boff*

***boff* is an integer variable or integer array element that receives the ANSI buffer offset in characters if the file is open with a file format of ANSI.**

For any other file format or if the file is not open, *boff* is undefined.

SEG=*ssize*

***ssize* is an integer variable or integerarray element that is assigned the value of the SDF record segment size in words. This value is defined only for files open for SDF sequential access.**

REREAD=*rrd*

***rrd* is a character variable or character array element that is assigned the value ON if the unit exists and is assigned as a reread unit, and a value OFF if the unit exists and is not assigned as a reread unit. When the unit doesn't exist, *rrd* is undefined.**

All other clauses except the UNIT, IOSTAT, ERR, and EXIST clauses are ignored when the unit is a reread unit (see 5.6.1.5).

PREP=*psize*

***psize* is an integer variable or integer array element that receives the physical prep factor specified for the device on which an SDF direct-access file is stored.**

When the file is not open for direct access, *psize* is undefined.

The PREP clause is ignored for shared direct-access files.

BUFR=*bufsize*

***bufsize* is an integer variable or integer array element that receives the buffer size used for an SDF direct-ccess file.**

When the file is not open for direct access, *bufsize* is undefined.

Description:

A variable or array element that receives a value in an INQUIRE statement can't be referred to by more than one of the clauses in the same INQUIRE statement.

When an INQUIRE by file statement is executed:

- The variables *nmd*, *fn*, *seq*, *dir*, *fmt*, and *unf* are assigned values only when the value of *fin* is acceptable and when a file by that name exists; otherwise, they are undefined.
- The variable *num* becomes defined if and only if *od* is defined with a value of .TRUE.
- The variables *acc*, *fm*, *rcl*, *nr*, *blnk*, ***rfm, mrcl, typ, bsize, boff, psize, bufsize,* and *ssize*** can be defined only if *od* is defined with a value of .TRUE.

When an INQUIRE by unit statement is executed, *num*, *nmd*, *fn*, *acc*, *seq*, *dir*, *fm*, *fmt*, *unf*, *rcl*, *nr*, *blnk*, ***rfm, mrcl, typ, bsize, boff, psize, bufsize,* and *ssize*** are assigned values only if the specified unit exists and is open; otherwise, they are undefined.

When an error condition occurs during execution of an INQUIRE statement, all of the optional clause variables and array elements except *ios* are undefined.

Variables *ex* and *od* always become defined unless an error condition occurs.

When the receiving area for a returned literal value is too small, a warning is issued and the literal is truncated on the right. Character literal values returned by INQUIRE are uppercase.

Example:

```
 CHARACTER*20   FN1,ACC1,SEQ1,DIR1, FORM1,FMT1,UNF1,RFORM1,
1BLANK1,TYPE1
 INTEGER   OUTKOM,UNIT1,RECL1,ASSOC,BLOCK1,BOFF1,SEG1
 LOGICAL   EXIST1,OPEN1,NAMED1
 INQUIRE  (10,IOSTAT=OUTKOM,ERR=50,EXIST=EXIST1,OPENED=OPEN1,NUMBER=UNIT1,
1NAMED=NAMED1,NAME=FN1,ACCESS=ACC1,SEQUENTIAL=SEQ1,DIRECT=DIR1,FORM=FORM1,
2FORMATTED=FMT1,UNFORMATTED=UNF1,RECL=RECL1,NEXTREC=ASSOC,BLANK=BLANK1,
3RFORM=RFORM1,MRECL=MRECL1,TYPE=TYPE1,BLOCK=BLOCK1,OFF=BOFF1,SEG=SEG1)
```

If an error condition is encountered while the INQUIRE statement is being executed, the variable OUTKOM receives the contents of the I/O status word, PTIOE, contained in the storage control table, and control transfers to the statement with label 50.

Assuming this INQUIRE statement is executed prior to the OPEN statement in the first example in 5.10.1 and that unit 10 is not already open, the following values are returned:

```
OUTKOM = 0
EXIST1 = .TRUE.
OPEN1 = .FALSE.
```

Since that unit is not open, all other variables are undefined.

Assuming the INQUIRE statement is executed immediately after the OPEN statement referred to previously, the following values are returned:

```
OUTKOM = 0
EXIST1 = .TRUE.
OPEN1 = .TRUE.
UNIT1 = 10
NAMED1 = .TRUE.
FN1 = 'TESTING*10.'
ACC1 = 'SEQUENTIAL'
SEQ1 = 'YES'
DIR1 = 'NO'
FORM1 = 'FORMATTED'
FMT1 = 'YES'
UNF1 = 'NO'
RECL1 = undefined       not applicable for SDF sequential
ASSOC = undefined       not applicable for SDF sequential
BLANK1 = 'NULL'
RFORM1 = undefined      not applicable for SDF sequential
MRECL1 = 33
TYPE1 = 'SDF'
BLOCK1 = 224
BOFF1 = undefined       not applicable for SDF sequential
SEG1 = 111
```

Section 6
Specification and Data Assignment Statements

6.1. Overview of Specification Statements

Specification statements provide information to the processor on how storage should be managed and how symbolic names should be interpreted and used. The specification statement description can specify the following information:

- Data type (which in turn specifies the storage required for the data and its internal representation)
- **Initial value**
- Array declaration
- Data that shares storage within a program unit
- Data that shares storage among several program units

The specification statements are: DIMENSION, IMPLICIT, explicit typing, EQUIVALENCE, COMMON, **BANK, VIRTUAL**, PARAMETER, EXTERNAL, INTRINSIC, and SAVE.

In an internal subprogram, names used in explicit typing, DIMENSION, COMMON, NAMELIST, PARAMETER, EXTERNAL, INTRINSIC, SAVE, statement function definition, and DEFINE statements are local to the internal subprogram, even if the names are also used in its external program unit. See 7.9 for a description of global and local names.

Specification statements are nonexecutable. They can appear at the beginning of the program unit prior to the first executable statement, **or they can appear following an ENTRY statement with no intervening executable statements. When specification statements follow an ENTRY statement, the variables described must not appear in a prior executable statement.**

How entities are assigned initial values is also discussed in this section. The DATA statement, a means of providing initial values for variables, arrays, array elements, and character substrings at compilation time, is discussed in 6.12.1.

Storage assignment is discussed in 6.13.

6.2. DIMENSION Statement

Purpose:

The DIMENSION statement declares arrays and specifies dimension information.

Form:

```
DIMENSION a(d[,d]...) [/x/] [ ,a(d[,d]...) [/x/] ]
```

where:

a

is a symbolic name.

d

is a dimension declarator and specifies the extent of *a*; *d* must satisfy the requirements for dimension declarators (see 2.4.4.1).

x

is initialization information for elements of *a*. It must satisfy the requirements of a constant list as specified in 6.12.1.

Description:

A symbolic name can be declared as an array name in a DIMENSION, COMMON, or an explicit type statement. However, only one array declaration for a symbolic name is permitted in a program unit.

The maximum number of elements a nonvirtual array can have is 262,143. Appendix M contains a description of virtual arrays. An array element whose relative address under its location counter in the relocatable element is over 65,536 can’t be initialized using a **constant list** or DATA statement.

When an array *a* has adjustable or assumed-size dimensions, it can be declared only in a subprogram and must be a dummy argument.

For a description of local-global rules for names in DIMENSION statements in internal subprograms, see 7.9.

Examples:

```
      DIMENSION ARR1(5,5,2)
C           Symbolic name ARR1 is declared as a three- dimensional
C           array name.  The maximum values that the three
C           subscripts can assume are 5, 5, and 2
C           respectively.  The lower bounds for the three
C           subscripts are assumed to be 1.

      DIMENSION FACTOR (0:4) /1.,2.,3.,4.,5./
C           Symbolic name FACTOR is declared as a one-dimensional
C           array with five elements.  A lower dimension bound
```

```
C           of zero and an upper dimension bound of four are
C           specified.  The five elements are assigned initial
C           values 1 through 5 respectively.

      SUBROUTINE ADJUST(ONE,THIRD,YES,ARR4)
      INTEGER ONE,THIRD
      DIMENSION ARR4(ONE,2,THIRD)
C           Symbolic name ARR4 is an adjustable dummy array.
C           The addresses of its contents will be calculated
C           using the variable values.  No storage is actually
C           assigned to the array since it is a dummy array.

      SUBROUTINE DETER(ARR5,ARR6,IDIM)
      DIMENSION ARR5(-IDIM:2+IDIM, -IDIM:*), ARR6(*)
C           ARR5 and ARR6 are assumed-size dummy arrays.
C           ARR5 is a two-dimensional array and ARR6 is
C           a one-dimensional array.  The value of a
C           subscript expression in a dummy element reference
C           must be in the range determined by the lower bound and
C           upper bound expressions specified for that dimension.
```

6.3. Type Statements

The type statements are used to specifically declare the characteristics of items. Otherwise, the normal convention specifies REAL type for any data item whose symbolic name begins with any of the letters A through H or O through Z, and INTEGER type for any data item whose symbolic name begins with any of the letters I through N.

You can use any data type from Table 6-1 in a type statement.

Table 6-1. Valid Data Type and Length

Type	**Permitted Length Specification**	
	Default	**Alternate**
INTEGER	**4**	**None**
REAL	**4**	**8**
COMPLEX	**8**	**16**
LOGICAL	**4**	**None**
CHARACTER	1	2,3,4, ...,511 or *
DOUBLE PRECISION	**8**	**None**

Type statements are either implicit or explicit.

Typical type and length specifications that you can use in type statements are INTEGER, LOGICAL, REAL (**or REAL*4**) for type single-precision real, DOUBLE PRECISION (**or REAL*8**) for type double-precision real, COMPLEX (**or COMPLEX*8**) for type single-precision complex, **COMPLEX*16 for type double-precision complex**, and CHARACTER**n* (where *n* is a positive unsigned integer constant) for type character.

6.3.1. IMPLICIT Statement

Purpose:

The IMPLICIT statement assigns a specific data type to a symbolic name based on the initial alphabetic character of its name.

Form:

```
IMPLICIT t[*i] (a [ ,a] ... ) [ ,t[*i] (a[ , a ] ... ) ] ...
```

where:

t

is a data type chosen from Table 6-1.

i

is an unsigned integer constant (or positive integer constant expression enclosed in parentheses if the type is character) and is a permitted length specification for the given type. When the length is omitted, the standard length for the given type is assumed (see Table 6-1). **ASCII FORTRAN allows **i* for all data types.** The FORTRAN 77 standard allows **i* for type character only.

a

is a single alphabetic letter or a range of alphabetic letters. A range is specified by the first and last letters of the range separated by a hyphen. For example, the notation B-F specifies letters B, C, D, E, and F and has the same effect as when all letters are listed separately.

Description:

The IMPLICIT statement affects the letter association used to assign data types by the name rule (see 2.4.2.1).

All letters, *a*, included in a parenthesized list are associated with the data type *t* and length *i* (if present) preceding that list. In the program unit, all variables, arrays, symbolic names of constants, function subprograms, and statement functions whose symbolic names begin with the letters *a* are given the associated data type if the names are not explicitly typed. IMPLICIT statements don't change the type of any intrinsic functions.

In a subprogram, IMPLICIT statements must appear after the SUBROUTINE, FUNCTION, or BLOCK DATA statement.

A program unit can contain any number of IMPLICIT statements, but IMPLICIT statements should precede all other specification statements, except PARAMETER statements. **ASCII FORTRAN lets you specify IMPLICIT statements out of the correct order. When an IMPLICIT statement occurs after other specification statements or after an executable statement, a warning is issued and only variables that have not yet appeared in any statement are affected by the typing implied by the IMPLICIT statement.**

A letter can neither appear in, nor be included in, a range specification of more than one IMPLICIT statement. Furthermore, a letter can appear only once in any IMPLICIT statement.

An internal subprogram gets the same IMPLICIT type associations as its external program unit. However, it can also have its own IMPLICIT statement that only affects its local names. Any following internal subprograms have their IMPLICIT types revert to that of their external program unit (see 7.3.2 and 7.3.3).

Examples:

```
      IMPLICIT LOGICAL (B)
C           Variables whose names begin with the letter B are given
C           the LOGICAL type unless explicitly typed otherwise.

      IMPLICIT CHARACTER*4(C-E,Z), COMPLEX*16(M-O)
C           Variables whose names begin with the letters C, D, E,
C           or Z are given the CHARACTER type and a length of 4.
C           Data whose names begin with letters M, N, or O are
C           given the COMPLEX type and their real and imaginary
C           parts are double precision.
```

6.3.2. Explicit Type Statements

Purpose:

Explicit type statements assign a particular data type to a data item based on its name rather than its initial letter. In addition to data type, the following can be specified: dimension information, **data length, and initial values.**

When an initial value list is present, it applies to the immediately preceding array or variable only. The syntax for initial values is different than in the DATA statement. Here, each data item has its own initial value list; in the DATA statement the values for many variables can be in one list.

A function name, symbolic name of a constant, statement function name, or dummy argument name can appear in an explicit type statement but can't be assigned an initial value.

When no data length is specified, the standard data length for that type is assumed. See Table 6-1.

When any intrinsic function name appears in an explicit type statement and the statement type is not the same as the predefined intrinsic type, the name becomes a user-defined name. When the types are the same, the definition doesn't change.

The FORTRAN 77 standard states that the name of an intrinsic function must appear in an EXTERNAL statement to define a user-defined function name. **ASCII FORTRAN doesn't require the occurrence of the intrinsic function name in an EXTERNAL statement to define a user-defined function name.**

6.3.2.1. INTEGER, REAL, DOUBLE PRECISION, COMPLEX, and LOGICAL Type Statements

Form:

```
typ [*len] name [/x/] [,name [/x/] ]...
```

where:

typ

is INTEGER, REAL, DOUBLE PRECISION, COMPLEX, or LOGICAL.

len

is an unsigned integer constant and is one of the permitted specifications for the given type *typ* (see Table 6-1). The length specified with type *typ* applies to all symbolic names in the statement except those names that are followed by a length specification.

name

is one of the forms:

```
v [*len]
ary [*len][(d)]
```

where:

v

is a variable name, symbolic name of a constant, function name, statement function name, or dummy argument.

ary

is an array name.

d

is dimension information that must satisfy the requirements of 2.4.4.

x

is initialization information that must satisfy the requirements of a constant list as specified in 6.12.1.

Examples:

```
      INTEGER     SUM, ITEM(10)
C           SUM is specified as type integer and length 4.
```

```
C            ITEM is a one-dimensional array with 10 elements.
C            Each element of the array is type integer with length 4.

      COMPLEX     C1, C2*16, C3
C            C1, C2 and C3 are specified as type complex.
C            C1 and C3 have a length of 8, C2 has length 16.

      DOUBLE PRECISION     ARR(2) / 2*1.0D+0 /
C            ARR is specified as a type real, one-dimensional
C            array with two elements.  Each
C            element is type real with length 8 or double
C            precision.  Both elements of the array are
C            initialized with the double precision constant 1.0D+0.
```

6.3.2.2. CHARACTER Type Statement

Form:

```
CHARACTER [*len[,]] name [/x/] [,name [/x/]]...
```

where:

len

is an unsigned integer constant or integer constant expression enclosed in parentheses with a positive value in the range 1 to 511. It can also be an asterisk in parentheses (*). The length specified with CHARACTER applies to all symbolic names that are not followed by a length specification.

name

is one of the forms:

```
v[*len]
ary[(d)][*len]
ary[*len][(d)]
```

where:

v

is a variable name, symbolic name of a constant, function name, statement function name, or dummy argument.

ary

is an array name.

d

is dimension information that must satisfy the requirements of 2.4.4.

x

is initialization information that must satisfy the requirements of a constant list as specified in 6.12.1.

Description:

An entity declared in a CHARACTER statement can have an asterisk as its length specification if the entity is a function subprogram, a dummy argument, a symbolic name of a constant, **or a statement function name.**

- When a dummy argument has an asterisk for *len*, the dummy argument assumes the length of the associated actual argument. When the associated actual argument is an array name, the length assumed by the dummy argument is the length of an array element in the associated actual array argument.
- When a function subprogram has an asterisk for *len*, the function name must appear in a FUNCTION or ENTRY statement in the same subprogram. When a reference to such a function is executed, the function assumes the length specified in the referencing program unit. The length specified for a character function in a program unit that refers to the function must be an integer constant expression and must agree with the length specified in the subprogram that specifies the function. There is always agreement of length when an asterisk for *len* is specified in the subprogram that specifies the function.
- When a symbolic name of a constant has an asterisk for *len*, the constant assumes the length of its corresponding constant expression in a PARAMETER statement.
- **When a statement function name has an asterisk for *len*, the expression generated by a statement function reference is left as is, with no length conversion (see 7.3.1.2).**

Examples:

```
      CHARACTER*4    CHARA, CHARB(10)*1, CHARC
C            Variables CHARA and CHARC are specified with
C            type character and length 4.  CHARB is a one-
C            dimensional array with 10 elements.  Each element
C            of the array has type character and a length of 1.

      CHARACTER    TITLE(2)*12 /'PROGRAMMER', 'ANALYST'/
C            TITLE is specified as a type character, one-
C            dimensional array with two elements.  Each
C            element of the array has length 12.
C            The first element is initialized with the character
C            constant 'PROGRAMMER' and the second is
C            initialized with the character constant 'ANALYST'.
```

6.4. EQUIVALENCE Statement

Purpose:

The EQUIVALENCE statement specifies sharing of storage by two or more data entities of a program unit.

Form:

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

where *n* is a variable name, an array element name, an array name, or a character substring name; *n* can't be a dummy argument name or function name. At least two names must appear in each parenthesized list.

Description:

All entities in a given parenthesized list share some or all of the same storage locations. The order of items in the list isn't important.

Although variables of different types can be equivalenced, neither mathematical equality nor type conversion of these entities is implied. Equivalencing entities of different classes (an array and a scalar, for example) doesn't cause the entities to assume the same class. The EQUIVALENCE statement lets you reduce the storage requirements of a program unit by causing two or more data entities to share the same storage locations. However, it is your responsibility to ensure that the logic of a program unit permits such storage sharing and that sharing doesn't destroy needed information.

A substring expression in an EQUIVALENCE list must be an integer constant expression.

Character entities can be equivalenced to one another. The length of the equivalenced entities is not required to be the same.

Character entities can be equivalenced to noncharacter type entities if the equivalence does not force the noncharacter entities to begin on nonword boundaries. See the example at the end of this section.

When an array name appears in an EQUIVALENCE statement, it can be followed by a parenthesized list of subscripts. Each subscript must be an integer constant expression and when evaluated can be positive, negative, or zero. Such a list specifies a particular array element.

Using an array name without a subscript list in an EQUIVALENCE list has the same meaning as using a subscript list that specifies the first array element. A subscript list can contain **either one subscript or** *d* subscripts, where *d* is the number of dimensions in the array declaration for that particular array. If the list has *d* subscripts, it refers to the array element in the usual way.

If the list has one subscript, it refers to the linear position in the array starting at one (that is, the first element of the array is referenced by a subscript value of 1). For example, if array ARR1 has the dimensions (3,3), the equivalence reference ARR1(8) refers to element ARR1(2,3).

A subscript list with one subscript creates an ambiguity for one-dimensional arrays with a lower-dimension bound other than 1. For this case, the subscript refers to the array element in the usual manner rather than the linear position. For example, if ARR2 has the dimensions (-3:3), the equivalence reference ARR2(1) refers to the fifth element and not the first.

Equivalencing elements in two different arrays can implicitly equivalence other elements of these arrays.

When an entity appears in more than one list, these lists are combined into a single list.

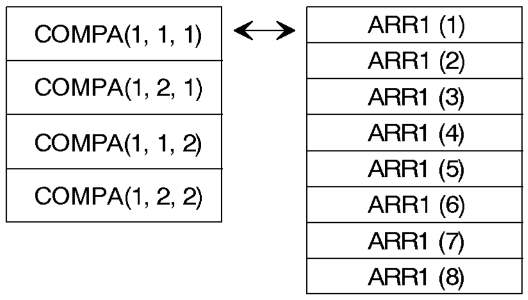
Sharing is accomplished on the basis of storage words. The number of words needed for internal representation of the various data types is listed in Table 6-2.

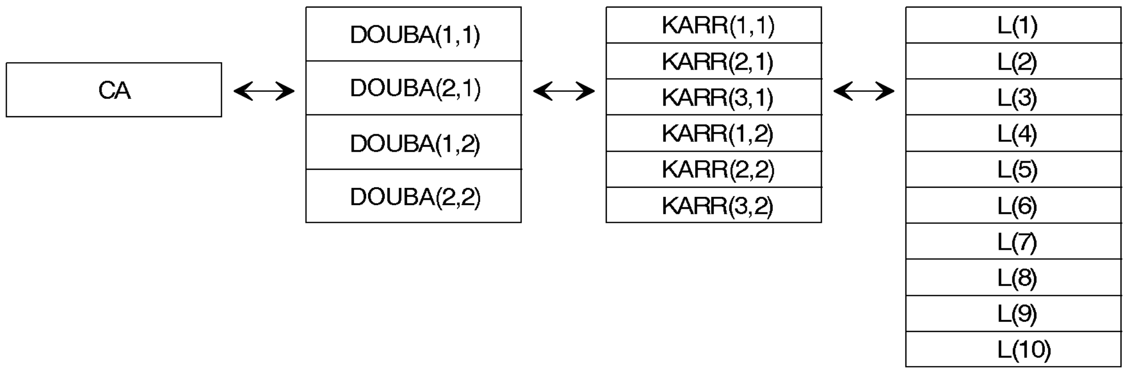
Table 6-2. Storage Units Required for Data Storage

Data Type	Words
INTEGER	1
REAL	1
DOUBLE PRECISION	2
COMPLEX	2
COMPLEX*16	**4**
LOGICAL	1
CHARACTER*n	(see 6.13.1)

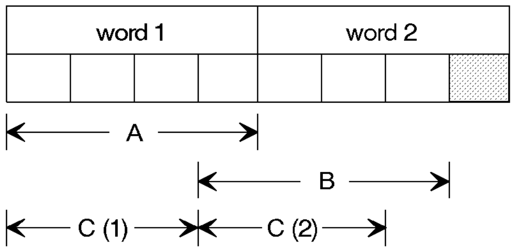
Examples:

```
      REAL AX(10,10), BX, T, Z
      INTEGER L
      EQUIVALENCE (AX(1,1),Z), (BX,L,T)
C            Array element AX(1,1) and variable Z share the
C            same storage sequence and entities BX, L and T
C            share the same storage sequence.

      REAL AA(10,10), BB, Z, CC(10)
      INTEGER M(20)
      EQUIVALENCE (AA(3,3),BB,CC(1)), (BB,Z,M(1))
C            Array element AA(3,3), variable BB, and array element
C            CC(1) share the same storage sequence.  BB, Z, and M(1)
C            share the same sequence.  Because entity BB appears in
C            both lists, the lists are combined so that AA(3,3), BB,
C            CC(1), Z, and M(1) all share the same storage sequence.

      DIMENSION ARR1(8), KARR(3,2)
      COMPLEX COMPA(1,2,2),CA
      DOUBLE PRECISION DOUBA(2,2)
      LOGICAL L(10)
      EQUIVALENCE(COMPA(1), ARR1(1)), (DOUBA(2), KARR(1,2), L(4),CA)
C            Based on storage word requirements, the EQUIVALENCE
C            statement causes the following locations to be shared:
```





```
      CHARACTER A*4, B*4, C(2)*3
      EQUIVALENCE (A,C(1)), (B,C(2))
C            Character variables whose names appear in equivalence
C            have the same first character storage unit.  The
C            EQUIVALENCE statement in this example causes the following
C            sequences to be shared:
```



```
C           The fourth character position of A, the first
C            character of B, and the first character position
C            of C(2) all occupy the same storage sequence.
C            The cross-hatched area is unallocated.

       INTEGER I, J
       CHARACTER*3 CX
```

```
            EQUIVALENCE (I, CX(3:3)), (J,CX)
      C           Once the equivalence of I is made to the third storage
      C           byte of CX, the equivalence of J to the first storage
      C           position of CX would require J to begin on a nonword
      C           boundary which is in error.
```

6.5. COMMON Statement

Purpose:

The COMMON statement provides a means of sharing storage among various program units.

Form:

```
COMMON [ / [c] / ] n [ ,n] ... [ [,] / [c] / n [,n] ... ] ...
```

where:

c

is a common block name. When the first *c* is omitted, the first two slashes are optional. The variable *c* must satisfy the definition of a symbolic name (see 2.4). A common block name is global to the executable program. **ASCII FORTRAN allows a common block name to be the same as an external subprogram name**.

A common block name in a program unit may also be the name of any local entity other than a constant, intrinsic function, or a local variable that is also an external function in a function subprogram. If a name is used for both a common block and a local entity, the appearance of that name in any context other than as a common block name in a COMMON or SAVE statement identifies only the local entity. Note that an intrinsic function name may be a common block name in a program unit that does not reference the intrinsic function.

n

is a common block member name. It is a variable name, an array name, or an array declaration.

Description:

The COMMON statement lets you name a storage sequence (a common block) to be shared by program units and to specify the names of variables and arrays that are to occupy this sequence. This permits:

- Different program units to refer to the same data without using arguments.
- Unrelated data from different program units to share the same storage.
- A single allocation of storage for variables and arrays, used by several program units.

The variables and arrays that appear in the list following a common block specification are declared to be in that common block.

When a common block name, *c*, is omitted, blank common is assumed. Otherwise, the common block is a labeled common block identified by the name *c*. There can be only one blank common block in an executable program; however, there can be many labeled common blocks.

A labeled or blank common specification can appear more than once in a COMMON statement or can appear in more than one COMMON statement within a program unit. The list of variables and arrays following such successive appearances of a common block name in a program unit is treated as a continuation of the list for that block name.

In a program unit, variables and arrays in a common block are allocated consecutive storage sequences in the same order as they appear in the list. Therefore, the order in which you enter them is significant when items in the common block are referenced in other program units.

Equivalencing of two quantities that appear in COMMON statements isn't possible since, by their appearance in a COMMON statement, storage is allocated for each.

Equivalencing an entity that was not named in any common block declaration in the program unit to an entity that was named in a common block declaration in the program unit draws the entity not named in any common block declaration into the common block.

An EQUIVALENCE list may cause storage to be appended to the end of a common block but may not cause storage to be allocated before the first member of the common block. See the examples below.

Character type entities can appear in the same common block as noncharacter type entities.

You can declare the dimensions of an array in a common block:

- in a type statement,
- in a DIMENSION statement, or
- by appending the dimension information to the array name in the COMMON statement.

However, the array must not be dimensioned more than once.

For a description of storage allocation on common variables, see 6.13.1.

A COMMON statement in an internal subprogram redefines (that is, is not a continuation of) the same common block declared in the external program unit. For a description of local-global rules for names used in COMMON statements in internal subprograms, see 7.9.

Execution of a RETURN or END statement does not cause entities in **either** blank **or labeled** common blocks to become undefined.

Blank common blocks **and labeled common blocks** need not be the same size in all program units of an executable program.

Entities in labeled common blocks or **blank common blocks** may be initially defined by a DATA statement **or data initialization associated with a type specification statement** in a BLOCK DATA subprogram **or in any other program unit.**

Examples:

```
      COMMON A,B,C(5,5) /COM1/D,M,V,S
C            Variables A, B and array C are placed in the blank
C            common block. Variables D, M, V, and S are placed in
C            the common block named COM1.

       COMMON L/C1/X,Y,Z //M,N
C            Variables L, M, and N are placed in the blank common
C            block.  Variables X, Y, and Z are placed in the
C            common block named C1.

       COMMON I
       EQUIVALENCE (I,J)
C            Variable J was not named in any common block declaration,
C            but since it is equivalenced to variable I (which is a
C            member of blank common), it is also a member of blank
C            common.

       DIMENSION B(3), E(3)
       COMMON I,J
       EQUIVALENCE ( I, B(2))
       EQUIVALENCE ( J, E(1))
C            The first member (first word) of the common block is a
C            variable I.  The first EQUIVALENCE statement is
C            illegal because it attempts to allocate B(1) in
C            front of I.  The second EQUIVALENCE statement is
C            legal because it appends storage to the end of the
C            common block.
```

6.6. BANK Statement

Purpose:

The BANK statement informs the compiler of the name of the bank containing data, functions, or subroutines referenced in the program unit. Use it to construct a multibank program.

Form:

BANK /*b*/ [&]*n* [,[&]*n*]...[/*b*/[&]*n*[,[&]*n*]...]...

where:

each *b*

is a bank name. It must satisfy the definition of a symbolic name, and is the name specified on a collector IBANK or DBANK directive.

each *n*

is the name of a subprogram or the name of a labeled common block that is included in the associated bank, *b*. A subprogram name must follow an ampersand (&). A labeled common block name must not follow an ampersand.

Description:

The compiler uses this information to generate linkage between multibank subprograms and data. The compiler doesn't actually construct the banks; it merely generates linkage between banks.

The COMPILER (BANKED=ALL) statement is much easier to use and more flexible than the BANK statement (see 8.5). However, using the BANK statement results in more efficient code for programs using multiple D-banks and can be used to make your program more efficient once it has been debugged.

A labeled common block name and a subprogram name cannot occur in the same bank. All labeled common blocks are assumed by the compiler to be in data banks and, therefore, are activated by LDJ or LBJ instructions. Subprograms are assumed to be in instruction banks and are accessed by LIJ instructions.

A subprogram name need not appear in a BANK statement when you use the COMPILER (LINK = IBJ$) statement (see 8.5). (The COMPILER (LINK=IBJ$) statement supersedes the use of the BANK statement on subprogram names.) However, they can be used together, and the BANK name supplied is used to link to the subprogram.

A bank name can appear more than once in a BANK statement or in more than one BANK statement. In either case, the effect is as though all the associated subprogram names or common block names are combined in one list.

The following cannot appear in a BANK statement:

- A FORTRAN V subprogram name (see 6.9).
- An internal subprogram name.
- A dummy subprogram name. (Use COMPILER statement option BANKED=DUMARG, LINK=IBJ$, or BANKED=ALL to inform the compiler of a banked dummy subprogram.)

A subprogram name used by an external program unit should not appear in a BANK statement in one of its internal program units.

An intrinsic function name should not appear in a BANK statement unless it was in an EXTERNAL statement first.

If a common block is banked, it must be in BANK statements in all of the program units the COMMON statements are in, or they must all have the BANKED=ALL COMPILER statement option.

In addition to the information present in the BANK statement, the compiler requires information concerning:

- the type of return from a subprogram (if a SUBROUTINE or FUNCTION is being compiled),
- the presence of bank information in dummy arguments to subprograms, and
- the presence of bank information in calling sequences of subprograms.

This information is specified by means of a COMPILER statement (see 8.5).

A comprehensive description of multibanking features appears in the *Series 1100 Executive System, EXEC, Reference,* UP-4144, and the *Guide to Installation and Use of Series 1100 Common Banks,* UP-10063.

Examples:

```
      BANK /BK/A,B,C
C            This statement indicates that common blocks A, B, and
C            C are in the data bank named BK.

      BANK /BANKA/&SUB1,&FUNC1/MRL/CBA,X, ADR
C            This statement indicates that subprograms SUB1 and FUNC1 are
C            in the instruction bank named BANKA.  Common blocks
C            CBA, X and ADR are in the data bank named MRL.
```

6.7. VIRTUAL Statement

Purpose:

The VIRTUAL statement informs the ASCII FORTRAN compiler that virtual objects are to be processed, and indicates which objects are to be placed in virtual space.

Form:

```
VIRTUAL [ (options[,options ])][object[,object] ... ]
```

where:

options

is one of three options: UNPACKED, ALL, or ALLBUT *cblist*.

The option UNPACKED indicates that each common block specified in this VIRTUAL statement that was not previously referenced by another program unit is allocated at the beginning of a new virtual page. (The default is to allocate a virtual object immediately after the previously allocated virtual object.) Use this option to separate seldom-used virtual objects from heavily used ones.

Note: The program unit must reference the common blocks for this option to have any effect.

The option ALL means that all named common blocks referenced by the program unit are in virtual space.

The option ALLBUT *cblist* indicates that all named common blocks referenced by the program unit (except those specified in *cblist* or in BANK statements) are in virtual space. *cblist* has the format:

```
/ cb / [ , / cb / ] ...
```

where *cb* is a common block name. The list of common blocks is local to the program unit.

object

is a common block name when it appears in slashes (for example, */name/*) or is a local variable name when it appears without the slashes (for example, *name*).

Description:

The appearance of the VIRTUAL statement automatically designates that arguments passed to a subprogram can be either in virtual or nonvirtual space. A called subprogram processes both virtual and nonvirtual arguments correctly.

The VIRTUAL statement can also specify named common blocks and local variables that are to be placed in virtual space.

The VIRTUAL statement puts the objects you select in virtual space. Objects that are in virtual space are exempt from the 262,143-word address space limitation of the OS 1100 architecture.

When *object* is a common block name in slashes, all program units declaring that common block must place it in virtual space or fatal run-time diagnostics can result. See Appendix M for a full description of how and when to use virtual space.

6.7.1. Placement of the VIRTUAL Statement

Place the VIRTUAL statement at the beginning of a FORTRAN program unit. IMPLICIT and COMPILER statements can precede or follow the VIRTUAL statement. When used in a subprogram, the VIRTUAL statement should be placed directly after the FUNCTION, SUBROUTINE, or BLOCK DATA statement. When the VIRTUAL statement is used in an internal subprogram, only local variable names can be specified, not common block names. For the first program unit of an element, the VIRTUAL statement can appear before the SUBROUTINE, FUNCTION, BLOCK DATA, or PROGRAM statement.

The VIRTUAL statement must appear in the first program unit when it is to appear in a FORTRAN element. It isn't necessary to place a VIRTUAL statement in the other program units in the element when they do not reference virtual common blocks or you don't want local arrays in virtual space. When the ALL option in a VIRTUAL statement is used in the first program unit, it is assumed by the other program units in the element. However, an explicit common-block-name list in a VIRTUAL statement is local to the program unit in which the VIRTUAL statement resides. Internal programs inherit the VIRTUAL statement of their parent external unit.

When the option ALLBUT *cblist* appears in a VIRTUAL statement in the first program unit, the ALL option is assumed by all other program units in the element (unless other ALLBUT options appear in those program units). *cblist* is always local to the program unit where it appears.

When neither the ALL or ALLBUT option appears in the first program unit, neither can appear in any other program unit in the element.

6.7.2. Local Variables in Virtual Space

Local variables (arrays or scalars) placed in virtual space default to being in the automatic storage class. This means that virtual storage is allocated for them on subprogram entry and released on subprogram exit. When you want local virtual space in the static storage class, place a simple SAVE statement in the specification statement area of the subprogram. A SAVE statement with no variable list means that all local variables are in the static storage class. A SAVE statement with a variable list puts only those variables in the static storage class.

6.7.3. Common Block Name in a BANK Statement

A common block name in a BANK statement overrides the ALL option of the VIRTUAL statement, and it is assumed that the common block is in banked space. For more information on the BANK statement, see Appendix H.

Blank common can't appear in banked space or in virtual space. It is always assumed to be in the control bank.

6.7.4. Arguments Passed to a Subprogram

When actual arguments passed to a subprogram are in virtual space, a VIRTUAL statement must be in the element containing the subprogram.

6.7.5. Use of Virtual and Banking Together

You can use virtual space in a program that also uses the already existing banking setup. Don't place a common block in both virtual and banked space. Construct a multibanked program when the I-bank code becomes too large. Constructing a multiple I-banked program is simpler than constructing a multiple D-banked program.

6.7.6. Examples of Using the VIRTUAL Statement

The following examples show the use of the VIRTUAL statement in a subroutine, main program, and function.

Example 1:

```
      SUBROUTINE SUB(ARG,N)
         VIRTUAL A,/C1/,C
         COMMON /C1/AR1(99000,10),AR2(10000)
         COMMON /C2/AR3(10,100),X,B(10)
         REAL A(99000),ARG(N),C(10000)
C             The local arrays A and C are in dynamic local
C             virtual space.  They are allocated on entry to subroutine
C             SUB and freed on exit.  Common block C1 is in virtual
C             space and common block C2 is not.  Processing of
C             arguments passed to subroutines SUB is correct no matter
C             where they reside.
                .
                .
                .
         END
```

Example 2:

```
      VIRTUAL(ALL)
         COMMON /C1/CX(10000)
         COMMON /C2/C2X(20000)
                 .
                 .
                 .
         END
         SUBROUTINE S(A2)
         REAL A2(*)
         COMMON /C1/AR1(99000,10)AR2(10000)
         COMMON /AXR/BIGX(2000,2000)
C              All named common blocks used in this program are in
C              virtual space.  Processing of arguments passed to
C              subroutine S is correct no matter where they reside.
                 .
                 .
                 .
         END
```

Example 3:

```
      REAL FUNCTION FFT(A,B,N)
         VIRTUAL ALT1,ALT2
         REAL A(N,N),B(N,N)
         REAL ALT1(400,400),ALT2(400,400)
         SAVE ALT1,ALT2
C             The local arrays ALT1 and ALT2 are in static local virtual
C             space.  They are allocated on first entry to the function
C             FFT and never released.  Processing of arguments
C             passed to function FFT is correct no matter where they
C             reside.
                   .
                   .
                   .
         END
```

6.8. PARAMETER Statement

Purpose:

The PARAMETER statement lets you refer to constants by symbolic names. This facilitates the updating of programs in which the only changes between compilations are in the values of certain constants. You can revise the PARAMETER statement instead of changing the constants throughout the program.

Form:

```
PARAMETER (n = e [,n = e] ...)
```

or

```
PARAMETER n = e [ ,n = e] ...
```

where:

n

is any symbolic name (identifier). It is called a symbolic name of a constant, or parameter constant.

e

is a constant expression.

Description:

The PARAMETER statement is nonexecutable. **In ASCII FORTRAN, the PARAMETER statement may appear anywhere in a program.** (The FORTRAN 77 standard requires a PARAMETER statement to appear before any DATA statements, statement function statements, and executable statements.) However, a PARAMETER statement defining a symbolic name of a constant must physically appear before that constant is referenced.

When the symbolic name *n* is of type integer, single- or double- precision real, single-precision complex, or **double-precision complex,** the corresponding expression *e* must be an arithmetic constant expression. When the symbolic name *n* is of type character or logical, the corresponding expression *e* must be a character constant expression or a logical constant expression, respectively.

When a symbolic name of a constant is not of default-implied type, its type must be specified by a type statement or IMPLICIT statement prior to its first appearance in a PARAMETER statement. When the length specified for *n* is not the standard length as specified in Table 6-1, its length must be specified in a type statement or IMPLICIT statement prior to the first appearance of the symbolic name of the constant. Its length must not be changed by subsequent statements, including IMPLICIT statements.

The expression *e* can be any expression that yields a constant result at compile time. If the expression *e* is not the same type as *n*, *e* is evaluated and then converted to the type of *n*. The operands used in *e* can be any constants, symbolic names of constants previously defined by other PARAMETER statements, **or any FORTRAN-supplied inline or library functions.** A symbolic name of a constant must not be redefined in the same program unit. The definition of symbolic names of constants occurs from left to right in a PARAMETER statement.

(The FORTRAN 77 standard restricts *e* to a constant expression. Intrinsic function calls aren't allowed. Exponentiation is allowed only when the exponent is type integer.)

The compiler can use the common-banked Common Mathematical Library (CML) to do compile-time arithmetic. Therefore, a PARAMETER statement requiring a CML call is not allowed at a site if the compiler is generated with the clause USING NO MATH BANKS on the compiler System Generation Statement (SGS) supplied to the ASCII FORTRAN compiler build.

PARAMETER statements requiring compile-time calls to the common mathematical routines are:

```
PARAMETER (IB=2**18)
PARAMETER (A=SIN(1.8))
```

On reference to a symbolic name of a constant in a FORTRAN source program, the constant value of that symbolic name of a constant is used by the compiler. No storage is assigned to symbolic names of constants whose values can be represented in 18 bits or less.

For a description of local-global rules for names used in PARAMETER statements in internal subprograms, see 7.9.

Examples:

```
      INTEGER A,B
      PARAMETER (A=3)
      PARAMETER (B=4*A)
      C=A+I*B
C            A and B have values 3 and 12 respectively at compile
C            time.  The third statement is interpreted as though
C            the statement C=3+I*12 was written there.

      PARAMETER D=(SIN(5.)+4)
C            SIN(5.) is evaluated at compile time, and D is assigned
C            this value plus 4, yielding 3.0410757.
```

You can't use a symbolic name of a constant as a constant in a FORMAT statement, or as a length specification in a typing statement (except for the CHARACTER type statement, where a constant expression is allowed in parentheses).

6.9. EXTERNAL Statement

Purpose:

The EXTERNAL statement informs the compiler that your subprogram exists external to the program unit.

Form:

```
EXTERNAL a[,a]...
```

or:

```
EXTERNAL [opt] a [(l)] [, [opt] a [(l)]] ...
```

where:

a

is a symbolic name.

opt

is & or *; & has no meaning and is provided for syntactic compatibility with other FORTRAN processors. The * indicates that *a* is a FORTRAN V external subprogram, and is also retained for compatibility.

l

is FOR, ACOB, or PL1.

Description:

The EXTERNAL statement establishes that each *a* is a subprogram (that is, function or subroutine) name, and is not a FORTRAN-supplied function or a variable.

ASCII FORTRAN does not require a subprogram name (internal or external), which is used as an actual argument in a program unit, to appear in an EXTERNAL statement in that program unit if that name has previously been referenced with an actual argument list.

The FORTRAN 77 standard specifies that any function or subroutine name that you supply and that is passed as an argument must appear in an EXTERNAL statement.

If a name *a* in an EXTERNAL statement is the same as a FORTRAN-supplied intrinsic function name, the name *a* is treated as a programmer-supplied subprogram name. The name *a* loses all properties that are associated with the FORTRAN-supplied function: type, automatic typing, required type, and number of arguments. If *a* is referenced as a function, its type is determined from the first letter of the name, unless *a* appears in an explicit type statement, or unless the first character of *a* appears in an IMPLICIT statement.

The optional keyword *l* following each name *a* can be FOR, ACOB, or PL1. It indicates that *a* is a FORTRAN V, ASCII COBOL, or PL/I external subprogram, and lets ASCII FORTRAN generate the correct linkage for each language.

The EXTERNAL statement is a specification statement and must precede any executable statements. In addition, any reference to a name *a* in a statement function definition, other than as a dummy argument of the statement function, must follow the EXTERNAL statement.

For a description of local-global rules for names you use in EXTERNAL statements in internal subprograms, see 7.9.

Example:

```
       EXTERNAL COMP1, COMP2
           .
           .
           .
        CALL PROG(COMP1, VAR1)
           .
           .
           .
        CALL PROG(COMP2, VAR2)
           .
           .
           .
        END
C           The first call passes the name of subprogram COMP1
C           as an argument of subroutine PROG.  The second call
C           passes the name of subprogram COMP2.
```

6.10. INTRINSIC Statement

Purpose:

An INTRINSIC statement identifies a symbolic name as representing an intrinsic function (see 7.7.1). It also permits use of a name that represents a specific intrinsic function as an actual argument.

Form:

```
INTRINSIC f [,f] ...
```

where *f* is an intrinsic function name.

Description:

Appearance of a name in an INTRINSIC statement declares that name to be an intrinsic function name.

If a specific name of an intrinsic function is used as an actual argument in a program unit, it must appear in an INTRINSIC statement in that program unit. The following names of intrinsic functions must NOT be used as actual arguments:

- Type conversions (INT, IFIX, **HFIX**, IDINT, **IDFIX**, FLOAT, REAL, DBLE, **DFLOAT, DREAL**, CMPLX, **DCMPLX,** ICHAR, CHAR, **UPPERC, LOWERC)**
- Lexical relationships (LGE, LGT, LLE, LLT)
- **Typeless functions (AND, OR, XOR, BOOL, COMPL)**
- **Pseudo-functions (BITS, SUBSTR)**
- **Intrinsic function SBITS**
- Largest or smallest value (MAX, MAX0, AMAX1, DMAX1, AMAX0, MAX1, MIN, MIN0, AMIN1, DMIN1, AMIN0, MIN1)

Only one appearance of a symbolic name in all of the INTRINSIC statements of a program unit is permitted. A symbolic name must not appear in both an EXTERNAL and an INTRINSIC statement in a program unit.

Example:

```
      INTRINSIC SIN
      CALL SUB(SIN,COS)
C           The intrinsic function name, SIN, is passed as the first
C           argument to subroutine SUB.  The real scalar variable,
C           COS, is passed as the second argument.
```

6.11. SAVE Statement

Purpose:

A SAVE statement is used to retain the values of entities as they are defined after the execution of a RETURN or END statement in a subprogram. On reentry of the subprogram, an entity specified by a SAVE statement contains the value as last defined by the execution of the subprogram.

Form:

```
SAVE [ n [,n] ... ]
```

where *n* is a named common block preceded and followed by a slash, a variable name, or an array name.

Description:

A SAVE statement is nonexecutable. Within a function or subroutine, an entity specified in a SAVE statement does not become undefined as a result of the execution of a RETURN or END statement in the subprogram. However, an entity in a common block **or a global entity in an internal subprogram** can become redefined in another program unit.

Dummy argument names and names of entities in a common block must not appear in a SAVE statement.

A SAVE statement without a list is treated as though it contained the names of all allowable items in that program unit. A SAVE statement is optional in a main program and has no effect.

The appearance of a common block name preceded and followed by a slash in a SAVE statement has the effect of specifying all of the entities in that common block.

The SAVE statement operates by ensuring static storage for the named or implied entities. ASCII FORTRAN normally has static storage for common blocks. **However, if the COMPILER statement with DATA=AUTO is present, local variables are not saved on execution of a RETURN or END statement, since storage is dynamically acquired.** The SAVE statement causes local variables to be placed under location counter 8, which is static storage in a program **that contains the DATA=AUTO option.** If the collector segmentation facilities are used to collect a program, the SAVE statement does *not* ensure that these entities are preserved, since ASCII FORTRAN has no control over the collection. Overlays of storage by any means, including collector segmentation, can cause the loss of values of the entities named in SAVE statements in the overlaid programs.

6.12. Assigning Initial Values

Initial value s can be assigned to variables, arrays, or array elements in the main program or any subprogram. Initial values are loaded with the executable program. Only variables to which initial values are assigned are defined at the start of execution; all other variables are undefined until values are assigned to them. Initial values can't be specified for the dummy arguments of a subprogram.

Initial value assignment is accomplished through the use of DATA statements **and of initial value lists in DIMENSION and type statements.** For variables and arrays in common blocks, initial values can be assigned in a BLOCK DATA subprogram (see 7.6).

A data item whose relative address under its location counter in the relocatable element is over 65,535 can't be initialized using a DATA, **DIMENSION, or typing** statement initialization unless the array is in virtual space. All local variables go under location counter 0 or 8, and each common block gets its own location counter. Attempting to initialize an entity whose relative address is over 65,535 (0177777 in octal) results in an ERROR 10 (see Appendix D).

6.12.1. DATA Statement

Purpose:

The DATA statement initializes variables, arrays, array elements, and character substrings at compile time.

Form:

```
DATA variable-list/const-list/ [[,]variable-list/const-list/ ] ...
```

where:

variable-list

a list of variables, arrays, array elements, substring names, and implied-DO groups, separated by commas. The format of an implied-DO group is:

```
(do-list , index = start , stop [ ,inc] )
```

where:

do-list

is a list of array element names and implied-DO groups.

index

is a scalar integer variable, called the implied-DO-variable. *Start*, *stop*, and *inc* are integer constant expressions. When implied-DO groups are nested, the *start*, *stop*, and *inc* expressions of an inner group can include references to the *index* of an outer group.

An iteration count and the values of the implied-DO-variable are established from *start*, *stop*, and *inc* exactly as for a DO-loop, except that the iteration count must be positive (see 4.5.4.2). The list items in *do-list* are specified once for each iteration of the implied-DO list with the appropriate substitution of values for any occurrence of the implied-DO-variable index.

Subscript expressions in a *do-list* must be integer expressions involving only constants, symbolic names of constants, and implied-DO index variables. However, subscript expressions in a *variable-list* but not in a *do-list* are limited to integer expressions involving only constants and symbolic names of constants. Substring expressions must be integer constant expressions.

const-list

is a list of constants separated by commas. You can optionally precede each constant in the list with "*n**", where *n* is an unsigned positive integer constant or a positive integer-valued symbolic name of a constant. This notation denotes repetition of the immediately following constant *n* times.

The constants in *const-list* can be of any type, **including three constant types that are supported for initial values only:**

1. **statement labels (see 6.12.2),**

2. **octal (see 6.12.3), and**
3. **Fieldata (see 6.12.4).**

Description:

The elements in the *variable-list* are matched to the elements in the *const-list* according to position. An array name matches *n* consecutive *const-list* items, where *n* is the number of elements in the array. The elements of the array are initialized in column-major order. The FORTRAN 77 standard requires a one-to-one correspondence between the number of elements in *variable-list* and *const-list.* **ASCII FORTRAN allows the lists to be of unequal length. When *const-list* is short, the last elements of the variable list aren't initialized. When *const-list* is too long, the last constants are ignored.** See 6.12.5 for allowed type matching between *variable-list* and *const-list.*

DATA statements are nonexecutable.

The FORTRAN 77 standard requires DATA statements to follow all specification statements. **ASCII FORTRAN, on the other hand, allows DATA statements to appear anywhere in the program unit after any SUBROUTINE, FUNCTION, or IMPLICIT statement and before the last statement in the program unit.**

ASCII FORTRAN places a few restrictions on the placement of DATA statements:

1. **DATA statements must follow all specification statements that declare variables referred to in the DATA statement.**
2. **When an array has its bounds declared in one statement and its type in a following statement, a DATA statement can't appear between the two specification statements.**

Because of these restrictions, the ASCII FORTRAN compiler issues a warning message whenever a DATA statement is followed by a specification statement.

The FORTRAN 77 standard doesn't permit blank common entities to be initialized and further restricts the initialization of entities in labeled common blocks to appear only in a BLOCK DATA subprogram. **ASCII FORTRAN, on the other hand, permits data initialization of entities in labeled and blank common blocks to appear in any program unit.**

Note the following cautions when initializing data:

- A variable, array element, or substring must not be initially defined more than once in an executable program.
- Storage association from the use of EQUIVALENCE can also result in incorrect multiple initialization.

- Be careful when initializing character type entities in the same common block from more than one program unit. When the program units are compiled as separate relocatable elements and both initialize different character type items, the initialization of some items can be destroyed during the collection process if the initialized items share the same word of storage. Character items in common are packed in storage and it is possible to have more than one item occupying the same word of storage. For this reason, limit initialization of character type items in common blocks to one program unit in the FORTRAN program.
- In a program unit, be careful when initializing character type entities. Multiple initializations of the same item or parts of the same item can cause some of the initialization information to be lost. Substring initializations only initialize the character positions specified. Other character positions in the same word and part of the same variable or array element have a value of binary zero unless initialized by other substring initializations.

See 6.12.6 for DATA statement examples.

6.12.2. Specifying Statement Labels in an Initialization Statement

You write a statement label in an initialization statement as an asterisk, a currency symbol, or ampersand, followed by one to five decimal digits.

The value of the digit string, interpreted as an integer, must be the same as that of one of the statement labels in the program. The corresponding integer scalar variable is initialized to the same value that it would receive if it were assigned that statement label in an ASSIGN statement.

6.12.3. Specifying Octal Constants in an Initialization Statement

You write an octal constant, allowed only in an initialization statement, as the letter O followed by a string of octal digits. If the string consists of more than 12 digits, only the last 12 digits are used. It is possible to have a symbolic name of a constant whose name is the same as an octal constant. Such a name is interpreted as a symbolic name of a constant when used as an initial value.

An octal constant can be used to initialize a character variable or a one-word noncharacter variable (that is, logical, integer, or single-precision real). When used with a character variable, an octal constant is extended with zeros on the left to a multiple of three digits (9, 18, 27, or 36 bits), then padded with blanks or truncated on the right to match the length of the character variable. When used with a logical, integer, or real variable, an octal constant of less than 12 digits is extended to 12 digits with zeros on the left.

6.12.4. Specifying Fieldata Constants in an Initialization Statement

You write a Fieldata constant, allowed only in an initialization statement, as a character or Hollerith constant immediately followed by the letter F. The

characters of the constant convert to the 6-bit Fieldata code and are stored six to a word. Fieldata constants are padded with Fieldata blanks on the right, or are truncated on the right, to achieve the required length. You can't use Fieldata constants to initialize character variables.

6.12.5. Type Matching Between Variable- and Constant-List Items

Table 6-3 indicates the requirements for type matching between a *variable-list* item and the matching *constant-list* item. Except for character, **statement label, octal, and Fieldata** constants, the types must match. If the types don't match but can be converted, a warning is issued and the constant is converted to the type of the variable. When no conversion is possible, the initialization isn't done.

Table 6-3. Data Initialization Type Matching Requirement

Constant Type	Variable Type	Size	Error or Warning	Initialization Done	Conversion Done
INTEGER	INTEGER	same size		yes	no
INTEGER	REAL	same size	warning	yes	yes
INTEGER	COMPLEX	same size	warning	yes	yes
INTEGER	LOGICAL	same size	error	no	
INTEGER	CHARACTER	same size	error	no	
REAL	INTEGER	same size	warning	yes	yes
REAL	REAL	same size		yes	no
REAL	REAL	**diff size**	**warning**	**yes**	**yes**
REAL	COMPLEX	same size	warning	yes	yes
REAL	LOGICAL	same size	error	no	
REAL	CHARACTER	same size	error	no	
COMPLEX	INTEGER	same size	warning	yes	yes
COMPLEX	REAL	same size	warning	yes	yes
COMPLEX	COMPLEX	same size		yes	no
COMPLEX	COMPLEX	**diff size**	**warning**	**yes**	**yes**
COMPLEX	LOGICAL	same size	error	no	

continued

* **ASCII FORTRAN permits any variable type to be initialized with a character constant.** The FORTRAN 77 standard permits only character variables to be initialized by character constants.

Table 6-3. Data Initialization Type Matching Requirement (cont.)

Constant Type	Variable Type	Size	Error or Warning	Initialization Done	Conversion Done
COMPLEX	CHARACTER	same size	error	no	
LOGICAL	LOGICAL	same size		yes	no
LOGICAL	nonlogical	same size	error	no	
CHARACTER	**any type***	same size		yes	truncation or blank fill
statement number	INTEGER	same size		yes	no
statement number	noninteger	same size	error	no	
Fieldata	CHARACTER	same size	error	no	
Fieldata	noncharacter	same size		yes	truncation or blank fill
octal	INTEGER	same size		yes	no
octal	REAL*4	same size		yes	no
octal	REAL*8	same size	error	no	
octal	COMPLEX	same size	error	no	
octal	LOGICAL	same size		yes	no
octal	CHARACTER	same size		yes	truncation or blank fill

* **ASCII FORTRAN permits any variable type to be initialized with a character constant.** The FORTRAN 77 standard permits only character variables to be initialized by character constants.

6.12.6. DATA Statement Examples

```
      DIMENSION C(10)
      DATA A, B/1.2,0.5/, C/10*1.0/
C           The constants 1.2 and 0.5 are assigned to A and B,
C           respectively, at compile time.  Each element in array C
C           is assigned a value of 1.0.

      DIMENSION IARY(10), LARY(10)
      DATA (IARY(I),I=1,10) /1,2,3,7*4/, (LARY(I),I=1,9,2) /5*0.0/
C           The array IARY is initialized by an implied-DO.
C           The first three elements in the array have values
C           1, 2, and 3 respectively.  The remaining seven elements in
C           that array have 4 as their value.  The elements LARY(1),
C           LARY(3), LARY(5), LARY(7), and LARY(9) are initialized
C           to 0.0.  The other elements of LARY aren't initialized.
```

```
      PARAMETER (O2 = 5)
      DATA I/O2/
C           O2 is a symbolic name of a constant which has 5 as its
C           value.  O2 in the DATA statement is interpreted as the
C           symbolic name of a constant and the integer constant 5 is
C           assigned to the variable I, rather than the octal constant
C           O2 (see 2.3.7).

      DIMENSION A(10)
      DATA A/'AB','CDE','FGHIJ'/
C           The first three elements in array A have values
C           AB∆∆, CDE∆, and FGHI, respectively.

      REAL C2(10)/10*98.6/
C           This example shows initialization in a type statement.
C           All elements of array C2 are assigned the value 98.6.

      DATA J /'ABCDEF'F/, K /O060710111213/
C           The integer variables J and K receive
C           the same initial value, J as a Fieldata constant
C           and K as an octal constant.

      CHARACTER*1 S1
      DATA S1(1:1) /'*'/
      CHARACTER*2 S2
      DATA S2(1:1), S2(2:2) /' ','*'/
      CHARACTER*3 S3
      DATA S3(1:2), S3(3:3) /' ','*'/
      CHARACTER*4 S4
      DATA S4(1:3), S4(4:4) /' ','*'/
C          This example shows character variables of length n being
C          initialized to n-1 blanks followed by an asterisk.

      CHARACTER*1 S1 /' '/
      DATA S1(1:1) /'*'/
      CHARACTER*2 S2 /' '/
      DATA S2(2:2) /'*'/
      CHARACTER*3 S3 /' '/
      DATA S3(3:3) /'*'/
      CHARACTER*4 S4 /' '/
      DATA S4(4:4) /'*'/
C           This example appears to show character variables
C           of length n being initialized to n-1 blanks
C           followed by an asterisk.  However, since there
C           are multiple initializations involving parts of
C           the same variable (which isn't allowed by the
C           standard), some initialization information is
C           lost.  The results are unpredictable and
C           can result in some character positions
C           being initialized to null characters (ASCII
C           octal code 000).  On some output devices nulls
C           are dropped, giving the appearance of characters
C           having been placed in the wrong position.
```

6.13. Storage Assignment

The ASCII FORTRAN compiler automatically allocates storage for program data and compiler-generated instructions.

The ordinary operation of the compiler assumes that data and program addresses in the generated object program are less than 65,536. This implies that the size of data storage for any FORTRAN program should be less than 65,536 words. However, with the O option on the processor call statement, the compiler assumes that all data addresses in the program can extend to 262,143 words for nonvirtual items. This permits the creation of much larger FORTRAN programs (see 9.5).

You can use this facility without restrictions on the type (scalar or array, common, or noncommon) of data involved. Consider several points when using this facility:

- It is necessary only when the last address of the program is greater than 65,535 (0177777 octal) words after collection. If automatic storage is used and an ER MCORE$ is done by storage management creating addresses greater than 65,535, the O option isn't needed.
- When the total collected program size exceeds 65,535 words, compile all FORTRAN program units present in the program with the O option.
- Use this facility only when it's required, since more instructions may be necessary to access arguments and arrays when you use the O option.
- When truncation messages naming the FORTRAN run-time routines are emitted during collection of the user absolute, use IN statements in the collector symbolic to name your elements and common block names. This results in the run-time routines being collected at addresses under 65,536.

When you require large programs (including those using a total of more than 262,143 words), they can be structured into smaller components using the BANK statement (see 6.6), and a multibanked collector symbolic (see Appendix H).

6.13.1. Data Storage Assignment

For data storage, the ASCII FORTRAN system assigns storage to your noncharacter type variables in word increments. Table 6-4 details the amount of storage allocated by variable type in bytes and the alignment of the variable in that storage.

The first item in a common block list is allocated storage on a word boundary. Subsequent character items in common are allocated storage in a packed form on byte boundaries that may or may not begin on word boundaries. In packed form, no unallocated character storage locations or bytes exist between two character entities in common. The common block structure determines the alignment of its members.

Equivalencing character items in common to noncharacter items is permitted only when the item in common is on a word boundary. When a noncharacter item follows a character item in common, there can be one or more unallocated bytes between the two items, since noncharacter data items must begin on a word boundary.

Table 6-4. Storage Alignment and Requirement

Type	Length	Storage	Alignment
INTEGER	4	1 word	word boundary
REAL	single (4)	1 word	word boundary
	double (8)	2 words	word boundary
COMPLEX	single (8)	2 words	real part in first word; imaginary part in second word
	double (16)	**4 words**	**real part in first 2 words; imaginary part in second 2 words**
LOGICAL	4	1 word	word boundary (only least significant bit is used)

Character items not equivalenced and not in common are allocated storage on word boundaries except for the following cases:

- Character items of length 2 are allocated storage on word and half-word boundaries.
- Character items of length 1 are allocated storage on byte boundaries.

Character arrays, in common or not, are allocated storage such that there are no unallocated bytes between the elements of the array.

Storage order and alignment for items not in common are not determined by their appearance in the program. Storage is allocated by traversing a chain of variables whose order is determined by a hash function. Don't try to predict storage order or alignment.

The COMPILER statement option STD=66 causes all character items, including array elements, to begin on word boundaries.

Examples:

```
REAL E(5).
```

Array E occupies five words:

1	E (1)
2	E (2)
3	E (3)

4	E (4)
5	E (5)

```
DOUBLE PRECISION D.
```

Variable D occupies two words:

1	
2	

```
COMPLEX*16 Z
```

Variable Z occupies four words:

1	Z real
2	part
3	Z imaginary
4	part

```
CHARACTER*2 A, B*3, C
COMMON /C1/ A, B, C
```

Since A, B, and C are scalars in common, they are allocated storage in a packed form:

1	A	A	B	B
2	B	C	C	unallocated

```
CHARACTER*6 F(3)
```

Assuming array F begins on a word boundary, F is allocated five words of storage in a packed form. The area designated by an asterisk (*)can be allocated to character items of length 1 or 2 or the area can be allocated to items in common or equivalenced to F:

1	F (1)	F (1)	F (1)	F (1)
2	F (1)	F (1)	F (2)	F (2)

3	F (2)	F (2)	F (2)	F (2)
4	F (3)	F (3)	F (3)	F (3)
5	F (3)	F (3)	*	*

6.13.2. Location Counter Usage

The instructions and data generated by the ASCII FORTRAN compiler are allocated by function to several location counters. The location counters and their functions are described in Table 6-5. For a description of the automatic storage feature, see 8.5.1 **(COMPILER statement options DATA=AUTO and DATA=REUSE)**.

Table 6-5. Compiler Options for Location Counter Usage

Location Counter	No Automatic Storage or O Option	O Option without Automatic Storage	Automatic Storage
0	local (noncommon and nonvirtual) variables	local (noncommon and nonvirtual) variables, I/O packets and list instructions, FORMAT and NAMELIST text, parameter lists, constants	not used
1	all program instructions not resulting from I/O lists	instructions	instructions
2	blank common	blank common	blank common
3	INFO-010 diagnostic tables (see 10.4.1)	diagnostic tables	diagnostic tables
4	I/O packets, I/O list instructions, FORMAT statement text, NAMELIST statement text	not used	not used
5	not used	not used	automatic storage initialization literals
6	parameter lists	not used	not used
7	not used	not used	automatic storage initialization code
8	not used	not used	local (noncommon and nonvirtual) variables appearing in SAVE statement

continued

Table 6-5. Compiler Options for Location Counter Usage (cont.)

Location Counter	No Automatic Storage or O Option	O Option without Automatic Storage	Automatic Storage
9	not used	not used	not used
10	constants	not used	not used
12 and greater	labeled common	labeled common	labeled common

Section 7 Function and Subroutine Procedures

7.1. Procedures

The term procedure refers to a code mechanism that performs a particular computation. FORTRAN supports three general types of procedures:

1. A function procedure that is called by the appearance of the function name followed by its argument list in an expression. A function returns a value that replaces the name in the evaluation of the expression.
2. A subroutine procedure that is called by the appearance of its name in a CALL statement. A subroutine doesn’t return a value, although it can modify the values of variables.
3. A main program procedure that is called by action of the operating system, usually as a result of the @XQT Executive control statement.

Subroutines and functions are known collectively as subprograms.

A program consists of a main program and zero or more subprograms.

Subprograms can be external **or internal.** One or more program unit groups can be compiled under one processor call (@FTN). A program unit group is composed of the external program unit and any internal subprograms contained in the external program unit. For example:

```
@FTN,IS NAME
       PROGRAM MAIN
           .
           .
           .
       END
       SUBROUTINE IN
           .
           .
           .
       END
       FUNCTION SOLVE
           .
           .
           .
       END
@EOF
```

An internal subprogram is considered as nested within the previous external subprogram, and can access the external subprogram's data as well as having its own local data. An external subprogram can have many internal subprograms. See 7.3.2 and 7.3.3 for a more detailed description of internal and external subprograms.

The source input to the ASCII FORTRAN compiler consists of one or more program units (see 9.2.1). If a main program is present in the source input, then it must physically be the first program unit in the input.

The relocatable binary elements produced by the ASCII FORTRAN compiler are combined into an executable absolute element by the Collector (@MAP).

In addition to subroutines and functions, FORTRAN supports a third type of subprogram, the BLOCK DATA subprogram. Such a subprogram contains no executable code and is used solely for the assignment of initial values to variables in COMMON blocks. A BLOCK DATA subprogram can't be called. BLOCK DATA subprograms are described in 7.6.

7.1.1. Procedure References

The code represented by a procedure name is executed when the name of the procedure is encountered. This name is followed by an actual argument list. When a function has no arguments, it is referred to with a void argument list, that is, ( ). Subroutine references must follow the keyword CALL in a CALL statement. If the subprogram is used as an argument for another subprogram, it must appear previously with an explicit actual argument list or be identified in an EXTERNAL statement (see 6.9). When a subprogram is used as an argument, it is never followed by an argument list and the code associated with the procedure is not executed at this point. When an intrinsic function is used as an argument for another subprogram, it must be identified in an INTRINSIC statement (see 6.10).

7.2. Main Program

A main program may begin with a PROGRAM statement, does not contain a BLOCK DATA statement, and is terminated by an END, FUNCTION, or SUBROUTINE statement.

There must be exactly one main program in an executable program. If a main program is present in the source input, then it must physically be the first program unit in the input.

A main program may not be referenced from a subprogram or from itself.

7.2.1. PROGRAM Statement

Purpose:

A PROGRAM statement is optional. Use it to associate a name with the main program.

Form:

```
PROGRAM pgm
```

where *pgm* is the symbolic name of the main program in which the PROGRAM statement appears.

Description:

A PROGRAM statement is not required to appear in an executable program. However, if it does appear, it must be the first statement of the main program.

The symbolic name *pgm* is global to the executable program. It must be unique relative to external procedure names, BLOCK DATA subprogram names, and common block names in the same executable program. In addition, *pgm* must not duplicate any local name in the main program.

The name of a main program has no explicit use in the FORTRAN language. It is available mainly for documentation.

7.3. Programmer-Defined Procedures

Due to the wide variety of FORTRAN applications, you may want to use procedures that are not supplied by the FORTRAN language. Such procedures can be defined by using one of the following:

- Statement function
- Function subprogram
- Subroutine subprogram

7.3.1. Statement Functions

A statement function is a procedure specified by a single statement that is similar in form to an arithmetic, logical, or character assignment statement. All statement function definition statements must precede any executable statements in the program unit, **and they should appear after all other specification statements.**

A statement function definition specifies an expression to be evaluated when that statement function name appears as a function reference in another statement in the same program unit. The expression can involve any appropriate combination of arithmetic, character, and logical operators.

The statement function generates inline code when it is referenced. This allows efficient references to the defining expression without writing the expression each time.

A statement function definition statement is classified as a nonexecutable statement; it is not part of the normal execution sequence.

7.3.1.1. Statement Function Definition Statement

Purpose:

A statement function definition statement associates a statement function name with an expression.

Form:

```
n ( [a [ ,a ] ... ] ) = exp
```

or:

```
DEFINE n [ ( [a [ ,a ] ... ] ) ] = exp
```

where:

n

is the name of the statement function.

each *a*

is a dummy argument used in *exp*. (The maximum number of dummy arguments allowed is 150.)

exp

is any FORTRAN expression that references the statement function dummy arguments *a*.

Description:

The names of each *n* and *a* take the forms of variable names (see 2.4.3). An argument, *a*, is a dummy name and serves only to indicate order, number, and type of arguments for the statement function.

The variable names that appear as dummy arguments of a statement function are local to that statement. They may be used to identify other dummy arguments of the same type in different statement function statements, but may not be repeated in the same statement function dummy argument list.

The name may be used to identify any other entity in a program unit. The FORTRAN 77 standard restricts the use to any scalar variable within the program unit, but includes its appearance as a:

1. dummy argument in a FUNCTION, SUBROUTINE, or ENTRY statement, or
2. common block name.

A type is associated with the name of the statement function and each of its arguments. The type of each name is determined by the normal typing conventions (IMPLICIT and type statements, and the I-N integer rule). If the type of a statement function is character, its length can be specified as a positive integer constant or as * (see 6.3.2).

Each argument of the statement function must be referred to in *exp*; *exp* can also contain references to other statement functions. However, statement functions can't be referred to before they are defined, and a statement function definition can't refer to itself. In addition, a statement function definition in a function subprogram must not contain a function reference to the name of the function subprogram or an entry name in the function subprogram.

For a description of local-global rules for names used in statement function definition statements in internal subprograms, see 7.9.

Examples:

```
      BRIEF(VAR1,VAR2) = (VAR1 + VAR2)/4.
C             Defines a REAL statement function named BRIEF which
C             calculates the sum of its two arguments and divides
C             their sum by four.

      INTEGER FLD
      FLD(I,J,A) = BITS(A,I+1,J)
C             Defines an integer statement function named FLD whose
C             arguments I, J, and A are used in its expression.  The
C             expression is a reference to the BITS pseudo-function.

      STAFUN(231) = 2*3-A+482**B-200
C             Defines a REAL statement function with no dummy
C             arguments.

        CHARACTER A*4,B*2
        DEFINE B(I)=A(I)(3:4)
C            Defines a CHARACTER*2 statement function that takes
C            character positions 3 through 4 of character array
C            element A(I).
```

7.3.1.2. Statement Function References

The reference to a statement function takes the form:

n ([e_1 [,e_2] ...])

or:

n [([e_1 [,e_2] ...])]

where:

n

is the name of the statement function.

e_i

is an actual argument of the statement function. It must be an expression **or an array name.**

The order, number, and type of the actual arguments in the statement function reference must be the same as the order, number, and type of the dummy arguments in the statement function definition. **If the type of an actual argument does not match the type of the corresponding dummy argument, no conversion is performed; instead, the actual argument's type is used in the expression.** The type of an actual argument e_i must be consistent with the usage of the corresponding dummy argument a_i in the expression *exp* in the statement function definition statement. For example, if .NOT. a_i appears in *exp*, the corresponding actual argument e_i must be a logical expression.

In effect, each reference to a statement function is replaced by a copy of expression *exp* in which each occurrence of a dummy argument *a* is replaced by the corresponding actual argument *e*. The type of the resulting expression is determined from the types of the actual arguments and the nonargument components of *exp* according to the rules for expression evaluation (see 2.5). When a statement function is referred to, it must generate a legal FORTRAN expression.

If the expression *exp* and the statement function *n* are both arithmetic type (integer, real, or complex), *exp* is converted, if necessary, to the type of *n*, according to the assignment rules in Table 3-1. If both *exp* and *n* are type character, the length of *exp* is changed, if necessary, to the length of *n* (as described in Table 3-2). **There is no character length conversion if the statement function is declared as CHARACTER*(*) (see 6.3.2.2).**

When an actual argument is an expression, it is evaluated only once and its value is substituted for the dummy argument.

Following its declaration, a statement function can be referenced wherever a reference to a function is permitted.

A statement function reference can sometimes be used as a receiving variable (for example, as the left-hand part of an assignment statement). The latter usage requires that after argument substitution, *exp* must be:

- **a simple variable,**
- **an array element,**
- **an array name,**
- **a character substring, or**
- **the BITS or SUBSTR pseudo-function. (See 7.3.2.2 and 7.7.2.3 for further restrictions on BITS and SUBSTR.)**

When a statement function reference appears as a receiving variable, no type conversion is performed between the type of the expression *exp* and the type of statement function *n*.

Example 1:

```
      I(A,J,K,L) = A+J/K**L-A
C           Defines integer statement function I with dummy
C           arguments of type real (A) and integer (J,K,L).
      NEWVAL = I(14.3,-5,2,4)
C           The statement function reference I is evaluated
C           as 14.3+(-5)/2**4-14.3.  This expression has a
C           result which is type real.  This result is then
C           converted to type integer (since I is type
C           integer).  The result of that conversion is stored
C           in NEWVAL (which is type integer).
```

Example 2:

```
C           Suppose that a program requires three 1000 x 1000 logical
C           arrays.  Using the FORTRAN LOGICAL data type would
C           require three arrays of 1,000,000 words each.  Since
C           ASCII FORTRAN can't support a single array of more than
C           262,000 words without the virtual feature, the program
C           could not be compiled.  However, by accepting a
C           slight decrease in readability, the problem can be
C           solved using statement functions.
      INTEGER BIT, L1,L2,L3
      INTEGER LOG1(27778),LOG2(27778),LOG3(27778)
      DEFINE BIT(A,I) = BITS(A((I+35)/36),
     1                  MOD(I+35,36)+1,1)
      DEFINE L1(I,J) = BIT(LOG1,(I-1)*1000+J)
      DEFINE L2(I,J) = BIT(LOG2,(I-1)*1000+J)
      DEFINE L3(I,J) = BIT(LOG3,(I-1)*1000+J)
C           Now by arbitrarily associating 0 with the value FALSE
C           and 1 with the value TRUE, L can be used in most of the
C           same ways that a 1000 x 1000 logical array can be used.
C           Thus, the following loop assigns the logical matrix
C           product of L2*L3 to L1.
       DO 100 I = 1, 1000, 1
          DO 100 J = 1, 1000, 1
              L1(I,J) = 0
              DO 100 K = 1, 1000, 1
100              L1(I,J) = OR(L1(I,J),AND(L2(I,K),L3(K,J)))
C           While this example is correct, it isn't
C           necessarily the most efficient way of performing the
C           desired computation.
```

7.3.2. Function Subprograms

A function subprogram is a separate program unit. It begins with a FUNCTION statement and ends with the next END, **SUBROUTINE, or FUNCTION** statement.

A function subprogram can contain any FORTRAN statement other than a BLOCK DATA, SUBROUTINE, or PROGRAM statement.

There are external and **internal functions**. A function is external if it appears as the first program unit or follows an END statement. **Otherwise, it is an internal function and is considered as local to the previous external program unit, whether it is a main program, a function, or a subroutine. The FORTRAN 77 standard doesn't have the concept of an internal subprogram; it is an ASCII FORTRAN extension. When another FUNCTION or SUBROUTINE statement is**

encountered (with no intervening END statement), the following source is considered to be a separate internal subprogram.

A function subprogram can't call itself, either directly or indirectly.

For a description of local-global rules for using symbolic names in internal function subprograms, see 7.9.

Example:

```
C Main program
         .
         .   (block 1)
         .
C Internal function A
       FUNCTION A
         .
         .   (block 2)
         .
C Internal subroutine B
       SUBROUTINE B
         .
         .   (block 3)
         .
       END
C External function C
       FUNCTION C
         .
         .   (block 4)
         .
C Internal subroutine D
       SUBROUTINE D
         .
         .   (block 5)
         .
       END
C External subroutine E
       SUBROUTINE E
         .
         .   (block 6)
         .
       END
```

This compilation unit contains a main program **with two internal subprograms (A and B)**, an external function (C) **with one internal subprogram (D)**, and an external subroutine (E) with no internal subprograms. Each block in the program is a group of statements containing no END, SUBROUTINE, FUNCTION, PROGRAM, or BLOCK DATA statements. See 7.3.3 for a description of subroutines.

7.3.2.1. FUNCTION Statement

Purpose:

The FUNCTION statement informs the compiler that the definition of a programmer-supplied function is being specified.

Form:

```
[type] FUNCTION n ( [a [ ,a] ... ] )
```

or:

```
[type] FUNCTION n [*s] [ ( [a [ ,a] ... ] ) ]
```

where:

type

is INTEGER, REAL, DOUBLE PRECISION, COMPLEX, CHARACTER, or LOGICAL. This optional field specifies the type of value returned by the function. CHARACTER**s* is also allowed, in which case **s* is not allowed after *n*.

n

is the symbolic name by which the function is known to other program units. In general, it is not advisable to use **the currency symbol ($)** in function names. This avoids conflicts with entry names in the ASCII FORTRAN library and the OS 1100 Operating System relocatable library.

s

is one of the length specifications allowed for *type* (see 6.3). This field can be specified only when *type* is specified. If *type* is CHARACTER, *s* can be one of the following: (1) An unsigned, nonzero, integer constant, (2) an integer constant expression that doesn't include a parameter constant, enclosed in parentheses and with a positive value, or (3) an asterisk in parentheses.

each *a*

is a dummy argument, which can be a variable name, array name, dummy procedure name, ***, or $. If there are no dummy arguments, the parentheses are not required. You can enclose any dummy argument in slashes (that is, /*a*/)**. The number of arguments can't exceed 250. The maximum number of character arguments allowed is 63.

Description:

The FUNCTION statement must be the initial statement of a function subprogram **(except possibly for a COMPILER or EDIT statement)**. It identifies the name of a function, its arguments, and possibly its type and length.

The type of the function (which is the type of the value returned by the function) is determined by the *type* field of the function, an explicit type specification statement in the function subprogram, or using the implicit typing convention described in 2.4.2.1. The function must be assigned the same type in all program units that refer to it. If a reference to the function is made from a program unit in which it has a type other than that assigned to the function in the function subprogram, the returned value is not converted to the type expected by the calling routine. The results of such a call are unpredictable.

The name of the function and each of its ENTRY names (see 7.5) are available throughout the subprogram for use as a variable of the same type as that of the function (or ENTRY name). The function and entry names are effectively equivalenced (see 6.4). If a function is type CHARACTER, then all of its ENTRY names must also be type character.

The function name or an ENTRY name must be assigned a value by appearing on the left-hand side of an assignment statement, as an element of a READ statement list, or as an argument to a subroutine or function that assigns a value to the corresponding formal parameter. The value of the function and ENTRY name can be subsequently referenced or changed. When execution reaches a RETURN or END statement in the FUNCTION subprogram, the value returned is that most recently assigned to any of the associated function or ENTRY names. If the type of the name to which the value was assigned differs from the type of the entry point by which the function is called, no conversion is done. The returned value is then undefined.

The function subprogram can also use one or more of its dummy arguments to return values to the calling program unit. See 7.3.4 for a complete discussion of actual and dummy arguments.

For a description of alternate return specifiers (that is, dummy arguments as asterisks), see 7.4. ASCII FORTRAN allows alternate returns from functions, but the FORTRAN 77 standard does not.

7.3.2.2. Function References

A programmer-supplied function is referenced by the appearance of its name with an explicit actual argument list. The form of a programmer-supplied function reference is:

```
f ( [a[ ,a] ... ] )
```

where *f* is the name of the programmer-supplied function. (The name *f* must not appear both as the name of a function and as the name of a subroutine in the same program unit.) Each *a* is an actual argument and must match the corresponding dummy argument of *f* in type and usage (see 7.3.4). The arguments can be:

- Expressions
- Array names
- Intrinsic functions
- External procedures
- **Internal procedures**
- Dummy procedures
- **Statement labels**

The number of arguments can't exceed 250.

When a statement label appears as an actual argument, write it as *n, &n, or $n, where n is the statement label. The corresponding dummy argument must be an

asterisk (*) or currency symbol ($). Any alternate returns result in a loss of the value normally returned by a function reference.

A reference to a function causes the computation indicated by the function definition to be performed. The value returned by the function determines the value of the expression in which the reference occurs. The value returned is assumed to be of the type indicated by the first character of the name *f*, unless *f* appears in a type statement or unless the first character of *f* appears in an IMPLICIT statement.

When an external procedure name appears in an actual argument list without an explicit actual argument list of its own, the procedure name is passed to the called function. When an external procedure name appears without an explicit actual parameter list, it must previously appear with an explicit actual argument list or in an EXTERNAL statement (see 6.9).

7.3.3. Subroutines

A subroutine is a separate program unit. It begins with a SUBROUTINE statement and ends with the next END, **SUBROUTINE, or FUNCTION** statement. There are **internal** and external subroutines in exactly the same manner as **internal** and external functions (see 7.3.2). A subroutine is external if it appears as the first program unit in the source input, or if its SUBROUTINE statement is immediately preceded by an END statement. **Otherwise, it is internal.**

The subroutine can contain any FORTRAN statements except a BLOCK DATA, FUNCTION, or PROGRAM statement. **Another FUNCTION or SUBROUTINE statement encountered (with no intervening END statement) terminates the subroutine and causes the following source to be an internal program unit.**

A subroutine can't call itself, either directly or indirectly.

For a description of local-global rules for symbolic names used in internal subroutine subprograms, see 7.9.

7.3.3.1. SUBROUTINE Statement

Purpose:

The SUBROUTINE statement notifies the compiler that a subroutine is being defined.

Form:

```
SUBROUTINE n [ ([a [ ,a] ...]) ]
```

where:

n

is the symbolic name by which the subroutine is known to other program units. **It is not advisable to use the currency symbol ($) in subroutine names.** This avoids conflicts with entry names in the ASCII FORTRAN library and the OS 1100 Operating System relocatable library.

each *a*

is a dummy argument, which can be a variable name, array name, dummy procedure name, *, **or $**. A name *a* can appear only once in the list of arguments. The number of arguments can't exceed 250. The maximum number of character arguments allowed is 63. If there are no dummy arguments, you can omit the parentheses.

Description:

The SUBROUTINE statement must be the first statement in a subroutine subprogram **(except possibly for a COMPILER [see 8.5] or EDIT statement [see 8.4])**. It specifies the name and arguments of the subroutine identified.

A dummy argument of a subroutine can be a symbolic name, an asterisk, **or a currency symbol.** An asterisk **or a currency symbol** indicates that the actual argument corresponding to this dummy argument is a statement label. Such labels can be referenced by the RETURN *i* statement (see 7.4). **You can enclose dummy arguments that are symbolic names in slashes (/*a*/)**.

The subroutine name can't appear in any other statement of the subroutine subprogram.

Example:

```
      SUBROUTINE SAMP(CONS, ARR, *)
C           Defines subroutine SAMP with three arguments.  The
C           asterisk indicates the actual argument is a statement label.
```

7.3.3.2. Subroutine References

A subroutine is referenced by a CALL statement (see 7.3.3.3).

7.3.3.3. CALL Statement

Purpose:

A subroutine is referred to by the appearance of its name following the keyword CALL in a CALL statement.

Form:

```
CALL s [ ( [a[ ,a] ...] ) ]
```

where *s* is the name of the subroutine. (The name *s* can't appear as both the name of a subroutine and as the name of a function in the same program unit.) Each *a* is an actual argument and must match the corresponding dummy argument of *s* in type and usage (see 7.3.4).

Description:

The arguments can be:

- Expressions
- Array names
- Statement labels
- Intrinsic functions
- External procedures
- **Internal procedures**
- Dummy procedures

The number of arguments can't exceed 250.

When a statement label appears as an actual argument, it is written as **n*, ***&n*, or $*n***, where *n* is the statement label. The corresponding dummy argument must be an asterisk (*) **or currency symbol ($)**. Use external procedure names in the same manner as function references (see 7.3.2.2).

A reference to a subroutine results in execution of the body of the subroutine. When execution of the subroutine is complete, the calling routine resumes at the statement following the call or at one of the statement labels appearing in the argument list of the CALL statement.

7.3.4. Function and Subroutine Arguments

On each call to a subprogram, the actual arguments are matched to the formal arguments based on their order in the respective list.

The actual arguments must match the formal arguments in number, type, and usage. ASCII FORTRAN automatically checks the types and number of actual arguments against those expected on subprogram entry, except for certain exceptions listed in 8.5.7 **(ARGCHK options of the COMPILER statement)**. If a dummy argument is an array, the corresponding actual argument must be either an array or an array element. In the first case, the size of the dummy array must not exceed the size of the actual array. In the second case, the effect is as if the first element of the dummy array is equivalenced to the indicated element of the actual array. The size of the dummy argument must not exceed the size of that portion of the actual array that follows and includes the indicated element.

A dummy argument is an array if it appears in a DIMENSION statement, or with dimensions in an explicit type statement in the subprogram. No dummy arguments can appear in an EQUIVALENCE, COMMON, DATA, PARAMETER, SAVE, or INTRINSIC statement.

If a dummy argument is assigned a value in the subprogram, the corresponding actual argument must be a variable, a character substring, an array element, or an array. Never use a constant or an expression as an actual argument if the corresponding dummy argument is assigned a value. Such an error is not detected by the ASCII FORTRAN system and the results are unpredictable.

If an actual argument is in COMMON, or appears more than once in the argument list, undesirable side effects can occur when the corresponding dummy argument is assigned a value. Consider the following example:

```
      SUBROUTINE MULT(A,B,C,LI,LJ,LK)
      DIMENSION A(LI,LJ),B(LJ,LK),C(LI,LK)
      DO 1 I = 1, LI
         DO 1 K = 1, LK
            C(I,K) = 0.0
            DO 1 J = 1, LJ
 1             C(I,K) = A(I,J)*B(J,K)+C(I,K)
      RETURN
      END
```

When this subroutine is called by:

```
CALL MULT(X,Y,X,5,5,5)
```

the result of the call is incorrect since X is modified before the multiplication is complete. Another type of problem is illustrated in the following:

```
      SUBROUTINE FC3(A,B,C)
      A = B + C
      C = SQRT(A**2+B**2+C**2)
      A = A - B
      END
```

When the call is:

```
CALL FC3(X,Y,X)
```

the result depends on whether or not optimization is requested. If there is no optimization, the code executed is:

```
X = Y + X
X = SQRT(X**2 + Y**2 + X**2)
X = X - Y
```

If optimization is requested, the code executed is:

```
T1 = Y + X
X = SQRT(T1*T1 + Y*Y + X*X)
X = T1 - Y
```

Similar results can happen if the dummy argument to which a value is assigned is associated with a variable in a COMMON block.

The compiler generates code for a subprogram, assuming that its arguments change their values only through explicit statements in the subprogram. If an argument is actually in a common block, and another subprogram is called which changes its value, the execution of the first subprogram may or may not reflect this change in the argument value. **The same situation exists on execution of a direct access I/O statement**

when the associated variable is unknown for the statement either because of a variable being used for a unit number, or because no OPEN or DEFINE FILE statement on the unit number exists in the subprogram. In this case, the compiler does not know which variable is the associated variable, so it may assume that the associated variable's value is not changed.

7.4. RETURN Statement

Purpose:

The RETURN statement provides a mechanism for returning control to the calling routine from a point in the subprogram other than the END statement.

Form:

```
RETURN [i]
```

where *i* is an optional integer expression whose value for example, *n* denotes the *n*th statement label in the argument list.

Description:

A RETURN statement in a main program is equivalent to a STOP statement (see 4.8).

When the RETURN statement is executed in a function subprogram, evaluation of the referencing expression is resumed with the function name replaced by its returned value. **However, if *i* is specified, execution continues at the statement whose label is selected by *i*. This results in a loss of the value normally returned from a function reference.**

Execution of a RETURN statement in a subroutine subprogram ordinarily causes execution to continue following the CALL statement. However, if *i* is specified, execution continues at the statement whose label is selected by *i*. The value of *i* must be positive and no greater than the number of statement labels passed as arguments.

If the value of *i* is outside of the correct range, a warning message is printed at run time:

```
RETURN ARGUMENT i OUT OF RANGE
```

A return is then executed as if *i* is not specified.

Examples:

```
SUBROUTINE SCALAR(I,K,A,B,IMAX,KMAX,N,TEST,*,*)
  .
  .
  .
RETURN 1
  .
  .
  .
RETURN
```

```
C          The first return passes control to the statement
C          identified by the first label in the argument list,
C          which is the ninth argument of subroutine SCALAR
C          (see 7.3.3.1).  The second return passes program control
C          to the point where the subroutine was called.
```

7.5. ENTRY Statement

Purpose:

The ENTRY statement defines an alternate point to begin execution of a subroutine or function subprogram, and an alternate name by which the subprogram can be referenced. The ENTRY statement can also be used to let a function subprogram return values of different types.

Form:

```
ENTRY n [ ([a[ ,a] ...]) ]
```

Description:

The entry name *n* is a symbolic name that can be referenced subject to the same rules as the corresponding subroutine or function name. **As with subroutine and function names, the currency symbol ($) should be avoided in entry names.**

Each dummy argument *a* is a symbolic name, which must be unique in the dummy argument list. The dummy arguments need not agree with those of the SUBROUTINE or FUNCTION statement, or any other ENTRY statement, in order, number, type, or usage. A dummy argument that appears in more than one dummy argument list need not occupy the same position in each list. **The use of slashes (/*a*/) is permitted as it is for a SUBROUTINE or FUNCTION statement.** A maximum of 250 dummy argument names can appear in a program unit (SUBROUTINE, FUNCTION, and all ENTRY statements). The maximum number of character dummy arguments allowed in a single ENTRY statement is 63.

Any dummy argument may use an asterisk **or a currency symbol** to denote that the corresponding actual argument is a statement label. (The FORTRAN 77 standard limits the use of asterisks for dummy arguments to subroutines; **ASCII FORTRAN allows them in functions also.)**

All references to a dummy argument or an entry name in executable statements must follow the first appearance of the name in a dummy argument list or the ENTRY statement that defines the entry name.

A program unit can contain multiple ENTRY statements. Each defines an alternate entry point to the subroutine or function and defines the name by which the entry point can be referenced. An entry point in a subroutine must be referenced as a subroutine; an entry point in a function must be referenced as a function. No more than 511 entry points can be specified for all program units in a compilation.

ENTRY statements are not executable. If the normal sequence of statement execution causes an ENTRY statement to be executed, it is skipped and the next executable statement is executed. When an ENTRY-defined name is referenced (in a CALL statement or function reference), execution of the subprogram begins with the first executable statement following the ENTRY statement. An ENTRY statement must not appear in the range of a DO-loop or in a block IF structure (that is, between a block IF statement and its corresponding END IF statement, where the IF-level is greater than zero).

If the ENTRY statement appears in a function subprogram, the entry name is available for use as a variable in the same manner as the function name; the entry names and function name are effectively equivalenced. The type of each function entry name is determined from explicit typing, the IMPLICIT statement, or the I-N integer rule, whichever applies to the name. At the time of return, if the last function/entry name variable assigned is of a different type than that required for the entry name referenced, the function value is undefined. When a function is type character, all of its entry names must be type character, **but all of its entry names need not be of equal length. A reference to a character function name or character entry name always uses the character length associated with the name most recently entered.**

The FORTRAN 77 standard requires that if a function is of type character with a length of *, all ENTRY names must also be type character and of length *; otherwise, the character length must be the same integer value for each ENTRY name.

A subprogram can't reference itself by the subprogram name or any of its entry names, either directly or indirectly. Doing so cannot always be detected by the compiler and results in an infinite loop at execution time.

If information for the object-time dimensioning of an array is passed in a reference to an ENTRY statement, the array name and all of its dimension parameters (except any that are in a common block) must appear in the argument list of the ENTRY statement.

When dummy arguments are not in the dummy argument list of the name through which the subprogram is referred to, they generally refer to the actual arguments with which they are most recently matched. The compiler can't detect this situation; don't attempt to use it since it is an unsupported side effect. Problems that can occur include:

- If an actual argument is an expression, its value is passed in a temporary area. Since this area is reused for following statements, its value can change even though the expression has the same value if it is reevaluated.
- If the segmentation facility of the collector is in use, the storage occupied by the actual argument can contain part of a different program unit when the subprogram is next called.
- If automatic storage is used, the dummy argument can find different contents on the stack the next time the subprogram is called.
- If the actual argument is a variable, the dummy argument remains associated with that variable and reflects any changes to the value of that variable. As a result, the dummy argument cannot retain the value that it had at the last execution of the subprogram.
- A dummy argument can't be used unless an entry point is previously entered where it exists in the argument list.

- Optimization generally can't know that this side effect is taking place, and produces code as if it has not taken place.

7.6. BLOCK DATA Subprograms

A BLOCK DATA subprogram allows the initialization of variables in blank or labeled common blocks to be combined in a single program unit. **ASCII FORTRAN allows the initialization of blank common, but the FORTRAN 77 standard does not.**

7.6.1. Structure

A BLOCK DATA subprogram is a separate program unit. It must begin with a BLOCK DATA statement (see 7.6.2). The only other statements that are permitted in a BLOCK DATA subprogram are IMPLICIT, PARAMETER, DIMENSION, COMMON, SAVE, EQUIVALENCE, DATA, END, type statements, and **the EXTERNAL and INTRINSIC statements.**

Since the association between common variables of different program units is by position in a common block, rather than by name, all elements of a common block should be listed in the COMMON statement, though they are not all initialized.

The FORTRAN 77 standard lets named common blocks be initialized only in BLOCK DATA subprograms. **ASCII FORTRAN allows any program unit (including BLOCK DATA subprograms) to initialize elements in one or more common blocks, and a common block can be initialized by more than one program unit. If an element of a common block is initialized by more than one program unit, or more than once in a program unit, one of the initial values is selected when the program is collected. Unless the initial values assigned to the element are all identical, the actual value assigned to the element at the beginning of execution is unpredictable.**

A BLOCK DATA subprogram requires no local storage. Any variable that appears in specification statements, but not in a COMMON statement, is noted in a warning message. No storage is reserved for variables that do not appear in a COMMON statement.

Since a BLOCK DATA subprogram defines no entry points, the automatic element inclusion facilities of the collector never cause inclusion of the relocatable generated for a BLOCK DATA subprogram. As a result, the use of a BLOCK DATA subprogram requires that the collection include an IN directive for the corresponding relocatable element.

7.6.2. BLOCK DATA Statement

Purpose:

The BLOCK DATA statement identifies a program unit as a BLOCK DATA subprogram.

Form:

```
BLOCK DATA [sub]
```

where *sub* is an optional n ame for the BLOCK DATA subprogram.

Description:

The BLOCK DATA statement is the initial statement in all BLOCK DATA subprograms.

The name *sub* is a global name and must not be the same as the name of an external procedure, main program, common block, or other BLOCK DATA subprogram in the same executable program. In addition, *sub* must not be the same as any local name in the subprogram.

There must not be more than one unnamed BLOCK DATA subprogram in an executable program.

The name of a BLOCK DATA subprogram has no explicit use in the FORTRAN language. It is available mainly for documentation.

The same common block must not be specified in more than one BLOCK DATA subprogram in the same executable program.

Examples:

```
      BLOCK DATA BLKA
      DIMENSION K(10)
      COMMON /B/K
      DATA K/1,2,3,4,5,6,7,8,9,10/
      END
C           This exemplifies a BLOCK DATA subprogram named BLKA.
C           The COMMON block named B is initialized by the
C           DATA statement.
      BLOCK DATA
      DIMENSION A(10),M(10)
      COMMON /X/ A,B,C /NAME1/M
      DATA M/2*1, 2*3, 2*5, 2*2, 2*4/,B,C/1.0,2.0/
      END
C           The elements of M are initialized to the sequence of
C           values (1,1,3,3,5,5,2,2,4,4).  The variables B and C
C           are set to 1.0 and 2.0, respectively.  No initial value
C           is assigned to any element of A.
```

7.7. FORTRAN-Supplied Procedures

The ASCII FORTRAN system provides a set of precoded procedures, called intrinsic procedures, for your convenience. These procedures are in three groups:

1. Intrinsic functions
2. **Pseudo-Functions and SBITS**
3. **Service subprograms**

These groups are described in detail in the following subsections.

References to certain intrinsic procedures cause generation of code that performs the requested action at the point of the reference. Such procedures are called inline procedures. References to the other intrinsic procedures result in calls to library subprograms; these procedures are called library procedures. The inline or library nature of each intrinsic procedure is specified in its respective description.

7.7.1. Intrinsic Functions

ASCII FORTRAN provides numerous intrinsic functions (sometimes called built-in functions). You do not write or modify them. However, their actual form and manner of referencing corresponds to that of a subprogram defined by a FUNCTION statement in a source module. These functions always return one value (function value) to the calling statement and perform computations frequently needed in FORTRAN programs.

The nonmathematical intrinsic functions arecalled typeless functions. Each argument of a typeless function is a single-word expression. If the argument is a character expression whose length is less than four characters (that is, less than one word), the argument is left-justified and blank-filled before performing the function. The function is performed bit-by-bit, treating a 0 as false and a 1 as true. These functions are summarized in Table 7-1. See 2.5.5.3 for more information on expressions involving typeless functions.

Table 7-1. Typeless Intrinsic Functions

Function Description	Function Name	Number of Args	Result	Sample Reference
Logical Product	AND	2	Typeless	AND(*word,word2*)
Logical Sum	OR	2	Typeless	OR(*word,word2*)
Exclusive OR	XOR	2	Typeless	XOR(*word,word2*)
Treat Argument as Typeless	BOOL	1	Typeless	BOOL(*word*)
Complement	COMPL	1	Typeless	COMPL(*word*)

The mathematical and character intrinsic functions are described in NO TAG. Some general rules for mathematical and character intrinsic functions follow:

- An intrinsic function is referred to in an expression as if it is a single value or user function.
- All trigonometric angles are expressed in radians.
- The type of an intrinsic function can be declared in the program unit that contains a reference to it. However, the declared type must match the type specified for the function in NO TAG. The intrinsic function name retains its automatic typing attribute.
- If an intrinsic function name appears in an EXTERNAL statement, the name becomes an external procedure name. If an intrinsic function name appears in an explicit typing statement that differs from the type associated with the name, the name loses its intrinsic properties. The name can be a function or a variable name that you supply, depending on the use of the name. Other names for the same function retain their automatic typing attribute.

For example, in the code:

```
INTEGER SIN,COS
Y = SIN(X) * DSIN(X)+COS
```

SIN(X) is a call to an external function that must be supplied. The reference DSIN(X) is a reference to the single-precision sine function, because of automatic type association. The reference to COS is a reference to the scalar variable COS.

- **When a reference is made to an intrinsic function with an argument of a type or length different from that specified in NO TAG, the ASCII FORTRAN compiler provides an automatic function selection facility. The compiler automatically selects, from the same group, the function that processes the type of argument passed. (The FORTRAN 77 standard has this automatic function selection facility only on the generic names.)**
- **Generally, if the argument is of a type for which there is no corresponding function in NO TAG or has a different number of arguments, the function is assumed to be a programmer-supplied function. If a reference is made to a function that doesn't handle an integer argument but does handle a real argument, the integer argument is converted to type real. The MAX and MIN functions constitute a special case. If the type of the arguments doesn't match the type specified for the function name used, the function is selected from MAX0/AMAX1/DMAX1 or MIN0/AMIN1/DMIN1, respectively.**
- If an intrinsic function name first appears in the program unit in an executable statement with no following parentheses or arguments, the name is assumed to be a scalar variable name.
- When an intrinsic function is used as an actual argument, only a specific intrinsic function name can be used. In addition, the specific intrinsic function name must appear in an INTRINSIC statement. See 6.10 for a list of intrinsic function names that cannot be used as actual arguments.
- **For intrinsic functions that require more than one argument, the arguments must all be of the same type. If they are not, they are converted to their highest common type before selection of the appropriate function.**
- An IMPLICIT statement doesn't change the type of an intrinsic function.
- AMOD, MOD, and DMOD are not defined when the value of the second argument is zero.

The procedure type (Proc Type) column in NO TAG indicates whether the associated reference causes generation of code that performs the necessary computation (inline) or results in a call to a library element. In the function descriptions, the notation *pv expression* denotes the principal value of the multiple-valued complex *expression*.

Table 7-2. Caption

Function Description	Function Name	Arguments		Function Value	Proc. Type	Generic Name
		No.	Type			
Natural logarithm $y = \ln x$ $(x > 0)$ $y = \text{pv} \ln x$	ALOG DLOG CLOG CDLOG	1	Real Double Complex D-complex	Real Double Complex D-complex	Library	LOG
Common logarithm $y = \log_{10} x$ $(x > 0)$	ALOG 10 DLOG 10	1	Real Double	Real Double	Library	LOG 10
Exponential $y = e^x$	EXP DEXP CEXP CDEXP	1	Real Double Complex D-complex	Real Double Complex D-complex	Library	EXP
Square root $y = \sqrt{x}$ $(x \geqq 0)$ $y = \text{pv} \sqrt{x}$	SQRT DSQRT CSQRT CDSQRT	1	Real Double Complex D-complex	Real Double Complex D-complex	Library	SQRT
Arc sine $y = \text{Arcsin } x$ (y in radians) $-1 \leqslant x \leqslant 1$	ASIN ARSIN DASIN DARSIN	1	Real Real Double Double	Real Real Double Double	Library	ASIN
Arc cosine $y = \text{Arcos } x$ (y in radians) $-1 \leqslant x \leqslant 1$	ACOS ARCOS DACOS DARCOS	1	Real Real Double Double	Real Real Double Double	Library	ACOS
Arc tangent $y = \text{Arctan } x$	ATAN DATAN	1	Real Double	Real Double	Library	ATAN
$y = \text{Arctan } (x_1 / x_2)$ (see note 14)	ATAN2 DATAN2	2	Real Double	Real Double	Library	ATAN2
Sine $y = \sin x$ (x in radians)	SIN DSIN CSIN CDSIN	1	Real Double Complex D-complex	Real Double Complex D-complex	Library	SIN
Cosine $y = \cos x$ (x in radians)	COS DCOS CCOS CDCOS	1	Real Double Complex D-complex	Real Double Complex D-complex	Library	COS
Tangent $y = \tan x$ (x in radians)	TAN DTAN CTAN CDTAN	1	Real Double Complex D-complex	Real Double Complex D-complex	Library	TAN

continued

Function Description	Function Name	Arguments No.	Arguments Type	Function Value	Proc. Type	Generic Name
Cotangent $y = \text{cotan } x$ (x in radians)	COTAN DCOTAN	1	Real Double	Real Double	Library	COTAN
Hyperbolic sine $y = (e^{x}-e^{-x}) / 2$	SINH DSINH CSINH CDSINH	1	Real Double Complex D-complex	Real Double Complex D-complex	Library	SINH
Cube root $y = x^{1/3}$ $y = pvx^{1/3}$	CBRT DCBRT CCBRT CDCBRT	1	Real Double Complex D-complex	Real Double Complex D-complex	Library	CBRT
Hyperbolic cosine $y = (e^{x}+e^{-x}) / 2$	COSH DCOSH CCOSH CDCOSH	1	Real Double Complex D-complex	Real Double Complex D-complex	Library	COSH
Hyperbolic tangent $y = (e^{x}-e^{-x}) / (e^{x}+e^{-x})$	TANH DTANH CTANH CDTANH	1	Real Double Complex D-complex	Real Double Complex D-complex	Library	TANH
Absolute value (see note 1) $y = \| x \|$ $y = ((\text{real } x)^2 + (\text{imag } x)^2)^{1/2}$	IABS ABS DABS CABS CDABS	1	Integer Real Double Complex D-complex	Integer Real Double Real Double	Inline Inline Inline Library Library	ABS
Error function $y = (2/\sqrt{Pi})\int_0^x e^{-u^2} du$	ERF DERF	1	Real Double	Real Double	Library	ERF
Complementary error function $y = (2/\sqrt{Pi})\int_x^\infty e^{-u^2} du$ $= 1 - ERF(x)$	ERFC DERFC	1	Real Double	Real Double	Library	ERFC
Maximum value y = largest value from $\{x1, . . . , xn\}$ $y \geqq x_i, 1 \leqslant i \leqslant n$	MAX0 AMAX1 DMAX1	≧2	Integer Real Double	Integer Real Double	Inline	MAX
	AMAX0 MAX1	≧2	Integer Real	Real Integer	Inline	-
Minimum value y = smallest value from $\{x1, . . . , xn\}$ $y \geqq x_i, 1 \leqslant i \leqslant n$	MIN0 AMIN1 MIN1	≧2	Integer Real Double	Integer Real Double	Inline	MIN
	AMIN0 MIN1	≧2	Integer Real	Real Integer	Inline	-

continued

Function Description	Function Name	Arguments		Function Value	Proc. Type	Generic Name
		No.	Type			
Gamma $y = \int_0^\infty u^{x-1} e^{-u}\, du$ For real: $0 <\lvert x\rvert< 34$ For double: $0 <\lvert x\rvert< 170$	GAMMA DGAMMA	1	Real Double	Real Double	Library	GAMMA
Log gamma $y = \int_0^\infty u^{x-1} e^{-u}\, du$ $(x > 0)$	ALGAMA DLGAMA	1	Real Double	Real Double	Library	LGAMMA
Remaindering - remainder of dividing x_1 by x_2 $y = x_1$ (modulo x_2) AMOD: x_1 / x_2 2^{27} DMOD: x_1 / x_2 2^{60}	MOD AMOD DMOD	2	Integer Real Double	Integer Real Double	Inline	MOD
TYPE CONVERSION Conversion to integer int(a) (see note 2)	— INT IFIX HFIX IDINT IDFIX — —	1	Integer Real Real Real Double Double Complex D-complex	Integer	Inline	INT
Conversion to real (see notes 3 and 13)	REAL FLOAT — SNGL — —	1	Integer Integer Real Double Complex D-complex	Real	Inline	REAL
Conversion to double (see note 4)	— DFLOAT — — — DREAL	1	Integer Integer Real Double Complex D-complex	Double	Inline	DBLE
Conversion to complex (see notes 5 and 13)	— — — — —	1 or 2	Integer Real Double Complex D-complex	Complex	Inline	CMPLX
Conversion to double complex (see note 5)	— — — — —	1 or 2	Integer Real Double Complex D-complex	D-complex	Inline	DCMPLX
Conversion to integer (see note 6)	ICHAR	1	Character	Integer	Inline	—

continued

Function Description	Function Name	Arguments		Function Value	Proc. Type	Generic Name
		No.	Type			
Conversion to character (see note 6)	CHAR	1	Integer	Character	Inline	—
Truncation (int(a), see note 2)	AINT DINT	1	Real Double	Real Double	Inline	AINT
Nearest whole number int(a+.5) if $a \geqq 0$ int(a-.5) if $a < 0$	ANINT DNINT	1	Real Double	Real Double	Library	ANINT
Nearest integer int(a+.5) if $a \geqq 0$ int(a-.5) if $a < 0$	NINT IDNINT	1	Real Double	Integer	Inline	NINT
Transfer of sign $y = \|x_1\|$ if $x_2 \geqq 0$ $= - \|x_1\|$ if $x_2 < 0$	ISIGN SIGN DSIGN	2	Integer Real Double	Integer Real Double	Inline	SIGN
Positive difference $y = x_1 - \min(x_1,x_2)$	IDIM DIM DDIM	2	Integer Real Double	Integer Real Double	Inline	DIM
Imaginary part of COMPLEX argument (see note 1)	AIMAG DIMAG	1	Complex D-complex	Real Double	Inline	IMAG
Complex conjugate (see note 1) y = a-bi for x = a+bi	CONJG DCONJG	1	Complex D-complex	Complex D-complex	Inline	CONJG
Double precision product $y = x_1 * x_2$	DPROD	2	Real	Double	Inline	—
Length y = length of character entity (see note 11)	LEN	1	Character	Integer	Inline	—
Index of a substring y = location of substring x_2 in string x_1 (see note 10)	INDEX	2	Character	Integer	Library	—
Lexically greater than or equal (see note 12)	LGE	2	Character	Logical	Inline	—
Lexically greater than (see note 12)	LGT	2	Character	Logical	Inline	—
Lexically less than or equal (see note 12)	LLE	2	Character	Logical	Inline	—

continued

Function Description	Function Name	Arguments		Function Value	Proc. Type	Generic Name
		No.	Type			
Lexically less than (see note 12)	LLT	2	Character	Logical	Inline	—
Uppercase - return a string identical to the argument, except that all lowercase ASCII characters have been converted to uppercase	UPPERC	1	Character	Character	Library	—
Lowercase - return a string identical to the argument, except that all uppercase ASCII characters have been converted to lowercase	LOWERC	1	Character	Character	Library	—
Trim length - return the length of the argument string with all trailing blanks removed	TRMLEN	1	Character	Integer	Library	—

Notes:

1. A complex value is expressed as an ordered pair of reals, (*xr*,*xi*), where *xr* is the real part and *xi* is the imaginary part.
2. For x of type integer, int(x)=x. For x of type real or double precision, there are two cases: if $|x| < 1$, int(x)=0; if $|x| \geq 1$, int(x) is the integer whose magnitude is the largest integer that does not exceed the magnitude of x and whose sign is the same as the sign of x. For example:

   ```
   int (-3.7) = -3
   ```

   For x of type complex, int(x) is the value obtained by applying the above rule to the real part of x. For x of type real, IFIX(x) is the same as INT(x).
3. For x of type real, REAL(x) is x. For x of type integer or double precision, REAL(x) is as much precision of the significant part of x as a real datum can contain. For x of type complex, REAL (x) is the real part of x.

   For x of type integer, FLOAT(x) is the same as REAL(x).

   The result of REAL is always single precision.
4. For x of type double precision, DBLE(x) is x. For x of type integer or real, DBLE(x) is as much precision of the significant part of x as a double-precision datum can contain. For x of type complex, DBLE(x) is as much precision of the significant part of the real part of x as a double-precision datum can contain.
5. CMPLX can have one or two arguments.

For x of type integer, real, or double precision, CMPLX(x) is the complex value whose real part is REAL(x) and whose imaginary part is zero.

For x of type complex, CMPLX(x) is x.

For x of type complex*16, CMPLX(x) is the complex value whose real part is REAL(x) and whose imaginary part is REAL(DIMAG(x)).

CMPLX(x_1,x_2) is the complex value whose real part is REAL(x_1) and whose imaginary part is REAL(x_2).

DCMPLX can have one or two arguments.

The result of DCMPLX is always double-precision complex.

For x of type integer, real, or double precision, DCMPLX(x) is the complex*16 value whose real part is DBLE(x) and whose imaginary part is zero.

For x of type complex*16, DCMPLX(x) is x.

For x of type complex, DCMPLX(x) is the complex*16 value whose real part is DBLE(x) and whose imaginary part is DBLE(AIMAG(x)).

DCMPLX(x_1,x_2) is the complex*16 value whose real part is DBLE(x_1) and whose imaginary part is DBLE(x_2).

6. ICHAR provides a means of converting from a character to an integer, based on the position of the character in the ASCII collating sequence. The first character in the collating sequence corresponds to position 0 (the ASCII control character NUL, which has octal representation 000), and the last one corresponds to position 255 (octal representation 0377).

   The value of ICHAR(x) is an integer in the range: $0 \le$ ICHAR(x) ≤ 255, where x is an argument of type character of length one. The position of that character in the ASCII collating sequence is the value of ICHAR.

   For any ASCII characters c_1 and c_2 , (c_1 .LE. c_2) is true if and only if (ICHAR(c_1) .LE. ICHAR(c_2)) is true, and (c_1 .EQ. c_2) is true if and only if (ICHAR(c_1) .EQ. ICHAR(c_2)) is true.

   CHAR(i) returns the character in the i^{th} position of the ASCII collating sequence. The value is of type character of length one. The parameter i must be an integer expression whose value must be in the range $0 \le i \le 255$.

   ```
   ICHAR(CHAR(i)) = i for 0 ≤ i ≤ 255.

   CHAR(ICHAR(c)) = c for any ASCII character c.
   ```

7. All angles are expressed in radians.
8. The result of a function of type complex is the principal value.
9. All arguments in an intrinsic function reference must be of the same type.
10. INDEX(x_1,x_2) returns an integer value indicating the starting position within the character string x_1 of a substring identical to string x_2. If x_2 occurs more than once in x_1, the starting position of the first occurrence is returned.

    If x_2 does not occur in x_1, the value 0 is returned. Zero is returned if LEN(x_1) < LEN (x_2).

11. The value (that is, the contents) of the argument of the LEN function need not be defined at the time the function reference is executed.

12. Using the intrinsic functions LGE, LGT, LLE, or LLT with arguments op1 and op2 (both character expressions) performs the same comparisons and returns the same result (.TRUE. or .FALSE.) as the character relational expression op1 relop op2, where relop is .GE., .GT., .LE., or .LT., respectively. These intrinsic functions are inline only if the arguments are CHARACTER*4 or smaller.

13. REAL, FLOAT, SNGL, and CMPLX are not generic names for DREAL and DCMPLX.

    Example:

    ```
    DOUBLE PRECISION D1,D2,D3
    COMPLEX * 16 CX1,CX2
       .
       .
       .
    D1 = REAL(CX1)
    CX2 = CMPLX(D2,D3)
    CX1 = CMPLX(SNGL(CX2), SNGL(D2))
    ```

    This example has a hidden, but severe, loss of precision due to the use of REAL and CMPLX. Arithmetic conversions are done that result in a loss of precision and sometimes cause an overflow condition. Use DREAL and DCMPLX to prevent conversions.

    Change all occurrences of REAL, FLOAT, SNGL, and CMPLX to DREAL and DCMPLX when you increase arithmetic accuracy in a program by changing:

    - all single-precision variables to double-precision variables
    - all single-precision complex variables to double-precision complex variables

14. The values for ATAN2 and DATAN2 for various cases are calculated as follows:

    ```
    y = Arctan(x1/x2)                when x2>0
    y = SIGN(x1)*Pi/2                when x2=0
    y = Arctan(x1/x2) + Pi*SIGN(x1)  when x2<0
    ```

    where:

    ```
    SIGN(x1) = 1  when x1 ≥ 0
    SIGN(x1) = -1  when x1 < 0
    ```

    The ranges of values for ATAN2 and DATAN2 are listed in the following table:

	x2<0	**x2=0**	**x2>0**
x1>0	*Pi>y>Pi/2*	*y=Pi/2*	*Pi/2>y>0*
x1=0	*y=Pi*	*(error)*	*y=0*
x1<0	*-Pi<y<-Pi/2*	*y=-Pi/2*	*-Pi/2<y<0*

7.7.2. BITS, SBITS, and SUBSTR

ASCII FORTRAN supplies two pseudo-functions, BITS and SUBSTR, to facilitate the manipulation of bit and character strings.

These functions are similar to intrinsic functions, except that pseudo-functions can also appear as the targets of assignment statements.

SBITS, a bit-manipulation intrinsic function, is also supplied.

7.7.2.1. BITS

Purpose:

The BITS pseudo-function selects a bit string of a specific length from an entity of any type.

Form:

```
BITS (e, i, l)
```

where:

e

is an expression, unless BITS is used as the target of an assignment statement, in which case *e* must be a scalar variable or array element.

i

is an integer expression specifying the initial bit position of the bit string to be selected, where $1 \le i \le t$ (*t* is the total number of bits in item *e*).

l

is an integer expression specifying the length (in bits) of the bit string being selected, where $1 \le l \le 36$. If BITS is used as the target of an assignment statement, then $1 \le l \le t$.

Description:

When BITS appears in an arithmetic expression (that is, not used as the target in an assignment statement), it returns a 36-bit integer result, which is set up as follows (note that bits are counted from left to right, starting at 1): the rightmost *l* bits are taken from bits *i* to $i+l-1$ of *e*, and the leftmost 36-*l* bits are zero-filled.

When you use BITS as a pseudo-function (target of an assignment), *e* must be a scalar variable or array element. The value from the right side of the assignment is stored in the indicated bits of *e*, and all other bits of *e* are unchanged. More than 36 bits of *e* can be set. Call the right side expression *r*, with length l_1 (in bits). If $l_1 > l$, then only the rightmost *l* bits of *r* are used. If

$l_1 < l$, then all of *r* is placed in *e* starting at bit *i* (that is, left-justified, in bits *i* to $i+l_1$-1) and the remaining portion of *e* to be filled (bits $i+l_1$ to $i+l$-1) contains zeros.

The relational expression $i+l-1 \leq t$ must be satisfied for all uses of BITS.

Examples:

```
      DATA L / 0000022000033/
      L1 = BITS(L,1,18)   @L1 is 18 (octal 022)

      N = 64              @octal 0100
      BITS(N,31,6) = 63      @N is 127 (octal 0177)

      CHARACTER C8*8/'abcdefgh'/, C4*4/'1234'/
      BITS(C8,37,18) = 'Z'
C           The fifth character of C8 gets 'Z', and the sixth character
C           gets all zero bits.
      BITS(C4,19,18) = 'abc' @C4 is '12bc'

      COMPLEX*16 C16
      DOUBLE PRECISION R8
      BITS(C16,73,72) = R8
C           R8 is placed in the imaginary portion of C16 (3rd and
C           4th words)
      BITS(C16,1,144) = R8
C           R8 is placed in the real portion of C16 (first 2 words),
C           with 0 in the imaginary part (3rd and 4th words).
```

7.7.2.2. SBITS

Purpose:

The SBITS intrinsic function is similar to BITS, except that the result is sign-extended.

Form:

```
SBITS (e, i, l)
```

where:

e

an expression, is the entity that the bit string is selected from.

i

an integer expression, is the initial bit position in *e* of the bit string being selected ($1 \leq i \leq t$, where *t* is the total number of bits in item *e*).

l

an integer expression, is the length in bits of the bit string being selected ($1 \leq l \leq 36$).

Description:

SBITS is a sign-extended BITS function, with a 36-bit integer result. You can't use it as a pseudo-function.

The 36-bit result is set up as follows (bits are counted from left to right, starting at 1): the rightmost *l* bits are taken from bits *i* to *i+l*-1 of *e*, and the leftmost 36-*l* bits are all set to bit *i* of *e* (the sign bit of the extracted bit string). (The BITS function sets the leftmost 36-*l* bits of the result to 0.) The relational expression *i+l*-1≤*t* must be satisfied.

The sign-extension feature of SBITS allows results to be negative numbers. For example, if several signed integers are packed into a single word, SBITS can properly extract individual signed (positive or negative) integers.

Examples:

```
      DATA N /0767574737271/
      I = BITS(N,31,6)
      J = SBITS(N,31,6)
C            I will be 57 (octal 071)
C            J will be -6 (octal 0777777777771)

      CHARACTER*10 /'1234567890'/
      I = BITS(A,76,6)
      J = SBITS(A,76,6)
C            I will be 57 (octal 071)
C            J will be -6
```

7.7.2.3. SUBSTR

Purpose:

The SUBSTR pseudo-function selects a character substring from a given character string.

Form:

```
SUBSTR (e,i,l)
```

where:

e

is a character expression, unless SUBSTR is used as the target of an assignment statement, in which case *e* must be a scalar character variable or character array element.

i

is an integer expression specifying the initial character position of the selected substring, where *i* ≥ 1 and *i* is less than or equal to the length of *e*.

l

is an integer expression specifying the length (in characters) of the selected substring, where *l* has the same restrictions as *i*.

Description:

You can use the SUBSTR pseudo-function in any character or relational expression.

The function selects a character string that is *l* characters long, starting at position *i* of expression *e*. Characters are counted from left to right, so if *i* is 1, the substring starts with the first character of expression *e*. The expression *i+l*-1 (representing the character position of the final character of the substring) must be less than or equal to the length of *e*.

You can also use this function as the target of an assignment statement. However, *e* must be a simple variable, an array element, or a statement function reference for which assignment is permitted. The value from the right side of the assignment is stored in the indicated substring. All other characters of the target remain unchanged.

You can also specify substrings by using the colon syntax specified in 2.4.5.

Examples:

```
C           If STR1 is typed character, then
      STR1 = SUBSTR('NEWYEARRESOLUTION',8,10)
C           selects substring 'RESOLUTION' and assigns it
C           to STR1.

C           If YEAR is character and contains the value '1983'
      SUBSTR(YEAR,4,1) = '4'
C           leaves YEAR with a value of '1984'.
```

7.7.3. Service Subprograms

OS 1100 ASCII FORTRAN provides a set of service subprograms for your convenience and information. Some of these subprograms aid you in pinpointing problem areas (DUMP and PDUMP). Others check for specific error conditions or convert information between ASCII and Fieldata forms, while another stops execution at any point that you indicate (EXIT). ERTRAN lets you refer to certain Executive Request (ER) functions. Another set lets you capture and diagnose arithmetic exceptions at execution time.

Service subprograms are similar in form to a subroutine or function subprogram in a FORTRAN source module, but you don't define them. Service subprograms may or may not return a value to the calling statement. Service subprograms are not treated in a special manner by the compiler, but rather are treated exactly the same as your program. Therefore, if a service subprogram can be called both as a subroutine and as a function, you can use only one type of reference from a given program unit. The use of service subprograms cannot be diagnosed by the compiler as nonstandard, since the compiler looks upon them as user-supplied subprograms. Their names are not treated as special names by the compiler.

Each service subprogram is explained in the following paragraphs.

7.7.3.1. DUMP

Purpose:

The DUMP subprogram dumps the contents of specified storage to the system output file and terminates execution.

Form:

```
CALL DUMP (a,b,f)
```

where:

a and *b*

are variables, array elements, or arrays that indicate the limits of storage to be dumped.

f

indicates the dump format. *f* can be any of the following:

0	Octal
1	Not used
2	LOGICAL
3	Not used
4	INTEGER
5	REAL
6	DOUBLE PRECISION
7	COMPLEX
8	COMPLEX*16
9	CHARACTER

Description:

A dump of storage contents between *a* and *b*, inclusive, is made according to format *f* and execution is then terminated.

Either *a* or *b* can be the upper or lower limit of storage, but both must be in the same program unit or the same common block.

Example:

```
       CALL DUMP (LOWER, UPPER, 0)
  C           This statement requests a storage dump
  C           in octal format starting at location LOWER and ending
  C           with location UPPER.  Execution then terminates.
```

7.7.3.2. PDUMP

Purpose:

The PDUMP subprogram dumps the contents of specified storage to the system output file and continues execution.

Form:

```
  CALL PDUMP (a,b,f)
```

where:

a and *b*

are variables, array elements, or arrays that indicate the limits of the storage dump.

f

indicates the dump format. The format possibilities are the same as for the DUMP subprogram (see 7.7.3.1).

Description:

A dump of storage contents between *a* and *b*, inclusive, is made according to format *f* and program execution resumes.

Either *a* or *b* can be the upper or lower limit of storage. Both must be in the same program unit or the same common block.

Example:

```
        CALL PDUMP (LOWER, UPPER, 5)
  C           A dump is made of information between locations
  C           UPPER and LOWER.  This data is output in single-
  C           precision real format to the system output file and
  C           normal program execution then resumes.
```

7.7.3.3. DVCHK

Purpose:

The DVCHK subprogram tests for a previous divide-check exception.

Form:

```
CALL DVCHK (i)
```

where *i* is an integer variable or array element that indicates whether the divide-check indicator is turned on or off.

Description:

DVCHK tests the divide-check indicator to see if either a fixed- or floating-point divide-check has occurred.

The variable *i* is then set to 1 if the divide-check indicator is on, or it is set to 2 if the indicator is off.

After testing, the divide-check indicator is turned off.

See DIVSET (7.7.3.9).

Example:

```
C           An example of a divide exception is a fixed-point divide
C           exception.  This is recognized when the division of a
C           fixed-point number by zero is attempted.  Such an
C           exception occurs during execution of the following
C           statements:
      INTEGER DIVISR,DIVDND,TROUBL
      DIVISR = 0
      DIVDND = 8
      QUOTNT = DIVDND/DIVISR
 C          This turns the divide-check indicator on, so the
 C          subsequent statement:
      CALL DVCHK(TROUBL)
 C          sets variable TROUBL to 1 and turns the indicator off.
```

7.7.3.4. OVERFL

Purpose:

The OVERFL subprogram tests for a previous exponent overflow.

Form:

```
CALL OVERFL (i)
```

where *i* is an integer variable or array element.

Description:

OVERFL determines whether or not an exponent overflow has occurred since the last call to OVERFL/OVUNFL (or the start of the program, if OVERFL/OVUNFL has not been previously called). The parameter *i* is assigned a value that corresponds to the current overflow status of the program.

When an overflow condition occurs, the value returned in *i* is 1. An exponent overflow occurs when the absolute value of the result of a floating-point addition, subtraction, multiplication, or division is greater than or equal to 2^{127} (approximately 10^{38}) for single precision or 2^{1023} (approximately 10^{307}) for double precision.

When no overflow occurs, the value returned in *i* is 2.

After the value of *i* is set, the overflow indicator is cleared.

See OVFSET (7.7.3.8).

Example:

```
      A = 1.0E20
          .
          .
          .
      B = A*A
      CALL OVERFL (LAST1)
C           Assuming that A has not been changed, the value returned
C           in LAST1 is 1, indicating exponent overflow occurred.
```

7.7.3.5. UNDRFL

Purpose:

The UNDRFL subprogram tests for a previous exponent underflow.

Form:

```
CALL UNDRFL (i)
```

where *i* is an integer variable or array element.

Description:

UNDRFL determines whether or not an exponent underflow has occurred since the last call to UNDRFL/OVUNFL (or the start of the program, if UNDRFL/OVUNFL has not been previously called). The parameter *i* is assigned a value that corresponds to the current underflow status of the program.

When an underflow occurs, the value returned in *i* is 3. An underflow occurs whenever the absolute value of the result of a floating-point addition, subtraction, multiplication, or division is not equal to 0 and is less than 2^{-128} (approximately 10^{-38}) for single precision or 2^{-1024} (approximately 10^{-308}) for double precision.

When no underflow occurs, the value returned in *i* is 2.

After the value of *i* is set, the underflow indicator is cleared.

See UNDSET (7.7.3.7).

Example:

```
       A = 1.0E-20
         .
         .
         .
       B = A*A
       CALL UNDRFL (LAST1)
 C           Assuming that A has not been changed, the value returned
 C           in LAST1 is 3.
```

7.7.3.6. OVUNFL

Purpose:

The OVUNFL subprogram tests for a previous exponent overflow or underflow.

Form:

```
 CALL OVUNFL (i)
```

where *i* is an integer variable or array element.

Description:

OVUNFL determines whether or not an exponent overflow or underflow has occurred since the last call to OVUNFL/OVERFL/UNDRFL (or the start of the program, if OVUNFL/OVERFL/UNDRFL have not been previously called). The parameter *i* is assigned a value that corresponds to the current overflow/underflow status of the program.

When only an overflow condition occurs, the value returned in *i* is 1. An exponent overflow occurs when the absolute value of the result of a floating-point addition, subtraction, multiplication, or division is greater than or equal to 2^{127} (approximately 10^{38}) for single precision or 2^{1023} (approximately 10^{307}) for double precision.

When only an underflow condition occurs, the value in *i* is 3. An underflow occurs when the absolute value of the result of a floating-point addition, subtraction, multiplication, or division is not equal to 0 and is less than 2^{-128} (approximately 10^{-38}) for single precision or 2^{-1024} (approximately 10^{-308}) for double precision.

When both overflow and underflow occur, the value returned in *i* is 4.

When no overflow or underflow occurs, the value returned in *i* is 2.

After the value of *i* is set, the overflow/underflow indicators are cleared.

Example:

```
       A = 1.0E20
           .
           .
           .
```

```
      B = A*A
      CALL OVUNFL (LAST1)
C           Assuming that A has not been changed, the value returned
C           in LAST1 is 1, indicating only overflow has occurred.
```

7.7.3.7. UNDSET

Purpose:

The UNDSET subprogram flags subsequent floating-point underflow exceptions.

Form:

```
CALL UNDSET (i)
```

where *i* is an integer expression.

Description:

A call to this subprogram lets the next *i* floating-point underflow exceptions be captured and diagnosed. The following message is generated:

```
WARNING: UNDERFLOW FAULT
```

When the program is compiled with the F or C option, the line number and program unit name of the offending statement are printed. After *i* messages, no more are printed unless another call to UNDSET is made. An *i*≤0 stops the capture of these faults.

Checkout debug mode (CZ options) receives an initial automatic default call to UNDSET with a count of 20.

Example:

```
@FTN,SIC UND
FTN 10R1 04/01/81-12:44(,0)
1.     COMMON A,B
2.     PRINT *,'TEST THE UNDSET ROUTINE'
3.     A = 1.0E-20
4.     CALL UNDSET(2)
5.     B = A*A
6.     PRINT *,'ONCE'
7.     B = A*A
8.     PRINT *,'TWICE'
9.     B = A*A
10.    PRINT *,'THREE TIMES'
11.    END

END FTN 28 IBANK 45 DBANK 2 COMMON

   ENTERING USER PROGRAM
TEST THE UNDSET ROUTINE

WARNING: UNDERFLOW FAULT
AT LN.   5 OF MAIN PROGRAM
ONCE

WARNING: UNDERFLOW FAULT
```

```
AT LN.   7 OF MAIN PROGRAM
TWICE
THREE TIMES
END PROGRAM EXECUTION
```

7.7.3.8. OVFSET

Purpose:

The OVFSET subprogram flags subsequent floating-point overflow exceptions.

Form:

```
CALL OVFSET(i)
```

where *i* is an integer expression.

Description:

A call to the OVFSET subprogram captures the next *i* floating-point overflows. The message:

```
WARNING: OVERFLOW FAULT
```

is printed. In addition, when the program is compiled with the F or C option, the line number and program unit name of the offending statement are printed.

After *i* messages, the next overflow fault causes the program to abort, unless another call to OVFSET is done first.

A negative *i* causes an abort on the first occurrence.

An *i* of zero stops the capture of these faults. Checkout debug runs (CZ options) automatically get an initial default call to OVFSET with a count of 20.

Example:

```
@FTN,SIC OVF
FTN 10R1 04/01/81-13:24(,0)
1.    COMMON A,B
2.    PRINT *,'TEST THE OVFSET ROUTINE'
3.    A = 1.0E20
4.    CALL OVFSET(2)
5.    B = A*A
6.    PRINT *,'ONCE'
7.    B = A*A
8.    PRINT *,'TWICE'
9.    B = A*A
10.   PRINT *,'THREE TIMES'
11.   END

END FTN 28 IBANK 45 DBANK 2 COMMON


   ENTERING USER PROGRAM
TEST THE OVFSET ROUTINE
```

```
WARNING: OVERFLOW FAULT
AT LN.   5 OF MAIN PROGRAM
ONCE

WARNING: OVERFLOW FAULT
AT LN.   7 OF MAIN PROGRAM
TWICE

WARNING: OVERFLOW FAULT
ARITHMETIC EXCEPTION COUNT EXPIRED - PROGRAM ABORTED
AT LN.   9 OF MAIN PROGRAM
ERR MODE ERR-TYPE: 00 ERR-CODE:00
ERROR ADDRESS: 032073  BDI: 300017
```

7.7.3.9. DIVSET

Purpose:

The DIVSET subprogram flags subsequent divide fault exceptions.

Form:

```
CALL DIVSET(i)
```

where *i* is an integer expression.

Description:

A call to the DIVSET subprogram captures the next *i* divide checks. The message:

```
WARNING: DIVIDE FAULT
```

is printed. In addition, when the program is compiled with the F or C option, the line number and program unit name of the offending statement are printed.

After *i* messages, the next divide fault causes the program to abort, unless another call to DIVSET is done first.

A negative *i* causes an abort on the first occurrence.

An *i* of zero stops the capture of these faults.

Checkout debug runs (CZ options) automatically get an initial default call to DIVSET with a count of 20.

Example:

```
@FTN,SIC DIV
FTN 10R1 04/01/81-13:58(,0)
1.     COMMON Q, DD, DR
2.     PRINT *,'TEST THE DIVSET ROUTINE'
3.     DR = 0.
4.     DD = 8.
5.     CALL DIVSET(2)
```

```
6.    Q = DD/DR
7.    PRINT *,'ONCE'
8.    Q = DD/DR
9.    PRINT *,'TWICE'
10.   Q = DD/DR
11.   PRINT *,'THREE TIMES'
12.    END

END FTN 29 IBANK 45 DBANK 3 COMMON

   ENTERING USER PROGRAM
TEST THE DIVSET ROUTINE

WARNING: DIVIDE FAULT
AT LN.   6 OF MAIN PROGRAM
ONCE

WARNING: DIVIDE FAULT
AT LN.   8 OF MAIN PROGRAM
TWICE

WARNING: DIVIDE FAULT
ARITHMETIC EXCEPTION COUNT EXPIRED - PROGRAM ABORTED
AT LN.   10 OF MAIN PROGRAM
ERR MODE ERR-TYPE: 00  ERR-CODE: 00
ERROR ADDRESS: 032074 BDI: 300017
```

7.7.3.10. CMLSET

Purpose:

The CMLSET subprogram flags subsequent errors occurring in the Common Mathematical Library (CML). This includes the mathematical intrinsic functions described in NO TAG, all exponentiation functions, and complex division.

Form:

```
CALL CMLSET(i)
```

where *i* is an integer expression.

Description:

A call to the CMLSET subprogram diagnoses the next *i* CML errors. Control returns to the point following the CML function call. A function result of zero is returned from the CML function for the error case.

After *i* messages, the next CML error causes the program to abort, unless another call to CMLSET is done first.

A negative or zero *i* causes an abort on the first CML error.

Example:

```
@FTN,SC CML
FTN 10R1 10/21/80-12:50(6,)
      1.   C
```

```
       2.   C   This program allows 2 Common Mathematical Library
       3.   C   errors to occur without the program being aborted.
       4.   C   On the 3rd math error, the program is aborted.
       5.   C
       6.   C
       7.       REAL A(5)
       8.       DATA A/25., -25., -16., -4., 4./
       9.       CALL CMLSET(2)     @Allow 2 CML errors before abort
      10.   C
      11.       DO 10 I = 1,5
1     12.           R = SQRT (A(I))
1     13.           WRITE (6,901) A(I), R
1     14.  10   CONTINUE
1     15.  C
      16.  901  FORMAT ('SQRT OF', F5.1, 'IS', F12.3)
      17.       END

END FTN 23 IBANK 37 DBANK

   ENTERING USER PROGRAM
SQRT OF 25.0 IS 5.000

ERROR CONDITION IN SQRT ROUTINE CAUSED BY
ARGUMENT UNNORMALIZED OR OUTSIDE ALLOWABLE RANGE
ARG1 = -25.000000
ARG1 OCTAL 572157777777
SQRT REFERENCED AT ABSOLUTE ADDRESS 120304 BDI 300020
AT LN.   12 OF MAIN PROGRAM
SQRT OF -25.0 IS  .000

ERROR CONDITION IN SQRT ROUTINE CAUSED BY
ARGUMENT UNNORMALIZED OR OUTSIDE ALLOWABLE RANGE
ARG1 = -16.000000
ARG1 OCTAL 572377777777
SQRT REFERENCED AT ABSOLUTE ADDRESS 120304 BDI 300020
AT LN.  12 OF MAIN PROGRAM
SQRT OF -16.0 IS  .000

ERROR CONDITION IN SQRT ROUTINE CAUSED BY
ARGUMENT UNNORMALIZED OR OUTSIDE ALLOWABLE RANGE
ARG1 = -4.0000000
ARG1 OCTAL 574377777777
SQRT REFERENCED AT ABSOLUTE ADDRESS 120304 BDI 300020
THIS ADDRESS IS AT LN.   12 OF MAIN PROGRAM
ER EABT$ ABORT ADR: 104754 BDI: 200020
PROGRAM INITIATED INTERRUPT: EABT$
```

7.7.3.11. SSWTCH

Purpose:

The SSWTCH subprogram tests one of 12 run condition switches to determine whether it is set or not.

Form:

```
CALL SSWTCH (i , j)
```

where:

i

is an integer expression indicating which of the 12 switches is to be tested.

j

is the integer variable or array element, set equal to 1 or 2, when the tested switch is set or not set, respectively.

Description:

SSWTCH regards the middle 12 bits (13-24) of the run condition word as 12 sense switches. Bit 24 corresponds to sense switch 1, bit 23 to sense switch 2, etc. The sense switches can be altered by means of the @SETC Executive control statement (see the *OS 2200 Exec System Software Executive Requests Programming Reference Manual*, 7830 7899).

If i is 1 through 12, the i^{th} sense switch is tested. If the i^{th} sense switch is set (1), then j is set equal to 1. If it is not set, then j is set equal to 2.

If $i < 0$ or $i > 12$, an error message is printed and program execution terminates.

Examples:

```
       CALL SSWTCH(3,JFLAG)
C           Assuming the control statement @SETC 0004 is issued
C           before execution, the variable JFLAG is set to
C           1 indicating sense switch 3 (bit 22 in the run condition
C           word) is set.

       I = 12
       CALL SSWTCH(I,JFLAG)
C           Assuming the control statement @SETC 0717 is issued
C           before execution, the variable JFLAG is set to 2
C           indicating sense switch 12 (bit 13 in the run condition
C           word) isn't set.
```

7.7.3.12. SLITE

Purpose:

The SLITE subprogram sets one, or resets all, of the six sense lights that are the sixths of the word labeled F2SLT$.

Form:

```
CALL SLITE(i)
```

where *i* is an integer expression indicating which of the six sense lights is set.

Description:

You can use the word F2SLT$ to set flags for conditions of your invention. These flags can then be tested with the SLITET function (see 7.7.3.13).

If *i* is 0, all sense lights are reset to 0. If *i* is 1 through 6, the i^{th} sense light is set to 1.

If *i* < 0 or *i* > 6, an error message is printed and program execution terminates.

Examples:

```
      CALL SLITE(0)
C              All six sense lights are reset to 0.
      CALL SLITE(3)
C              The third sense light (S3 of F2SLT$) is set to 1.
```

7.7.3.13. SLITET

Purpose:

The SLITET subprogram tests and resets one of the six sense lights that are the sixths of the word labeled F2SLT$.

Form:

```
CALL SLITET (i , j)
```

where:

i

is an integer expression that indicates which of the six sense lights is to be tested.

j

is an integer variable or array element set equal to 1 or 2 when the tested light is set or reset, respectively.

Description:

If the value of *i* is from 1 through 6, the i^{th} sense light is tested. If the i^{th} sense light is set (value of 1), then *j* is set to 1 and the sense light is reset to 0. If the i^{th} sense light is not set (0), then *j* is set equal to 2.

If *i* < 0 or *i* > 6, an error message is printed and program execution terminates.

Example:

```
      CALL SLITET(3,JFLAG)
C              Assuming sense light 3 is set, JFLAG is set to 1
C              and sense light 3 is reset to 0.  Assuming
C              sense light 3 isn't set, JFLAG is set to 2.
```

7.7.3.14. EXIT

Purpose:

EXIT terminates execution.

Form:

```
CALL EXIT
```

Description:

The EXIT service subprogram lets you terminate the execution of your programs.

The EXIT subprogram serves the same purpose as the STOP statement (see 4.8). It is provided primarily for compatibility with previous FORTRAN systems.

7.7.3.15. ERTRAN

The ERTRAN subprogram provide access to several Executive Request (ER) functions. (For details on ER functions, see the *OS 2200 Exec System Software Executive Requests Programming Reference Manual*, 7830 7899).

7.7.3.15.1. Input/Output Executive Requests

Purpose:

ERTRAN lets you use some of the input/output Executive Request functions. These are:

- IO$ (requests an operation on an input/output device and returns immediately),
- IOW$ (requests an operation on an input/output device and returns on completion of the operation),
- IOI$ (identical to IO$ except that when the operation has completed, an interrupt activity is initiated),
- IOWI$ (combines the features of IOI$ and IOW$ - control is returned on completion of input/output and a specified interrupt activity is initiated),
- IOXI$ (identical to IOI$ except that the activity making the request exits),
- WAIT$ (delays execution until the input/output operation controlled by a specified I/O packet has been completed),
- WANY$ (delays execution until any current input/output operation is completed),
- TSWAP$ (closes the current reel for a tape file and requests loading of the next reel of the file),
- UNLCK$ (enables an input/output interrupt activity to reduce its switching priority to the priority of the activity that initiated the input/output request), and

- SYMB$ (a packet-driven ER providing a means to request certain symbiont ER functions, character transfer, and some expanded READ$ capability).

Form:

$$\text{CALL}\left\{\begin{array}{l} sub \\ sub_1\ (pkt) \\ sub_2\ (pkt\ [,label]) \\ sub_3\ (pkt,label_1,label_2) \end{array}\right\}$$

where:

sub

is FWANY or FUNLCK.

sub$_1$

is FIO, FIOI, FIOWI, FIOXI, FTSWAP, or FSYMB.

sub$_2$

is FIOWor FWST.

sub$_3$

is FSTAT.

pkt

is an 8-word or 10-word table filled by you, with input/output packet information.

label

is an optional error return.

label$_1$

identifies the return used if the input/output operation is still pending.

label$_2$

identifies the return used if an input/output operation terminates abnormally.

Description:

Two basic steps are necessary when using these routines:

1. Construct an Executive input/output packet (*pkt*) exactly as described in the *OS 2200 Exec System Software Executive Requests Programming Reference Manual*, 7830 7899. There must be one 8-word table declared for each file to be used simultaneously. For FSYMB, the table can be 8 or 10 words. The FIOP procedure (see 7.7.3.15.2) can be used to set fields in the packet.
2. Call the required I/O Executive Request function, passing the I/O packet as the first argument of the call.

The Executive Request is done with register A0 pointing to the program-constructed input/output packet. This provides you with full control over the input/output operation and complete error analysis capability.

There are no buffers external to these routines. They are all reentrant at both the program and the activity level.

This input/output mechanism is entirely independent of standard FORTRAN file control. Therefore, no file control tables are built by the FORTRAN system for FIO files.

If FIO is called, the parameter *pkt* that points to the input/output packet is picked up, the Executive Request function IO$ is referenced, and control returns immediately to the caller.

If FIOW is called, the parameter *pkt* that points to the input/output packet is picked up, the Executive Request function IOW$ is referenced and it waits for completion before returning to the caller. If the calling syntax does not include the optional error return, you should check the status area of the input/output packet by using the ISTAT statement function as provided in FIOP. If the calling syntax does include the error return option, then all abnormal statuses cause transfer to that label.

If FIOI is called, the parameter *pkt* that points to the input/output packet is picked up, the Executive Request function IOI$ is referenced, and control is returned immediately to the caller. On completion of the input/output operation, the interrupt activity specified in the packet is initiated.

If FIOWI is called, the parameter *pkt* that points to the input/output packet is picked up and the Executive Request function IOWI$ is referenced. On completion of the operation, control returns to the caller and a specified interrupt activity is initiated.

If FIOXI is called, the parameter *pkt* that points to the input/output packet is picked up, the Executive Request function IOXI$ is referenced, and the calling activity is terminated. On completion of the input/output operation, a specified interrupt activity is initiated.

If FWST is called, the parameter *pkt* that points to the input/output packet is picked up, the Executive Request function WAIT$ is referenced, and a wait is done if input/output is outstanding on the packet. On completion, a status check is made. In the case of an abnormal completion status, a return is done to *label*, if present. If the calling syntax doesn't include the error return option, control returns to the instruction following the CALL.

If FWANY is called, the Executive Request function WANY$ is referenced and a wait is done for completion of any input/output. On completion, control is returned to the caller.

If FTSWAP is called, the parameter *pkt* that points to the input/output packet is picked up and the Executive Request function TSWAP$ is referenced.

If FUNLCK is called, the Executive Request function UNLCK$ is referenced to enable an input/output interrupt activity that reduces its switching priority to the priority of the activity which initiated the input/output request. Control then returns to the caller.

If FSYMB is called, the parameter *pkt* pointing to an 8- or 10-word packet is picked up, the Executive Request function SYMB$ is referenced and control is returned to the caller.

An additional routine, which does not refer to an Executive Request, is FSTAT. If FSTAT is called, the parameter *pkt* that points to the input/output packet is picked up and a check is done on the status of the input/output operation. If the input/output operation is still pending, a return is made to *label*$_1$. If the input/output operation terminated abnormally, a return is made to *label*$_2$. If the input/output operation terminated normally, a normal return is taken.

Examples:

```
      SUBROUTINE FASTIO(BUF,NADDR)
C           This subroutine reads 112 words from track NADDR
C           into the array BUF.
      LOGICAL INIT/.FALSE./
C           Declare 8-word table for packet.
      INTEGER IOT(8)
C           Include the FORTRAN procedure in the program.
C           File name may have to be changed depending on
C           site conventions.
      INCLUDE SYS$LIB$*FTN.FIOP
C           Initialize first two words with file name in Fieldata,
C           words 3 through 8 to 0.
C           A unit specifier could have been used.
      DATA IOT(1), IOT(2), (IOT(I), I = 3, 8) /'MYFILE'F,'  'F,6*0/
      IF (INIT) GO TO 110
      INIT = .TRUE.
C           Assign the file.
      STAT=FACSF2('@ASG,A MYFILE . ')
C           Check that the file assignment is OK.
      IF (STAT.LT.0) CALL FABORT
      NWRDS(IOT) = 112
C            112 words to be read.
      FUNCT(IOT) = FR
110   CONTINUE
      BUFAD(IOT) = LOC(BUF)
C           Set file track address for input.
      TRKAD(IOT) = NADDR
C            Call to do input/output
      CALL FIOW(IOT,*500)
      RETURN
C            Following is error handler
500   CONTINUE
      PRINT *,'DID NOT WORK'
      CALL FEXIT
      RETURN
      END
```

The following program shows one way to use FSYMB:

```
      PROGRAM AWRITE
C           This program writes 132 characters from BUFFER through
C           SYMB$ if the printer allows 132 characters
      CHARACTER*204 BUFFER
      INTEGER PACKET(10)
C           Include the statement functions and parameters to use
C           SYMB$
      INCLUDE FIOP
C           Set the file name to the PRINT$ ER code
      PACKET(1)=14
      PACKET(2)=0
C           Set the function code to PRINT$
      INFUNC(PACKET)=FW
C           Set the mode to ASCII
      IMODE(PACKET)=IASC
C           Set the number-of-characters transferred
      ICHCT(PACKET)=132
C           Set the image address
      IIMGAD(PACKET)=LOC(BUFFER)
C           Clear the remainder of the SYMB$ packet
      DO 10 I-6,10,1
10       PACKET(I)=0
C           Initialize the 132-character BUFFER
      BUFFER='TEST BUFFER LENGTH OF 132 CHARACTERS'
      BUFFER(111:132)='END OF 132 CHARACTERS'
      BUFFER(194:204)='END OF LINE'
C           Call SYMB$
      CALL FSYMB(PACKET)
      STOP
      END
```

7.7.3.15.2. FIOP Procedure

Purpose:

FIOP is a FORTRAN procedure that facilitates creation and manipulation of the packets for input/output Executive Requests. The FIOP procedure is included in the ASCII FORTRAN library file. This procedure contains statement function and PARAMETER declarations as detailed below.

Form:

```
INCLUDE libfile.FIOP
```

where *libfile* is the file name of the FORTRAN run-time library file. The file name is site-dependent. See 9.5.3 for more information on finding the element FIOP.

Description:

The information required in the Executive Request packet is placed in the various fields by use of statement function and PARAMETER statements in the FORTRAN procedure FIOP. The statement functions and symbolic names of constants provided by FIOP are declared as type integer. This FORTRAN procedure must be included with the FORTRAN source program (see 8.2). There are statement function names for all of the fields in the Executive Request packet except the first two words, which contain the Fieldata internal file name or unit specifier left-justified and space-filled. You can initialize these two words with the DATA statement using the Fieldata representation of either the internal file name or the unit specifier. This is most efficient since it is done at compile time. However, an alternate method is to use the FASCFD run-time conversion routine (see 7.7.3.18).

The following list of statement function names represents the various fields of the Executive Request packet, as described in the *OS 2200 Exec System Software Executive Requests Programming Reference Manual*, 7830 7899. Each is contained in FIOP. Each statement function reference requires one argument that is the input/output packet name.

Statement Functions of the Executive Request Packet contained in FIOP:

ACTID

int-act-id (numeric identity that identifies the interrupt activity)

ACTADR

interrupt-activity-addrs (interrupt activity address)

ISTAT

status (status of the last function performed)

FUNCT

function (denotes function to be performed)

AFC

AFC (abnormal frame count for tape I/O)

SUBST

final-word-count-returned-by-I/O

GINC

G (incrementation flag)

NWRDS

word-count (number of words transferred)

BUFAD

buffer-addr (memory address for transfer - use LOC intrinsic function)

TRKAD

mass-storage-addr (relative track/sector address for start of I/O)

SRCHS

search-sentinel

SFDA

search-find-drum-addr (search find address)

The following list of statement function names represents the various fields of the Executive Request packet for SYMB$. Each statement function reference requires one argument, which is the input/output packet name.

Statement Functions of the Executive Request Packet for SYMB$:

IFUNC

function (describes the action performed on the specified file)

IMODE

mode (provides further description of the function)

ISTAT

status (status of last function performed)

IERCD

I/O-status (I/O error code if SYMB$ is terminated because of an I/O error)

ICCIN

control-card-index (CLIST index for an image read)

ISUBST

sub-status returned (further information status)

ICHCT

character-count (number of characters to transfer)

IIMGAD

image-address (address to put/get an image)

ITTNF

TT-number-field (which translate table to use)

ICHSZ

character-size-field (6-bit or 9-bit byte)

IFCHCT

final-character-count-transferred (number of words or characters transferred)

ISPCE

spacing (number of lines to space before an image is written)

IEOFS

EOF-sentinel (Fieldata or ASCII character returned in column 6)

IWRCC

character-count (number of characters to transfer)

IWRAD

image-address (address of an image)

IWRFC

final-character-count-transferred (number of characters or words transferred)

The following is a list of names from PARAMETER statements declared in FIOP that define the values for the I/O functions for the FUNCT field. For more information, see the *OS 2200 Exec System Software Executive Requests Programming Reference Manual*, 7830 7899.

Parameter	Octal	Function
FW	10	Write
FWEF	11	Write end of file
FCW	12	Contingency write
FSW	13	Skip write
FABW	14	TIP recoverable write
FGW	15	Gather write
FACQ	16	Acquire
FABSW	17	Absolute write whole unit/extended acquire
FR	20	Read
FRB	21	Read backward
FRR	22	Read and release
FREL	23	Release
FBRD	24	Block read drum
FRDL	25	Read and lock
FUNL	26	Unlock
FABR	27	Absolute read, TIP recoverable read
FTSA	30	Track search all words
FTSF	31	Track search first word
FPSA	32	Position search all words
FPSF	33	Position search first word

continued

Parameter	Octal	Function
FSD	34	Search drum
FBSD	35	Block search drum
FSRD	36	Search read drum
FBSRD	37	Block search read drum
FREW	40	Rewind
FREWI	41	Rewind with interlock
FSM	42	Set mode
FSCR	43	Scatter read
FSCRB	44	Scatter read backward
FWR	45	Write, then read
FABSR	47	Absolute read whole unit
FMF	50	Move forward
FMB	51	Move backward
FFSF	52	Forward space file
FBSF	53	Backspace file
FCN	55	Mode set

The following is a list of names from PARAMETER statements declared in FIOP defining the values for the mode field for SYMB$:

Parameter	Octal	Description
IASC	1	ASCII request
ITRUN	2	Truncate an image
IUNTR	4	Untranslate request
IPUALT	10	Punch alternate file
ISPEC	20	Special translation index

The following is a list of the names declared by statement function statements in FIOP of the fields for the set mode function (FORTRAN PARAMETER - FSM):

Field Name	Bits
FSMF1	34 - 35
FSMF2	32 - 33
FSMF3	30 - 31
FSMF4	28 - 29
FSMF5	26 - 27
FSMF6	22 - 25
FSMF7	20 - 21
FSMF8	18 - 19
FSMF9	0 - 17

The fields of the set mode function are set as described in the *OS 2200 Exec System Software Executive Requests Programming Reference Manual*, 7830 7899.

7.7.3.15.3. Miscellaneous Executive Requests

Purpose:

These Executive Requests let you refer to each of the Executive Request functions:

- ABORT$ (abort run)
- ACSF$ (generate control statement)
- ERR$ (error exit)
- EXIT$ (program exit)
- SETC$ (set condition word)
- COND$ (retrieve condition word)
- DATE$ (request date and time)

Form:

$$\texttt{CALL}\left\{\begin{array}{l} r_1 \\ r_2\ (arg) \\ r_3\ (arg1,\ arg2) \end{array}\right\}$$

where:

r1

is FABORT, FERR, or FEXIT.

r2

is FACSF, FACSF2, FSETC, or FCOND.

r3

is FDATE or ADATE.

arg

is a character array or a character expression for FACSF and FACSF2, an integer expression for FSETC, or an integer variable or array element for FCOND.

arg1, *arg2*

are variables or array elements.

Description:

If FABORT is called, the Executive Request function ABORT$ is referred to. All current activities are terminated and the run is terminated in an abort condition immediately; files are not closed.

If FERR is called, the Executive Request function ERR$ is referred to. Only the activity in error is terminated.

If FEXIT is called, the Executive Request function EXIT$ is referred to. The routine provides program termination. No file-closing actions are performed if FEXIT is called.

If FACSF is called, the Executive Request function ACSF$ is referred to. The routine submits an Executive control statement image (arg) for interpretation and processing. The image submitted must be a character array or a character expression containing one of the control statements in the list that follows. The control statement must not be longer than 84 characters and must be terminated by the character sequence: blank, period, blank.

Valid control statements for arg are:

Statement	Use
@ADD	Add to runstream
@ASG	Assign a file
@BRKPT	Breakpoint symbiont output files
@CAT	Catalog a file
@CKPT	Produce checkpoint dump of this run
@FREE	Deassign a file

continued

Statement	Use
@LOG	Message to the master log file
@MODE	Set mode and/or noise constant for tape file
@QUAL	File qualification
@RSTRT	Restart run whose checkpoint dump is saved by @CKPT
@START	Schedule an independent run
@SYM	Queue files for symbiont processing
@USE	Associate internal to external file name

The control statement image should not exceed 84 characters. If the control statement image doesn't contain the sequence of blank, period, blank before 84 characters, only 84 characters are used as the length of the control statement image. Extraneous characters following the actual control statement within the 84 characters used as the control statement image can cause an Exec error.

When the control statement image is less than 84 characters and does not contain the sequence of blank, period, blank before the end of the image, a total of 84 characters becomes the length of the control statement image. This causes inclusion of unknown characters in the actual control statement. This can cause an Exec error on the ER to ACSF$. It can also cause a guard mode when the control statement image is at the end of a bank where 84 characters can't be used.

If FACSF is called as a function, FACSF must be typed as integer, and, if a few special statuses are returned from the ACSF$ call, those statuses are returned as the value of the function. The FACSF function call does not return all nonzero statuses. (See the *OS 2200 Exec System Software Executive Requests Programming Reference Manual*, 7830 7899). A subprogram name (for example, FACSF) can be used only in one way in a given program. Thus, FACSF can't be used in a CALL statement and as a function name in the same program.

FACSF2 is the same as FACSF, except that no ERTRAN error message is printed if an error status is returned from the ACSF$ Executive Request. You must perform your own error checking (using the status returned as the FACSF2 function value) and processing.

If FSETC is called, the Executive Request function SETC$ is referred to. The subroutine places (sets) the contents of the lower third (bits 25-36) of *arg* in the corresponding third of the run condition word. The lower two-thirds of the run condition word are used as a flag that can be tested by the control statement @TEST or retrieved by the FORTRAN call CALL FCOND(a) and then tested.

If FCOND is called, the Executive Request function COND$ is referred to. The subroutine retrieves the condition word and makes it available to you in *arg*.

If FDATE is called, the Executive Request function DATE$ is referred to. The subroutine supplies you with the current date and time in *arg1* and *arg2*, respectively. The data in *arg1* is the Fieldata character form MMDDYY where MM represents the month (01-12), DD the day (01-31), and YY the last two digits of the year (00-99). The time in *arg2* is in the Fieldata character form HHMMSS, where HH represents the hours (00-24), MM the minutes (00-60), and SS the seconds (00-60). The first words of *arg1* and *arg2* (which are both variables or array elements) are filled with the 6-bit Fieldata characters described previously.

If ADATE is called, the Executive Request function DATE$ is also referred to as in FDATE, but the date and time are returned in ASCII character form. Variables *arg1* and *arg2* must be character variables or character array elements of eight characters in length. The first six character positions of each variable are filled with the ASCII character form of the date and time in the format described for FDATE above. The remaining two characters are space-filled.

Examples:

```
      INTEGER FACSF2
      CHARACTER ASG*20, DATE*8, TIME*8
      DATA ASG /'@ASG,A FILENAM . '/
      CALL FACSF(ASG)
C           This call would attempt to assign the file FILENAM by
C           referring to the Executive request function ACSF$.
      ISTAT = FACSF2(ASG)
C           This function reference attempts the assignment and
C           puts the result status in ISTAT.
      IF(ISTAT .LT. 0) CALL FERR  @ terminate on error
      CALL ADATE(DATE, TIME)
C           This call supplies you with the date ('MMDDYY ') in DATE
C           and the time ('HHMMSS ') in TIME.
      PRINT *, DATE, TIME
      END
```

7.7.3.16. NTRAN$

Purpose:

The NTRAN$ service subprogram provides a tool for reading or writing binary information on tape or mass storage, and also provides for I/O buffering. NTRAN$ I/O processing is completely separated from normal FORTRAN I/O processing. NTRAN$ I/O is accessible only by calls to the NTRAN$ service subprogram, and normal FORTRAN I/O is accessible only by FORTRAN I/O statements such as READ, WRITE, BACKSPACE, ENDFILE, OPEN, CLOSE, and INQUIRE. An NTRAN$-created file can't be referred to by normal FORTRAN I/O statements.

The ASCII FORTRAN NTRAN$ service subprogram has exactly the same syntax as the FORTRAN V NTRAN subprogram (see the *FORTRAN V Library Reference*, UP-7876). The only difference is that NTRAN in the FORTRAN V call is replaced by NTRAN$ in the ASCII FORTRAN call.

Form:

```
CALL NTRAN$(unit,sequence-of-operations)
```

where *unit* is an integer expression designating the logical unit, and the sequence-of-operations is any list of I/O operations (as specified in 7.7.3.16.1) performed in order on the specified unit.

The I/O operations provided by NTRAN$ are as follows:

Argument Group	Description
1.	Write (tape or mass storage)
2.	Read (tape or mass storage)
3.	Block read (tape or mass storage)
4.	Search read (tape or mass storage)
5.	Search mass storage
6.	Position mass storage
7.	Position tape by block (tape)
8.	Position tape by files (tape)
9.	Write end of file (tape)
10.	Rewind (tape or mass storage)
11.	Rewind/interlock (same as rewind for mass storage)
12.	Set tape density medium (tape)
13.	Set tape density low (tape)
14.	Set tape parity odd (tape)
15.	Set tape parity even (tape)
16.	Initialize multireel file (tape)
17.	Swap reels for multireel file (tape)
18.	Deassign unit (tape or mass storage)
19.	Assign unit to external file name (tape or mass storage)
20.	NOP (tape or mass storage)
21.	Get device

continued

Argument Group	Description
22.	Wait and unstack then release unit (tape or mass storage)
23.	Set tape density high (tape)

Description:

If the unit is not busy, NTRAN$ initiates the first operation, stacks the rest in a waiting list, and then returns to the calling program. If the unit is already busy, then the entire sequence is stacked in a waiting list and chained to any previously stacked operations. The exceptions are operations 16 to 22; when they are encountered, NTRAN$ waits for the completion of all previous operations for that unit before returning to the calling program. When an interrupt occurs, NTRAN$ records the transmission status, initiates the next operation in the chain, and returns control to the interrupted calling program. Priority tasks aren't supported.

To use NTRAN$, a FORTRAN program must have some way to check the status of the transmission. For this reason, every block of main storage used for I/O operations has a block status word (an integer variable) associated with it; the name of the status word is specified in the argument list of the CALL statement.

When NTRAN$ is called, the list of arguments is searched for status words, and these are all set to a value (-1) that indicates transmission isn't complete. When an interrupt occurs, the corresponding status word is set by NTRAN$ to a value that indicates the nature of completion, whether normal (a positive value indicating the number of words transmitted), abnormal (value = -2), or in error (value = -3 or -4). The status words for each operation are defined in 7.7.3.16.1.

When NTRAN$ generates -2 or -3, it releases all operations stacked for the unit that have not started. The offending operation is marked to abort and remains stacked. Any further calls of NTRAN$, requesting the above-described unit (except operation 22) aren't performed or stacked, but generate a particular status code (-4). Operation 22 can release the abort condition for a unit. This lets you regain control after trying to read or write past an end of file, end of drum, or end of tape.

An attempt to read or write zero words (n=0) results in the function being ignored.

The following errors generate a status word of -3:

- Hardware errors
- Parity and character count errors
- Illegal unit specified

Note: *Legal units are all tapes and mass storage files. An NTRAN$ read beyond the end of an NTRAN$ mass storage file generates a status word of -3.*

At compile time, the ASCII FORTRAN compiler prints out a warning each time NTRAN$ is called from the same program unit with a different number of arguments than was specified on the first call in the program unit. You can ignore these warnings.

7.7.3.16.1. Operations

An operation is defined in the argument list by a group of arguments. The first argument for an operation identifies the type of operation. It is followed by the parameters for the operation; these are fixed in number and order of occurrence by the type of operation. Several operations can be grouped in a single call to NTRAN$.

When referring to a mass storage file, the current mass storage address for that file is the starting address for the file only if the mass storage file was never referred to in the current run. If the mass storage file was referred to before, the current mass storage address is the current address before the last CALL statement using the file plus the number of words transmitted or positioned in that CALL statement. To reach the starting address of the file, you can use operations 10 and 22.

For example, in CALL NTRAN$ (3, 9, 10, 22):

3 = unit number

9 = end of file when operation is completed

10 = rewind unit

22 = all operations on unit must be completed before another function is issued

The example above is a stacked operation.

Note: *For sector-formatted mass storage I/O, the specified mass storage address is a sector count and not a word count as for word-addressable mass storage I/O. However, with normal termination, the status variable associated with a main storage transfer indicates the actual number of words transmitted. It is up to you to perform the covered divide with the sector size to retrieve the corresponding sector count.*

For search operations on sector-formatted mass storage, if a find is made, the mass storage address points to the sector containing the matching item; a following read function does not necessarily start reading the matched item.

1. Write

   The argument group is: 1,*n*,*b*,*l*

   in which *n* is an integer expression that specifies the number of words to write. *b* is a variable or array element from which data is written. *l* is an integer variable or integer array element; it identifies the status word, which is set by NTRAN$ as follows:

   -1 = transmission not complete

   -2 = end of the tape or mass storage file

   -3 = device error

-4 = transmission aborted (previous operation had -2 or -3 status)

If the transmission is completed normally, *l* receives the number of words transmitted (*n*). After a normal write operation to a sector-formatted mass storage file, NTRAN$ positions the file to the next sector (28 word) address.

2. Read

   The argument group is: 2,*n*,*b*,*l*

   in which *n* is an integer expression that specifies the number of words to read. *b* is a variable or array element into which the data is read. *l* is an integer variable or integer array element; it identifies the status word, which is set by NTRAN$ as follows:

   -1 = transmission not complete

   -2 = end of file (no words read from mass storage)

   -3 = device error or end of NTRAN$ mass storage file

   -4 = transmission aborted (previous operation had -2 or -3 status)

   If the transmission terminates normally, *l* receives the number of words transmitted (*n* if reading a mass storage file, a maximum of *n* if tape). After a normal read operation from a sector-formatted mass storage file, NTRAN$ positions the file to the next sector (28 word) address.

   For tape files, each write operation writes a block of words to tape. When the number of words *n*, specified on the read operation, is greater than the block length written to the tape, only the number of words in the block is read. The tape is then positioned at the start of the next block. When the number of words *n*, specified on the read operation, is less than the block length written to the tape, *n* words are read, and the tape is then positioned at the start of the next block.

   For mass storage files, the number of words read is *n*. It it not dependent on the number of words written.

3. Block read

   The argument group is: 3,*n*,*b*,*l*

   A block read for tape and sector-formatted mass storage is the same as an ordinary read. For word-addressable mass storage, transmission is terminated by reading a word of all 1 bits (called end-of-block word). *n* is the maximum number of words that can be transmitted. *l* is an integer variable, the status word, which receives the actual number of words transmitted if the operation is completed normally; otherwise *l* is set as in Read. *b* has the same definition as in Read.

4. Search read

   The argument group is: 4,*s*,*n*,*b*,*l*

   in which *s*, a sentinel word, is a constant or a variable used to search tape or mass storage.

   For tape, the first word of each block is compared to the sentinel and, when a match is found, that block (including the sentinel word) is read. For mass storage, starting at the current mass storage address, each word is compared to the sentinel until a match is found or until all remaining words of the granule (track or position) are tested. An unsuccessful search results in status (-2) for *l*. If no find is made, you can request additional searches by setting the mass storage address of a different granule (track or position). For sector-formatted mass storage, a track search is employed; if no find is made, you can request additional searches.

The search-read operation in NTRAN$ reads a certain number of words after a match is found. When a match is found on word-addressable mass storage, the block of *n* number of words beginning with the matched word is read into the buffer *b*. When a match is found on sector-formatted mass storage, *n* number of words from the beginning of the sector containing the matched sentinel is read into buffer *b*.

5. Search mass storage

   The argument group is: 5,*s*

   where *s* is a constant or variable sentinel word. Starting at the current mass storage address, each word is compared to the sentinel until a match is found or until all remaining words of the granule (track or position) are tested. The mass storage address of the match becomes the new current mass storage address (the first mass storage address to be read or written is that of the matched mass storage address). When a match is found on a sector-formatted mass storage device, the mass storage address points to the sector containing the matched sentinel. If a match is not made, the address doesn't change.

6. Position mass storage

   The argument group is: 6,*n*

   where *n* is a positive or negative integer constant or variable that is added to the current mass storage address to form a new current mass storage address. If *n* is negative and the current mass storage address plus *n* is less than the starting address of the mass storage file, the current mass storage address is set to the starting address of the mass storage file. *n* is the word count for word-addressable mass storage and the sector count for sector-formatted mass storage.

7. Position tape by blocks

   The argument group is: 7,*n*

   where *n* is an integer constant or variable that specifies the number of blocks to space over on tape; positive *n* indicates forward spacing and negative *n* indicates backspacing.

8. Position tape by files

   The argument group is: 8,*n*

   where *n* is an integer constant or variable that specifies the number of file marks to space over; positive *n* indicates forward spacing and negative *n* indicates backspacing. The operation is terminated by moving over the *n*th file mark, by reaching the load point (backspacing), or by reaching the end of tape (forward spacing).

9. End file

   The argument group is: 9

   For tape, an end-of-file mark is written.

10. Rewind

The argument group is: 10

11. Rewind/interlock

    The argument group is: 11

    For tape, a rewind/interlock is given. For word-addressable mass storage and sector-formatted mass storage, the operation is the same as a rewind.

Note: *The following four operations pertain to magnetic tape density and parity setting (available only on UNISERVO 7-track tape units). If not specified, the setting is system standard.*

12. Set tape density medium (556 bpi)

    The argument group is: 12

13. Set tape density low (200 bpi)

    The argument group is: 13

14. Set tape parity odd (binary standard)

    The argument group is: 14

15. Set tape parity even (BCD standard)

    The argument group is: 15

Note: *Density and parity setting routines set density and parity for all tape units tied to a logical unit during multireel processing.*

16. Initialize multireel file

    The argument group is: 16

    This operation reinitializes the cycle of tape swapping when making more than one pass over a multireel file.

17. Swap reels

    The argument group is: 17

    This operation accesses the next physical unit in a multireel file. The old physical unit is not rewound. Use a sequence of operation 17's to assign any number of physical units to a given unit.

18. Deassign unit

    The argument group is: 18

    This operation frees the unit using an @FREE,B init statement. The internal name (unit) associated with the file is released. If this is the only internal name attached, the external file name is also freed.

19. Assign unit to external file name

    The argument group is: 19,*x*,*l*

    This operation assigns a temporary file to an external file name when the file is not assigned. It also links an internal name to a file that is already assigned. The array *x* contains a two-word file name, which is left-justified in Fieldata.

Integer variable *l* is set as follows:

0 = assignment made

1 = error in call statement, no assignment made

20. NOP for compatibility (old function dropped)

    The argument group is: 20,*l*

21. Retrieve device

    The argument group is: 21,*x*

    This operation retrieves the device code for the specified unit. The code is stored in the integer variable *x*.

22. Wait and unstack

    The argument group is: 22

    This operation causes a wait in NTRAN$ until all previous operations for the specified logical unit are complete before stacking any further operations or returning to the user's program. It also removes any operation that causes an abnormal or error status that is still stacked against the unit specified.

23. Set tape density high (800 bpi - not compatible)

    The argument group is: 23

7.7.3.16.2. Using the Virtual Feature with NTRAN$

Banked and virtual arguments can be passed to NTRAN$. This can cause problems when a program is running asynchronously. When the buffer to be filled is a banked or virtual argument, a synchronous Executive Request (ER) is performed for the I/O operation. This ensures continuous control of the banks containing the buffer until the buffer is filled. The buffer length can now be over 65K words and, in fact, the buffer can be a large area in virtual space that spans several banks. NTRAN$ handles bank spanning that occurs in a virtual buffer automatically. Therefore, you can request an I/O through NTRAN$ on a buffer that is several million words long.

When the buffer on a write operation to a tape is a virtual argument that spans a bank, the data written by this operation must be read into the same buffer on the read operation of a tape, with the length returned by the write operation. When this is not possible, the write operation buffer and the read operation buffer must both start at the beginning of a virtual page (or have the same offset within a virtual page), have the same page size; the read operation must also use the size returned by the write operation. When the block length n spans virtual banks or is greater than 65K on a write operation to tape, several blocks are written to the tape. This means that read, move, or search operations don't know how many blocks to span for one record.

7.7.3.16.3. NTRAN$ Error Messages

NTRAN$ produces an error message under certain error conditions. The unit number is identified in the message. The FORTRAN program and the line number where the call to NTRAN$ is made are identified under the error message.

The possible error messages produced by NTRAN$ are:

1. **NTRAN ERROR* UNIT *n*: NO PACKET SPACE AVAILABLE.

   This message indicates that all available NTRAN$ packets are in use and that another packet is requested.

   Suggested Action: Reassemble the ASCII FORTRAN library element F2NTRAN$ and increase the number-of-packets parameter (NPKTS).

2. **NTRAN ERROR* UNIT *n* IS NOT AVAILABLE FOR NTRAN.

   A reference is made to a unit already in use by normal FORTRAN I/O processing.

   Suggested Action: Change unit number.

3. **NTRAN ERROR* UNIT *n* NOT ASSIGNED.

   A reference to an unassigned unit is made with a function other than write function 1 (see 7.7.3.16) or assign function 19.

   Suggested Action: If a write function 1 is used as the first reference, a dynamic assign of a mass storage file (scratch) would be made. If a scratch file is not intended, an assignment must be made either by assign function 19 or by an @ASG control image.

4. **NTRAN ERROR* UNIT *n* HAS IMPROPER DEVICE.

   Requested function is not available for the device assigned to this unit. The requested function is ignored.

   Suggested Action: If you want action for the requested function, a unit with another device assigned must be used.

5. **NTRAN ERROR* UNIT *n* HAS ILLEGAL FUNCTION CODE.

6. **NTRAN ERROR* UNIT *n*: NUMBER OF ARGUMENTS IN STACK EXCEEDS TABLE LENGTH.

   This message indicates that the number of arguments in the call is greater than the maximum calling sequence table length.

   Suggested Action: Reassemble ASCII FORTRAN library element F2NTRAN$ and increase the NCT length (NCTLT).

7. **NTRAN ERROR* UNIT *n*: SYNTAX ERROR FOR FILE NAME.

   This message indicates an illegal character in the file name for NTRAN$ function 19 (see 7.7.3.16).

Note: *You must not change any argument of an argument group before the function is completed, that is, before the status word (if any) is changed from -1 to another value. All NTRAN$ functions are executed in sequence; the completion of one function implies completion, successful or unsuccessful, of all preceding functions.*

7.7.3.17. CLOSE

Purpose:

The CLOSE subprogram closes a FORTRAN data file and frees all main storage associated with the file such as the file control table and buffer areas.

Form:

```
CALL CLOSE (i,j)
```

where *i* and *j* are integer expressions.

Description:

This subroutine closes the file associated with the unit identifier designated by the first argument.

The file is rewound when the second argument is nonzero (for tape only).

Examples:

```
      I = 3
      J = 0
      CALL CLOSE (I,J)
C             The file associated with unit identifier 3 is closed
C             and no rewind of the tape file occurs.
```

7.7.3.18. FASCFD and FFDASC

Purpose:

The FASCFD and FFDASC subprograms provide a FORTRAN interface to the FDASC conversion routine. FDASC allows conversion from ASCII to Fieldata and from Fieldata to ASCII.

Form:

```
CALL r (i, a, b)
```

where:

r

is FASCFD or FFDASC.

i

is a positive integer variable or array element specifying the number of words to be converted. On return from the FASCFD/FFDASC call, *i* contains the length in words of the converted string. You should not use an integer constant for this argument.

a

is the source (an expression) that is converted.

b

is the target (variable or array element) for the converted characters.

Description:

ASCII/Fieldata characters are converted as described in the *Series 1100 System Service Routines Library (SYSLIB)*, *Programmer Reference*, UP-8728 (applicable version). These routines are not intended to convert trailing blanks in a string. Trailing blanks in the source string may or may not be included in the word count of the converted string.

In the following descriptions, ASCII characters are assumed to be packed four characters per word and Fieldata characters are assumed to be packed six characters per word.

If FASCFD is called, the 4*i* ASCII characters in *a* are converted to Fieldata characters and stored in *b*. If 4*i*/6 has a remainder *R* other than zero, then 6-*R* Fieldata spaces follow the last character in *b*, space-filling the last word. However, if this causes the last word of *b* to be all spaces, the word count returned in *i* does not include this word.

If FFDASC is called, the 6*i* Fieldata characters in *a* are converted to ASCII characters and stored in *b*. If 6*i*/4 has a remainder of *R* other than zero, then 4-*R* ASCII spaces follow the last character in *b*, space-filling the last word. If this causes the last word of *b* to be all spaces, the word count returned in *i* does not include this word.

For either call, if *a* and *b* are the same character variable, then the source characters are destroyed.

For ASCII FORTRAN level 10R1A and lower, when the source is too long, the source is truncated and no warning message is issued. The source and target items must be in the same D-bank.

For ASCII FORTRAN level 11R1 and higher, the source and target items do not have to be in the same D-bank. When the source and target items are not in the same bank, the source is moved to a buffer where the conversion is completed. The contents of this buffer are moved to the target item. The source item must not exceed 131 words when a move of the source item to the buffer is done via subprogram FASCFD. The source item length must not exceed 87 words when the source item moves to a buffer in subprogram FFDASC. A warning message occurs when the source item length exceeds these maximum lengths and only the maximum length is moved and converted.

When the source item does not begin on a word boundary, it is moved to a buffer where it begins on a word boundary. The source item length is limited to 131 words for FASCFD and 87 words for FFDASC. A warning message occurs when the source length exceeds the maximum length. Only the maximum length is moved.

Example:

```
      IWC = 13
      CALL FASCFD (IWC,ASC,FD)
C            The first 52 ASCII characters in ASC are converted to
C            Fieldata and stored in FD.  On return, IWC equals 9.
C            If the last four characters (49-52) of ASC are spaces,
C            then IWC is set to 8 on return.
```

The following call to FASCFD can result in a serious error:

```
CALL  FASCFD(3,ASC,FD)
```

Although a constant is passed, the value of the constant changes on return from FASCFD. Since the compiler assumes that the values of constants never change and reuses them throughout a compilation, logic errors can easily result. This can happen any time a constant is passed as an argument to a routine that changes the value of the corresponding dummy argument.

7.7.3.19. MAXAD$

Purpose:

The MAXAD$ subprogram changes one or more items in the Common Storage Management System (CSMS) packet and returns a status. CSMS is used mainly for I/O buffers.

Form:

```
CALL MAXAD$ (max, mcore, lcore)
```

where:

max

4 is the maximum address that the program can reach (that is, CSMS never requests main storage past this address). If *max* is -1 or 0, the default 0777774 (decimal value 262,140) is assumed.

mcore

4 is the request main storage increment size (that is, the minimum amount of main storage obtained each time an ER MCORE$ is done). If *mcore* is -1, the default 010000 (decimal value 4,096) is assumed. If *mcore* is 0, no requests for main storage are made.

lcore

4 is the minimum amount of freed storage that remains in the allocated area after an ER LCORE$ is performed by the CSMS (to release storage that has been freed). If *lcore* is -1, the default 020000 (decimal value 8,192) is assumed. If *lcore* is 0, no release of main storage is done. It is recommended that if *lcore* is specified and is

nonzero, it should be at least twice as large as the request for main storage increment size *mcore*.

Description:

The MAXAD$ service subprogram can't be called from checkout mode, as checkout doesn't use the CSMS feature.

If any of the three parameters is passed as -2, the MAXAD$ subprogram doesn't change the corresponding packet location.

This function returns 0 or -1, which indicates good or bad status, respectively. You can test this return status when MAXAD$ is referenced as a function (rather than as a subroutine). A 0 status indicates no errors. A bad status (-1) indicates that one of the following has occurred:

- One of the parameters was greater than 0777774 (decimal value 262,140). The parameter is ignored.
- The current maximum address in the packet (that is, the current end of the control bank) was greater than *max* (first parameter). In this case, the absolute maximum address in the packet (that is, the address that the program never exceeds) is set to the current maximum address.

If the request for main storage increment size in the packet is already zero when this routine is entered, then no action is performed. This is because an initial reserve is probably specified in F2FCA (see G.7), and the CSMS routines use that main storage with no ER MCORE$ or ER LCORE$. A good status (0) is returned in this case.

Use caution when specifying mcore = 0, or when specifying *max* as anything but the default. CSMS terminates in error in certain cases.

7.7.3.20. LOC

Purpose:

The LOC integer function returns the absolute address of its argument.

Form:

```
LOC(name)
```

where *name* is the argument whose address is returned as the function value.

Description:

LOC should be typed as integer in the calling routine if an IMPLICIT statement types the letter L as noninteger, since LOC is in the service subroutine class and doesn't have an inherent type.

See Appendix M for a description of LOCV$, which returns the virtual address of the argument.

7.7.3.21. MCORF$ and LCORF$

Purpose:

The MCORF$ and LCORF$ service subprograms give a primitive entry into the ASCII FORTRAN storage allocator, so that dynamic pseudo-arrays can be allocated and released.

Form:

```
iadr=MCORF$(isize)        @ Function call

CALL LCORF$(iadr)
```

where:

iadr

is the address of the buffer to be allocated or freed (*iadr* is returned as the corresponding MCORF$ function result).

isize

is the number of words you want in the dynamic pseudo-array (passed to function MCORF$, which returns an address *iadr*).

Description:

Since it is difficult to use an address in the FORTRAN language, base-offset type referencing must be done to use the address returned by MCORF$.

The pseudo-arrays created by this process do not have all of the attributes of a normal FORTRAN array. For example, the pseudo-array names (D1 and D2 in the first example below) with no following subscripts cannot be passed to a subprogram or to I/O (since they are actually statement functions that require an argument). However, individual array elements of the pseudo-arrays can appear anywhere.

Examples:

The following example uses a dummy array, DUMMY, and statement functions to create two dynamic pseudo-arrays, D1 and D2, which are both single-precision real.

```
        DIMENSION DUMMY(1)
        DEFINE D1( I ) = DUMMY( I + ID1OFF )
        DEFINE D2( I ) = DUMMY( I + ID2OFF )
        ID1OFF = 2000  @ Make D1 2000 elements
        ID2OFF = 3000  @ Make D2 3000 elements
        CALL GET( DUMMY, ID1OFF )
        CALL GET( DUMMY, ID2OFF )
C
C            Initialize arrays
```

```
C
        DO 10 J = 1, 2000
10        D1( J ) = 0.
        DO 20 J = 1, 3000
20        D2( J ) = 0.
C
C           Now use D1 and D2 arrays.
C
              .
              .
              .
C
C           Now free D1 and D2. Note that ID1OFF and ID2OFF
C           were never changed in the above example.
       CALL FREE ( DUMMY, ID1OFF )
       CALL FREE ( DUMMY, ID2OFF )
         END

       SUBROUTINE GET ( DUM, ISZ )
       ISZ = MCORF$ ( ISZ ) - LOC ( DUM )
       END

       SUBROUTINE FREE ( DUM, ISZ )
       CALL LCORF$ ( ISZ + LOC ( DUM ))
       ISZ = 0    @ Preventive medicine
       END
```

The following example is similar to the preceding example, except that it creates a pseudo-array, ADU(1000), with two words per element (in other words, double-precision real).

```
      PARAMETER (IMAXX = 1000)            @ Dimension of pseudo-array
      DOUBLE PRECISION AD(1), ADU
      INTEGER ADI
      DEFINE ADU(IX) = AD(IX+ADI)
      ADI = IMAXX                         @ Cleared by free routine
      CALL GET2(AD,ADI)                   @ Get storage for array ADU
         .
         .   USE ARRAY ADU
         .
      CALL FREE2(AD, ADI)                 @ Free storage for ADU
      END

      SUBROUTINE GET2(D,IDIM)
C
      DOUBLE PRECISION D
C             For 2 words/element array
C
      COMMON /STRG2$/INUMD,ISVDA(40),ISVDX(40)
      DATA INUMD/0/
      IA=MCORF$(IDIM*2+3)
      IDIM=(IA-LOC(D))/2 + 1
      IX = IDIM*2+LOC(D)                 @ Approx addr
      IF (INUMD.GE.40) STOP 'GET2 MAX'
      INUMD = INUMD+1
      ISVDA(INUMD)=IA                    @ Actual addr obtained
      ISVDX(INUMD)=IX                    @ Approx addr
      END

      SUBROUTINE FREE2(D,IDIM)
      DOUBLE PRECISION D
      COMMON /STRG2$/INUMD, ISVDA(40),ISVDX(40)
C
```

```
C            First calculate the approx. address, then
C            search for the actual address
       IX=IDIM*2+LOC(D)
       DO 10 I=1,INUMD
       IF(ISVDX(I).EQ.IX) THEN
            CALL LCORF$(ISVDA(I))
            IDIM = 0
            INUMD=INUMD-1
            IF(I.EQ.INUMD+1)RETURN
            IF(INUMD.EQ.0)RETURN
            ISVDA(I)=ISVDA(INUMD+1)
            ISVDX(I)=ISVDX(INUMD+1)
            RETURN
       ENDIF
10     CONTINUE
       STOP 'FREE2 NO-FIND'
       END
```

The following example creates a CHARACTER*3 pseudo-array, A3U (1000).

```
       PARAMETER (IMAXX = 1000)
       CHARACTER*3 A3(1), A3U
       INTEGER A3I
       DEFINE A3U(IX) = A3(IX+A3I)
       ADI=IMAXX                  @ Cleared by free routine
       CALL GETC3 (A3,A3I)        @ Get storage for array A3U
        .
        .   USE A3U
        .
       CALL FREEC3(A3,A3I)        @ Free storage for A3U
       END

      SUBROUTINE GETC3(D,IDIM)
      CHARACTER*4 D
C           Get storage for a three char/elt array.
      COMMON /STRG$/INUM3,ISV3A(40),ISV3X(40)
      DATA INUM3/0/
      IN=MCORF$(IDIM*3/4+2)
      IDIM=(IA-LOC(D))*4/3 + 1     @ Element separation
      IX = 3*IDIM/4 + LOC(D)       @ Approx addr
      IF (INUM3.GE.40) STOP 'GETC3 MAX'
      INUM3=INUM3+1
      ISV3A(INUM3)=IA
      ISV3X(INUM3)=IX              @ Approx addr
      END

      SUBROUTINE FREEC3(D,IDIM)
      CHARACTER*4 D
      COMMON /STRG$/INUM3,ISV3A(40),ISV3X(40)
      IX=3*IDIM/4+ LOC(D)          @ Approx addr
      DO 10 I=1,40
      IF (ISV3X(I).EQ.IX) THEN
          CALL LCORF$(ISV3A(I))
          IDIM = 0
          INUM3=INUM3-1
          IF (I.EQ.INUM3+1.OR.INUM3.EQ.0)RETURN
          ISV3A(I)=ISV3A(INUM3+1)
          ISV3X(I)=ISV3X(INUM3+1)
          RETURN
      ENDIF
10    CONTINUE
      STOP 'FREEC3 NO-FIND'
      END
```

When dynamic pseudo-arrays have a different element size, you must specify a separate GET and FREE routine for each element size.

7.7.3.22. F2DYN$

Purpose:

This service subprogram lets you have a measure of dynamic storage allocation for dummy arrays.

Form:

```
CALL {F2DYN$|FFDYM$} (sub, isize 1,...,isize n-1)
```

where:

sub

is the subprogram name.

isizei

is an integer expression denoting array size in words ($1 \leq i \leq n$-1).

Description:

In the ASCII FORTRAN I/O complex, the storage management routines can be accessed via subprogram F2DYN$. It has *n* arguments. The first argument is the name of an ASCII FORTRAN subprogram to call. The second through *n*th arguments are the sizes (in words) of *n*-1 arrays that are passed to this subprogram. The service subprogram acquires core via the I/O complex, creates *n*-1 dynamic arrays, and passes them on to the subprogram. The subprogram can dimension them dynamically on entry. The acquired storage is released on the return. The F2DYN$ subprogram can be called recursively.

In a program with several subprograms (such as in the following example), the D-bank savings can be substantial because there is much less dead static storage as a result of a smaller amount of dynamic storage being allocated at any one time.

An alternate entry point FFDYN$ is provided, so the allocation mechanism can be used for calling both subroutines and functions from the same program unit. The compiler issues warnings (that can be ignored for this case) when either F2DYN$ or FFDYN$ is called from the same program unit with a differing number of arguments.

Example:

Subprogram REDUCE needs some local temporary arrays to do its computations:

```
SUBROUTINE REDUCE(M,MD)
REAL M(MD) @ MD range is 10 to 1000
DIMENSION M2(1000),M3(1000),M4(1000)
REAL M2
DOUBLE PRECISION M3
COMPLEX*16 M4
```

```
              .
              .
              .
            END
```

It can be changed to:

```
            SUBROUTINE REDUCE(M,MD)
            REAL M(MD) @ MD range is 10 to 1000
            COMMON /SIZES/I
            EXTERNAL REDINT
            I=MD
            CALL F2DYN$(REDINT,MD,MD*2,MD*4)
      C
      C  The internal subroutine does the processing now.
      C
            SUBROUTINE REDINT(M2,M3,M4)
            DIMENSION M2(I),M3(I),M4(I)
            REAL M2
            DOUBLE PRECISION M3
            COMPLEX*16 M4
              .
              .
              .
            END
```

7.7.3.23. IOFLG$

Purpose:

The IOFLG$ service subprogram prevents the checking for a type mismatch of an I/O list item and a format edit descriptor during a formatted I/O statement.

Form:

```
  CALL IOFLG$ (a)
```

where:

a

is an integer expression indicating the action taken during format-directed input and output statements when a type-match check of an I/O list item and a format edit descriptor occurs.

Description:

When the value of *a* is greater than 0, the check is not done to determine if the type of the I/O list item is legal for the format edit descriptor specified. A warning is not printed and the input/output status specifier (IOSTAT = *ios*) is not set to an error condition. See 5.2.8.

When the value of *a* is 0, the check on a match of the type of the I/O list item and the format edit descriptor is made. A warning appears whenever a mismatch occurs and the ERR clause or the IOSTAT clause is not present. When the ERR clause is present, control transfers to the statement specified by that clause. When the IOSTAT clause is present, the value of the I/O status specifier is set to a specified value and a warning is not issued. See 5.2.8.

When only the IOSTAT clause is present, the result of the type check on I/O list items and edit descriptors is similar. That is, a warning message is not printed when a mismatch occurs. The only difference between the use of these two methods of turning off warning messages is in setting the I/O status specifier. The default is to perform the type check of I/O list items and edit descriptors.

7.8. Non-FORTRAN Procedures

Because FORTRAN is a high-level language oriented to the solution of certain types of problems, it is sometimes expedient to code some parts of a program in FORTRAN and other parts in another language.

These non-FORTRAN parts can be included in the FORTRAN program unit when the program is collected. Program parts written in assembly language (that is, MASM) must be prepared to accept the calling sequences generated by the ASCII FORTRAN compiler. Parts written in FORTRAN V, ASCII COBOL, or PL/I can be called directly (see EXTERNAL statement [6.9]).

See Appendix K for details on the interfaces with FORTRAN V, ASCII COBOL, PL/I, and MASM.

7.9. Local and Global Names

Entities that can be referred to from both external and internal program units are called global. Entities that can be referred to only within a particular internal subprogram are called local, that is, local to that subprogram.

Entities used in an external program unit cannot be referred to by another external program unit unless they are in common blocks or passed as arguments. Variables, arrays, statement functions, parameter constants, and so forth, which are declared or used in an external program unit, can be referred to by any of their internal subprograms. Internal subprograms can’t reference each other’s data except through common blocks and arguments.

External program units can contain any number of internal subprograms. An internal subprogram can call another internal subprogram if both are in the same external program unit. BLOCK DATA subprograms can’t be internal and can’t contain internal subprograms.

Any data declarations encountered in an internal subprogram create variables local to that internal subprogram. The following statements create local names when in an internal subprogram:

- **Explicit typing statements (such as REAL, INTEGER)**
- **DIMENSION (the array names)**

- **COMMON (the variable names)**
- **BANK (the bank names)**
- **NAMELIST (the namelist name)**
- **PARAMETER (the name being defined)**
- **EXTERNAL**
- **INTRINSIC**
- **DEFINE and statement function definitions (the statement function name)**
- **EQUIVALENCE**
- **SAVE**
- **DATA (symbolic names not defined in a common block)**
- **Dummy arguments in FUNCTION, SUBROUTINE, and ENTRY statements**

For rules on IMPLICIT type associations in internal subprograms, see 6.3.1.

A COMMON statement in an internal subprogram redefines (in other words, is not a continuation of) the same common block declared in the external program unit.

In addition, all statements create local variables for names encountered that don't exist in their external program unit.

Names that are used in an internal subprogram, but are not defined locally in the aforementioned statements, come from the external program unit environment if they exist there. If they are local to the internal subprogram, their declarations must occur before they are referred to.

Example 1:

```
INTEGER P
DIMENSION X(5)
PARAMETER (P=10)
CALL INT(X)
SUBROUTINE INT(Y)
DIMENSION Y(P)   @ uses 10
PARAMETER (P=5)  @ does not help Y
    .
    .
    .
```

Example 2:

```
A=10.               @ global A
CALL INT
SUBROUTINE INT
NAMELIST/LIST/A,B   @ global A, local B
REAL A              @ local A
B,A=3.              @ local A
WRITE(6,LIST)       @ write global A, local B
END
```

Statement labels are local to the program unit in which they appear.

An internal subprogram can't be referred to by any external program unit outside its program unit group (a program unit group is composed of the external program unit and any internal subprograms contained in the external program unit).

However, if the name of an internal subprogram is passed as an argument to an external program unit outside its program unit group, the external program unit can, in general, refer to the internal subprogram by referring to the dummy argument name. There are two exceptions to referring to an internal subprogram through an argument name:

- In a program compiled with the automatic storage stack feature (COMPILER statement options DATA=AUTO or DATA=REUSE), an external program unit can refer to internal subprograms only from its program unit group. This restriction is required because stack storage is acquired for all parts of a program unit group at one time upon entry to the external program unit.
- In multibanked programs, an internal subprogram cannot be called from a program unit that is not in the same I-bank as the internal subprogram. This restriction is required because internal subprograms do not use an LIJ for returns; they always jump back.

When an intrinsic function name such as SIN is used as an internal function, it must be placed in an EXTERNAL statement in its external program unit. This is a special use of the EXTERNAL statement to indicate that the intrinsic function is not desired. In this case, all references to the function SIN in the external program unit or in any of its internal subprograms refer to the internal subprogram SIN.

Section 8
Program Control Statements

8.1. General

The program control statements are used to temporarily modify the source program being compiled, modify the code generated, and control the compiler listing produced. **The statements (all nonstandard) are:**

- **INCLUDE**
- **DELETE**
- **EDIT**
- **COMPILER**

Use the INCLUDE statement to insert additional statements into the compilation and the DELETE statement to delete source statements from the compilation. The EDIT statement dynamically controls the compiler listing. Use the COMPILER statement to alter the compilation process.

8.2. INCLUDE Statement

Purpose:

The INCLUDE statement inserts an externally defined set of ASCII FORTRAN statements into the program being compiled.

Form:

```
INCLUDE [f. ]n [ , LIST ]
```

where:

f

if specified, is a file name specifying the file to be searched for the procedure (PROC) *n*.

n

is the name of a FORTRAN procedure (PROC) created by the Procedure Definition Processor (PDP). See the *Procedure Definition Processor (PDP)*, *Programmer Reference*, UP-10070, for details on PDP. The name *n* contains from 1 to 12 characters from the same character set as a FORTRAN symbolic name.

LIST

when specified, causes the included statements to be listed whenever the source program is listed.

Description:

The included statements don't become part of the updated source program produced. Therefore, the physical line numbers of the source program don't change.

INCLUDE statements can't be nested; that is, an INCLUDE statement must not be among a group of included statements; any other FORTRAN statement can be included.

One would normally use a single INCLUDE statement to represent a block of FORTRAN source statements. For example, one such application would be to include a set of statement functions that are frequently used at a particular installation.

A useful application for INCLUDE procedures is to use one or more INCLUDE elements containing a set of data declarations that are shared among different portions of a larger user program system. These data declarations can be, for example, many of the COMMON, DIMENSION, EQUIVALENCE, and PARAMETER statements, as well as statement function definitions. You should be cautious, however, when using these FORTRAN procedures. All of the statements must be processed, slowing down the compilation. An example of poor usage is a small subroutine including a procedure (PROC) that defines several hundred variables, but uses only one of these definitions.

8.2.1. Creating FORTRAN Procedures (PROCs)

Use the Procedure Definition Processor (PDP) to create a FORTRAN procedure (PROC). A PROC is a set of FORTRAN statements that can be inserted into source language with an INCLUDE statement.

PDP accepts source language statements defining FORTRAN procedures and builds an element in the user-defined program file. Using INCLUDE, these procedures can be subsequently referenced in a compilation without redefinition.

A FORTRAN procedure has the following format (see Example 8-1):

1. The PROC line is the first line of a FORTRAN procedure. It contains the procedure name (1 to 12 characters), starting in column 1, and PROC starting in column 7 or after.
2. The procedure images follow the PROC line. These images are the FORTRAN statements that are to be included.
3. The final line must be the word END, appearing in columns 2 through 4.

A PDP source element can contain more than one FORTRAN procedure, as in Example 8-1.

A FORTRAN procedure is not analogous to an Assembler procedure. For a FORTRAN procedure, the F option must be in the @PDP command.

The sample procedures in Example 8-1 (SPECS1, SPECS2, and SPECS3) can be included (using the INCLUDE statement) in a FORTRAN program.

```
@PDP,FIL STUFF
PDP12R1 R72-16 02/09/77 09:18:37 (,0)      RI


 PE0001   SPECS1 PROC
  0002         IMPLICIT INTEGER (O-Q)
  0003         PARAMETER (IN=5, OUT=6)
  0004         PARAMETER (P1=1,P2=2,P3=3)
  0005         PARAMETER (M=MAX(MOD(P3,P2),DIM(P3,P1)))
  0006         COMPLEX C(10),D(5)/5*(1,0,-1.0)/
  0007         DIMENSION O(5),P(4)
  0008         COMMON /CB1/C,K,O  / /A(5),Q
  0009
  0010   9999  FORMAT( )
  0011         EXTERNAL FUNEX
  0012    END
 PE0013   SPECS2 PROC
  0014         COMPLEX X(5)
  0015         COMMON E,F,G
  0016    END
 PE0017   SPECS3 PROC
  0018         COMPLEX C
  0019         COMMON /CB1/ C(5),K,O
  0020      END
```

Example 8-1. Sample PROC

Example:

```
  .
  .
  .
 INCLUDE SPECS1
  .
  .
  .
 INCLUDE SPECS3,LIST
  .
  .
  .
```

For PDP to add the procedure names to the FORTRAN procedure table of the specified program file, one of three conditions must be satisfied:

1. The I option must be specified and an element name given in the *spec1* field of the PDP control card; or

2. The U option must be specified and an element name given in the *spec1* field of the PDP control card; or
3. Neither the I nor the U option is specified and element names are given in the *spec1* and *spec2* fields of the PDP control card.

8.2.2. Specifying the Procedure File Name

When you specify a file name in the INCLUDE statement, the FORTRAN procedure table of that file is searched for the PROC *n*. The file name can be a fully specified name (qualifier, file, cycle, keys) or a @USE attached name. The usual Executive file name dropout rules apply. If the file has a read key, it must be specified in the INCLUDE statement or in an @ASG statement prior to the compilation, or the Executive may abort the compilation.

If a file name is not given in the INCLUDE statement, the PDP element that contains the procedure (PROC) to be included must be in a file that is assigned to the run in which the compilation is being performed.

The rules for determining which files are searched and the order in which the search is performed are as follows:

1. You have the option of specifying the file that is searched first. You do this by assigning the mass storage file with the @USE attached name FTN$PF to the run and by specifying the M option on the @FTN statement for the compilations in which the optional file search is desired.

   If the file FTN$PF is assigned to the run and the M option is specified, FTN$PF is the first file searched for the PROC to be included. If FTN$PF is not assigned to the run or the M option isn't specified or the PROC isn't found in FTN$PF, the search continues as specified below.
2. If the source input is coming from a program file on mass storage, that file is searched.
3. If the source input is coming from a program file on tape, the relocatable binary output file is searched.
4. If the source input is coming from cards, the source output file is searched, if one exists. Otherwise, the relocatable binary output file is searched.
5. If the PROC isn't found in any of the files previously listed, the system library files SYS$LIB$*PROC$ and SYS$*RLIB$ are searched.
6. If the PROC isn't found in any of the files previously listed, an error message is printed. Compilation continues as if the INCLUDE statement is not present.

8.3. DELETE Statement

Purpose:

The DELETE statement prevents compilation of a set of source lines contained in the input source program.

Form:

```
DELETE n [ , [d] [ ,l] ]
```

where:

n

is a statement label in the same program unit as the DELETE statement.

d

is the delete parameter. It must be an integer constant or symbolic name of an integer constant.

l

is the list parameter. It must be an integer constant or symbolic name of an integer constant.

Description:

When the DELETE statement is used, the compiler ignores all source lines following the DELETE statement up to, but not including, the statement labeled *n*. However, if the value of the delete parameter, *d*, is 0, the DELETE statement is ignored.

If the value of the list parameter, *l*, is nonzero, the deleted source lines are listed in the compilation listing. If the value of *l* is 0, the deleted lines are not listed.

The lines deleted are made transparent to the compiler, but they are not actually removed from the updated source program.

Several simplifications of the form of the DELETE statement are allowed:

- If the delete parameter is 1 (*d*=1) and the list parameter is 0 (*l*=0), you can use the following reduced form:

  DELETE *n*

- If the deleted lines are never to be listed (*l*=0), you can use the following form:

  DELETE *n*,*d*

- If the images are always to be deleted (*d*=1), and the list option is to be specified, you can use the following form:

  DELETE *n*, ,*l*

Any statement (including a specification statement) can have a label (see 9.3.1), so DELETE *n* can refer to any statement in the program unit.

8.4. EDIT Statement

Purpose:

The EDIT statement is used to start or stop the compiler listing produced for any portion of the source program.

Forms:

```
START EDIT PAGE
START EDIT SOURCE
START EDIT CODE
STOP EDIT SOURCE
STOP EDIT CODE
```

Description:

The word PAGE causes a page eject in the source listing, the word SOURCE controls the listing of the source program statements, and the word CODE controls the listing of the generated object code.

The START forms cause the compiler to initiate the type of listing specified (SOURCE or CODE) if it has been suppressed by a preceding EDIT statement or by a processor call option. A START EDIT CODE statement initiates both source and code listings.

The STOP forms cause the compiler to terminate the type of listing specified if it is initiated by a preceding EDIT statement or by a processor call option. A STOP EDIT SOURCE statement terminates both source and code listings.

The START EDIT PAGE form causes the current page of the compiler listing to eject before listing the next statement of the source program. This has the effect of placing the next statement following this command at the top of a new page. If the source is not currently being printed, this statement has no effect on the output.

8.5. COMPILER Statement

Purpose:

The COMPILER statement allows you to place certain code-generation or storage-related compiler directives within a program's structure.

Form:

```
COMPILER (op) [ , (op) ] ...
```

where *op* is a compilation option. All options are of the form *a*=*b*, where *a* is the option name and *b* is the option type. The allowable options are:

- DATA = AUTO
- DATA = REUSE

- PARMINIT = INLINE
- STACK = KEEP
- STACKADR = KEEP
- STACK = LIST
- BANKED = RETURN
- BANKED = DUMARG
- BANKED = ACTARG
- BANKED = ALL
- BANKACT = NOTEST
- BANKACT = CALL
- LINK = IBJ$
- STD = 66
- U1110 = OPT
- ARGCHK = ON
- ARGCHK = OFF
- PROGRAM = BIG
- PAGESIZE = nK
- NBRPAGES = n
- VIRTUAL = NCCB
- VIRTUAL = STATIC

Description:

When you use the COMPILER statement, it affects the code generated for the entire program unit or compilation, and it should be one of the first statements in the program unit or compilation.

The BANKED = RETURN option is local to the program unit in which it appears. All other options are global to the compilation and should appear near the beginning of the first program unit in the source.

8.5.1. DATA, PARMINIT, and STACK Options

DATA=AUTO and DATA=REUSE

These COMPILER statement options cause the ASCII FORTRAN compiler to generate code that has no local D-bank (other than common blocks), to acquire and release space for local use by calls to run-time routines, and to dynamically initialize this space on entry, thus giving ASCII FORTRAN an automatic storage facility.

The DATA=AUTO and DATA=REUSE options also cause the LINK=IBJ$ option to be turned on. The DATA=REUSE option is identical to the DATA=AUTO option, except that a run-time prologue entry point is used that does not initially push the stack before allocating local storage. This means that the stack storage of the previous routine is reused. This conserves stack space, but since the stack of its caller is destroyed, a RETURN can never be done from a subprogram with a DATA=REUSE option. You have to somehow handle program termination yourself. Since arguments are passed in the stack, arguments cannot be passed to routines that have the DATA=REUSE option; that is, they must not have any dummy arguments.

The compiler generates the code and data for initializing the stack frame under location counters $(5) and $(7) (see 6.13.2). If I-bank size is a problem in programs that use the DATA=AUTO option, $(5) and $(7) for some or all elements may be collected into a separate bank. (Be sure to keep $(5) and $(7) for a given element in the same bank.)

Because of the highly volatile nature of this automatic storage, some side effects of static storage allocation are not available when using automatic storage:

- Local variables not initialized by DATA statements don't have the value of 0 on entry.
- Local variables don't have their last value on reentry to a subprogram unless they appear in a SAVE statement. A SAVE statement forces them to static storage.
- Arguments set by entering at one entry point are not available when the subprogram is entered at another.
- If one reads into a Hollerith or literal field of a FORMAT statement, it will hold that value only as long as the subprogram containing it is active.
- Names of internal subprograms can't be passed to other external subprograms and have the external call the internal.

Full-debug checkout and FTNPMD (see 10.3 and 10.4) use is severely restricted when using automatic storage because:

- Walkback terminates when a subprogram using automatic storage is encountered.
- No local variables or arguments can be set or dumped for subprograms that use automatic storage.

PARMINIT=INLINE

This COMPILER statement option has effect only when used with the DATA=AUTO or DATA=REUSE option. When the DATA=AUTO or DATA=REUSE option is specified, a considerable amount of initialization code is required for the automatic storage stack. Items such as I/O packets, format lists, argument lists, and data lists require instructions to initialize the list information onto the automatic storage stack. This initialization code is normally executed for the entire program unit upon entry to the main program or subprogram. That is, all of the stack initialization for the entire program unit is executed at program unit initialization before executing any of the statements of the program unit. This is inefficient if only a small part of the program unit is executed each time.

The PARMINIT=INLINE option can be used to make certain FORTRAN programs execute more efficiently. This option causes the initialization of argument lists to take place at their reference, rather than at program unit initialization time. This is more efficient for program units that execute only a portion of their statements each time called and do not loop heavily around subprogram calls. However, if the program unit has many loops, with subprogram calls in the loops, using this option can take more execution time instead of less.

STACK=KEEP

This COMPILER statement option is used with a DATA=AUTO or DATA=REUSE automatic storage option. It specifies that the acquired D-bank stack frame is not released. The stack frame is acquired and initialized on the first entry to the subprogram. However, it is not released on subprogram exit. Subsequent entries to the subprogram skip the acquisition and initialization of the stack frame, resulting in a significant performance gain over the standard automatic storage options. Use of this option is valuable to a program composed of many modules where only a small number of the modules are entered on any given execution. In this situation, the collected size of the control D-bank is minimal, but there is no performance degradation due to initialization code.

STACKADR=KEEP

This COMPILER statement option is an alternate for the STACK=KEEP option. The D-bank stack frame is acquired and initialized on the first entry to the subprogram, and is not released on subprogram exit, just as with the STACK=KEEP option. However, the method used internally to keep track of which stack frame belongs to which subprogram is different. This alternate method may result in better performance than the STACK=KEEP method, and will work better in segmented programs, but the “cost” of this alternate method is at least one static local D-bank word per subprogram. (“Static” in this context means it is not allocated on the DATA=AUTO stack.)

STACK=LIST

This COMPILER statement option specifies that the stack frame sizes for each external program unit in a compilation appear in a short listing just before the compiler sign-off line. This option only affects programs using automatic storage, that is, programs using the DATA=AUTO or DATA=REUSE options of the COMPILER statement or programs using the virtual feature. This option is assumed when you specify an L, Y, or D option on the ASCII FORTRAN processor call control statement.

8.5.2. BANKED Options

The ASCII FORTRAN banking mechanism is described in H.2.

BANKED=ALL

This COMPILER statement option is a general banking option used to indicate that banking of some form appears in your program.

The BANKED=ALL option indicates that all labeled common blocks can be in paged data banks or in the control bank. In addition, BANKED=ALL turns on the following COMPILER statement banking options:

LINK=IBJ$ (subprograms can be banked)

BANKED=DUMARG (input arguments can be banked)

When BANKED=ALL appears, the specific subprogram (I-bank) and common block (D-bank) bank structure of the user program need not be known at compile time. This makes the process of generating a multibank user program more flexible, since changes in the bank structure can be made entirely at collection time, with no ASCII FORTRAN recompilations required.

When you have paged data banks and BANKED=ALL is specified, BANK statements (see 6.6) naming common blocks can be completely omitted from ASCII FORTRAN source programs, although they can appear for reasons of efficiency. See H.2.3 for a description of banking efficiency.

BANKED=DUMARG

This COMPILER statement option applies to dummy arguments in subprograms. (A dummy argument is an item appearing in argument lists in SUBROUTINE, FUNCTION, or ENTRY statements.) When this option appears, dummy arguments that are data items can be in paged data banks or in the control bank, and dummy arguments that are subprograms can be in any I-bank or in the control bank.

On every reference to a dummy argument in a subprogram, the compiler must generate code to base the item's bank, if it isn't already based.

The BANKED=DUMARG feature is automatically turned on if BANKED=ALL is specified.

BANKED=RETURN

When this COMPILER statement option appears in a subprogram, the subprogram returns to its calling program via one of two instructions: LIJ or a jump. The decision is made in generated code with a test sequence: an LIJ is executed if the subprogram is entered via an LIJ, and a jump is executed if the subprogram was entered via an LMJ.

Use this feature when the subprogram's I-bank may be different from the I-bank of any calling program.

The BANKED=RETURN feature is automatically turned on if LINK=IBJ$ or BANKED=ALL is specified.

BANKED=ACTARG

BANKED=ACTARG is an obsolete option that is supplied for compatibility with previous levels of ASCII FORTRAN. It was previously used to pass banking information for arguments on subprogram calls, but this information is automatically passed now (when you specify any form of banking, for example, BANKED=ALL).

8.5.3. BANKACT Options

The following COMPILER statement options change the type of code generated to activate a D-bank to reference your variable in that D-bank. These can be D-banks used for virtual space as well as D-banks used for the existing banking setup. Code to activate that D-bank consists of an inline LBJ instruction, or a link to a run-time routine that does the LBJ. (The link instruction can be an SLJ or LMJ.) A test sequence sometimes takes place to test around the activation instruction. A test sequence speeds up execution when the bank is already based.

BANKACT=NOTEST

The compiler generates a test around a D-bank activation instruction unless there is a good chance that a bank-switch from the last reference can occur. This option tells the compiler not to generate a test sequence, but rather to generate a direct D-bank activation sequence. Only use this option on FORTRAN elements when the compiler makes the wrong decision on TEST/NOTEST sequences, and performance suffers enough to warrant intervention. The default lets the compiler select a TEST/NOTEST sequence based on previous references, but the selection is not predictable.

Note: *A test sequence generation references an array dummy argument when the VIRTUAL statement is present.*

BANKACT=CALL

The compiler sometimes generates an inline LBJ instruction to activate a D-bank or to link to a run-time activation routine by an LMJ or SLJ. This option ensures use of an out-of-line routine in element F2ACTIV$. You can insert statistical code in element F2ACTIV$ (at tag VSTATS$) to determine reference patterns on D-banks and detect the source of performance problems. There is no default because the compiler can generate several code sequences to reference your virtual or banked variables. Therefore, this option simply ensures that an out-of-line sequence is picked.

8.5.4. LINK=IBJ$ Option

The LINK=IBJ$ COMPILER statement option makes it easier to construct multi-I-bank programs without using the BANKED=RETURN COMPILER statement option or BANK statements. This option is meaningful for banked subprograms, although it can also be used on nonbanked programs. The LINK=IBJ$ option has no effect on paged data banks.

See H.2 for a description of the ASCII FORTRAN banking mechanism. The LINK=IBJ$ feature is automatically turned on if BANKED=ALL is specified.

The LINK=IBJ$ option causes the compiler to generate the following reference to your subprograms:

```
LXI,U    X11, BDICALL$+subprg
IBJ$     X11,subprg
```

BDICALL$ and IBJ$ are special external reference items that are resolved at collection time; *subprg* represents the name of a FORTRAN external subprogram. The collector determines if *subprg* is contained in a different instruction bank than the one from which the call is made. If it is, then BDICALL$+*subprg* is the BDI of the bank in which *subprg* is located, and IBJ$ is an LIJ. If *subprg* is in the same instruction bank, BDICALL$+*subprg* is zero, and IBJ$ is an LMJ. The LINK=IBJ$ option also causes the compiler to generate the following instructions to return from subprograms:

```
LA,H1     A4, X11-save-location
JZ        A4,0,X11
LXI       X11,A4
LIJ       X11,0,X11
```

Use of the LINK=IBJ$ option, as opposed to the BANKED=RETURN option and BANK statement, also allows for modification of bank structure collections without change to the source program, and thus without recompilation. An IBJ$ reference does an LMJ or LIJ to the correct bank, and the subprogram returns correctly regardless of the bank structure.

8.5.5. U1110=OPT Option

The U1110=OPT COMPILER statement option invokes a code reordering algorithm during the code generation phase of the compiler. This option is meaningful on hardware that incurs register conflicts (most hardware for which ASCII FORTRAN is supported). It shuffles generated code, which minimizes the hardware register conflicts to speed execution.

This code-reordering algorithm can also be called by the use of the E option on the ASCII FORTRAN processor call statement (@FTN,E). See 9.6.3.

8.5.6. STD=66 Option

The STD=66 COMPILER statement option is provided to let relocatables created with level 8R1 coexist with relocatables created with level 9R1 and higher.

ASCII FORTRAN levels 9R1 and higher conform to the FORTRAN 77 standard (ANSI X3.9-1978). ASCII FORTRAN levels lower than 9R1 conform to ANSI X3.9-1966.

Compatibility problems arise because conforming to the FORTRAN 77 standard requires certain features that conflict with the implementation of previous ASCII FORTRAN levels.

If the option STD=66 is not specified in a compilation, ASCII FORTRAN conforms to the FORTRAN 77 standard. If STD=66 is specified, ASCII FORTRAN implements the features in Table 8-1 as done in previous levels (that is, in a nonstandard manner).

If any prelevel 9R1 ASCII FORTRAN relocatables appear in a collection, you must do the ASCII FORTRAN level 9R1 or higher compilations with the COMPILER (STD=66) compatibility option. Also, all relocatables in a collected program must have matching compatibility options. If one program is compiled with STD=66, all FORTRAN programs to be collected with that program must have STD=66.

Using the STD=66 option also implies using the ARGCHK=OFF option.

Table 8-1. Use of the STD=66 COMPILER Statement Option

Feature	Implementation with STD=66 Option	Implementation without STD=66 Option
Character data	Unpacked character data - all character items (including array elements) begin on word boundaries. Substrings passed as arguments are passed as expressions (they are not passed by address).	Packed character data for arrays, common blocks, etc. - character items need not begin on word boundaries (see 6.13.1). If either a dummy or actual argument is type character, then the other must also be type character (see 7.3.4).
DO-loops	A DO-loop (with DO statement DO *n i* = e_1,e_2,e_3) must be traversed at least once, even if e_1 >e_2 and e_3>0. -1 is assumed as the increment value if e_3 is not specified, and e_1 and e_2 are both constant expressions, where e_1 > e_2.	A DO-loop is not traversed if e_1 >e_2 and e_3>0, or if e_1<e_2 and e_3<0 (see 4.5.4.1). 1 ia assumed as the increment value if e_3 is not specified.
Typing	Symbolic names of constants and statement functions have no associated types. Types are determined by usage in expressions.	Symbolic names of constants (see 6.8) and statement functions (see 7.3.1) have associated types, based on normal typing conventions, and previous IMPLICIT and type statements.

8.5.7. ARGCHK Options

ASCII FORTRAN automatically checks the types and number of actual arguments against those expected on subprogram entry. However, if the optimization options Z or V are specified on the @FTN processor call, type checking is not done automatically. If you want type checking when optimization is used, you must use the ARGCHK=ON COMPILER statement option.

If type checking is not desired, you can use the ARGCHK=OFF option. When either the calling routine or the routine being called has disabled type checking by using the

ARGCHK=OFF option or by compiling with optimization (without the ARGCHK=ON option), type checking is not done.

8.5.8. PROGRAM=BIG Option

If the PROGRAM=BIG COMPILER statement option appears, the O option (see processor call options, 9.5.1) is turned on. The compiler generates code that allows D-bank addresses to exceed octal 0200000 (decimal 65,535). This option eliminates problems that result when a program that needs the O option is mistakenly compiled without it.

8.5.9. Options Affecting Virtual Space

The following COMPILER statement options change the characteristics of virtual space when they are put in the main program.

PAGESIZE = ***n*****K**

where:

n

has the possible values 4, 8, 16, 32, 64 (or 65), 128 (or 131).

K

means 1,024 words.

This option specifies the size of the D-banks used for virtual space. Each D-bank holds a page of virtual space. The values of 65K and 131K are acceptable, allowing you to think of K as 1,000 words. The default value is 32K, allowing about 32,000-word banks for virtual space.

Note: *Virtual sizes are always a power of 2. The D-banks containing virtual pages are slightly larger than a virtual page because a small ID area precedes the portion used for a virtual page.*

NBRPAGES=*n*

where *n* is a value greater than 1, but less than 2,048.

This option specifies the number of pages used for virtual space. These pages are defined in your main program as void D-banks from INFO-11 directives. The default is 200 pages.

Note: *The Executive currently limits the number of banks defined in an absolute to 251.*

The maximum virtual address space available to a program is defined by the number of pages times the virtual page size. Table 8-2 gives the virtual address limits for specific numbers and sizes of virtual pages.

Table 8-2. Virtual Address Limits

Number of Pages	Page Size					
	128K	64K	32K	16K	8K	4K
2,000	262M	131M	65M	32M	-	-
1,000	131M	65M	32M	16M	8M	-
500	65M	32M	16M	8M	4M	2M
250	32M	16M	8M	4M	2M	1M
200	26.2M	13.1M	6.5M	3.3M	1.6M	0.8M

* M *m*eans millions of words. The blank line defines the current Exec limit. The default is 200 pages at 32K words each.

VIRTUAL=NCCB

This option specifies that the D-banks used for virtual space have the S option, indicating that these are shared banks. Shared banks (often called nonconfigured common banks or NCCBs) do not disappear when program execution is done; separate runs can simultaneously access NCCBs. However, D-banks are writable and this can lead to problems.

When the Executive initially loads a program that references NCCBs, it scans a list of currently defined NCCBs to see if it can link the program to existing banks. For missing NCCBs, it attempts to load a copy from the program. When the program has a void copy, the Exec prints an error diagnostic and treats it as a program-local bank. Since ASCII FORTRAN uses INFO-11 directives to define void D-banks for virtual space, this makes the VIRTUAL=NCCB option currently unusable. Therefore, this option will be dropped if void banks cannot be NCCBs.

VIRTUAL=STATIC

This option specifies that the D-banks used for virtual space are static rather than dynamic. This means that each time a program executes in main storage, all D-banks used for virtual space are resident in main storage. This can cure thrashing problems that you may be having, but it also means that other concurrent users may be locked out. When the amount of virtual space that you define is greater than available main storage, your program may never be swapped-in again. The default is to define dynamic D-banks for virtual space.

Examples:

```
1.  VIRTUAL (ALL)
2.  COMPILER (PAGESIZE=16K)
```

```
3.  COMPILER (NBRPAGES=100)
4.  COMPILER (VIRTUAL=STATIC)
5.  COMPILER (BANKACT=NOTEST)
```

Statement 1 shows that all named common banks referenced in this element are placed into virtual space.

Statement 2 shows that 16,384-word pages are used for virtual space.

Statement 3 shows that a maximum of 100 pages can be used for virtual space. This gives a maximum virtual address range of 1.6 million words for this program.

Statement 4 shows that the D-banks used for virtual space are static (always resident in main storage during program execution).

Statement 5 shows that D-bank activation code sequences generated by the compiler to reference virtual (or banked) space do not generate a test around the D-bank activation instruction.

Section 9
Writing a FORTRAN Program

9.1. General

This section discusses the organization of FORTRAN programs and the rules for writing them.

9.2. FORTRAN Program Organization

A FORTRAN program is made up of one main program and as many subprograms and procedures as you require. The main program contains the primary steps required to solve a given problem. The subprograms (either function, subroutine, or BLOCK DATA) are subordinate units used by the main program. All can be referred to as program units.

9.2.1. Program Unit

A program unit consists of a sequence of statements and optional comment lines. A program unit is either a main program, a function or subroutine subprogram, or a BLOCK DATA (specification) subprogram. Both **internal** and external subprograms are program units. An executable program contains one main program and zero or more subprograms. If the source input to the ASCII FORTRAN compiler consists of a main program and one or more subprograms or specification subprograms, then the main program must physically be the first program unit.

9.2.2. Types of Program Units

The various types of program units, illustrated in Figure 9-1, are:

- A main program is the program unit that receives control when an executable program is placed in execution by the processor. A main program consists of a series of comments and statements that may begin with a PROGRAM statement, does not contain a BLOCK DATA statement, and is terminated by an END, **FUNCTION, or SUBROUTINE** statement.
- A function subprogram is an executable subprogram that consists of a series of comments and statements (it does not contain a BLOCK DATA or PROGRAM statement) starting with a FUNCTION statement and terminated by an END, **FUNCTION or SUBROUTINE** statement.
- A subroutine subprogram is an executable subprogram that consists of a series of comments and statements (it does not contain a BLOCK DATA or PROGRAM statement) starting with a SUBROUTINE statement and terminated by an END, **FUNCTION, or SUBROUTINE** statement.
- A specification (BLOCK DATA) subprogram is a nonexecutable subprogram that consists of a series of comments and specification statements starting with a BLOCK DATA statement and terminated by an END statement. A BLOCK DATA subprogram does not contain executable code and is used solely for the assignment of initial values to variables in common blocks. A BLOCK DATA subprogram can't be called.
- **A program unit is an internal subprogram if the previous program unit is terminated by the internal's FUNCTION or SUBROUTINE statement instead of an END statement. Thus, both function subprograms and subroutine subprograms can be internal subprograms.**

  An internal subprogram is considered as nested within the previous external subprogram, and can access the external subprogram's data as well as have its own local data. An external subprogram can have many internal subprograms.
- A program unit is external if it is a main program or a BLOCK DATA subprogram if it is the first program unit in the compilation **or if the previous program unit is terminated by an END statement.** Thus, any of the program units described above can be external, i.e., free-standing program units.

See 7.3.2 and 7.3.3 for descriptions and examples of functions and subroutines (both external and **internal**).

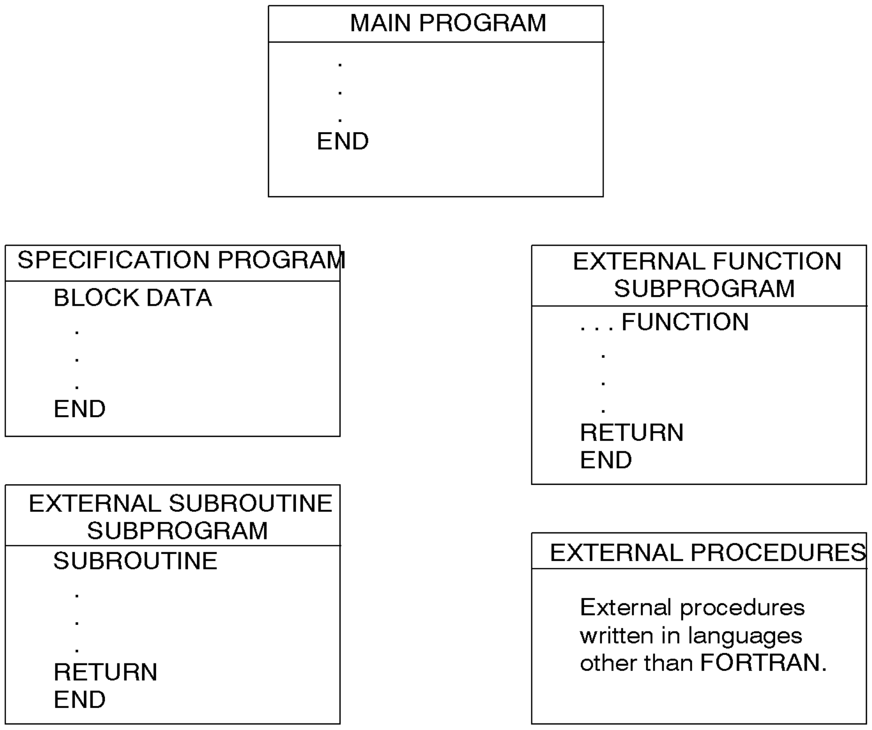


Figure 9-1. Program Units Within a FORTRAN Program

Subprograms are useful because they eliminate repetitive coding of procedures used many times in a program. In addition, a library of mathematical external procedures, called intrinsic functions (see 7.7.1), is present and contains debugged procedures for computation of mathematical functions such as square root, sine, etc. You can code the main program of a large FORTRAN program as a logical skeleton consisting primarily of references to subprograms; you can independently code these subprograms. The subprograms may or may not be compiled with the main program.

The OS 1100 Collector links together all compiled program units to form an executable program starting with the main program unit. You can write program units in languages other than FORTRAN but must conform to the rules for FORTRAN subprograms. (See Appendix K.) Such program units and procedure subprograms (function and subroutine subprograms) are classed as external procedures.

9.2.3. Program Unit Organization

A program unit consists of comments and statements. A FORTRAN statement falls into one of two categories: an executable statement or a nonexecutable statement. An executable statement specifies an action. A nonexecutable statement describes the characteristics and arrangement of data, editing information, statement function definitions, and classification of program units. Nonexecutable statements are generally intended as instructions to the compiler; in most cases, no executable machine language instructions are generated. Executable statements result in executable machine language instructions (object program).

Statement classification and ordering are described in 9.3. The actual formats of the lines of a FORTRAN source program are outlined in 9.4.

Many program units can be put together into one source element and compiled into one relocatable element.

9.2.4. Execution Sequence

Execution of a FORTRAN program begins with the performance of the first executable statement of the main program. The difference between an executable statement and a nonexecutable statement is outlined in 9.2.3.

A subprogram, when referenced by the name of the subprogram, starts execution with the first executable statement of that subprogram. When an entry name is used to reference a subprogram, execution starts with the first executable statement following the proper ENTRY statement. When subprogram activity is complete, execution of the calling program is resumed at the statement following the call, or at one of the statement labels appearing in the argument list of the CALL statement.

Nonexecutable statements or comment lines do not affect execution sequence.

Every program unit except a BLOCK DATA subprogram must contain at least one executable statement. A program unit should not contain a statement that can't be executed (for example, an unlabeled statement following an unconditional GO TO).

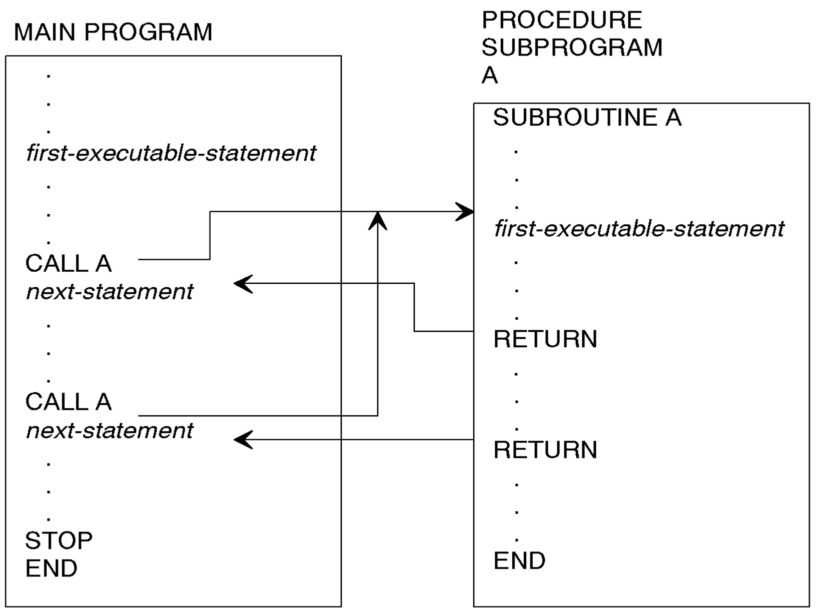
If the execution sequence attempts to proceed beyond the last executable statement of a main program, the effect is the same as the execution of a STOP statement. If the execution sequence attempts to proceed beyond the last executable statement of a procedure subprogram, the effect is the same as the execution of a RETURN statement.

In the execution of a program, a subprogram can't be referenced twice without an intervening RETURN statement being executed in that subprogram. That is, no recursion is allowed.

In the first example shown in Figure 9-2, the main program proceeds until it encounters a reference (CALL) to the external procedure. The external procedure assumes control until it encounters a RETURN statement, which sends control back to the calling program unit (in this case, the main program). The main program then continues processing until another reference transfers control to the external procedure. The external procedure assumes processing until it encounters a RETURN statement (not necessarily the same one as the first RETURN statement) and transfers control back to the main program. The main program then resumes processing until it encounters the STOP statement, which transfers job control to the operating system.

The second example in Figure 9-2 shows how a procedure subprogram calls on another procedure subprogram during execution.

Example 1:



Example 2:

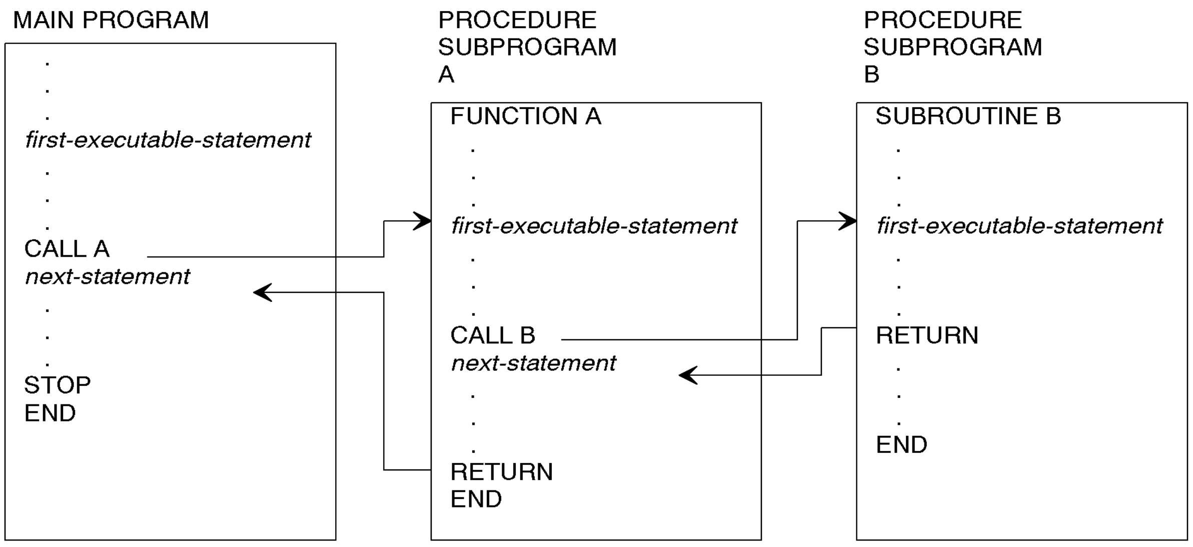


Figure 9-2. Sample Control Paths During Execution

9.3. Statement Categories

A given FORTRAN statement performs any one of the following functions:

1. It performs certain operations (addition, multiplication, branching, etc.)
2. It specifies the nature of the data being handled.
3. It defines the characteristics of the source program.

FORTRAN statements usually are composed of certain FORTRAN keywords used with the basic elements of the language: constants, variables, and expressions. The categories of FORTRAN statements are:

1. Assignment statements: The basic mechanism by which the result of an expression is stored (and therefore, saved) in a variable, array element, or substring for future reference.
2. Control statements: Govern the order of execution of the object program and terminate its execution.
3. Input/output statements: Control input/output devices and let you transfer data between internal storage and an input/output medium.
4. FORMAT statement: Used with certain input/output statements to specify the form in which data appears in a FORTRAN record or an input/output device.
5. **NAMELIST statement: Used with certain input/output statements to specify data appearing in a special kind of record.**
6. DATA initialization statement: Assigns initial values to variables and array elements.
7. Specification statements: Declare the properties of variables, arrays, and functions (such as type and amount of storage reserved).
8. Statement function definition statement: Specifies operations that are performed when the statement function name appears in an executable statement.
9. Subprogram/Program statements: Used to name and specify arguments for functions and subroutines. Used to identify program units and subprograms.
10. **Program control statements: Temporarily modify the source program being compiled, modify the code generated, and control the compiler listing produced.**
11. **Debug facility statements: Set the conditions for operation of the debug facility, identify the beginning of the debug packet, and designate actions to be taken at specific points in the program unit.**

9.3.1. Statement Classification

Each FORTRAN statement is classified as executable or nonexecutable.

Executable statements specify actions and form an execution sequence in an executable program. All assignment, control, and input/output statements are executable statements.

Nonexecutable statements describe characteristics and arrangement of data, editing information, statement function definitions, and classification of program units. Nonexecutable statements are not a part of the execution sequence.

See Appendix F for a breakdown of executable and nonexecutable statements.

Statement labels are allowed on any statement (except continuation lines of a statement). However, only labeled executable statements (except ELSE IF, ELSE, and **DEFINE FILE**) and FORMAT statements can be referred to by the use of statement labels. There is one exception to this rule: the **DELETE statement** can refer to any statement label.

You can't use a label on a FORMAT statement as a transfer label for another statement in the program unit (that is, it can't be used to control the execution sequence).

9.3.2. Ordering of Statements in a Program Unit

The order of FORTRAN statements in a program unit (other than a BLOCK DATA subprogram) is:

Placement	Statements
1	**COMPILER statement**, if desired.
2	Subprogram (FUNCTION or SUBROUTINE) statement if subprogram, PROGRAM statement if main program (optional), or BLOCK DATA statement if specification subprogram.
3	**VIRTUAL statement,** if desired.
4	IMPLICIT statements, if any.
6	Statement function definitions, if any.
7	Executable statements, at least one of which should be present.
8	**Debug packets,** if desired.
9	END statement.

In a program unit, FORMAT, **DEFINE FILE, and EDIT** statements and comment lines can appear anywhere. ENTRY statements may appear anywhere except in ranges of DO-loops and block IF structures, or in main programs or BLOCK DATA subprograms.

PARAMETER statements can occur before and among other specification statements. DATA statements should occur after all specification statements.

Figure 9-3 shows the required order of statements for a program unit. Vertical lines delineate varieties of statements that may be interspersed. For example, FORMAT statements can be interspersed with specification statements and executable statements. Horizontal lines delineate varieties of statements that can't be interspersed. For example, specification statements can't be interspersed with statement function definitions, and statement function definitions can't be interspersed with executable statements.

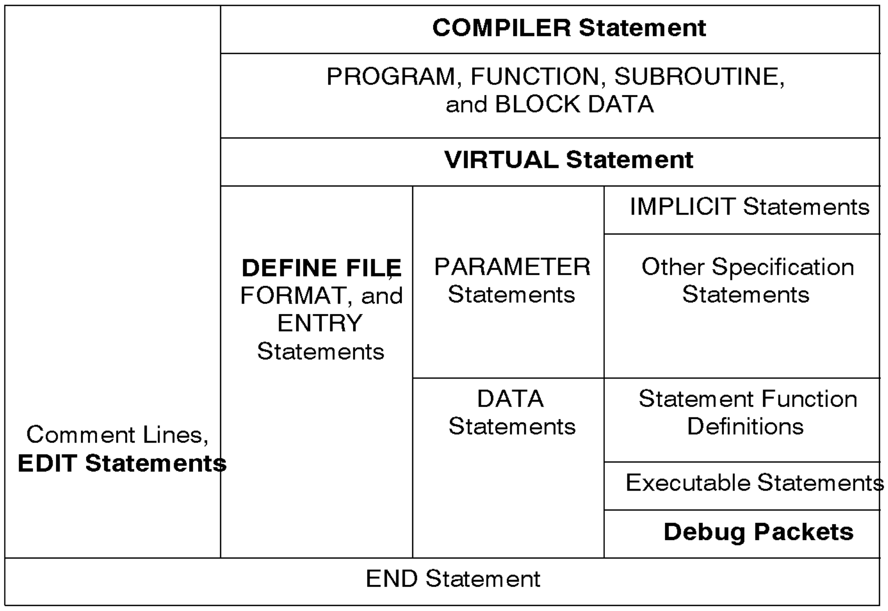

COMPILER Statement
PROGRAM, FUNCTION, SUBROUTINE, and BLOCK DATA
VIRTUAL Statement
Comment Lines, EDIT Statements
DEFINE FILE, FORMAT, and ENTRY Statements
PARAMETER Statements
IMPLICIT Statements
Other Specification Statements
DATA Statements
Statement Function Definitions
Executable Statements
Debug Packets
END Statement

Figure 9-3. Order of Statements in a Program Unit

9.4. Source Program Representation and Control

9.4.1. Source Program Format

An ASCII FORTRAN program unit, such as a main program or subprogram, consists of a sequence of lines that are ordered by the succession in which they are presented to the processor. A line in a program unit is a string of 72 characters. The character positions in a line are called columns and are consecutively numbered 1, 2, 3, . . ., 72. The number indicates the sequential position of a character in the line starting at the left and proceeding to the right. Lines are classified as initial lines, continuation lines, or comment lines.

9.4.1.1. General Statement Form

The following is an illustration of the two basic forms of a source program line. Use the first form for all FORTRAN statements. The appearance of a character in column 6 indicates a continuation line. Use the second form for comment lines.

1 5	6	7	73 80
number or blank		*statement*	nonprocessed documentation

1	2	73 80
C or *	*comment*	nonprocessed documentation

A FORTRAN line uses only columns 1 through 72. The information in columns 73 through 80, shown only in the program listing, is used for documentation. Any text (except comments) beyond column 72 is moved over three columns and a space-period-space is inserted before the source line is printed. For statements, columns 1 through 5 are used for labels. Column 6 is used to indicate the continuation of a statement. Columns 7 through 72 contain the statement or its continuation. If the line is a comment line, columns 2 through 72 contain the comment information.

9.4.1.2. Initial Line

The first line of a FORTRAN statement is called the initial line. Columns 1 through 5 of an initial line must be blank or contain a statement label. Column 6 must be blank or contain a zero. A statement or portion of a statement is placed in columns 7 through 72. The remaining columns, if any, of an initial line may be blank or used for identification and a statement sequence number.

9.4.1.3. Continuation Line

A continuation line contains a continuation of a FORTRAN statement. Column 6 must contain a nonblank, nonzero character. Columns 1 through 5 must be blank.

The FORTRAN 77 standard allows up to 19 continuation lines. **ASCII FORTRAN allows as many lines as necessary as long as the number of significant characters is under 1,320.** Blanks aren't significant characters except in character constants, **Hollerith constants, apostrophe edit descriptors, and H edit descriptors.** An ASCII FORTRAN statement containing only significant characters can thus continue for a maximum of 19 lines (assuming lines with statement coding in columns 7 through 72).

9.4.1.4. Comment Line

Any line with the characters C (**which may be lowercase**) or * in column 1 is treated as a comment and not as a FORTRAN statement. A line that contains only blank characters in columns 1 through 72 is treated as a comment line.

Any character capable of being represented in the processor can appear in a comment line, occupying columns 2 through 72. Comments are included with listings and in the symbolic elements as a documentation/information device, but they aren't compiled as FORTRAN statements.

In ASCII FORTRAN, comment lines can appear anywhere within a program unit. The FORTRAN 77 standard doesn't allow comment lines following an END statement.

9.4.1.5. Inline Comment

To retain compatibility with FORTRAN V, you can attach a comment to any line by placing a master space character (@) before the comment. An inline comment can't start in columns 1 through 6.

Example:

```
X = 5.0      @ INITIALIZE X
```

9.4.1.6. Statements

A statement consists of an initial line and, if required, continuation lines that follow in sequence. Write each line of the statement in columns 7 through 72. An initial line must have either the digit 0 or a blank character in column 6 and must not have the character C or * in column 1. Each continuation line must have a character other than the digit 0 or blank character in column 6, and must have blanks in columns 1 through 5. Precede each continuation line with an initial line or another continuation line.

Except as part of a logical IF statement, no statement can begin on a line that contains any part of the previous statement. For example,

```
IF (LOG) THEN I=I+1
```

is illegal, because a block IF statement and an assignment statement appear on the same line.

Blank characters in a statement don't change the interpretation of a statement except when they appear within the data strings of **Hollerith constants**, character constants, apostrophe edit descriptors, and H edit descriptors. Nonsignificant blank characters don't count as characters in the limit of 1,320 characters in any one statement.

Blank lines, like blank columns, have no effect on the code generated. Use blank lines wherever you want to insert spaces in the listing of the source program. A line that is blank in columns 1 through 72 is not regarded as an initial line.

For example:

```
  40   FORMAT(17H AVERAGE VALUE = , F10.5)
```

could be written as:

```
  40   FORMAT(
     A17H AVERAGE VALUE =
     B,F10.5)
```

However, it can't be written as:

```
  40   FORMAT(17H
     AAVERAGE VALUE =
     B,F10.5)
```

because the 17H of the FORMAT statement means that the 17 characters immediately following the H are Hollerith data. Since blank characters are significant in Hollerith data, the next 17 blanks are interpreted to be the data value rather than the intended characters.

9.4.1.7. Statement Labels

A statement label provides a means of referring to individual statements.

A statement label is an unsigned integer (1 through 99999) that identifies a FORTRAN statement and is written in columns 1 through 5 of the initial line of a statement. Use only the digits 0 through 9 and blank characters in a statement label. You can't define the same statement label more than once in a program unit, but the same statement label can appear in more than one program unit. The value of a statement label does not affect the order in which statements are executed; it merely identifies the statement it is associated with. This lets other statements of the program unit reference it. See 9.3.1 for rules stating which statement labels may be referenced.

Use a maximum of five digits in a statement label. All blanks and leading zeros are ignored.

For example, in the sequence:

```
     .
     .
     .
   157 F = (A + B - C) / D
     .
     .
     .
   GO TO 157
     .
     .
     .
```

the statement label of the arithmetic assignment statement can also be written as:

```
  1Δ5Δ7  F = (A + B - C) / D
```

or:

```
01△57  F = (A + B - C) / D
```

or:

```
15△△7  F = (A + B - C) / D
```

but not as:

```
157△0  F = (A + B - C) / D
```

because this label is 1570 rather than 157.

9.4.2. Compiler Listing

For every program compiled, a compiler listing is produced. The composition of the information listed is a function of the listing options that you specify.

9.4.2.1. Listing Options

The options available to control the composition of the compiler listing are specified in the @FTN processor call (see 9.5) and in EDIT statements. The processor call options specify the initial setting for the type of listing desired. They can be overridden at any point in the source program by the EDIT statement (see 8.4).

If you use no listing options, a minimum of default output is produced. This output consists of a single compiler identification line with time and date, a single line indicating the end of compilation with the number of errors and warnings, and the amount of storage for I-bank, D-bank, and common blocks. Error and warning messages are printed. (See 9.8 for information on the format of diagnostic messages.)

All compiler options are listed in 9.5.1. The options that can be specified on the @FTN processor call to control the compiler listing are:

D

A listing of the storage map, common blocks, entry points, and external references is generated.

K

When the source lines are being printed (S or L option), both the update and input (base) line numbers appear in front of the source images. In addition, SIR correction lines are interspersed in the source code listing.

L

In addition to the information listed for the S option, the cross-reference listing, storage map, common block list, entry point list, external reference list, and octal and mnemonic representation of the generated object code are also printed.

N

The N option causes warnings to be suppressed.

R

A cross-reference listing is generated.

S

A listing of the source program statements is produced in addition to the default output. Errors or warnings detected during syntax analysis can be interspersed in the source program listing and are usually printed immediately following the offending statement.

T

Print diagnostics for nonstandard FORTRAN usage (see 9.5.1).

W

The correction lines used by the source input routine (SIR) are printed at the top of the listing (before the source code listing, if any).

Y

A listing containing the storage map, common blocks, entry points, external references, and a shorter mnemonic representation of the object code is generated.

9.4.2.2. Composition of a Compiler Listing

The composition of a listing generated under control of the L option is described in this subsection. Refer to 9.4.2.1 to determine which parts of the listing described are not generated under control of the other listing options.

See 9.4.2.2.2 for an example of an L option listing.

9.4.2.2.1. Identification Line

The identification line printed at the beginning of each compiler listing has the following form:

```
FTN ssRii - mm/dd/yy - hh:mm - (i,o)
```

where:

FTN

is the ASCII FORTRAN compiler used to perform the compilation.

*ss*R*ii*

is the level number of the compiler; *ss* denotes the symbolic update number; *ii* denotes the incremental update number. Following *ii*, a letter can be used to indicate minor modifications, such as emergency fixes, recollections to include modified RLIB$ elements, etc. (for example, 9R1A). Release levels begin with 1R1.

mm/dd/yy

indicates the month, day, and year on which the compilation is performed.

hh:mm

is the hour and minute at which the compilation begins.

(i,o)

indicates the input cycle number and the output cycle number, respectively. If *i* is blank, the input was from the runstream. If *o* is blank, there is no symbolic output element.

9.4.2.2.2. Source Code Listing

The source code listing contains all source language input lines processed by the compiler. The errors detected during the production of this listing are printed on the first available line following the offending source line.

Below is a sample source code listing with a main program, an internal subroutine in the main program, an external subroutine subprogram, and an internal subroutine in that external subroutine.

The first (leftmost) field of the listing contains the level number of the statement (if greater than zero). The level number is a counter containing the DO-level plus the IF-level of the statement. The DO-level is a counter containing the number of nested DO-loops; the IF-level is a counter containing the number of nested block IF structures.

The level number column contains blanks, if the level number is zero, and the letter D if the source line being printed is deleted by the DELETE statement.

Since the example contains no block IF or DO statements or deleted lines, the level number column is always blank.

The second field of this listing is the source line sequence number. The line sequence numbers are consecutive integers, one per line, followed by a period. There is one exception: if the source element was created by CTS (see the *Series 1100 Conversational Time Sharing (CTS)*, *Programmer Reference*, UP-7940), then the CTS line number appears in the second field.

When you specify the K option, the update line number is followed by the input (base) line number. This option is useful when you use Source Input Routine (SIR) correction lines.

The final field is the FORTRAN source line.

If the source input to the compiler consists of more than one external program unit, then spacing is inserted before the source code listing of the next external program unit.

L Option Listing

```
@FTN,L A.TEST,B.
FTN 11R1 04/21/84-08:11(1,)
         1.    I = 1
         2.    PRINT *,'MAIN: I = ' , I
         3.    CALL INTMN
         4.    CALL SUB1
         5.  C
         6.  C *** INTERNAL SUBPROGRAM (INTMN) IN MAIN PROGRAM ***
         7.  C
         8.    SUBROUTINE INTMN
         9.    I = 2
        10.    PRINT *,'INTMN: I = ' , I
        11.    RETURN
        12.    END

        13.    SUBROUTINE SUB1
        14.    COMMON I
        15.    I = 3
        16.    PRINT *,'SUB1: I = ' , I
        17.    CALL INTSUB
        18.    RETURN
        19.   C
        20.   C *** INTERNAL SUBPROGRAM (INTSUB) IN EXTERNAL SUBPROGRAM (SUB1) ***
        21.   C
        22.    SUBROUTINE INTSUB
        23.    INTEGER I
        24.    COMMON /C/ I
        25.    I = 4
        26.    PRINT *,'INTSUB: I = ' , I
        27.    RETURN
        28.    END

 F O R T R A N   C R O S S   R E F E R E N C E   L I S T I N G

 MAIN PROGRAM

 NAME   USE   LINE NUMBER

 I     SET    1
       USED   2
 INTMN USED   3
 SUB1  USED   4

 SUBROUTINE INTMN : MAIN PROGRAM

 NAME   USE   LINE NUMBER

 I     SET    9
       USED  10
 INTMN SPEC   8

 SUBROUTINE SUB1

 NAME   USE   LINE NUMBER

 I     COMMON 14
       SET    15
       USED   16
 INTSUB USED   17
 SUB1   SPEC   13

 SUBROUTINE INTSUB : SUB1

 NAME  USE    LINE NUMBER

 C     SPEC   24
 I     COMMON 24
       SPEC   23
       SET    25
       USED   26
 INTSUB SPEC   22

 F O R T R A N   O B J E C T   C O D E   L I S T I N G

MAIN PROGRAM
```

```
RELATIVE   INSTRUCTION SUBFIELDS   U  FLD    LABEL        SYMBOLIC INSTRUCTION                            LINE
ADDRESS   F   J  A  X  HI   U       LC                    F,J       A,U,X                               NUMBER
                                             $(1)         AXR$
                                                          ASCII
                                             1G
000000   74  13  13 00 0  000000X    4       $(1)         LMJ       X11,FINT2$                                 0
000001   10  16  04 00 000001                0G           LA,U      A4,1                                       1
000002   01  00  04 00 0  000000R    0                    SA        A4,I                                       1
000000             0777777777776             $(4)         +         0777777777776                              2
000001             0030400000000                          +         0030400000000                              2
000002             0000000000000                          +         0000000000000                              2
000003             0000000000000                          +         0000000000000                              2
000004             0000000000000R   10                    +         0000000000000                              2
000005             0060012000000                          +         0060012000000                              2
000006             0000000000000R    0                    +         0000000000000                              2
000007             0010004000000                          +         0010004000000                              2
000010   74  13  13 00 0  000000X    5                    LMJ       X11,FMTE$$                                 2
000003   26  16  14 00 000000    R   4       $(1)         LXM,U     A0,FTEMP$                                  2
000004   46  17  13 00 000000    X   6                    LXI,XU    X11,BDICALL$+F2SE$$                        2
000005   00  00  13 00 0  000000X    6                    IBJ$      X11,F2SE$$                                 2
000006   10  16  00 00 000000                             LA,U      A0,0                                       3
000007   46  17  13 00 000000                             LXI,XU    X11,0                                      3
000010   74  13  13 00 0 000017R     1                    LMJ       X11,INTMN                                  3
000011   10  16  00 00 000000                             LA,U      A0,0                                       4
000012   46  17  13 00 000000                             LXI,XU    X11,0                                      4
000013   74  13  13 00 0 000000X     7                    LMJ       X11,SUB1                                   4
000014   74  13  13 00 0 000000X     8                    LMJ       X11,FEXIT$                                 8
000000             0115101111116             $(10)        +         0115101111116                              8
000001             0072040111040                          +         0072040111040                              8
000002             0075040040040                          +         0075040040040                              8

SUBROUTINE INTMN : MAIN PROGRAM

RELATIVE   INSTRUCTION SUBFIELDS   U  FLD    LABEL        SYMBOLIC INSTRUCTION                            LINE
ADDRESS   F   J  A  X  HI   U       LC                    F,J       A,U,X                               NUMBER
                                             $(1)         AXR$
                                                          ASCII
000015   27  00  13 00 0 000011R     4                    LX        X11,FTEMP$+9                               8
000016   74  04  00 13 0 000000                           J         0,X11                                      8
000017   06  00  13 00 0 000011R     4       INTMN        SX        X11,FTEMP$+9                               8
000012             0111116124115             $(4)         +         0111116124115                              8
000013             0116040072040                          +         0116040072040                              8
RELATIVE   INSTRUCTION SUBFIELDS   U  FLD    LABEL        SYMBOLIC INSTRUCTION                           LINE
ADDRESS   F   J  A  X  HI   U       LC                    F,J       A,U,X                                NUMBER
000014             0115101111116                          +         0115101111116                               8
000015             0041040040040                          +         0041040040040                               8
000016             0000000000012R    4                    +         0000000000012                               8
000020   10  00  01 00 0 000016R     4       $(1)         LA        A1,FTEMP$+14                                8
000021   46  16  13 00 000000  X     9                    LXI,U     X11,BDICALL$+ARGC1$                         8
000022   00  00  13 00 0 000000X     9                    IBJ$      X11,ARGC1$                                  8
000023   10  16  04 00 000002                2G           LA,U      A4,2                                        9
000024   01  00  04 00 0 000000R     0                    SA        A4,I                                        9
000017             0777777777776             $(4)         +         0777777777776                              10
000020             0030400000000                          +         0030400000000                              10
000021             0000000000000                          +         0000000000000                              10
000022             0000000000000                          +         0000000000000                              10
000023             0000000000003R   10                    +         0000000000003                              10
000024             0060013000000                          +         0060013000000                              10
000025             0000000000000R    0                    +         0000000000000                              10
000026             0010004000000                          +         0010004000000                              10
000027   74  13  13 00 0 000000X     5                    LMJ       X11,FMTE$$                                 10
000025   26  16  14 00 000017  R     4                    $(1)      LXM,U A0,FTEMP$+15                         10
000026   46  17  13 00 000000  X     6                    LXI,XU    X11,BDICALL$+F2SE$$                        10
000027   00  00  13 00 0 000000X     6                    IBJ$      X11,F2SE$$                                 10
000030   74  04  00 00 0 000015R     1                    J         $-11                                       11
000003             0111116124115             $(10)        +         0111116124115                              12
000004             0116072040111                          +         0116072040111                              12
000005             0040075040040                          +         0040075040040                              12
SUBROUTINE   SUB1

RELATIVE   INSTRUCTION SUBFIELDS   U  FLD    LABEL        SYMBOLIC INSTRUCTION                           LINE
ADDRESS   F   J  A  X  HI   U       LC                    F,J       A,U,X                                NUMBER
                                             $(1)         AXR$
                                                          ASCII
000031   27  00  13 00 0 000030R     4                    LX        X11,FTEMP$+24                              12
000032   74  04  00 13 0 000000                           J         0,X11                                      12
000033   06  00  13 00 0 000030R     4       SUB1         SX        X11,FTEMP$+24                              12
000031            0123125102061              $(4)         +         0123125102061                              13
```

```
000032          0041040040040                       +        0041040040040                  13
000033          0000000000031R    4                 +        0000000000031                  13
000034   10  00  01 00 0 000033R    4       $(1)         LA       A1,FTEMP$+27                   13
000035   46  16  13 00 000000  X    9                    LXI,U    X11,BDICALL$+ARGC1$            13
000036   00  00  13 00 0 000000X    9                    IBJ$     X11,ARGC1$                     13
000037   10  00  04 00 0 000001R    0       3G           LA       A4,$bentr                      13
000040   74  11  04 00 0 000046R    1                    JB       A4,4G                          13
                                            $(1),5G
000000          0000000000006R   10         $(6)         +        0000000000006                  13
000001          0000000000007R   10                      +        0000000000007                  13
000002          0000002000000R    6                      +        0000002000000                  13
RELATIVE   INSTRUCTION SUBFIELDS   U  FLD   LABEL        SYMBOLIC INSTRUCTION                  LINE
ADDRESS    F   J  A  X  HI   U       LC                  F,J      A,U,X                        NUMBER
000041   10  00  00 00 0 000002R    6       $(1)         LA       A0,FTEMP$$+2                   13
000042   46  17  13 00 000000                            LXI,XU   X11,0                          13
000043   74  13  13 00 0 000000X   10                    LMJ      X11,OLDCM$                     13
000044   10  16  04 00 000001                            LA,U     A4,1                           13
000045   01  00  04 00 0 000001R    0                    SA       A4,$bentr                      13
000046   10  16  04 00 000003                   4G       LA,U     A4,3                           15
000047   01  00  04 00 0 000000R    2                    SA       A4,I                           15
000034          0777777777776               $(4)         +        0777777777776                  16
000035          0030400000000                            +        0030400000000                  16
000036          0000000000000                            +        0000000000000                  16
000037          0000000000000                            +        0000000000000                  16
000040          0000000000011R   10                      +        0000000000011                  16
000041          0060012000000                            +        0060012000000                  16
000042          0000000000000R    2                      +        0000000000000                  16
000043          0010004000000                            +        0010004000000                  16
000044   74  13  13 00 0 000000X    5                    LMJ      X11,FMTE$$                     16
000050   26  16  14 00 000034  R    4       $(1)         LXM,U    A0,FTEMP$+28                   16
000051   46  17  13 00 000000  X    6                    LXI,XU   X11,BDICALL$+F2SE$$            16
000052   00  00  13 00 0 000000X    6                    IBJ$     X11,F2SE$$                     16
000053   10  16  00 00 000000                            LA,U     A0,0                           17
000054   46  17  13 00 000000                            LXI,XU   X11,0                          17
000055   74  13  13 00 0 000061R    1                    LMJ      X11,INTSUB                     17
000056   74  04  00 00 0 000031R    1                    J        $-21                           18
000006          0000000000001               $(10)        +        0000000000001                  22
000007          0100505050505                            +        0100505050505                  22
000010          0000000000000                            +        0000000000000                  22
000011          0123125102061                            +        0123125102061                  22
000012          0072040111040                            +        0072040111040                  22
000013          0075040040040                            +        0075040040040                  22

SUBROUTINE   INTSUB : SUB1

RELATIVE   INSTRUCTION SUBFIELDS   U  FLD   LABEL        SYMBOLIC INSTRUCTION                  LINE
ADDRESS    F   J  A  X  HI   U       LC                  F,J      A,U,X                        NUMBER
                                            $(1)         AXR$
                                                         ASCII
000057   27  00  13 00 0 000045R    4                    LX       X11,FTEMP$+37                  22
000060   74  04  00 13 0 000000                          J        0,X11                          22
000061   06  00  13 00 0 000045R    4       INTSUB       SX       X11,FTEMP$+37                  22
000046          0111116124123               $(4)         +        0111116124123                  13
000047          0125102040072                            +        0125102040072                  13
000050          0040123125102                            +        0040123125102                  13
000051          0061041040040                            +        0061041040040                  13
000052          0000000000046R    4                      +        0000000000046                  13
000062   10  00  01 00 0 000052R    4       $(1)         LA       A1,FTEMP$+42                   13
000063   46  16  13 00 000000  X    9                    LXI,U    X11,BDICALL$+ARGC1$            13
RELATIVE   INSTRUCTION SUBFIELDS   U  FLD   LABEL        SYMBOLIC INSTRUCTION                  LINE
ADDRESS    F   J  A  X  HI   U       LC                  F,J      A,U,X                        NUMBER
000064   00  00  13 00 0 000000X    9                    IBJ$     X11,ARGC1$                     13
000065   10  16  04 00 000004               6G           LA,U     A4,4                           25
000066   01  00  04 00 0 000000R   12                    SA       A4,I                           25
000053          0777777777776               $(4)         +        0777777777776                  26
000054          0030400000000                            +        0030400000000                  26
000055          0000000000000                            +        0000000000000                  26
000056          0000000000000                            +        0000000000000                  26
000057          0000000000014R   10                      +        0000000000014                  26
000060          0060014000000                            +        0060014000000                  26
000061          0000000000000R   12                      +        0000000000000                  26
000062          0010004000000                            +        0010004000000                  26
000063   74  13  13 00 0 000000X    5                    LMJ      X11,FMTE$$                     26
000067   26  16  14 00 000053  R    4       $(1)         LXM,U    A0,FTEMP$+43                   26
000070   46  17  13 00 000000  X    6                    LXI,XU   X11,BDICALL$+F2SE$$            26
000071   00  00  13 00 0 000000X    6                   IBJ$      X11,F2SE$$                     26
000072   74  04  00 00 0 000057R    1                   J         $-11                           27
000014          0111116124123               $(10)       +         0111116124123                  28
```

```
000015          0125102072040                        +         0125102072040                        28
000016          0111040075040                        +         0111040075040                        28
                                                     END
F O R T R A N   S T O R A G E   M A P

NAME    TYPE       MODE      RELATIVE    LOC      ELEMENT  NUMBER OF    COMMON     PROGRAM
                             ADDRESS     COUNT    LENGTH   ELEMENTS     SIZE       UNIT

I       INTEGER    SCALAR    000000          0        4                            MAIN PROGRAM
I       INTEGER    SCALAR    000000          2        4                            SUBROUTINE SUB1
I       INTEGER    SCALAR    000000         12        4                            SUBROUTINE INTSUB : SUB1

Common blocks
C                  COMMON    000000         12                              1
        BLANK      COMMON    000000          2                              1

Entry points
FMAIN$             ENTRY     000000          1
SUB1               ENTRY     000033          1

External references
IBJ$       BDICALL$      BDIREF$      FMAIN$      FINIT$     FMTE$$      F2SE$$      SUB1      FEXIT$
ARGC1$     OLDCM$

END  FTN  59  IBANK  73  DBANK  2  COMMON
```

9.4.2.2.3. Cross-Reference Listing

The cross-reference listing contains all references to statement labels, subprograms, and variables that appear in the source program. Statement numbers appear first in the listing and are in numerical order. Symbolic names (variables, arrays, and so on) appear second and are in alphabetical order.

A separate cross-reference listing is generated for each program unit. A heading appears before each cross-reference listing, identifying the program unit. The heading has one of the following formats:

MAIN PROGRAM [*p*]

SUBROUTINE *n* [:*e*]

FUNCTION *n* [:*e*]

BLOCK DATA *b*

BLOCK DATA (DEFINED AT LINE *m*)

In these formats, *p* indicates the optional program name, *n* indicates the subprogram name, *e* (specified only if *n* is an internal subprogram) indicates the name of the external program unit (MAIN PROGRAM or external subprogram name), *b* indicates the BLOCK DATA program name, and *m* indicates the source line number where the unnamed BLOCK DATA program unit begins.

Column headings printed at the top of each page of the listing are:

NAME

Entity named in the program.

USE

Indication as to whether the entity is set, defined, specified, equivalenced, or used in a COMMON statement.

LINE NUMBER

The line numbers of the source statements where the entity is referenced. If an item appears in a specification statement in a FORTRAN procedure (named in an INCLUDE statement), then the line number is printed as *proc-name.proc-line-number*.

9.4.2.2.4. Object Code Listing

The L option listing produces an object program listing containing all instructions, data, and constants generated by the compiler for the program.

A heading appears before the object code listing of each program unit, identifying the unit. This heading is the same as that generated for the cross-reference listing.

At the top of each page of this listing, column headings are printed that refer to the fields under them:

RELATIVE ADDRESS

The relative address, in octal, of the memory location.

INSTRUCTION SUBFIELDS

Machine representation for F, J, A, X, H, I, and U instruction subfields. Letters R and X are also printed for the relocation information and the external reference information, respectively. This portion is suppressed, except the R and X, when the Y option is specified. For a detailed explanation of each subfield, see the *Series 1100 Assembly Instruction Mnemonics (AIM)*, *Supplementary Reference*, UP-9047.

U FLD LC

The location counter number applied to the U-field for relocation purposes (if R appears under INSTRUCTION SUBFIELDS), or the index to the external reference table applied to the U-field for external reference purposes (if X appears under INSTRUCTION SUBFIELDS).

LABEL

Shows the label associated with the storage location, if any. This label is composed of a decimal number and the letter L or G following the number. The letter L is used for a user-defined label, and the letter G is used for a compiler-generated label.

SYMBOLIC INSTRUCTION

Contains the symbolic representation of the object code shown under INSTRUCTION SUBFIELDS. It follows assembly language code conventions as closely as possible, with the F and J fields followed by the A, U, and X fields.

LINE NUMBER

Contains the source code line number that causes the instruction to be generated. Instructions moved by optimization can be associated with a previous statement label.

9.4.2.2.5. Storage Assignment Map

A storage map of entities defined in the source program is produced after the object code listings. Statement labels appear first in the listing and are in numerical order. Symbolic names (variables, arrays, and so on) are in alphabetical order.

The heading AUTOMATIC STORAGE SIZES is printed when automatic storage is specified through the COMPILER statement option DATA = AUTO or through the VIRTUAL statement with local variables. When local virtual variables are specified, the subheadings “Nonvirtual” and “Virtual” follow the main heading. The amount of storage for local virtual items that don’t appear in a SAVE statement in a program unit group appears under the subheading Virtual. Nonvirtual refers to compiler-generated data and to local variables that don’t appear in a SAVE statement when you specify the option DATA = AUTO.

A single storage map is generated for all program units.

At the top of each listing, a heading line identifying the fields of each line in the listing is printed:

NAME

Name used for the entity defined in the source program.

TYPE

An indication of what the entity identifies. Data entities are: integer, real, character, logical, and so on.

MODE

An indication of the way the entity is used (that is, scalar, array).

RELATIVE ADDRESS

Relative octal address (within the location counter) of the identifier. Its value is DUMMY when the item is a dummy argument or character function entry point name.

OFFSET

Byte number in decimal, where a character item begins (in the word defined by the relative address field). 0Q1, 1Q2, 2Q3, 3Q4.

LOC COUNT

Location counter in decimal, under which the symbol is allocated. (Not used for dummy arguments.)

ELEMENT LENGTH

Element length in decimal, of the entity in bytes.

NUMBER OF ELEMENTS

Decimal number of elements that an array entity contains.

COMMON SIZE

Size of the common block in words. (Used only for the common block listing; see 9.4.2.2.6.)

PROGRAM UNIT

Program unit the name is defined in.

ILLEGAL PAGE SIZES - 4 8 16 32 64 128

An *x* in the *n* column indicates that an illegal page span error will occur when the program is executed with page size *nk*. For a description of page size, see 8.5.9.

9.4.2.2.6. Common Block Listing

Following the storage map, the heading (common blocks) is printed. A list of all common blocks referenced in the source program is then printed.

The storage map column headings are used for the common block listing. TYPE is BLANK for blank common or VIRTUAL for a common block in virtual space. NAME identifies the name of a named common block.

Only one common block listing is generated, even if the source program contains more than one program unit.

9.4.2.2.7. Entry Point Listing

Following the common block listing, a heading (entry points) is printed. The list of entry points to the source program is then printed in the format NAME, MODE (always ENTRY), RELATIVE ADDRESS, and LOC COUNT. These are similar to the entities of the storage map listing.

Only one entry point listing is generated, even if the source program contains more than one program unit.

9.4.2.2.8. External References

The external reference listing follows the entry point listing. Under the printed heading (external references), the names of all external entry points referenced in the source program are listed. These include the names of FORTRAN subprograms and all the library and miscellaneous routines referenced.

Only one external reference listing is generated, even if the source program contains more than one program unit.

9.4.2.2.9. Static Virtual Storage Sizes

The static virtual storage sizes listing follows the external references listing for compilation units that specify:

- virtual local variables that don’t appear in a SAVE statement
- virtual common blocks

The line contains “Static virtual storage sizes:” followed by the amount of virtual common block storage and the amount of local D-bank storage for virtual local variables.

9.4.2.2.10. Termination Message

At the conclusion of the compiler listing, a termination line is printed. On this line, the total number of errors and warnings detected in the program and the amount of storage for I-bank, D-bank, and common blocks are printed. All fields are optional. If any field is zero, it isn't printed. The format is:

```
END FTN [n ERRORS] [m WARNINGS] [i NON-STD USAGES] [x IBANK]
[y DBANK] [z COMMON]
```

where:

n

is the number of errors detected.

m

is the number of warnings detected.

i

is the number of nonstandard usage messages printed (T option only).

x

indicates the number of words of storage used in the instruction bank.

y

indicates the number of words of storage used in the data bank, except for common blocks.

z

indicates the number of words of storage used in common blocks.

9.5. Calling the ASCII FORTRAN Compiler

The ASCII FORTRAN processor call format is:

```
@FTN [ ,option[option] ...] [si] [ , [ ro] [ , so] ]
```

where:

option

is a processor call letter option as described in Table 9-1. Options identified as SIR options control the source input routine. The listing options are discussed in more detail in 9.4.2.1.

si

is the name of the program file symbolic source element that contains the program to compile if the I option is not specified. If you don't specify the source element name, the FORTRAN source program must directly follow the processor call in the runstream. If the I option is specified, the FORTRAN source program must follow the processor call in the runstream. If *si* is present with I, it designates the program file symbolic element in which the source program is saved.

ro

is the name of the program file relocatable element that is to receive the compiled program. If *ro* is omitted, the file name and element name of *si* are used. If *si* is also omitted, the temporary file TPF$.NAME$ is used.

so

is the name of the program file symbolic element into which the updated source program will be placed. *so* is not used if the I or U option is specified or if *si* is not specified. If neither *so* nor U is specified and there are corrections, the corrected source is compiled but not saved.

Use the COMPILER statement to obtain additional control over the compilation process. (See 8.5.)

9.5.1. Processor Call Options

ASCII FORTRAN processor call options are listed in Table 9-1. The listing options are described in detail in 9.4.2.1.

Table 9-1. Processor Call Letter Options

Letter	Action
A	Don’t enter ERR mode on serious errors (that is, always perform ER EXIT$ to terminate the compilation). Without the A option, certain errors (such as I/O ERRORS) result in ERR$ termination.
B	In checkout mode, this option inhibits clearing of core before loading the user program.
C	Invokes FORTRAN checkout mode, generating code in core and executing it. (See 10.3.)
D	Print the storage map, common block listing, entry point listing, and external reference listing.
E	1110 code reordering. See 9.6.3.
F	Generate diagnostic tables under location counter 3 for use by walkback and the interactive PMD (see 10.4).
I	SIR option; source input is from the runstream.
K	List update and input (base) line numbers (if source lines are being printed), and print SIR correction lines.

continued

Table 9-1. Processor Call Letter Options (cont.)

Letter	Action
L	Generate all available listing output.
M	Search file FTN$PF first for INCLUDE statements.
N	If used with no other listing options, then list only source program error messages. If used with other listing options, then suppress printing of warning messages.
O	Generate code that allows D-bank addresses to exceed octal 0200000 (decimal 65,536).
P	Allows a character or Hollerith constant to be compared for equality to an integer or real item.
R	Print the cross-reference listing.
S	List only the source program and error and warning messages.
T	Flag items that are nonstandard (that is, not compatible with the American National Standard Programming Language FORTRAN, ANSI X3.9-1978) with a diagnostic. N option does not turn off these nonstandard usage messages. The flagger compiler always generates these messages unless you use the T option to turn the messages off.
U	SIR option; generate an updated cycle of the source input element.
V	Perform local code optimization. This option is ignored during checkout mode.
W	SIR option; list all SIR correction lines at the head of the printer listing.
X	Perform an ER EABT$ at the end of the compilation if the FORTRAN program had any errors or if any serious errors (such as I/O errors) occurred.
Y	List the generated code in pseudo-assembler format, without the octal representation of each word. Also print the listings generated by the D option. The L option overrides the Y option.
Z	Perform full optimization (when not in checkout mode). This includes the operations controlled by the V option. In checkout mode, this option turns on interactive debugging (see 10.3).

9.5.2. Execution of the Object Program

9.5.2.1. Execution Using Checkout

If a program or a portion of it is in one source element, it can be executed in checkout mode by putting the C or CZ options on the compiler call (see Table 9-1). If only a subprogram is tested, use the CZ options. Then, upon entering checkout debug mode, test the subprogram using the CALL and DUMP commands (see 10.3).

Extensive debugging facilities are available in interactive checkout debug mode.

9.5.2.2. Collection and Execution

If there are multiple source elements or if the absolute is to be executed more than once, collect an absolute with the collector (@MAP processor). First, the FORTRAN programs must be compiled.

Next, you must collect your program. If your site has the FORTRAN run-time library in a file that is automatically searched by the collector, you can do the following:

```
@MAP,SI MAPSOURCE,TALLY
IN TALLEY
@XQT TALLY
```

This assumes that all of your programs are in the standard temporary file TPF$.

If your site doesn’t have the FORTRAN run-time elements in a file that is automatically searched by the collector (see 9.5.3), you must specify the appropriate file in the collector symbolic using a LIB statement:

```
@MAP,FSI MAPSOURCE,TALLEY
LIB FTN*LIB.
IN TALLEY
@XQT TALLEY
```

If the collector generates truncation error messages during the collection, the program is too large to fit into the standard 65K addressing space. In this case, use the O option on all ASCII FORTRAN compilations. See Appendix H for a description of large programs and multibanking, or Appendix M for a description of the ASCII FORTRAN virtual feature.

For collected absolutes, the debugging features of the debug facility statements (see 10.2) and the run-time interactive PMD and walkback are available (see 10.4).

9.5.3. Using the Run-Time Library

Your site can have a type1 or type2 library for ASCII FORTRAN. A type1 library contains I/O routines that are accessed by an LMJ linkage from the ASCII FORTRAN-generated code. A type2 library does not contain any I/O routines; the I/O routines are in system common banks that are accessed by an LIJ linkage from compiler-generated code. The ASCII FORTRAN compiler generates an IBJ$ instruction linkage to all I/O routines; when this IBJ$ instruction mechanism is used, the collector decides whether to resolve the IBJ$ to an LMJ or LIJ linkage, depending on the type of library that is specified in the LIB statement in the collection.

The ASCII FORTRAN library is normally installed by COMUS in SYS$LIB$*FTN or manually installed in the file SYS$*RLIB$. The collector automatically searches these files for the relocatable elements needed by the compiler-generated code in your program. If the library was not installed in one of these files, you must specify the file name in a collector LIB statement, as in the example in 9.5.2.2.

The library also contains three symbolic elements that contain procedures that some programmers may wish to use. These elements are FTNPROC (MASM procedures to change the I/O and virtual environments; see Appendixes G and M), FIOP (FORTRAN procedure for the ERTRAN I/O packets; see 7.7.3.15), and INFO-PROC (MASM procedures for a FORTRAN-to-MASM interface; see 10.4.3.6 and Appendix K).

If the library has not been installed in SYS$*RLIB$, or the procedure elements not installed in SYS$LIB$*PROC$, the FTN and MASM processors are not able to find the procedures when they need them. In this situation, you need to use PDP to put a copy of the proper element in a file that is searched by the FTN or MASM processor (usually your source-input file). For example, if FORTRAN was installed in SYS$LIB$*FTN, and you want to use a procedure from the element FTNPROC, and you use TPF$ as your source file, use the following:

```
@PDP,M SYS$LIB$*FTN.FTNPROC,TPF$.FTNPROC
@EOF
```

9.6. Compiler Optimization

The compiler always performs the following computational optimizations in addition to its other functions (see 1.3.1):

- The reduction of expressions involving constants to a single constant (for example, 3.14159**2 is replaced by the single constant 9.8695877).
- The reordering of logical expressions to reduce the amount of time spent evaluating the expression (for example, IF(A.OR.B) GO TO 10 is replaced by IF(A) GO TO 10, and IF(B) GO TO 10).
- All available registers are used to hold intermediate results of calculations and reduce the number of references to slower storage.

Using processor options, you can direct the compiler to devote additional time to optimizing the FORTRAN statements before generating the relocatable binary object program from the object code. This is done at the expense of compilation speed, but the resulting output is usually executed significantly faster than without this additional optimization.

9.6.1. Local Optimization

At your option, the compiler partitions the FORTRAN program into basic blocks; that is, a sequence of statements with no entry or exit points interior to the sequence. In each of these blocks it performs the following additional optimizations:

- The evaluation of most expressions that are constant at compilation time. For example, the calculations of the following statement sequence are done at compile time rather than at execution time:

```
PARAMETER (J=1)        may be compiled as           I = 3
I = J + 2
```

- The elimination of a computation that, at execution time, is already computed in a previous statement, and replacement of the second computation by a reference to the temporary storage location or arithmetic register saving the result of the first computation. For example,

```
A = X + Y        may be compiled as          TEMP = X + Y
B = X + Y                                    A = TEMP
                                             B = TEMP
```

- The propagation of constants when a variable is assigned a constant value, to the places that the variable is used. For example,

```
J = 1            may be compiled as          J = 1
I = I + J + K                                I = I + 1 + K
```

- The elimination of unneeded references to storage, in a basic block. For example,

```
A = X + Y        may be compiled as          B = C + D
B = C + D                                    A = X + Y + Z
A = X + Y + Z
```

- The replacement of an operation with a faster, equivalent one. For example, J*2 is replaced by a shift operation.
- Simplification of more complex expressions. For example, the expression A**2 is replaced by A*A.

Specify local optimization by using the V option on the ASCII FORTRAN processor call (@FTN).

9.6.2. Global Optimization

At your option, the compiler performs an analysis of the interconnection patterns between basic blocks and performs the following additional optimizations:

- The elimination of redundant computations between basic blocks and the replacement of the second computation by a reference to the temporary storage location or arithmetic register which, at execution time, holds the result of the first computation.
- The elimination of unneeded stores to variables across the whole program unit. If the variable on the left side of an assignment statement is never used, the entire assignment statement is eliminated. Because evaluation of the right side of the assignment statement doesn't occur, any possible side effects of the eliminated statement will not occur.
- Movement of computations that are constant relative to a loop to a point outside the loop, and the replacement of the computation in the loop by references to the temporary storage location or arithmetic register that holds the result of the computation. (ASCII FORTRAN does not move out of loops portions of a computation that involve intrinsic functions.)
- Replacement of loop control variables, and expressions involving loop control variables by temporaries that are incremented on each pass through the loop.

- The maintenance of the result of computations and frequently used values in registers when execution crosses between blocks.

Global optimization is effected by specifying the Z option on the ASCII FORTRAN processor call (@FTN). For more information about optimization of register conflicts, see code reordering in 8.5.5.

9.6.3. Code Reordering

The E option on the ASCII FORTRAN processor call (@FTN,E) invokes a code reordering algorithm during the code generation phase of the compiler. This option is meaningful on hardware that incurs register conflicts (most hardware supporting ASCII FORTRAN). This option shuffles generated code, which minimizes the hardware register conflicts in order to obtain faster execution speeds.

Example of code reordering:

Source:

```
J = I
N = K + M
P = 0
```

Generated code:

```
1)   L    A4,I
2)   S    A4,J
3)   L    A6,K
4)   A    A6,M
5)   S    A6,N
6)   SZ   P
```

There is a major register conflict between lines 1 and 2 and between lines 4 and 5. A large delay occurs at each of the two points. After code reordering, the sequence is:

```
1)   L    A4,I
2)   L    A6,K
3)   A    A6,M
4)   SZ   P
5)   S    A4,J
6)   S    A6,N
```

This sequence is faster than the unordered code.

The algorithm used is a simple look-ahead that stops when labels, calls, etc., are encountered. In general, a decrease of 3% to 8% in central processing unit (CPU) time can be expected for execution of the generated code. However, I/O-bound programs see no change and a heavily looping program can execute up to 30% faster. The effect is most pronounced on programs that also are compiled with global optimization (the Z option).

This code-reordering algorithm can also be called by using the U1110=OPT COMPILER statement option. See 8.5.5.

9.6.4. Optimization Pitfalls

The optimizer cannot fully optimize expressions involving local variables shared between an internal subprogram and its external program unit. Therefore, to complete optimization of an element, be careful of which and how much data is shared by internal and external program units. The same situation exists with common blocks and arguments to subprograms. The optimizer operates on only one program unit at a time and can optimize operations on data local to the program unit most efficiently.

Example:

```
  X=22.
  CALL Y(Z)
  X=X+1.
```

In this situation, if X is a local variable, the optimizer can keep the value in a register across the CALL to Y, since no other program unit can access purely local variables not passed as arguments.

If X were in common or shared between the external program unit and any internal subprograms, this optimization could not be done since the procedure Y could conceivably change the value of X.

ASCII FORTRAN issues the following warning message when global optimization of an assigned GO TO statement is not possible.

```
2008  Presence of assigned GO TO inhibits global optimization
```

The presence of some assigned GO TO statements inhibit global optimization and result in only local optimization being performed. This situation is limited to only those assigned GO TO statements that transfer control to the head of a loop that is not a DO-loop.

The backward movement and strength reduction optimizations performed on a loop require an area (commonly referred to as an initialization block) to which operations that are invariant in the loop can be moved.

When transfers of control are made to the head of a non-DO-loop from outside the loop, optimization must create an initialization block inserted immediately before the head of the non-DO-loop. The transfers of control from outside the non-DO-loop are then modified to transfer control to the newly created initialization block.

Since the point to which an assigned GO TO statement is transferred depends on the contents of a variable that contains the address of any one of a list of labels, the modification cannot be made.

Conversion of the assigned GO TO statement to a computed GO TO statement allows global optimization.

9.6.5. Numerical Differences Caused by Optimization

A program compiled with optimization may produce different answers than one compiled without optimization. These differences can come from several sources, including:

- conversion of a division by a constant to a multiplication of the reciprocal of that constant (if the compiler was built without the INHIBIT DIVIDE OPTIMIZATION SGS). This can cause slight differences because the hardware floating point representation of the reciprocal is not exact.
- conversion of an exponentiation by certain integers to a series of multiplications. This can cause slight differences because the run-time algorithm computes exponentiation using a generalized method, not the series of multiplications.
- evaluation of some additional expressions at compile time rather than at run time. This can cause slight differences because all compile-time arithmetic is done in double precision (even when the corresponding run-time arithmetic would be single precision), and because compile-time double precision to single precision conversions are rounded and run-time conversions are truncated.
- some operations that replaced slower, mathematically equivalent ones give different results in the case of underflows, overflows, and other exceptions.

You should ignore small differences in the final results due to optimization. Large differences in final results may indicate that the program contains an algorithm that is too sensitive to the data (for example, the algorithm is operating at or near a mathematical “singularity”), or that an undetected exception is occurring (for example, division by zero), and that the normally insignificant differences due to optimization have become a significant portion of the final results.

9.7. Hints for Efficient Programming

Follow this list of suggestions to obtain the most efficient object programs from the ASCII FORTRAN compiler:

1. Simplify program structure as much as possible.

   Efforts in debugging and maintaining a FORTRAN program are greatly reduced, and more optimal code is produced by the compiler if unnecessary complexity is avoided. Individual statements should be simple enough to be easily understood. Loop nesting should be simple and straightforward with little inside branching. Program flow should be simple enough to minimize the number of paths through the program and make the purpose and method of the program easily understandable. Adhere to structured programming practices. Use the blocking statements (see 4.4) instead of GO TO statements whenever possible. Avoid unstructured methods such as extended range of DO-loops.

2. Optimization operates most efficiently on modularized, small program units. Limitations on table sizes in optimization affect the optimizations performed on a large program unit.

3. Although the compiler is designed to handle variables in COMMON or EQUIVALENCE statements in the most optimal way, the compiler must assume the worst possible cases when references are made to your subroutines or functions. Avoid any unnecessary use of COMMON or EQUIVALENCE statements to aid optimization.
4. Because of the additional complexity involved in addressing variables with over 65K addresses (the O option), this alternative should be used only for programs that actually exceed 65K. This particularly applies to arrays in very tight loops.
5. Since references to dummy arguments (which are not arrays) require indirect references to storage, it is best to assign such variables to local variables for use in the subprogram, especially within loops.
6. Excessive nesting of loops may require the loading and unloading of registers, adding to the execution time of the program, and should be avoided.
7. Because of the way logical expressions are optimized, you should try to order the logical expressions so that the most frequently satisfied comparison is performed first in a left-to-right sequence.
8. The initialization portion of a loop is assumed to be executed less frequently than the loop. When this assumption is violated, a FORTRAN program can take longer to execute with optimization than without.
9. The use of lists with ASSIGNED GO TO statements helps optimization by reducing the worst case assumptions that must be made without the lists. However, the list must include all labels that can be branched to at execution time or the compiler may produce incorrect code, and therefore, the execution of the program may produce incorrect results. Even with a complete list of labels, global optimization is inhibited when an assigned GO TO statement branches to the head of a loop that is not a DO-loop.
10. One portion of global optimization involves attempting to take advantage of the linear allocation of storage for multidimensional arrays. This involves the attempt to convert a data transfer operation in a simple loop into a single, linear operation referred to as a block transfer. A simple loop is defined as involving only the transfer of one array to another, or a constant into an array. To take advantage of this form of optimization, each array should be initialized or redefined within its own loop. Character arrays that start on word boundaries and whose element lengths are a multiple of 4 can also be optimized in this way.

9.8. Diagnostic System

The compiler contains a large number of self-explanatory diagnostic messages, some of which contain symbolic names from the source code. Diagnostic messages are printed adjacent to the source program statements that contain the errors or they are printed following the source statement listing if they pertain to the entire program unit.

Some messages are merely reminders to you and do not affect the generated code. This type of message is prefaced with the word WARNING when it is printed.

Other messages indicate more serious errors. A message prefaced by the word ERROR indicates that the code generated is possibly incorrect due to ambiguous or invalid usage of the source language. When an error is detected during a compilation, the relocatable binary element produced is marked as being in error. If the program is being run in batch mode, this allows some execution where execution is normally stopped, as in the case of detection of an I/O error.

Nonstandard usage messages occur only when you specify the T option. See 9.5.1 (Processor Call Options). (The flagger compiler always generates these messages unless you specify the T option.)

The format of an error/warning/nonstandard usage message is:

```
    ERROR
 *  WARNING          x [at line y] description
    NON-STD USAGE
```

where *x* is a code number indicating the type of error or warning. See Appendix D for details on these codes. *y* indicates a source line number or a line number within an included procedure. If it is the latter, the line is given in the form:

```
 proc.proc-line-number
```

where *proc* is the name of a procedure. (See 8.2.)

All diagnostic message descriptions appear in Appendix D.

Regardless of the type of errors detected in the program, all statement are scanned to detect syntactical errors. At the end of the compiler listing (in the END FTN message), the numbers of warnings, errors, and nonstandard usages detected (if any) are printed.

Section 10
Debugging a Program

10.1. Debugging Methods for ASCII FORTRAN

ASCII FORTRAN provides three facilities for debugging programs. Different problems and programs lend themselves to different debugging methods.

It is important to understand that the three debugging facilities are completely separate; in particular, commands for one debugging method do not apply to the others (except where commands are duplicated).

The three debugging facilities are:

1. The (noninteractive) debug facility (see 10.2.)
2. Checkout mode and the interactive checkout debugger (see 10.3.)
3. Walkback and the interactive Post Mortem Dump (PMD) (see 10.4)

The debug facility is most useful during program development. The interactive checkout debugger is probably the most powerful of the three, but can only be used on programs that can be run in checkout mode. Walkback and interactive PMD can be compiled into a production program and will be available when unexpected problems occur in a program.

10.2. Debug Facility Statements

The debug facility is used for debugging a program while you are creating it. Ordinarily, the debug facility is not a part of the final program unit.

The debug facility provides these debugging aids:

1. subscript checking
2. label tracing
3. tracing of changes in values
4. tracing of entry and exit for subprograms
5. simple output

The DEBUG statement sets the conditions for operation of the debug facility and designates debugging operations that apply to the entire program unit (such as subscript checking). The debug packet identification statement (AT) identifies the beginning of the debug packet and the point in the program unit at which the statements in the debug packet are to be executed. The three executable statements (TRACE ON, TRACE OFF, and DISPLAY) designate actions to be taken at specific points in the program unit.

Only one DEBUG statement can appear in a program unit; if a DEBUG statement appears, then any number (including zero) of debug packets can appear in the program unit. The AT, TRACE ON, TRACE OFF, and DISPLAY statements can't appear before the DEBUG statement. In the program unit, debug packets must be located after all regular code of the FORTRAN main program or subprogram, but preceding the END statement (or following the last statement in the program unit, if the program unit has no END statement). Any normal FORTRAN executable, data, or format statement can also occur in a debug packet. The debug packet is terminated by: (1) another AT statement, (2) the END statement for that FORTRAN program unit, or (3) the FUNCTION or SUBROUTINE statement that signifies the start of an internal subprogram.

The debug facility and compiler optimization (V or Z option on the ASCII FORTRAN compiler call statement) are incompatible. When used together, bad code could be generated, so instead, the compiler emits an error and turns off optimization.

The individual debug statements, which are all nonstandard, are explained in detail in the following subsections.

10.2.1. DEBUG

Purpose:

The DEBUG statement indicates the existence of a debug facility for the given FORTRAN program unit and specifies the debugging environment.

Form:

```
DEBUG [option [,option] ... ]
```

where *option* is any of the debugging environment specifications:

- UNIT (See 10.2.1.1.)
- SUBCHK (See 10.2.1.2.)
- TRACE (See 10.2.1.3.)
- INIT (See 10.2.1.4.)
- SUBTRACE (See 10.2.1.5.)

Description:

Any combination of the debugging environment options can appear in the option list following the DEBUG keyword. They can be in any order.

There must be a single DEBUG statement for each program unit to be debugged and it must immediately precede the first debug packet, if one exists.

If you don't specify the UNIT option, any debugging output is put in the standard program output file.

If you omit the TRACE option from the DEBUG option list, there is no display of program flow by statement labels in the program unit.

Examples:

```
      DEBUG
C           Indicates debugging is enabled.  Debug action is specified
C           in an associated AT statement.  Output is put in
C           the standard system output file.

       DEBUG SUBTRACE,UNIT(4),SUBCHK(ARRAY1,BUNCH2,GROUP3),INIT
C           Subscripts are checked for arrays ARRAY1, BUNCH2,
C           and GROUP3.  Changes in values of all variables are
C           noted.  Debug output is put on unit number 4.

       DEBUG TRACE,INIT(C,LIST1,E),SUBCHK
C
C           Debugging will include subscript checking on all arrays,
C           list of program flow by statement label passage (assuming
C           a TRACE ON statement appears in an AT packet following the
C           DEBUG statement), and notation of changes in value of C,
C           LIST1, and E.
```

10.2.1.1. UNIT

Purpose:

The DEBUG statement UNIT option designates a particular output file for debug information.

Form:

```
UNIT(c)
```

where *c* is an integer constant representing a unit specifier.

Description:

All debugging output goes to the designated file.

The unit specifier can't change in an executable program; for example, if the FORTRAN main program specifies UNIT(8), a subprogram called by this main program must specify UNIT(8) if it has a DEBUG statement.

When this option is absent, all debugging output is put in the standard output file. No DEBUG UNIT message is printed in this case.

Example:

```
      DEBUG UNIT (25)
C           Sends all debug output to the file associated with unit
C           25.
```

10.2.1.2. SUBCHK

Purpose:

The DEBUG statement SUBCHK option checks the validity of subscripts of array elements referenced in the program unit.

Form:

```
SUBCHK [(n [,n]...)]
```

where each *n* is an array name.

Description:

If the list of array names is not given following the SUBCHK option, subscript checking is done for all arrays in the program unit.

The check is made by comparing the size of the array with the product of the subscripts. A message (listing the source code line number and array name) is placed in the debug output file if an out-of-range subscript expression is encountered. The incorrect subscript is still used in the continued program execution.

When this option is omitted, no subscript checking is performed.

Subscript checking can’t be done for assumed-size arrays (see 2.4.4.1).

Examples:

```
DEBUG SUBCHK

DEBUG SUBCHK(ARRAY1, LIST2)
```

10.2.1.3. TRACE

Purpose:

The DEBUG statement TRACE option indicates that you want statement label tracing in the FORTRAN program unit in which the DEBUG statement appears.

Form:

```
TRACE
```

Description:

This option only enables label tracing. Tracing is not actually performed until a TRACE ON statement is encountered in the program flow. It is terminated on encountering a TRACE OFF statement (see 10.2.3 and 10.2.4).

TRACE ON and TRACE OFF statements have no effect on a program unit in which the TRACE option is not specified.

Example:

```
      DEBUG TRACE
C             The trace debug facility is enabled.
C             It can be activated with a TRACE ON statement.
```

10.2.1.4. INIT

Purpose:

The DEBUG statement INIT option traces the changes in values of variables and arrays during execution.

Form:

```
INIT [(m [,m]...)]
```

where *m* is the name of a variable or array in the program unit for which a value trace is performed.

Description:

When no list is given after the INIT option, a value trace is done on every variable or array in the program unit. This includes changes in value of any particular element of an array.

The value trace consists of placing, in the debug output file, a display of the variable name or array element name along with its new value each time it is assigned a value in an assignment statement, a READ statement (except a namelist READ), a DECODE statement, or an ASSIGN statement. The source code line number where the variable is set is also listed.

Example:

```
      DEBUG INIT (A,VAR1)
C             Starts debug facility and initiates trace of array
C             A and variable VAR1.
```

10.2.1.5. SUBTRACE

Purpose:

The DEBUG statement SUBTRACE option indicates entrance and exit of a subprogram during program execution.

Form:

```
SUBTRACE
```

Description:

When you include the SUBTRACE option in the DEBUG statement within a function or subroutine, a trace on entrance to and exit from that subprogram is enabled.

The message ENTER SUBPROGRAM *s* (where *s* is the subprogram entry point name) is placed in the debug output file each time *s* is entered, and RETURN FROM SUBPROGRAM *s* is inserted in the file each time *s* completes execution.

Example:

```
DEBUG SUBTRACE
```

10.2.2. AT

Purpose:

The AT statement identifies the beginning of a debug packet and indicates the point in the program unit at which the packet is to be activated (that is, the point at which the statements in the debug packet are to be executed).

Form:

```
AT s
```

where *s* is a statement label of an executable statement in the program unit to be debugged.

Description:

There must be one AT statement for each debug packet. Each AT statement indicates the beginning of a new debug packet. The end of the debug packet is indicated by: (1) an END statement, (2) the FUNCTION or SUBROUTINE statement that signifies the start of an internal subprogram, or (3) another AT statement.

The statements in the debug packet are executed whenever the statement associated with statement label *s* is executed in the program flow. They are executed immediately prior to the execution of *s*.

Example:

```
          .
          .
          .
        DEBUG
        AT 100
        DISPLAY X, Y, A
        END
C               The DISPLAY statement is executed each time the statement
C               with label 100 is executed.
```

10.2.3. TRACE ON

Purpose:

The TRACE ON statement initiates display of the flow of execution by statement label.

Form:

```
TRACE ON
```

Description:

After TRACE ON is encountered and until the next TRACE OFF is encountered, a record of the associated statement label is placed in the debug output file each time a labeled statement is executed in the program.

TRACE ON remains in effect through any level of subprogram call or return. If the TRACE option isn't used on a DEBUG statement in a particular program unit, label trace doesn't occur during execution of that program unit.

TRACE ON can occur anywhere in a debug packet.

There can be no display of program flow by statement label in this program unit if the TRACE option is omitted from the DEBUG option list.

Example:

```
      DEBUG TRACE, INIT(A,B)
       .
       .
       .
      AT 104
      TRACE ON
C           The flow of execution is displayed starting at
C           statement 104.
```

10.2.4. TRACE OFF

Purpose:

The TRACE OFF statement terminates statement label tracing.

Form:

```
TRACE OFF
```

Description:

TRACE OFF can occur anywhere in a debug packet. This statement terminates tracing of program flow by statement label in the program.

Example:

```
        .
        .
        .
       DEBUG TRACE, INIT(A,B)
        .
        .
        .
      AT 104
      TRACE ON
        .
        .
        .
      AT 950
      TRACE OFF
C           The flow of execution is displayed from the point
C           where statement 104 is executed to the point where
C           statement 950 is executed.
```

10.2.5. DISPLAY

Purpose:

The DISPLAY statement provides a simple debug output mechanism.

Form:

```
DISPLAY list
```

where *list* is a series of variables, array element names with constant subscripts, or array names separated by commas. A dummy argument name of a function or subroutine isn't permitted in *list*.

Description:

The DISPLAY statement is equivalent to the following FORTRAN statements:

```
NAMELIST /name/ list
WRITE (n,name)
```

where *list* is as defined above, *name* is a unique name generated for DISPLAY, and *n* is the debug unit specifier (from the UNIT option). DISPLAY provides a simple means of putting results of debugging operations for the program unit in the debug output file without needing FORMAT, NAMELIST, or WRITE statements. The output to the debug output file is in NAMELIST format. The DISPLAY statement can appear anywhere in a debug packet.

Example:

```
    AT 100
    DISPLAY A,B,C,D(1,2),E
C          The values of variables A, B, C, and E and array
C          element D(1,2) are listed each time before
C          statement 100 is executed.
```

10.2.6. Debug Facility Example

The following example uses all of the debug facility statements (including all of the DEBUG statement options). It includes a main program without a DEBUG statement and an external subroutine (A) with a DEBUG statement and three AT packets.

FORTRAN source:

```
   CALL A
    END

    SUBROUTINE A
    DIMENSION B(4)
    DATA I, K /2, 5/
5   B(I) = 2.
10  J = B(2)
15  L = B(K)
20  RETURN
    DEBUG UNIT(6), SUBCHK, TRACE, INIT, SUBTRACE
    AT 5
    TRACE ON
    DISPLAY I, K
    AT 15
    TRACE OFF
    AT 20
    DISPLAY J, B
    END
```

Program execution:

```
(a) DEBUG UNIT      6
(b) ENTER SUBPROGRAM A
(c) TRACE ON
(d)   $0001
(e) I =    2,K =      5
(f) $END
(g) TRACE     5
(h) AT LINE      6:  ELEMENT          2 OF B           =        2.0000000
(i) TRACE     10
(j) AT LINE      7:J      =              2
(k) TRACE OFF
(l) AT LINE      8:  SUBSCRIPT IS OUT OF RANGE FOR ARRAY B
(m) AT LINE      8:  L     =              0
(n)   $0002
(o) J =      2,
(p) B = .00000000     ,     .20000000+001,   .00000000  ,     .00000000
(q) $END
(r) RETURN FROM SUBPROGRAM A
```

The lines printed out during the preceding execution are a result of the following options and statements:

UNIT option - line a.

SUBCHK option - line l.

TRACE option and TRACE ON, TRACE OFF, and AT statements - lines c, g, i, k.

INIT option - lines h, j, m.

SUBTRACE option - lines b, r.

DISPLAY and AT statements - lines d - f, n - q.

10.3. FORTRAN Checkout Mode

You can use the ASCII FORTRAN compiler as a compile-and-go processor by calling the checkout mode of operation.Add the C option to the processor call command (see 9.5). This directs the compiler to generate code into main storage and immediately execute it when compilation is complete. This mode results in increased throughput in cases where the object program is to be executed only once and when execution is relatively short. No relocatable element is produced.

FORTRAN checkout mode also provides a powerful interactive debugging system. Add the Z option in addition to the C option to the processor call command (see 9.5) to enable this feature. By combining the C and Z options, you can trace the execution, halt the execution, dump variable values, and perform other debugging activities.

Since optimization is not practical for programs that are executed only once and since it interferes with interactive debugging, the optimization features of ASCII FORTRAN are disabled in checkout mode.

Although no relocatable element is produced, a write-enabled RO file (or SI file if no RO is specified) must be available, since an omnibus element is produced under certain circumstances when in checkout mode.

10.3.1. Calling Checkout Mode

A checkout run is entered in much the same manner as a normal compilation. However, you must follow special procedures to enter data for the executing program or to call the debugging features provided. Enter a simple execution as follows:

```
@FTN,CI file.element
program
@EOF
data images
```

The *program* consists of your main program (which must physically be first) and any necessary subprograms (internal and external subroutines, internal and external functions, or BLOCK DATA subprograms). An END statement (see 4.9) must terminate each program unit group. As in noncheckout mode, the only variables shared among external program units are arguments passed between program units and variables defined in common blocks.

On encountering the @EOF control statement, all program units are processed as one program, and execution is initiated following compilation. The *data images* following the @EOF control statement are read and used, as necessary. The program can be read in from a file by omitting the I option. The @EOF control statement is still required, however, preceding the data images in order to indicate that no corrections are to be applied to the file.

If the Z option is used to enable debugging, enter the following commands:

```
@FTN,CIZ file.element
program
@EOF
GO
data images
```

The GO command is required since the debugging routine allows entry of debugging commands immediately preceding the execution of the program. These commands can be entered before the GO command.

10.3.2. Interactive Debug Mode in the Checkout Compiler

10.3.2.1. Entering Interactive Debug Mode

You can only enter the interactive debug mode in the checkout compiler when the Z option is specified on the checkout processor call (@FTN,CZ). It is entered:

- Before the first executable statement in the FORTRAN program. Debug mode is automatically entered at this point.
- When a contingency interrupt occurs during execution of the FORTRAN program (see 10.3.4).
- When the FORTRAN program executes the statement CALL PAUSE. PAUSE has no parameters.
- When execution of the FORTRAN program reaches the END statement of the main program (that is, just before termination of the program).

  Entry into debug mode at this point lets you execute debug commands to do the following: dump the final values of variables (see DUMP command, 10.3.3.4), restore execution to a previous state (see RESTORE command, 10.3.3.11), or dump the final contents of the program (see SNAP command, 10.3.3.15). The following message is printed before debug mode is entered:

  ```
  END PROGRAM EXECUTION
  ```

- When the special RESTART processor (FTNR, see 10.3.6) is called to reenter a previous debugging session. For example:

  ```
  @FTNR ROFILE.MYPROG
  ```

- When a breakpoint interrupt occurred in the previous FORTRAN statement. The SETBP command (see 10.3.3.14) is used to set the breakpoint register, which causes hardware breakpoint interrupts. The following message is printed on entry to debug mode, where *n* is the current source line number:

  ```
  SETBP BREAK AT LINE n
  ```

- When a step break is set at the current statement using the STEP command (see 10.3.3.16). The following message is printed on entry to debug mode, where *n* is the current source line number:

  ```
  STEP BREAK AT LINE n
  ```

- When a line number break is set at the current statement using the BREAK command (see 10.3.3.1). The following message is printed on entry to debug mode, where *n* is the current source line number:

  ```
  BREAK AT LINE n
  ```

- When a statement label break is set at the current statement using the BREAK command (see 10.3.3.1). The following message is printed on entry to debug mode, where *n* is the statement label associated with the current statement:

  ```
  LABEL BREAK AT nL
  ```

  Another line follows the above message, stating which program unit in the FORTRAN program contains the label.

- When the subprogram called by the CALL command (see 10.3.3.2) returns. The following message is printed on entry to debug mode:

  ```
  ENTER DEBUG MODE (RETURN FROM CALL COMMAND)
  ```

For the first five cases, the following message is printed on entry to debug mode, where *n* is the source line number of the statement at which execution in the FORTRAN program is interrupted:

```
ENTER DEBUG MODE AT LINE n
```

10.3.2.2. Soliciting Input

When the checkout compiler is in interactive debug mode, commands are solicited with ER ATREAD$. The solicitation message is:

```
C:
```

If the command is read from an @ADD stream, if the @FTN program began execution in @BRKPT mode, or if the program is executing in batch mode, then the command image is printed.

To leave debug mode, use the GO, EXIT, and CALL commands. The debug commands are discussed individually in 10.3.3.

10.3.3. Debug Commands

You can abbreviate all debug command names with one letter (the initial one), except for the following (with their abbreviations in parentheses): CALL (CA), LINE (LIN), SAVE (SA), SETBP (SETB), SNAP (SN), and STEP (ST).

The following syntax rules apply for the debug commands:

- No blank characters are allowed inside a field of a debug command.

  The only exception to this rule occurs when a character variable is specified in the *v* subfield of the first field of the SET command. In this case, blanks can appear inside apostrophes in the character constant in the *c* field.

  This rule applies when a command contains a *p* subfield. This subfield, if specified in a command, is part of the first field of the command. Therefore, no blanks can appear before or after the slash (/) that separates the *p* subfield from the previous subfield, or before or after the colon (:) separator in the *p* subfield.

- Any number of blank characters (or none) can appear between fields.
- The *v* (variable name) subfield in the DUMP, SET, and SETBP commands must be one of the following:
  - scalar variable name (including a function subprogram entry point name)
  - array element name (with constant subscripts)
  - array name (DUMP command only)

  A scalar or array subprogram argument is specified by using one of these forms.

  The variable *v* must appear in an executable statement in the designated program unit *p*, unless *p* is the main program or a block data program. If the COMPILER statement option DATA=AUTO or DATA=REUSE appeared in the program, then *v* must be a variable appearing in a common block or a SAVE statement.
- The *p* (program unit) field in the PROG command and the *p* subfield in the DUMP, SET, SETBP, BREAK, CLEAR, and GO commands has the following format:

```
progname[:extname]
```

where:

progname

represents the desired program unit in the ASCII FORTRAN program. It can be specified as (1) * (to represent the main program), (2) a FORTRAN program unit (main program, subroutine, function, or BLOCK DATA subprogram) name, or (3) an unsigned positive integer *n* (to represent the *n*th unnamed BLOCK DATA subprogram in the FORTRAN source program).

extname

represents the program unit name of the external program unit corresponding to the internal subprogram *progname*. Therefore, *extname* can be specified only if *progname* is specified as a subprogram name that represents a FORTRAN internal subprogram. The variable *extname* can be specified as: (1) asterisk (*) (if the external program unit is the main program), or (2) a FORTRAN program unit (main program, subroutine, or function) name.

If *progname* is specified as a program unit name and *extname* is not specified, then the external program unit with name *progname* is taken. If no such external program unit exists, then the first internal subprogram with name *progname* is taken.

Examples of the *p* format:

*

main program

3

third unnamed BLOCK DATA subprogram in the source

SUB1

external program unit SUB1 (or, if no external program unit SUB1 exists in the program, the first internal subprogram in the source with name SUB1)

INT1:SUB2

internal subprogram INT1, whose external program unit is SUB2

INT2:*

internal subprogram INT2, whose external program unit is the main program

The checkout debug commands are described individually in the following subsections.

10.3.3.1. BREAK

Purpose:

The BREAK checkout debug command (abbreviated B) sets break points at statement labels or source line numbers.

Form:

```
BREAK n [L [/ p ] ] [ , n [ L [ / p ] ] ] ...
```

where:

n

is an unsigned positive integer.

p

represents a program unit in the FORTRAN source program. The *p* subfield is described under syntax rules (see 10.3.3).

Description:

The BREAK command specifies labels or a line number as points at which execution of the FORTRAN program is interrupted and interactive debug mode entered. These points are known as breakpoints.

If the *n* L [/*p*] format is used, then the breakpoint is the beginning of the FORTRAN statement with statement label *n*, where *p* designates the program unit in which *n* resides. If *p* is not specified, then the breakpoint is label *n* in the default program unit . (See PROG command, 10.3.3.10.)

If the *n* format is used, then the breakpoint is the beginning of the FORTRAN statement at source line number *n*.

Set a maximum of eight label breaks and eight line number breaks at any one time.

Two other debug commands are used in connection with the BREAK command. The CLEAR command clears one or more breakpoints. The LIST command lists all breakpoints.

Example:

```
BREAK 10L/SUB1, 9
```

Set breakpoints at (1) statement label 10 in program unit SUB1, and (2) source line 9. Debug mode is reentered at these points. The command GO should follow so that the program resumes execution.

10.3.3.2. CALL

Purpose:

The CALL checkout debug command (abbreviated CA) calls a FORTRAN subprogram.

Form:

```
CALL s[ (a[,a] . . . ) ]
```

where:

s

is a subprogram entry point name. *s* has the following format:

```
ent[:extname]
```

where:

ent

is the entry point called; *ent* may be any entry point in any subprogram in the FORTRAN program, except for an alternate entry point (that is, an entry point specified in an ENTRY statement) in an internal subprogram.

extname

represents the program unit name of the external program unit corresponding to the internal subprogram *ent*. Therefore, *extname* can be specified only if *ent* is specified as an internal subprogram name. The variable *extname* can be specified as: (1) asterisk (*) (if the external program unit is the main program), or (2) a FORTRAN program unit (main program, subroutine, or function) name.

a

is an actual argument that is passed to the subprogram. *a* must be specified in one of the following forms:

- A FORTRAN constant.
- A variable in program unit *p*, where *p* is the default program unit (set by the PROG command). It must be specified as a scalar variable name, an array name, or an array element name (with constant subscripts).
- A subprogram entry point name, immediately preceded by an asterisk (*). The name *a* can be any entry point in any subprogram in the FORTRAN program, except for an alternate entry point (that is, an entry point specified in an ENTRY statement) in an internal subprogram. If the entry point specified exists as an external subprogram entry point, then that one is taken. If no such external entry point exists, then the first internal subprogram with the specified name is taken.

Description:

The CALL command calls a FORTRAN subprogram with the given arguments. This lets you test only a given subprogram without having to execute the entire FORTRAN program. For example, a subprogram can be repeatedly called with different sets of arguments.

If *extname* is not specified, then the external subprogram entry point with name *ent* is taken. If no such external subprogram entry point exists, then the first internal subprogram with name *ent* is taken.

Each *a* is an actual argument and must match the corresponding dummy argument of *s* in type and usage. (In this way, the CALL command closely resembles a subprogram reference in a FORTRAN program.)

A statement label can't be passed as an actual argument via the CALL command. Therefore, a subprogram with any RETURN *i* statements (that is, a subprogram with * as any dummy argument) can't be called with this command.

Use a maximum of 20 arguments.

When the subprogram returns (by the RETURN statement), control transfers back to interactive debug mode. The following message is printed:

```
ENTER DEBUG MODE (RETURN FROM CALL COMMAND)
```

In addition, if the subprogram called is a function, the following message is printed, followed by the actual value:

```
FUNCTION VALUE RETURNED:
```

Examples:

```
CALL FUNC1(A(1,1), 1.6E4, V)
```

Refer to function FUNC1, passing as arguments array element A(1,1) (from the default program unit), the real constant 1.6E4, and variable V (also from the default program unit). After FUNC1 returns control, the function value is printed, and debug mode is reentered.

```
CALL SUB1(5, *SUB2)
```

Call subroutine SUB1, passing as arguments the integer constant 5 and subprogram entry point SUB2. After SUB1 returns control, debug mode is reentered.

When debug mode is reentered on return from the CALL command, you can't resume normal execution of the program (at the line where the CALL command was executed) using the GO command. Instead, use the SAVE and RESTORE commands for this purpose, since the CALL command interrupts normal execution.

For example, when you wish to execute a portion of the program, interrupt execution to test subprogram SUB (using the CALL command), and then resume normal execution of the program, enter the following commands:

```
SAVE
CALL SUB
RESTORE
```

10.3.3.3. CLEAR

Purpose:

The CLEAR checkout debug command (abbreviated C) clears breakpoints set by the BREAK and SETBP commands.

Form:

where:

n

is an unsigned positive integer.

p

represents a program unit in the FORTRAN source program. The *p* subfield is described under syntax rules (see 10.3.3).

k

is one of the following keywords: LABEL, LINE, BRKPT, or ALL.

Description:

The CLEAR command clears one or more breakpoints established by the BREAK or SETBP commands.

The *n* [L [/*p*]] format is the same as in the BREAK command. The format *n* L [/*p*] clears the breakpoint set at statement label *n* in program unit *p*. The format *n* clears the breakpoint at line *n*.

The rest of the formats clear one or both break lists or the SETBP break. CLEAR LABEL clears all label breakpoints. CLEAR LINE clears all line number breakpoints. CLEAR BRKPT clears the breakpoint register set by the SETBP command. CLEAR and CLEAR ALL clear all label and line number breaks and the SETBP break.

You can abbreviate LABEL, LINE, BRKPT, and ALL to LA, LI, B, and A, respectively.

Example:

```
CLEAR 10L/SUB1, 9
```

Clear the breakpoints (previously set by the BREAK command) at (1) statement label 10 in program unit SUB1, and (2) source line 9.

10.3.3.4. DUMP

Purpose:

The DUMP checkout debug command (abbreviated D) prints the values of FORTRAN variables.

Form:

```
DUMP [, opt]   {  v [/p] [,v[/p]] . . .   }
               {  /p                      }
               {  !                       }
```

where:

opt

is an option letter; A (ASCII) or O (Octal) are allowed.

v

is a variable in program unit *p*. The *v* subfield is described under syntax rules (see 10.3.3).

p

represents a program unit in the FORTRAN source program. The *p* subfield is described under syntax rules (see 10.3.3).

Description:

The DUMP command prints the current values of one or more FORTRAN variables.

When you use the O option on the DUMP command, the values are printed in octal format. If the A option is specified, they are printed in ASCII character format. If neither O nor A is specified, they are printed in a format corresponding to the variable's data type (INTEGER, REAL, COMPLEX, and so on).

When the value of a variable is printed, it follows a heading line in the format:

```
v   /p
```

where *v* is the variable name and *p* is the program unit name. (See the description of the *p* subfield in 10.3.3.)

If an entire array is dumped, the values of all elements in the array are printed in column-major order.

The formats are described as follows:

- DUMP [,*opt*] *v*[/*p*] [, *v* [/*p*]] . . .

  The *v*[/*p*] format prints the value of variable *v* in program unit *p*.

  If *v* is a scalar, array element, or function entry point, then one value is printed. If *v* is an array name, then the values of all elements in the array are printed.

  If *p* is not specified, then variable *v* is taken from the default program unit (set by the PROG command).

- DUMP [,*opt*] /*p*

  This format prints the values of all variables in program unit *p*.

- DUMP [,*opt*] !

  This format prints the values of all variables in all program units in the FORTRAN program.

When the second or third formats are used, the order that the variables appear in the output is as follows:

- In a program unit, the variables appear in alphabetical order.
- In the FORTRAN program (format 3), the program units appear in the order that they appear in the source input.

```
DUMP !
```

Print the values of all variables in all program units.

```
DUMP /SUB2
```

Print the values of all variables in program unit SUB2.

```
DUMP,O A/*, B(1,1), C/SUB3
```

Print the values of variables A (from the main program), B(1,1) (from the default program unit), and C (from program unit SUB3), all in octal format.

10.3.3.5. EXIT

Purpose:

The EXIT checkout debug command (abbreviated E) terminates the ASCII FORTRAN processor.

Form:

```
EXIT
```

Description:

The EXIT command terminates the processor with a call to the FEXIT$ system routine. This routine terminates all input/output and does an ER EXIT$.

An @ image (control statement) read in checkout debug mode is treated like an EXIT command (that is, the processor is terminated).

10.3.3.6. GO

Purpose:

The GO checkout debug command (abbreviated G) resumes execution of an ASCII FORTRAN program.

Form:

```
GO [nL [/p]]
```

where:

n

is an unsigned positive integer.

p

represents a program unit in the FORTRAN source program. The *p* subfield is described under syntax rules (see 10.3.3).

Description:

The GO command causes an exit from interactive debug mode; execution of the FORTRAN program is then resumed.

When GO is specified (that is, no command fields), execution of the program continues at the point at which it was interrupted to go into debug mode.

If the *n*L [/*p*] format is used, execution of the program continues at statement label *n* in program unit *p*. If *p* isn't specified, the default program unit (set by the PROG command) is assumed.

Be cautious when specifying the *n*L [/*p*] format, since registers may not be set up correctly when jumping to a statement label. For instance, jumping to a label inside a DO-loop or jumping to a label in another program unit (that is, not the one currently being executed) can cause execution problems.

Examples:

```
GO
```

Resume execution of the FORTRAN program at the point at which it was interrupted to go into debug mode.

```
GO 15L/S1
```

Resume execution of the program at statement label 15 in program unit S1.

10.3.3.7. HELP

Purpose:

The HELP checkout debug command (abbreviated H) prints information about debug commands.

Form:

$$\texttt{HELP}\left[\texttt{[,opt]}\begin{Bmatrix}\texttt{ALL}\\ \textit{cmd}\end{Bmatrix}\right]$$

where:

opt

is an option letter; F (formats) or D (descriptions) are allowed.

cmd

is one of the checkout debug command names. Don't use abbreviations.

Description:

The HELP command prints information about debug commands. This lets you continue debugging without having to consult a manual about command descriptions or formats.

The format HELP lists all of the debug command names.

The format HELP *cmd* prints all available information about the designated debug command *cmd*, including a list of all formats, a description of the individual items specified in the formats, and a general description of the command.

The format HELP ALL lists all available information for all debug commands. A large amount of output is generated.

When you specify the F option, only command formats are printed.

When you specify the D option, only command descriptions are printed.

Examples:

```
HELP
```

List all of the debug command names.

```
HELP ALL
```

List all information for all debug commands.

```
HELP,F DUMP
```

List all formats for the DUMP command.

10.3.3.8. LINE

Purpose:

The LINE checkout debug command (abbreviated LIN) prints the current source line number.

Form:

```
LINE
```

Description:

The LINE command prints the source line number of the statement in the FORTRAN program where execution is interrupted to go into debug mode.

10.3.3.9. LIST

Purpose:

The LIST checkout debug command (abbreviated L) lists all breakpoints set by the BREAK command and the default program unit.

Form:

```
LIST
```

Description:

The LIST command lists all breakpoints set by the BREAK command. This includes all line number breaks and all statement label breaks.

The default program unit (set by the PROG command) is also listed.

10.3.3.10. PROG

Purpose:

The PROG checkout debug command (abbreviated P) sets the default program unit for variables and statement labels.

Form:

```
PROG p
```

where *p* represents a program unit in the FORTRAN source program. The *p* field is described under syntax rules (see 10.3.3).

Description:

The PROG command sets the default program unit in the FORTRAN symbolic element that is implied for variables (in the DUMP, SET, SETBP, and CALL commands) and statement labels (in the BREAK, CLEAR, and GO commands) to *p*.

If no PROG command is entered in debug mode during execution of the FORTRAN program, then the first program unit in the FORTRAN symbolic element is set as the default.

The default program unit set by this command may be overridden in an individual command (DUMP, SET, SETBP, GO, BREAK, or CLEAR) by specifying a program unit subfield *p* in that command.

The LIST command prints the default program unit.

Examples:

Assume the following commands are entered sequentially:

```
PROG SUB1
```

Set program unit SUB1 as the default program unit.

```
DUMP X
```

Print the value of variable X in the program unit SUB1.

```
DUMP X/2
```

Print the value of variable X in the element's second unnamed BLOCK DATA subprogram.

```
BREAK 10L
```

Set a break at statement label 10 in program unit SUB1.

```
BREAK 10L/*
```

Set a break at statement label 10 in the main program.

10.3.3.11. RESTORE

Purpose:

The RESTORE checkout debug command (abbreviated R) restores your program to a previous point of execution.

Form:

```
RESTORE [n]
```

where *n* is an integer consisting of 1 to 12 digits.

Description:

The RESTORE command restores the state of your program that a previous corresponding SAVE command preserved (see 10.3.3.12). It essentially restarts your program at the state it was in at the SAVE point. The optional version number *n* can be used to preserve several stages of execution while debugging.

When reentering your program, the following message is printed:

```
ENTERING USER PROGRAM: prog-name [ VERSION: version-no ]
```

Note: *You are responsible for the assignment of files and their positioning. File contents, assignments, and positioning (tapes) are not saved or restored. Only your variables and point of execution are saved and restored, along with several debug mode parameters: breakpoints set by the BREAK and STEP commands, the default program unit set by the PROG command, and the trace mode value set by the TRACE command. Also, the same level of ASCII FORTRAN must be used to do the corresponding SAVE command.*

You can reenter a checkout debugging run at a later date by use of the special restart processor (FTNR) released with ASCII FORTRAN. All that is necessary is that the relocatable output file from the previous session still be available. (This is where the SAVE information is saved; see 10.3.3.6.)

Examples:

```
@FTN,SCZ IN.ELT
@EOF
```

State is automatically saved in omnibus element IN.ELT before entry to debug mode.

```
BREAK 7
GO
```

ASCII FORTRAN responds with BREAK AT LINE 7.

```
SAVE 2
```

State saved in omnibus element IN.ELT/2.

```
RESTORE
```

State restored from omnibus element IN.ELT.

ASCII FORTRAN responds with ENTERING USER PROGRAM: ELT.

```
BREAK 3
GO
```

Your program now restarts execution from the original save point. ASCII FORTRAN responds with BREAK AT LINE 3.

```
RESTORE 2
```

State restored from omnibus element IN.ELT/2.

ASCII FORTRAN responds with ENTERING USER PROGRAM: ELT VERSION: 2.

```
GO
```

Your program resumes execution at line number 7, where the save was done.

```
@FTN,CZ IN.GAMES,SAVE.GAMES
       .
```

```
    .
    .
@EOF
```

The state is automatically saved in SAVE.GAMES before entry to debug mode.

```
GO
    .
    .
    .
```

Your program executes.

```
    .
    .
    .
@FIN
    .
    .
    .
```

Next day you want to do more testing on GAMES.

```
@RUN SMITH,123456,TRNG
@FTNR SAVE.GAMES
```

FTNR responds with a sign-on line and the state of GAMES is restored from yesterday's SAVE.

```
GO
    .
    .
    .
```

Your GAMES program now executes again.

10.3.3.12. SAVE

Purpose:

The SAVE checkout debug command (abbreviated SA) saves the present state of your program for later resumption.

Form:

```
SAVE [n]
```

where *n* is an integer consisting of 1 to 12 digits.

Description:

The SAVE command saves the present state of your program by writing it out to an omnibus element in a Relocatable Output (RO) file. The element name used is the RO element name. The version name used is either your RO version, if any, or the up to 12-digit field on the SAVE command.

Use only an all-digit field on the SAVE command. Since the element created is typed as omnibus, this command doesn't destroy your symbolic, relocatable, or absolute elements of the same name in your RO file.

The RESTORE (see 10.3.3.11) command can restore the FORTRAN program to the state of execution of a corresponding SAVE command.

An automatic SAVE command is done for you just before initially entering debug mode (after the END FTN message).

Examples:

```
@FTN,SCZ IN.ELT
@EOF
SAVE
```

This saves the present state of your program in omnibus element IN.ELT. The RESTORE command can restore the current state.

```
@FTN,CZ IN.ELT,OUT.ELT/TEST
@EOF
SAVE
```

This saves the present state of your program in omnibus element OUT.ELT/TEST. The RESTORE command can restore the current state.

```
@FTN,CZ IN.ELT,OUT.ELT/TEST
@EOF
SAVE 99
```

This saves the present state of your program in omnibus element OUT.ELT/99. The RESTORE command 99 can restore the current state.

10.3.3.13. SET

Purpose:

The SET checkout debug command (abbreviated S) changes the value of a FORTRAN variable.

Form:

```
SET v [/p] = c
```

where:

v

is a variable in program unit *p*. The *v* subfield is described under syntax rules (see 10.3.3).

p

represents a program unit in the FORTRAN source program. The *p* subfield is described under syntax rules (see 10.3.3).

c

is a FORTRAN constant.

Description:

The SET command sets the value of variable *v* in program unit *p* to the constant *c*. If *p* isn't specified, then *v* is from the default program unit (set by the PROG command).

The variable *c* must be the same data type as *v*. There are no conversions between data types for the SET command. For example, if *v* is declared as type COMPLEX*16 in program *p*, then *c* must be a double-precision complex constant.

When *v* is a character variable, *c* must be a character constant. Hollerith constants are not allowed.

Examples:

```
SET I=5
```

Set the value of variable I (in the default program unit) to 5. I must be type integer.

```
SET CD(2,3)/S2 = (1.4D2, 2.3D3)
```

Set the value of array element CD(2,3) in program unit S2 to the COMPLEX*16 constant (1.4D2,2.3D3). Array CD must be type COMPLEX*16.

10.3.3.14. SETBP

Purpose:

The SETBP checkout debug command (abbreviated SETB) sets a breakpoint so that debug mode is reentered when a specific variable is set or referred to.

Form:

```
SETBP[,opt] v[/p]
```

where:

opt

is an option letter; R or W are allowed.

v

is a variable in program unit *p*. The *v* subfield is described under syntax rules (see 10.3.3).

p

represents a program unit in the FORTRAN source program. The *p* subfield is described under syntax rules (see 10.3.3).

Description:

The SETBP command sets a breakpoint so that debug mode is reentered whenever the designated FORTRAN variable, *v*, in program unit *p*, is set or referenced during execution of a FORTRAN program. If *p* is not specified, *v* is taken from the default program unit (set by the PROG command).

Only use the SETBP command when execution is on a machine where the ER SETBP$ mechanism is available. This Executive Request sets the programmable breakpoint register, which causes a breakpoint interrupt whenever the specified condition is met. Checkout debug mode is reentered at the beginning of the next executable FORTRAN statement after the specified variable is set or referenced.

If the R option is specified on the SETBP command, then debug mode is reentered whenever the designated variable *v* is referenced from storage. This occurs when the variable is referenced in an assignment statement (on the right side of the assignment [=] operator) or an I/O write statement.

If the W option is specified, then debug mode is reentered whenever the variable *v* is stored into. This occurs when the variable is set in an assignment statement (on the left side of the assignment [=] operator) or an I/O read statement.

If neither the R nor the W option is specified, then both are assumed; that is, debug mode is reentered whenever the variable is set or referenced.

A breakpoint interrupt occurs when the storage that the variable occupies is involved in a load (R option) or store (W option) instruction. Therefore, the FORTRAN statement where the breakpoint interrupt occurs (that is, the executable statement immediately preceding the statement where debug mode is reentered) may not actually reference the variable name specified in the SETBP command; the interrupt may be caused by the setting or referencing of a variable that occupies the same storage as the variable designated in the command. Variables that may overlap in storage in a FORTRAN program include those used in EQUIVALENCE or COMMON statements or those passed as subprogram arguments.

The breakpoint set by the SETBP command remains in effect during execution until it is cleared by the CLEAR command (either CLEAR or CLEAR BRKPT format), or until you set another breakpoint with the SETBP command. You can only set one SETBP breakpoint at a time.

Examples:

```
SETBP,W C(6)/*
```

Set a breakpoint so that debug mode is reentered whenever array element C(6) in the main program is set. The command GO should follow so that the program resumes execution.

```
SETBP,R V
```

Set a breakpoint so that debug mode is reentered when variable V (in the default program unit) is referred to.

```
SETBP A/S2
```

Set a breakpoint so that debug mode is reentered when variable A in program unit S2 is set or referred to.

10.3.3.15. SNAP

Purpose:

The SNAP checkout debug command (abbreviated SN) produces a dump of the FORTRAN program or registers.

Form:

```
SNAP [l]
```

where *l* is one of the following letters: I, D, or R.

Description:

The SNAP command dumps all or part of the FORTRAN program. The Executive Request ER SNAP$ dumps the contents of one location counter at a time. Your program's registers can also be dumped.

The SNAP format dumps the entire FORTRAN program and all registers.

The SNAP I format dumps the contents of location counter 1 of the program. $(1) contains all program instructions not resulting from input/output lists.

The SNAP D format dumps the contents of all location counters except 1.

The SNAP R format dumps all registers.

Since ER SNAP$ lists absolute addresses only, an L option checkout compiler listing of the FORTRAN program is helpful when you wish to decode the location counter information that is dumped. This listing includes the location counters and relative addresses for variables and code in the program.

10.3.3.16. STEP

Purpose:

The STEP checkout debug command (abbreviated ST) sets a breakpoint at a certain point ahead in the program.

Form:

```
STEP [n]
```

where *n* is an unsigned positive integer.

Description:

The STEP command specifies a breakpoint at which execution of the FORTRAN program is interrupted and interactive debug mode is reentered.

After the GO command is entered, *n* FORTRAN statements are executed and then debug mode is reentered.

If *n* is omitted, 1 is assumed.

Example:

```
STEP 3
```

Set a breakpoint so that debug mode is reentered after three FORTRAN statements are executed. The command GO should follow so that the program resumes execution.

10.3.3.17. TRACE

Purpose:

The TRACE checkout debug command (abbreviated T) turns trace mode on or off.

Form:

```
TRACE [m]
```

where *m* is one of the following keywords: ON or OFF.

Description:

Either TRACE or TRACE ON turns trace mode on. TRACE OFF turns trace mode off.

If trace mode is on, then a message in the form:

```
LINE n
```

is printed at the start of execution of each FORTRAN statement, where *n* is the source line number of the statement. Trace mode is initially off.

10.3.3.18. WALKBACK

Purpose:

The WALKBACK checkout debug command (abbreviated W) traces the general flow of program execution through FORTRAN subprograms.

Form:

```
WALKBACK
```

Description:

The WALKBACK command gives a step-by-step trace of FORTRAN subprogram references that have occurred during program execution. The trace begins at the current statement in the subprogram that is executing (that is, the point in your program at which execution was interrupted to go into debug mode) and ends at the main program.

During execution of the walkback trace, one line is printed at each step indicating which FORTRAN subprogram (subroutine or function) was referenced at a certain line number of another subprogram (or the main program). Walkback can occur over any number of subprograms.

Any FORTRAN subprogram named in a walkback message is a main entry point, regardless of which entry point in that subprogram is actually referenced.

One or more of the following messages is printed during the checkout walkback process:

```
WALKBACK INITIATED AT ADDRESS absadr IN USER PROGRAM

THIS ADDRESS IS AT LN. line OF prog2

prog1 REFERENCED AT LN. line OF prog2

THIS ADDRESS IS AT elt $(lc) reladr

prog1 REFERENCED AT elt $(lc) reladr

prog1 REFERENCED AT ADDRESS absadr BDI bdi

THIS ADDRESS IS IN THE I/O COMPLEX

X11 LINK ADDRESS DESTROYED - WALKBACK TERMINATED

WALKBACK TERMINATED BECAUSE OF AUTO. STORAGE

prog1 REFERENCED BY CALL COMMAND
```

Example:

```
1.      I = 5
2.      CALL S(I)
3.      END

4.      SUBROUTINE S(I1)
5.      J = F(I1)
```

```
6.     PRINT *,J
7.     RETURN
8.     END

9.     FUNCTION F(I2)
10.    F = I2**3
11.    RETURN
12.    END
```

Execution of the following three checkout debug commands:

```
BREAK 11
GO
WALKBACK
```

causes the following lines to be printed:

```
WALKBACK INITIATED AT ADDRESS 031470 IN USER PROGRAM
THIS ADDRESS IS AT LN.    11 OF F
F     REFERENCED AT LN.     5 OF S
S     REFERENCED AT LN.     2 OF MAIN PROGRAM
```

See 10.4.3 for more walkback examples and a more complete description of the walkback mechanism. That subsection describes FTNWB, the run-time walkback routine, which is very similar to the checkout walkback process.

10.3.4. Contingencies in Checkout Mode

When an illegal operation (IOPR), guard mode (IGDM), or error mode (EMODE, including a math or I/O library error) contingency interrupt occurs during checkout execution of your program, the following action is taken (after a contingency message is printed), depending on the options specified on the @FTN control statement:

- C option (but no F or Z option)

  A walkback trace is performed (from the point of the error back to the main program). See 10.3.3.18 for a description of the walkback mechanism.

- CF options (but no Z option)

  (1) A walkback trace is performed, and (2) the current values of all variables in your program are printed (that is, the automatic 'DUMP !' command).

- CZ options

  (1) A walkback trace is performed, and (2) interactive debug mode is entered (so that you can find the problem). Execution of the program may not be resumed after a contingency of this type.

When a break keyin (@@X C) occurs during checkout execution of your program, the following action is taken (depending on the @FTN options):

- C option (but no Z option)

  No action is taken.

- CZ options

(1) The message BREAK KEYIN is printed; (2) the current FORTRAN statement completes execution (so that debug mode is not entered in the middle of a statement); and (3) debug mode is entered. Execution of your program resumes with the GO checkout command.

When a divide fault (IDOF), floating-point overflow (IFOF), or floating-point underflow (IFUF) contingency interrupt occurs during checkout execution of the program, the following is performed (if the appropriate run-time counter is positive): (1) a contingency message is printed, (2) the line number and program unit where the contingency occurred is listed, and (3) execution of the program is resumed. See subsections on service subprograms DIVSET (7.7.3.9), OVFSET (7.7.3.8), and UNDSET (7.7.3.7) for a description of how the run-time counters are set. All three counters are initialized to 20 on CZ options, and to 0 otherwise.

10.3.5. Checkout Mode Restrictions

Due to the generation of code in main storage, only simple program structure is provided in checkout mode. Links cannot be generated to subprograms that are not physically in the source program. In addition, multibanking and segmentation are not provided (that is, the BANK statement and the COMPILER statement with options BANKED=DUMARG, BANKED=ACTARG, BANKED=RETURN, or BANKED=ALL present cannot be used). The maximum size for your program in checkout mode is somewhat smaller than the maximum size for a program in noncheckout mode, due to the addressing space used by checkout execution-time requirements. A warning message is printed if this maximum size is exceeded.

The error diagnostics associated with the checkout mode appear in Appendix I.

10.3.6. Restart Processor (FTNR)

FTNR is a small, separate processor, released with ASCII FORTRAN, which restarts previous checkout debugging sessions. It is called with the SI field holding a file-element name that is an omnibus-type element holding one of your runs saved from a previous session. (See SAVE command, 10.3.3.12.) The processor signs on in a standard manner and goes to interactive mode after reloading the program.

Example:

```
▶@FTNR   SAVEFILE.MYPROG
▶FTNR IIRI 01/12/84  17:20:41
ENTERING USER PROGRAM: MYPROG
```

If you give a version name on the FTNR call statement, this element/version is the base level that is restored by a parameterless RESTORE command.

Example:

```
▶@FTNR F.GAME/A
▶FTNR IIRI 01/12/84 17:20:41
ENTERING USER PROGRAM: GAME VERSION: A
▶RESTORE
```

The RESTORE command in this example restores GAME/A again.

All checkout debugging commands are available in the FTNR (restart) processor. You must use the same level of FTN and FTNR, or the restart doesn't work. This is true of the RESTORE command also (see 10.3.3.11); the same level of FTN and FTNR must do the corresponding SAVE.

10.4. Walkback and the Interactive FTNPMD

When a contingency interrupt occurs during execution of an ASCII FORTRAN program, the following debugging aids are automatically executed:

- the ASCII FORTRAN walkback process (FTNWB)
- the ASCII FORTRAN interactive postmortem dump (FTNPMD)

The walkback mechanism gives a step-by-step trace of FORTRAN subprogram references that occur during program execution, from the point of the error condition back to the main program. Only subprograms in relocatable elements generated by ASCII FORTRAN compilations or in MASM elements using the walkback procedures (see 10.4.3.6) are traced by the walkback process.

FTNPMD lets you interactively dump the current values of FORTRAN variables in the executing program. FTNPMD won't dump variables that are in virtual space. The variables that you can dump are those that exist in elements generated with F-option ASCII FORTRAN compilations. If optimization is used, the values provided for the variables may not be the same as the value you expect. This difference is the result of the elimination of unneeded stores. If the program is running in batch mode (see 10.4.4.1), FTNPMD is executed only if you specify the F option on the @XQT control statement.

In addition to being called by the contingency routine, both FTNWB and FTNPMD can be initiated by calls from your program.

If the walkback and interactive PMD debugging aids are to execute correctly, neither the Z nor R option should be specified on the @MAP control statement used for collection of the program. The @MAP,Z statement suppresses generation of diagnostic tables in the absolute element. The @MAP,R statement generates a collector relocatable element; no @MAP,R relocatables should appear in your program.

In addition, the collector directive TYPE EXTDIAG should be used if any FORTRAN subprogram main entry points are unreferenced in the program. This statement causes all entry points (referenced or not) to be inserted in the collector's diagnostic tables.

If the program is multibanked, it must follow the specifications listed in Appendix H, since both FTNWB and FTNPMD make assumptions about the program's banking structure.

FTNPMD and FTNWB do not execute correctly if you change the name of a relocatable element produced by ASCII FORTRAN with the @CHG or @COPY control statement.

10.4.1. Diagnostic Tables Generated by ASCII FORTRAN

When you specify the F option on the ASCII FORTRAN processor control statement (@FTN,F), the compiler produces a large set of diagnostic tables (also known as INFO-010 text) under location counter 3 of the generated relocatable element.

When you don't specify the F option on the @FTN control statement, a smaller set of diagnostic tables (containing only the program unit information) is generated. These tables only allow walkback through the program units in the element, with no line numbers appearing in the walkback messages.

When the relocatable element is mapped into an absolute element, the collector handles the INFO-010 location counter (3) in a special manner. The contents of $(3) are inserted in the absolute element, but are never part of the executing program. Therefore, the size of the diagnostic tables does not affect the size of the loaded program. FTNWB and FTNPMD read necessary INFO-010 tables from the absolute element to local buffers.

For the FTNPMD and FTNWB routines to get mapped in an absolute element, at least one @FTN,F relocatable must be mapped in.

10.4.2. Initiating FTNWB and FTNPMD

Both walkback and the interactive PMD are automatically executed (in that order) if any of the following contingency interrupts occur:

- Error detected by the math library
- Error detected by the I/O library (see G.9)
- Illegal operation (IOPR)
- Guard mode (IGDM)
- Error mode (EMODE)

No registers are destroyed in FTNWB or FTNPMD. Therefore, when the registers are dumped by the ER EABT$ (which is performed by the contingency routine after walkback and the interactive PMD have completed), the contents of all registers are the same as when the error occurred.

The interactive PMD (but not walkback) is executed:

- When a break keyin (@@X C) contingency occurs during execution of the program
- When the program executes the FORTRAN statement CALL FTNPMD (the routine has no parameters)

Walkback (but not the interactive PMD) occurs when the program executes the FORTRAN statement CALL FTNWB (the routine has no parameters). In this case, the walkback process begins at the point of the call and traces back to the main program. Program execution continues at the statement after the call.

If the program is executing in checkout mode (see 10.3), then neither FTNWB nor FTNPMD can be executed by any of the methods mentioned previously.

For FTNPMD to execute in batch mode in the above cases, specify an F option on the @XQT control statement. Walkback doesn't require a special @XQT option for batch mode operation.

When you specify the F option on ASCII FORTRAN compilations (@FTN,F), normal line numbers appear in the walkback messages; otherwise, all line numbers are printed as 0 (the subprograms' names are correct, however).

10.4.3. Walkback (FTNWB)

During execution of the walkback process, one line is printed at each step indicating that FORTRAN subprogram (subroutine or function) is referenced at a certain line number of another subprogram or the main program. Walkback can occur over any number of subprograms.

The walkback process terminates when the trace reaches (1) the main program or (2) a routine that is in an element that was not generated by an ASCII FORTRAN compilation. The latter case includes ASCII FORTRAN service subprograms (see 7.7.3) and routines in MASM elements.

When you specify the F option on ASCII FORTRAN compilations (@FTN,F), normal line numbers appear in the walkback messages; otherwise, all line numbers are printed as 0 (the subprograms' names are correct, however).

In the examples listed in the following subsections, it is assumed that all ASCII FORTRAN compilations are performed with the F option and that a type2 (banked I/O) library file is used as a LIB file during collection.

Walkback may not work correctly in segmented programs.

10.4.3.1. Errors Detected by the Mathematical Library (CML)

The mathematical library (common bank or relocatable) detects the following errors:

- Unnormalized argument
- Argument value out of range
- Function value out of range

If any of these errors are detected during execution, messages are printed, listing the following:

- The mathematical routine that detected the error
- The nature of the error
- The decimal and octal representations of the argument that caused the error

- A walkback trace of your subprogram references, from the call that referenced the mathematical library, back to the main program (if possible). If the CMLSET service subprogram (see 7.7.3.10) is called (and the CMLSET run-time counter is positive), a one-step walkback is performed (listing the line number and program unit where the mathematical library was called), and program execution is resumed.

Example:

ASCII FORTRAN source input:

```
1.      CALL S1
2.      END

3.      SUBROUTINE S1
4.      CALL S2(-4.)
5.      RETURN
6.      END

7.      SUBROUTINE S2(X)
8.      Y=SQRT(X)
9.      RETURN
10.     END
```

Program execution:

```
ERROR TERMINATION IN SQRT         ROUTINE CAUSED BY
ARGUMENT UNORMALIZED OR OUTSIDE ALLOWABLE RANGE
ARG1=        -4.0000000
ARG1 OCTAL  574377777777
SQRT      REFERENCED AT ABSOLUTE ADDRESS 007740 BDI 000004
THIS ADDRESS IS AT LN.     8 OF S2
S2   REFERENCED AT LN.     4 OF S1
S1   REFERENCED AT LN.     1 OF MAIN PROGRAM

***** ENTER FTN PMD *****

-> (enter FTNPMD commands)
```

10.4.3.2. Errors Detected by the I/O Library

When a fatal error is detected during execution of any I/O library (common bank or relocatable) routine, messages are printed, listing the following:

- The nature and address of the error
- The I/O common bank or I/O relocatable element where the address resides
- A walkback trace of your subprogram references, from the call that referenced the I/O library, back to the main program (if possible).

Example:

ASCII FORTRAN source input:

```
1.        CALL S1
2.        END

3.        SUBROUTINE S1
4.        CALL S2
5.        RETURN
6.        END

7.        SUBROUTINE S2
8.        DIMENSION A(2)
9.        PRINT *,A(100000)
10.       RETURN
11.       END
```

Program execution:

```
GUARD MODE       ERR-CODE: 02
ERROR ADDRESS:  007603        BDI: 500025
THIS ADDRESS IS IN COMMON I/O BANK C2F$
I/O      REFERENCED AT LN.     9 OF S2
S2       REFERENCED AT LN.     4 OF S1
S1       REFERENCED AT LN.     1 OF MAIN PROGRAM

***** ENTER FTN PMD *****

- (enter FTNPMD commands)
```

When a nonfatal I/O error is detected, messages are printed, listing the following:

- The nature of the error
- The line number and program unit where the error occurred

Program execution then continues.

Example:

ASCII FORTRAN source input:

```
1.       CALL S1
2.       END
3.       SUBROUTINE S1
4.       READ (5,10) I
5. 10    FORMAT (I12)
6.       RETURN
7.       END
```

Program execution (assuming the character string ‘A’ is read by I/O):

```
FTN ERR ON UNIT-5   INPUT DATA DOES NOT CORRESPOND TO TYPE
I/O    REFERENCED AT LN.     4 OF S1
```

10.4.3.3. Errors Detected in Your Program

If an error (illegal operation, guard mode, or error mode) is detected during execution of your routine, then messages are printed, listing the following:

- The nature and address of the error
- A walkback trace of your subprogram references, from the subprogram where the error occurred, back to the main program (if possible)

Example:

ASCII FORTRAN source input:

```
1.         CALL S1
2.         END

3.         SUBROUTINE S1
4.         CALL S2
5.         RETURN
6.         END

7.         SUBROUTINE S2
8.         DIMENSION A(2)
9.         A(100000) = 2.
10.        RETURN
11.        END
```

Program execution:

```
GUARD MODE      ERR-CODE: 02
ERROR ADDRESS:  003120       BDI: 000004
THIS ADDRESS IS AT LN.         9 OF S2
S2   REFERENCED AT LN.         4 OF S1
S1   REFERENCED AT LN.         1 OF MAIN PROGRAM
***** ENTER FTN PMD *****

- > (enter FTNPMD commands)
```

If a divide fault, floating-point overflow, or floating-point underflow contingency interrupt occurs during execution and the appropriate run-time counter is positive, messages are printed, listing the following:

- The nature of the error
- The line number and program unit where the error occurred

Program execution then resumes.

See subsections on service routines DIVSET (7.7.3.9), OVFSET (7.7.3.8), and UNDSET (7.7.3.7) for a description of how the run-time counters are set.

Example:

ASCII FORTRAN source input:

```
1.        CALL S1
2.        END

3.        SUBROUTINE S1
4.        CALL DIVSET( 5 )
5.        A = 0.
6.        B = 1. / A
7.        RETURN
8.        END
```

Program execution:

```
WARNING: DIVIDE FAULT
AT LN.     6 OF S1
```

10.4.3.4. FTNWB Routine Call

If routine FTNWB is called during the execution of your program, messages are printed, listing the following:

- The address from which FTNWB is called
- A walkback trace of your subprogram references, from the subprogram that called FTNWB, back to the main program (if possible)

Example:

ASCII FORTRAN source input:

```
1.         CALL S1
2.         END

3.         SUBROUTINE S1
4.         CALL S2
5.         RETURN
6.         END

7.         SUBROUTINE S2
8.         CALL FTNWB
9.         RETURN
10.        END
```

Program execution:

```
FTNWB CALLED AT ADDRESS 003116 BDI 000004
THIS ADDRESS IS AT LN.            8 OF S2
S2      REFERENCED AT LN.         4 OF S1
S1      REFERENCED AT LN.         1 OF MAIN PROGRAM
```

10.4.3.5. Walkback Messages

One or more of the following messages is printed during the walkback process:

```
FTNWB CALLED AT ADDRESS absadr BDI bdi

THIS ADDRESS IS AT LN. line OF prog2

prog1 REFERENCED AT LN. line OF prog2

THIS ADDRESS IS IN THE I/O COMPLEX

THIS ADDRESS IS AT elt $(lc) reladr

prog1 REFERENCED AT elt $(lc) reladr

prog1 REFERENCED AT ADDRESS absadr BDI bdi

X11 LINK ADDRESS DESTROYED - WALKBACK TERMINATED

AUTO. STORAGE ELEMENT ENCOUNTERED - WALKBACK TERMINATED
```

The values inserted in these messages are described as follows:

absadr

absolute address (octal)

bdi

index of the bank in which *absadr* resides (octal)

line

source line number of a designated FORTRAN statement (decimal)

prog2

one of the following:

- MAIN PROGRAM (if the walkback process reaches the last step of a successful trace)
- an external subprogram name
- *i*:*e*, where *i* is an internal subprogram name and *e* represents *i*'s external program unit. The variable *e* is either MAIN PROGRAM or an external subprogram name.

prog1

one of the following:

- I/O (to represent the I/O library complex)
- an external subprogram name
- *i*:*e* (see *prog2*)

elt

an element name

lc

a location counter in element *elt* (decimal)

reladr

a relative address under location counter *lc* (octal)

Any FORTRAN subprogram named in a walkback message is a main entry point, regardless of which entry point in that subprogram is actually referenced.

The last five walkback messages listed above are printed only when the trace reaches a termination point that isn't in the main program.

10.4.3.6. Walkback Procedures for MASM Subprograms

There are two MASM procedures in the ASCII FORTRAN library, F$EP and F$INFO (in element INFO-PROC, see 9.5.3). These procedures can be referenced in your MASM element that is generating a subprogram used in an ASCII FORTRAN system. If these procedures are used correctly, ASCII FORTRAN's run-time walkback mechanism properly handles the MASM subprogram.

10.4.3.6.1. F$EP

The form of the F$EP call line is:

```
F$EP  sub
```

where *sub* is a field of six Fieldata characters (left-justified, blank-filled) denoting the subprogram name that is externalized by the procedure.

F$EP generates some epilogue and prologue code in the MASM subprogram, namely:

- Epilogue: restore registers and return to the caller of the subprogram. The return is done via the IBJ$ return mechanism (see 8.5.4); that is, a jump or LIJ is performed, depending on the contents of H1 of X11.
- The subprogram entry point *sub* (F$EP externalizes the entry point, so *sub* must not appear as a label in your MASM code).
- Prologue: save registers (including X11).

The prologue-epilogue code is the same as that generated by an @FTN,O compilation (that is, over-65K D-bank code). Volatile registers A2 (prologue) and A4 (epilogue) are destroyed in the generated code.

10.4.3.6.2. F$INFO

The form of the F$INFO call line is:

```
F$INFO  elt
```

where *elt* is a field of 12 Fieldata characters (left-justified, blank-filled) denoting the relocatable element name from the MASM control statement.

F$INFO generates the following:

- The jump back up to the epilogue code (generated by F$EP)
- The INFO-010 diagnostic text (readable by the walkback routines in the ASCII FORTRAN library)

The INFO-010 tables are never loaded at run time along with your program's I-bank and D-bank. Instead they are in the absolute element's diagnostic tables, which are read in by ASCII FORTRAN's run-time walkback routines. The tables contain information that allows these routines to determine the program unit and line number for walkback messages.

10.4.3.6.3. Description

If procedures F$EP and F$INFO are used, only one subprogram can appear in an element (that is, each of the two procedures should be referenced only once per MASM assembly).

The MASM element which references the two procedures and which contains the externalized subprogram SUB1 appears as follows:

```
@MASM,IS ELT1,ELT2
   AXR$
      .
      .
      .
   F$EP  'SUB1 '
 $(1)
      .
      .   $(1) subprogram code except for epilog
      .   (restore registers and return to caller),
      .   prolog (save registers), and jump to epilog.
      .   Note that subprogram label (SUB1 in this
      .   case) is externalized by procedure F$EP.
      .   It should not appear as a label in the
      .   user's $(1) code.
      .
      .
      .
      .
```

```
     F$INFO 'ELT2  '
     END
```

The epilogue and prologue code generated by these two procedures does not handle arguments (passed to the subprogram) or functions (that is, the MASM subprogram as a function). You can process these items in the $(1) subprogram code, if applicable. Arguments should be handled after the F$EP reference. If the subprogram is a function, then A0 (and possibly A1, A2, and A3, depending on the function type) should be loaded with the function result before the F$INFO call.

You should not use $(3) or $(4) in a MASM element referencing procedures F$EP and F$INFO, since these location counters are used by the two procedures.

A MASM error (E-flag) results when you have an incorrect reference to one of the two procedures. Possible errors include:

- One of the procedures referred to more than once
- F$INFO referred to without a previous reference to F$EP
- $(3) or $(4) used in your code
- More than one field with one subfield passed to either procedure
- Parameter passed to either procedure is not a Fieldata string
- The length of the Fieldata string passed to one of the procedures is incorrect (six for F$EP, 12 for F$INFO)

If the element containing the two procedures (ASCII FORTRAN library element INFO-PROC, marked as type ASMP) has a PDP performed on it by you, an M option must appear on the PDP call statement. (See 9.5.3 to find INFO-PROC.)

Example:

```
@MASM,IS TEST
     AXR$
     F$EP   'SUB1 '
 $(1)
     LX,U    A0,0
     LXI,U   X11,0
     LMJ     X11,SUB2
     F$INFO  'TEST  '
     END

@FTN,ISF MAIN

 1.   CALL SUB1
 2.   END

 3.   SUBROUTINE SUB2
 4.   CALL FTNWB
 5.   END
```

If these two relocatable elements are collected into an absolute element, the resulting program is executed as follows:

```
FTNWB CALLED AT ADDRESS 066431 BDI 000004
THIS ADDRESS IS AT LN.      4 OF SUB2
SUB2   REFERENCED AT LN.    0 OF SUB1
SUB1   REFERENCED AT LN.    1 OF MAIN PROGRAM
```

The line number in a MASM subprogram is always 0 in a walkback message.

10.4.4. Interactive Postmortem Dump (FTNPMD)

10.4.4.1. Soliciting Input

When execution is in interactive PMD mode, commands are solicited with ER TREAD$. The solicitation message is -.

If the command is read from an add stream (@ADD) or if the program began execution in breakpoint mode (@BRKPT), the command image is printed.

The following message is printed on entry to interactive PMD mode:

```
***** ENTER FTN PMD *****
```

To leave interactive PMD mode, use the EXIT command (see 10.4.4.2.2).

If the program is executing in batch mode, then PMD mode does not go interactive. Instead, if the F option is specified on the @XQT control statement, the two commands DUMP ! and EXIT are automatically executed. FTNPMD is not executed in batch mode if the F option was not specified.

10.4.4.2. PMD Mode Commands

You can abbreviate both of the interactive PMD commands described below to one letter (the initial one).

10.4.4.2.1. DUMP

Purpose:

The DUMP command (abbreviated D) prints the values of FORTRAN variables.

Forms:

```
DUMP [,opt] [v]/[p]/e
```

or

```
DUMP [,opt] !
```

where:

opt

is an option letter; A (ASCII) or O (octal) is allowed.

v

is the name of a FORTRAN variable in program unit *p* in element *e*. It must be one of the following:

- scalar variable name (including a function subprogram entry point name)
- array name
- array element name (with constant subscripts)

You can specify a scalar or array subprogram argument by using one of the these forms.

The variable *v* must appear in an executable statement in *p*, unless *p* is the main program or a BLOCK DATA subprogram. If the ASCII FORTRAN compilation for *e* had a COMPILER statement option DATA=AUTO or DATA=REUSE, then *v* must be a variable in a COMMON block or a SAVE statement.

p

represents a program unit in element *e*. It has the format:

```
progname [ : extname]
```

where:

progname

represents the desired program unit in the ASCII FORTRAN program. Specify as: (1) * (to represent the main program); (2) a FORTRAN program unit (main program, subroutine, function, or BLOCK DATA subprogram) name; or (3) an unsigned positive integer *n* (to represent the n^{th} unnamed BLOCK DATA subprogram in the FORTRAN source program).

extname

represents the program unit name of the external program unit corresponding to the internal subprogram *progname*. Therefore, *extname* can be specified only if *progname* is specified as a subprogram name that represents a FORTRAN internal subprogram. The variable *extname* can be specified as: (1) * (if the external program unit is the main program); or (2) a FORTRAN program unit (main program, subroutine, or function) name.

If *progname* is specified as a program unit name and *extname* is not specified, then the external program unit with name *progname* is taken. If no such external program unit exists, then the first internal subprogram with name *progname* is taken.

e

is the name of a relocatable element that is mapped into the executing program and that was produced by an F-option ASCII FORTRAN compilation.

Description:

The DUMP command prints the current values of one or more FORTRAN variables.

If the O option is specified on the DUMP command, the values are printed in octal format. If the A option is specified, they are printed in ASCII character format. If neither O nor A is specified, they are printed in a format corresponding to the variable's data type (INTEGER, REAL, COMPLEX, etc.).

Whenever the value of a variable is printed, it follows a heading line in the format:

```
v     /p     /e
```

where *v* is the variable name, *p* is the program unit name (see the description of the *p* subfield above), and *e* is the relocatable element name.

If the *v* subfield is specified and is a scalar, array element, or function entry point, one value is printed. If *v* is specified and is an array, all elements of the array are printed in column-major order.

The following syntax rules apply for the DUMP command:

- One or more blanks must appear between the command name (or command name with option) and the command's only field.
- No blank characters are allowed inside the command's only field. A blank character in the field terminates the syntax scan.

Form Descriptions:

1. DUMP [,*opt*] *v/p/e*

   This format prints the value of variable *v* in program unit *p* in element *e*.

2. DUMP [,*opt*] *v//e*

   This format prints the value of variable *v* in the first program unit in element *e*.

3. DUMP [,*opt*] */p/e*

   This format prints the values of all variables in program unit *p* in element *e*.

4. DUMP [,*opt*] *//e*

   This format prints the values of all variables in all program units in element *e*.

5. DUMP [,*opt*] !

   This format prints the values of all variables in all program units in all relocatable elements produced by F-option ASCII FORTRAN compilations that were mapped into the executing program. Before the variables in a given element *e* are dumped, a heading line is printed:

```
    > > > ELEMENT e          < < <
```

When you use formats 3 through 5, the variables appear in the dump as follows:

- In a program unit, the variables appear in alphabetical order.
- In an element, the program units appear in the order that they appear in in the FORTRAN element.
- If format 5 is used, the elements appear in the order that their diagnostic text appears in in the absolute element's diagnostic tables.

Formats 1 through 4 can be appended with two subfields after the *e* subfield. The format is:

```
DUMP [ ,opt] [v] / [p] /e[ / [s]/i]
```

where *s* and *i* represent the segment name and bank name, respectively, under which $(3) of the relocatable element *e* is mapped. If *i* is specified and *s* is not, then the segment in bank *i* with segment index 0 is assumed.

You can specify the two extra subfields when the executing program is multibanked, although they are never required. If specified, they cause the interactive PMD to search more efficiently through the absolute element's diagnostic tables for the diagnostic text corresponding to relocatable element *e*. The search is more efficient because the absolute element's pointer tables for the diagnostic text are organized by segment and bank, not by element.

If the fourth and fifth subfields are not specified, FTNPMD searchs sequentially through the absolute element's diagnostic tables when looking for the diagnostic text of relocatable element *e*. The diagnostic pointer tables in the absolute element are not used in this case.

Examples:

```
DUMP !
```

Print the values of all variables in all program units in all elements.

```
DUMP //ELT1
```

Print the values of all variables in all program units in element ELT1.

```
DUMP,O /SUB1/E2
```

Print the values of all variables in subprogram SUB1 in element E2 (in octal format).

```
DUMP,A A//E3
```

Print the value of variable A in the first program unit of element E3 (in ASCII character format).

```
DUMP B(6,3,1)/*/E4
```

Print the value of array element B(6,3,1) in the main program (in element E4).

10.4.4.2.2. EXIT

Purpose:

The EXIT command (abbreviated E) terminates the interactive PMD mode.

Form:

```
EXIT
```

Description:

The EXIT command causes program execution to leave the interactive PMD mode.

- If walkback and the PMD mode are initiated by a call from the contingency routine because of one of the five possible error conditions (mathematical library error, I/O library error, illegal operation, guard mode, or error mode), control returns to the contingency routine, where an ER EABT$ is performed.
- If the PMD mode is initiated by a break keyin (@@X C) contingency, control returns to the point of interrupt in the executing program.
- If the PMD mode is initiated by execution of the FORTRAN statement CALL FTNPMD, normal program execution resumes.

The following message is printed when the interactive PMD mode terminates:

```
***** EXIT FTN PMD *****
```

Program execution is immediately terminated if an @ image (control statement) is read in PMD mode. The FEXIT$ termination routine is called.

10.4.4.3. FTNPMD Diagnostics

The diagnostics printed by the interactive PMD are explained in the following list.

```
BANK NOT FOUND
```

The bank specified in the *i* subfield of the DUMP command does not exist in the executing program, or no entry exists in the absolute element's diagnostic pointer tables corresponding to segment *s* (specified in the *s* subfield) in bank *i*.

```
ELEMENT MUST BE SPECIFIED
```

The *e* subfield in the DUMP command is required (unless you use the format DUMP [,*opt*] !).

```
ELEMENT NOT FOUND
```

The element specified in the *e* subfield of the DUMP command was not mapped into the executing program, or no diagnostic text was generated for the element.

```
ENTIRE ASSUMED-SIZE ARRAY CANNOT BE DUMPED
```

The range of an assumed-size array is not known. Only individual elements of an assumed-size array can be dumped.

```
FTEMP$ STORAGE DESTROYED
```

A subprogram's temporary storage area (for saving registers and the argument list) was destroyed because of an error in your program. The specified variable can't be dumped.

```
FUNCTION HAS NOT BEEN CALLED
```

An attempt was made to dump a character function value (that is, the *v* subfield of the DUMP command was specified as a character function subprogram entry point), but the function has not yet been called during execution.

```
I MUST BE SPECIFIED
```

The *s* subfield (segment name) is specified in the DUMP command, but the *i* subfield (bank name) is not.

```
ILLEGAL COMMAND
```

An illegal PMD command name is specified when interactive PMD mode solicits input.

```
ILLEGAL SYNTAX
```

A general syntax error is found in the command image. This includes specifying a field for a command when none is allowed, or not specifying a field when one is required.

```
INCORRECT NUMBER OF SUBSCRIPTS
```

The number of subscripts specified for the array element in the *v* subfield of the DUMP command doesn't equal the number of dimensions declared for the array in the specified program unit (*p* subfield) and element (*e* subfield).

```
NO DIAGNOSTIC TABLES - WALKBACK & FTN PMD TERMINATED

USE F OPTION ON @FTN CARDS TO ENABLE THESE FEATURES
```

In the collection process for the executing program, there were no elements with INFO-010 text mapped in (that is, no relocatable elements produced by ASCII FORTRAN compilations). Therefore, neither walkback nor the interactive PMD can execute, since they both require the use of the INFO-010 diagnostic tables.

```
PARAMETER'S SUBPROGRAM HAS NOT BEEN CALLED
```

An attempt is made to dump a subroutine or function parameter (that is, the *v* subfield of the DUMP command is specified as a subprogram parameter), but the subprogram has not yet been called during program execution.

```
PROGRAM UNIT NOT FOUND
```

The program unit specified in the *p* subfield of the DUMP command doesn't exist in the specified element (*e* subfield).

```
SEGMENT NOT FOUND
```

The segment specified in the *s* subfield of the DUMP command doesn't exist in any bank of the executing program.

```
SUBSCRIPT OUT OF RANGE FOR ARRAY
```

The constant subscripts specified for the array in subfield *v* of the DUMP command is too large or too small.

```
VARIABLE IS NOT AN ARRAY
```

The variable in subfield *v* of the DUMP command is appended with a subscript list, but the variable is not declared as an array in the specified program unit (*p* subfield) and element (*e* subfield).

```
VARIABLE NOT FOUND
```

The variable in subfield *v* of the DUMP command does not exist in the specified program unit (*p* subfield) and element (*e* subfield), or *e* was not compiled with @FTN,F (thereby producing no diagnostic information about your variables).

```
WARNING: BAD DIAGNOSTIC TEXT FOUND; SEARCH TERMINATED
```

It is not possible to continue reading diagnostic (INFO-010) text from the absolute element's diagnostic tables, because bad text was encountered. The bad text could be from (1) a processor other than ASCII FORTRAN or FORTRAN V (which has generated a relocatable element with INFO-010 text), or (2) an old level of ASCII FORTRAN (which has generated a format of INFO-010 text that cannot be interpreted by the current FTNPMD and walkback run-time routines).

Appendix A
Differences Between FORTRAN Processors

A.1. General

The differences in language capabilities and equivalent syntax for ASCII FORTRAN and its predecessor, FORTRAN V, are discussed in this appendix.

A.2. ASCII FORTRAN Extensions Not in FORTRAN V

1. An expanded character set (which handles exclusively ASCII data sets) is available.
2. Double-precision complex data type (COMPLEX*16) is permitted.
3. A character data type is allowed.
4. The $ is allowed in symbolic names, except for the first character of a name.
5. General expressions for subscripts are allowed.
6. Concatenation of character strings is allowed.
7. Multiple assignments are implemented.
8. Character assignments and comparisons are permitted.
9. An integer expression for the index of a COMPUTED GO TO is permitted.
10. An optional comma in a DO statement is permitted.
11. Integer expressions for DO parameters are allowed.
12. The last statement of a DO range can be any statement that permits execution of the following statement.
13. The PAUSE statement is extended to allow longer messages. FORTRAN V STOP and PAUSE arguments should be enclosed in apostrophes to conform with ASCII FORTRAN.

14. The STOP statement is extended.
15. Expressions are permitted in an output I/O list.
16. Expressions are permitted in an implied-DO.
17. List-directed I/O statements are permitted.
18. ASCII FORTRAN enhances the use of storage greater than 65K words. ASCII FORTRAN programs that are larger than 65,535 words are made possible by the O option on the @FTN control command (see 6.10 and 9.5). Programs larger than 262,143 words are possible by also using the BANK statement (see 6.6 and 6.10). Accessing of arrays greater than 65K words was possible in FORTRAN V using the XM = 1 compiler option.
19. Initialization of ASSIGN variables in a DATA statement is permitted.
20. The BANK statement permitting utilization of banks is implemented.
21. An extension to the EXTERNAL statement (&, *) is implemented.
22. An expanded set of mathematical library functions is allowed.
23. New service subroutines are provided.
24. The SUBSTR and BITS pseudo-functions are implemented.
25. A new DEBUG facility is implemented.
26. An extended form of the FORTRAN V PARAMETER statement is implemented.
27. A statement label variable can be used for the format number in an input/output statement.
28. Exponentiation between variables of all arithmetic types and lengths is permitted.
29. In data initialization statements, ASCII FORTRAN requires that the variables or array elements match the type of the corresponding constants if the type is real, complex, double precision, integer, or logical. When such a mismatch occurs, the compiler prints a diagnostic and converts the constant to match the type of the variable. FORTRAN V does not require element type to match the value type but does not convert the constant regardless of its type, possibly causing problems later in the FORTRAN program.
30. Checkout mode is available.
31. INCLUDE statements can specify a file name as well as an element name.
32. ASCII FORTRAN levels higher than 8R1 conform to FORTRAN 77 (ANSI X3.9-1978), and FORTRAN V conforms to FORTRAN 66 (ANSI X3.9-1966). Therefore, all features in FORTRAN 77 that were not in FORTRAN 66 are implemented in ASCII FORTRAN.
33. The virtual feature is provided.

A.3. Incompatibilities between ASCII FORTRAN and FORTRAN V

1. Character data is represented in the ASCII code set rather than the Fieldata code set.

   Although most Fieldata characters have corresponding representations in ASCII, characters are four to a word rather than six and have different internal representations.

   The three Fieldata graphic characters △, □, and ≠ have no equivalents in the ASCII graphic set. They are converted to caret (^), quotation mark ("), and underscore (_), respectively, on an ASCII printer.

2. ASCII FORTRAN doesn't allow transfer of control to labels on nonexecutable statements. However, FORMAT statements are allowed as targets of DO-loops.

3. The interpretation of RETURN *k* is a branch to the *k*th statement label, and not to the *k*th argument. To obtain the effect of the FORTRAN V statement RETURN *k*, you should:

   - Create an explicit typing statement for an integer array ARR with as many elements as there are arguments in the subroutine argument list.
   - Initialize the array such that when the *k*th argument of the list is a statement label, the *k*th element of the array is the sequential position of that statement label among all the statement labels in the list. If there are ENTRY points in the subprogram, the array may need to be set separately in the flow of control after each ENTRY.
   - Set all other array elements to 0.
   - Change RETURN *k* to RETURN ARR (*k*).

   For example, the subroutine:

```
SUBROUTINE SAM (Y,*,*,X)
       .
       .
       .
               I=2
RETURN I
       .
       .
       .
RETURN 3
END
```

   should be changed to:

```
SUBROUTINE SAM (Y,*,*,X)
INTEGER IA (4)/0,1,2,0/
      .
      .
      .
              I=2
RETURN IA(I)
      .
      .
      .
```

```
RETURN IA(3)
       END
```

4. The ABNORMAL statement is not implemented. (The compiler assumes that all FORTRAN mathematical functions are normal and all external procedures are abnormal. It generates an error message and ignores all ABNORMAL statements.)
5. No FORTRAN V hardware IF statements are allowed (subroutine calls are available).
6. FORTRAN V permits a dummy argument to appear in an EQUIVALENCE statement with another dummy argument. ASCII FORTRAN does not allow any dummy arguments in an EQUIVALENCE statement.
7. The FORTRAN V function INSTAT is not available in ASCII FORTRAN. It has been replaced by the functions IOC, IOS, and IOU (see 5.8.1).
8. Exponents have been standardized. ASCII FORTRAN exponents contain three digits rather than two digits used by FORTRAN V.
9. ASCII FORTRAN does not support any of the FORTRAN V options for the COMPILER statement. The ramifications of this and your possible actions are listed by option and variation in the following discussion.
   - FLD Option

     Although there is no counterpart for the FORTRAN V FLD option in ASCII FORTRAN, the FLD function is equivalent to the BITS pseudo- function. It can also be simulated using a statement function.

     The effect of the FLD option is to alter the compiler interpretation of the FLD function. These alterations can be carried over to the BITS function as follows:

     – FLD = L

       You should:

       1. change FLD to BITS
       2. reorder the arguments
       3. change I to I + 1

For example:

```
FLD(5,6,A)
```

should be translated to:

```
BITS(A,6,6)
```

Or, you can insert the statements:

```
INTEGER FLD
FLD(I,J,A)=BITS(A,I+1,J)
```

at the beginning of the program unit.

– FLD = R

  This results in the same action as the L option except for the third step:

  1. change FLD to BITS
  2. reorder the arguments
  3. change I to 36-I

  For example:

```
FLD(5,6,Z)
```

  should be translated to:

```
BITS(Z,31,6)
```

  or:

```
INTEGER FLD
FLD(I,J,A)=BITS(A,36-I,J)
```

  can be inserted at the beginning of the program unit.

– FLD = T

  Used only for optimization and should be deleted.

– FLD = Q

  Used only for optimization and should be deleted.

– FLD = ABS

  Causes all nonconstants in the bit location and length fields to be passed as absolutes. These nonconstants should be made absolute.

  For example:

```
FLD(I,5,C)
```

should be translated to:

```
BITS(C,ABS(I) + 1,5)
```

- **DATA option:**
  - DATA=SHORT
  - DATA=IBM

  These options are used in FORTRAN V to permit partial initialization of arrays. To accomplish the same effect in ASCII FORTRAN, an implied DO-loop is required or else a warning message is issued.

  For example:

```
COMPILER (DATA=SHORT)
DIMENSION A(10)
DATA A/1.,2.,3./
```

  should be translated to:

```
DIMENSION A(10)
DATA (A(I),I=1,3)/1.,2.,3./
```

- 1110=OPT option:

  This activates a code reordering algorithm. The corresponding ASCII FORTRAN option is U1110=OPT.

- Other options:
  - ADR=IND
  - LIST=DEF
  - DIAG=N
  - EQUIV=CMN
  - MATH=A
  - PROP=DPON
  - PROP=DPOF
  - PROP=RLON
  - PROP=RLOF
  - PROP=NONE
  - XM=1
  - CONT=RFOR

These options can be deleted, but you should check the ramifications for the processing of each program. Special coding may be required.

10. Execution of FORTRAN V-formatted WRITE statements may indicate the presence of the control character by printing a blank character, so that printing starts with position 2 of a print line. ASCII FORTRAN makes no such indication, and printing starts in position 1 of a print line.
11. FORTRAN V lets you initialize a whole array with one Hollerith literal in a DATA statement, whereas ASCII FORTRAN requires each element of the array to have a Hollerith literal.
12. FORTRAN V lets you ENCODE/DECODE beyond the area you have given to the ENCODE/DECODE statement. ASCII FORTRAN may give a warning diagnostic and may not ENCODE/DECODE beyond the actual area given or assumed.
13. FORTRAN V allows the unit-number field, the maximum-number-of-records field, and the maximum-record-size field in the DEFINE FILE statement to be an integer constant, an unsubscripted integer variable, or an integer symbolic name of a constant. ASCII FORTRAN does not allow it to be an unsubscripted integer variable.
14. FORTRAN V does not rewind an SDF tape file at termination time unless a CALL CLOSE has been done on that file. ASCII FORTRAN does a rewind of the SDF or ANSI tape file, even if CALL CLOSE or CLOSE has not been done for the file.
15. FORTRAN V checks via an ER FACIT$ to determine if a unit is assigned prior to the program execution. If not assigned to the run, FORTRAN V does an @ASG,T on the unit. ASCII FORTRAN does the ER FACIT$ but does an @ASG,A on the unit to determine if a cataloged file already exists for the unit. If it doesn't exist, an @ASG,T is done on the unit.
16. A FORTRAN V direct-access DEFINE FILE statement can be used to extend a direct-access file. However, an ASCII FORTRAN direct-access DEFINE FILE statement cannot be used to extend a direct-access file. In ASCII FORTRAN level 10R1 or higher, you can use the OPEN statement to extend a direct-access file.

A.4. Differences in Syntax

1. The FLD function is replaced by the BITS pseudo-function. To change a FORTRAN V reference to the FLD function to an ASCII FORTRAN reference, you should:
   - change FLD to BITS;
   - reorder the arguments; and
   - change I to I+1 or insert statements such as:

```
INTEGER FLD
FLD(I,J,A)=BITS(A,I+1,J)
```

To illustrate the first method, the reference:

```
FLD(5,6,A)
```

can be changed to:

```
BITS(A,6,6)
```

2. Four input/output statements in FORTRAN V are expressed differently in ASCII FORTRAN. They are:

   ```
   READ INPUT TAPE unit,f,list
   WRITE OUTPUT TAPE unit,f,list
   READ TAPE unit,list
   WRITE TAPE unit,list
   ```

   To change them to their ASCII FORTRAN equivalents, you should:

   - eliminate the words INPUT, OUTPUT, and TAPE; and
   - enclose *unit* and *f* (if present) in parentheses and remove the comma before *list*.

   For example:

   ```
   READ INPUT TAPE 5,6, A
   ```

   should be changed to:

   ```
   READ (5,6) A
   ```

   and:

   ```
   WRITE TAPE 6, D
   ```

   should be translated to:

   ```
   WRITE (6) D
   ```

3. A Hollerith constant can be specified in FORTRAN V in the following ways:

   ```
   7HEXAMPLE
   7REXAMPLE
   7LEXAMPLE
   'EXAMPLE'
   ```

   Of these, only the first and last are permitted in ASCII FORTRAN.

Change the R and L forms to the H form. Left and right space-fill and shifts may not be significant. ASCII FORTRAN uses four characters per word rather than the six used by FORTRAN V.

4. FORTRAN V permits the concatenation of two arithmetic operators if one of them is unary and it follows **, /, or *. However, this syntax isn't allowed in ASCII FORTRAN.

   For example, the following are permitted in FORTRAN V but not in ASCII FORTRAN:

   ```
   A=4.0/+B
   A=4.0*-B
   A=4.0*+B
   A=4.0**-I
   A=4.0**+I
   ```

   To make them acceptable, separate the two consecutive operators by enclosing the unary operator and its operand in parentheses.

   For example:

   ```
   A=4.0/-B
   ```

   should be translated to:

   ```
   A=4.0/(-B)
   ```

5. Complex variables in list-directed input/output statements are not enclosed by parentheses in FORTRAN V.

6. In ASCII FORTRAN, the first parameter in a direct-access DEFINE FILE statement must be an integer constant.

7. Expressions of the form A**B**C are somewhat ambiguous in interpretation in FORTRAN V. ASCII FORTRAN evaluates the expression as A**(B**C).

Appendix B
ASCII Symbols and Codes

The ASCII symbols and codes are given in Table B-1 and Table B-2.

Table B-1. ASCII Control Characters

Octal Code	Symbol	Meaning
00	NUL	Null - can be used as time-fill
01	SOH	Start of heading
02	STX	Start of text
03	ETX	End of text
04	EOT	End of transmission
05	ENQ	Inquiry. *Who are you?*
06	ACK	Acknowledgment. *Yes.*
07	BEL	Bell. Your attention is required.
10	BS	Backspace.
11	HT	Horizontal tabulation.
12	LF	Line feed.
13	VT	Vertical tabulation.
14	FF	Form feed.
15	CR	Carriage return.
16	SO	Shift out. Nonstandard code follows.
17	SI	Shift in. Return to standard code.
20	DLE	Data link escape.
21	DC1	Device control for turning on auxiliary device.
22	DC2	Device control for turning on auxiliary device.
23	DC3	Device control for turning on auxiliary device.

continued

Table B-1. ASCII Control Characters (cont.)

Octal Code	Symbol	Meaning
24	DC4	Device control for turning on auxiliary device.
25	NAK	Negative acknowledgment. *No.*
26	SYN	Synchronous idle.
27	ETB	End of transmission block.
30	CAN	Cancel previous data.
31	EM	End of medium.
32	SUB	Substitute character for one in error.
33	ESC	Escape. For code extension.
34	FS	File separator.
35	GS	Group separator.
36	RS	Record separator.
37	US	Unit separator.

Table B-2. ASCII Graphic Characters

Octal Code	Symbol	Octal Code	Symbol
40	Space	120	P
41	!	121	Q
42	”	122	R
43	#	123	S
44	$	124	T
45	%	125	U
46	&	126	V
47	Apostrophe	127	W
50	(	130	X

continued

Table B-2. ASCII Graphic Characters (cont.)

Octal Code	Symbol	Octal Code	Symbol
51	)	131	Y
52	*	132	Z
53	+	133	[
54	Comma	134	\
55	Hyphen	135	]
56	Period	136	Circumflex
57	/	137	Underscore
60	0	140	Grave Accent
61	1	141	a
62	2	142	b
63	3	143	c
64	4	144	d
65	5	145	e
66	6	146	f
67	7	147	g
70	8	150	h
71	9	151	i
72	:	152	j
73	;	153	k
74	<	154	l
75	=	155	m
76	>	156	n
77	?	157	o
100	@	160	p
101	A	161	q
102	B	162	r

continued

Table B-2. ASCII Graphic Characters (cont.)

Octal Code	Symbol	Octal Code	Symbol
103	C	163	s
104	D	164	t
105	E	165	u
106	F	166	v
107	G	167	w
110	H	170	x
111	I	171	y
112	J	172	z
113	K	173	{
114	L	174	|
115	M	175	}
116	N	176	Tilde
117	O	177	Delete

Appendix C
Programmer Check List

C.1. General

This list points out the most commonly encountered programming errors for the novice programmer and therefore helps avoid unnecessary errors.

First, common language errors are listed. Then some coding techniques are given to help you avoid execution errors.

C.2. Language Errors

The simplest errors are straightforward language-use errors. These are discovered by the compiler during compilation and are flagged in the listing. You can let the compiler do the checking for this type of error and continually change the program until no more errors are flagged. Alternatively, machine time and money can be saved by manually doing a quick check of the program for the following common errors:

1. Do the main program and subprograms contain statements in the following order?
   - COMPILER statement
   - Program declaration statements
     – In a main program, a PROGRAM *pgm* statement is optional.
     – In a subprogram, one of the three program declaration statements must be the first statement of the program unit:
       a. type FUNCTION f(a_1,a_2,...,a_n)
       b. SUBROUTINE s(a_1,a_2,...,a_n)
       c. BLOCK DATA [*sub*]
   - IMPLICIT statements

- Explicit type statements
- PARAMETER statements
- DIMENSION statements
- COMMON statements
- BANK statements
- EQUIVALENCE statements
- EXTERNAL statements
- INTRINSIC statements
- SAVE statements
- DATA statements
- NAMELIST statements
- FORMAT statements
- Statement functions
- Executable statements
- The END statement

Observance of this ordering prevents the omission or misplacement of necessary program statements.

Some deviation from this order is acceptable. Before changing the order, you should refer to the section of this manual that specifically discusses the statement in question.

2. Are bodies of source statements contained in positions 7 through 72?

   - Do comment lines contain a C or * in column 1?
   - Do continuation lines contain a continuation character in position 6 and blanks in columns 1 through 5?
   - Are statement labels contained in positions 1 through 5?

3. Are all arrays dimensioned?

   If not, the program may attempt to treat references to the array as function calls. Such errors may show up as undefined references when the program is collected. If an array is not dimensioned and considered as a function by the compiler, its name appears in the EXTERNAL REFERENCES listing produced by the compiler (see 9.4.2.2.8).

4. Do any array references contain subscripts that are out of bounds?
5. Do all statement labels referenced in GO TO statements exist in the same program unit as the reference?
6. Do function subprograms contain a statement assigning a value to the function?

   Since a function reference is used as a value, a value must be assigned to it somewhere in the subprogram.
7. Are DO-loop termination statements placed properly so that only the desired statements are encompassed?
8. Does the number of right parentheses match the number of left parentheses in FORMAT statements, etc?

C.3. Coding Techniques

Avoid execution errors, as well as language errors, by using certain coding techniques. Some guidelines follow.

1. Are there any repeated sets of code that could be converted to subroutine references?
2. Can the program be broken down into a series of smaller subprograms? If so, does the number warrant the use of common storage?
3. What corrective or diagnostic actions are taken if:
   - incorrect data is read in?
   - numeric overflow, underflow, or divide fault occurs?

   A good echo check (to verify that the data read in is what is wanted) is to insert a list-directed output statement immediately following the READ statement (to write the input values with some explanatory text). What you think you are reading in may not have any resemblance to what you are actually reading.

   For example:

   ```
   PRINT *,'READ X,Y,I AS: ', X,Y,I
   ```
4. Insert abundant comments into the code. They help you remember what is actually done in a particular coding section and what else is needed by this code. Comments may be particularly helpful at the beginning of a program unit that is used often. Such comments might contain the following information:
   - What is the program designed to do?
   - What input data does it need, if any? In what format and from what device?
   - What subprograms, if any, does the subprogram use? What do these subprograms do?

- What tapes, files, disks, etc., are used by the program? What are their unit numbers?
- What forms of data failure can occur during execution of this program? What action is taken in each such case?

5. Use the blocking statements (block IF, ELSE IF, ELSE, and END IF) instead of GO TO statements whenever possible, to make the code more understandable.

Appendix D
Diagnostic Messages

This appendix lists the diagnostic messages (errors, warnings, and nonstandard usages) that may be issued during compilations of ASCII FORTRAN programs. The notation *xx* indicates that situation-dependent data is filled into the message when the error occurs. Appendix I contains messages that may be issued during checkout (C option on the ASCII FORTRAN processor call) runs, and Appendix G has messages at the end that may be issued at run time by the I/O handler.

1 `Compiler cannot assign spill file`

The compiler assigns the temporary file PSF$ to be used to hold various pieces of information needed during the compilation that don't all fit into main storage. For some reason, the compiler cannot assign this file.

2 `I/O error on compiler spill file` *xx*

Some kind of I/O error has occurred on the compiler spill file.

3 `Compiler spill file limit exceeded`

Your program is so large that the compiler has run out of spill file space. The program either has too many variables and equivalences (which fill up its dictionary area), or it has too many executable statements (which fill up its text area).

4 `Compiler limit or error` *xx* `encountered`

An unrecoverable internal error has occurred within the compiler. Either some internal limit such as the number of data names is exceeded, or there is an internal error. This event should be reported to your local Systems Analyst who may fill out a User Communication Form (UCF) to send to Unisys for analysis.

5 `Too many external references`

The external reference table is full. All references to additional external names are deleted.

6 `Data size exceeded` *`xx`* `location limit`

With or without the presence of the compiler O option (except as noted below), no collected program can exceed 262,143 decimal words of address space at any one time. Local noncommon data and compiler-generated data must not exceed 65,535 decimal words in a given compilation.

If the O option is not used, the total D-bank size allowed in your collected program is 65,535 decimal words. This includes local declarations and compiler-generated data. Each common block is also limited to 65,535 decimal words in size.

If the O option is used, the total D-bank size allowed in your collected program is 262,143 decimal words. This includes local declarations and compiler-generated data. The size of each common block is also limited to 262,143 decimal words.

Programs can exceed these sizes if you use the VIRTUAL data or the multibanking features of ASCII FORTRAN. See the VIRTUAL, BANK and COMPILER statements and Appendixes M and H.

Even if the O option is used, the size of location counter 0 in a compilation cannot exceed 65,535 decimal words. $(0) consists of all noncommon data (including local variables, packets, temporaries, and so forth). If your location counter 0 exceeds 65,535 words, you must make the compilation smaller by splitting your program into separate elements.

7 `Too many program blocks for optimization`

The complexity of the program exceeds the ability of the optimization phase to completely analyze it. As much of the program as possible receives global optimization. The rest of the program receives only limited local optimization. You should simplify the program by reducing the number of branches (IF, GO TO, DO statements) or by breaking the program into several subprograms.

8 `Too many program variables for optimization`

There are more variables in the program than can be handled by the optimization phase of ASCII FORTRAN. It is assumed that the most frequently used variables are in the most deeply nested loops, and usage of these is optimized on a global basis. Expressions involving other variables are only optimized on a local basis.

9 `Too many user entry points`

The entry point table is full. This entry point and all of your subsequent entry points are not inserted. You should drop some entry points by eliminating some subroutines or ENTRY statements in your source element.

10 `Initialization of xx is not possible`

Initialization is not possible for variables or array elements that have addresses over 65K decimal. This occurs on local variables once their accumulated size adds up to this much, or on individual COMMON groups adding up to this much. For example:

```
DIMENSION A(2,33000), B(2,33000)
DATA A(1,1), B(1,1) /1.,2./
```

One of the above initializations is bad. However, the following works:

```
DIMENSION A(2,33000), B(2,33000)
COMMON A
DATA A(1,1), B(1,1) /1.,2./
```

The solution to this problem is to make smaller COMMON groups.

11 `USED_NUM_INTS GT MAX_NUM_INTS`

There are more intervals in the program than can be handled by the information gathering portion of the optimization phase.

12 `USED_NUM_LEVELS GE MAX_NUM_LEVELS`

The number of levels resulting from the interval analysis algorithm exceeds the maximum number of levels that can be handled by the optimization phase.

15 `LASTIBLK GT MAX_NUM_INTS`

During the information-gathering portion of the optimization phase, the newest initialization block created is assigned an interval number greater than the maximum number of intervals. Global optimization is unable to continue. Local optimization is attempted.

16 `Missing END statement, next program unit assumed external`

1. A BLOCK DATA subprogram is missing an END statement. (BLOCK DATA subprograms cannot have internal subprograms.)
2. An END is missing from an external program unit and a BLOCK DATA subprogram followed. (BLOCK DATA subprograms can't be internal subprograms.)

17 `Compiler error - text chain ends prematurely`

A text entry with a zero forward link is encountered before an END text entry is found. This most likely occurs during code generation.

18 `MCORE failure - compiler terminated`

Optimization (V or Z option) requires extra table space. An ER to MCORE$ is attempted but fails, since your system does not have enough available storage. The compilation is terminated. This message may also appear if table space for storage map (D, L, or Y options) or INFO-010 table generation (F or C options) is not available.

19 `Program unit has no executable statements`

The program unit (main program, external subprogram, or internal subprogram) that just terminated has no executable statements. If the program unit is a subprogram, then the code generated causes it to return control to the caller. If the program unit is a main program, then the code generated consists of calls to FINIT$ (initialization routine) and FEXIT$ (termination routine).

20 `MCORE failure - local optimization attempted`

If an MCORE failure occurs while attempting to obtain the initial storage for global optimization tables, local optimization is attempted.

21 `Source line contains more than 80 columns`

If columns 81-132 appear in the source line, the line is truncated to 80 columns, and is then accepted by the compiler. If more than 132 columns appear in the source line, the line is rejected. In either case, you must edit the source line so that it is 80 columns or less.

22 `ROR$ call error, bad ROR packet encountered`

ROR detected bad contents in a ROR packet. The compilation terminates.

23 `I/O error during ROR$ call, I/O status =` *xx*

An I/O error occurred during a ROR call. The I/O status is printed out as a decimal integer. The compilation terminates.

24 `ROR error,` *xx* `status =` *xx*

1. ROR error, ER-PFWL$ status = *xx*

   A start ROR call received a bad status from an ER PFWL$ request. The status *xx* is printed out as a decimal integer. The compilation terminates.

2. ROR error, I/O status = *xx*

   An end ROR call received an I/O error. The status *xx* is printed out as a decimal integer. The compilation terminates.

25 `ROR Table Write error, xx status = xx`

1. ROR Table Write error, I/O status = *xx*

   An ROR Table Write request resulted in an I/O error. The status *xx* is printed out as a decimal integer. The compilation terminates.

2. ROR Table Write error, ER-PFI$ status = *xx*

   An ROR Table Write request resulted in a bad status from an ER PFI$ request. The status *xx* is printed out as a decimal integer. The compilation terminates.

26 `Automatic storage exceeds 65K for a program unit group`

When you specify the COMPILER statement option DATA=AUTO, the combined size of automatic data storage (i.e., all data not declared in COMMON or SAVE statements) cannot exceed 65,535 words for a program unit group (an external program unit and all of its internal subprograms). You must change some of the local arrays in automatic storage to be static (using either COMMON or SAVE), or divide the program unit group to make it smaller.

27 `Duplicate user entry point xx`

The specified user entry point appears in more than one SUBROUTINE, FUNCTION, or ENTRY statement in the compilation unit.

101 `Delimiter is missing before xx`

A required delimiter is missing at the indicated point in the statement. In some cases, the delimiter is inserted (for example, a missing comma after a DATA value list). In other cases, the statement is deleted.

102 `Delimiter xx is used incorrectly`

The indicated character appears in a position where it is not permitted. The character may be ignored (for example, an excess comma in a FORMAT statement), or it may cause deletion of the statement.

103 `xx is incorrect outside literal constant`

The character printed occurs outside a character or Hollerith constant, but is not a letter, digit, currency symbol, or FORTRAN delimiter. The character is deleted from the statement; the effect is as if the character is a blank.

104 `Numeric constant xx is out of range`

This message indicates that a constant is outside the limits for its type. This can indicate an octal constant that exceeds 12 digits; the last 12 digits are used. For an integer, this message indicates that the value cannot be represented in 35 bits; the last 35 bits of the value are used. For a real constant, this message indicates that the exponent is too large (positive or negative); the value is replaced by 0 in the statement.

105 `Long name truncated to` *xx*

The indicated name is more than six characters long. The first six characters of the name are used.

106 `Constant` *xx* `is incorrect`

The indicated constant is incorrectly formatted. For example, 3.0E-I receives this message because the exponent is not an integer constant. In this example, 3.0E-I is interpreted as ((3.0E0) - I). This message also applies to character constants that exceed 511 characters in length; for such constants, the last 510 characters are used.

107 `Constant` *xx* `is used incorrectly`

The indicated symbol is used where a particular type of constant is required. The error may cause the statement to be deleted, or it may cause just the affected part of the statement to be ignored.

108 `Statement ends prematurely`

The end of the statement is encountered while the syntax of the statement is incomplete. The statement may be completed in an arbitrary manner, or the statement may be deleted.

109 `Statement contains excess right parentheses`

The statement contains at least one right parenthesis for which there is no matching left parenthesis. The statement is deleted.

110 `Statement contains excess left parentheses`

The statement contains at least one left parenthesis for which there is no matching right parenthesis. The statement is deleted.

111 `Statement is too long`

There are more than 1,320 significant characters in the statement, excluding the statement label, if any, and the continuation markers in column 6. This error only occurs for statements having more than 19 continuation lines. The statement is ignored.

112 `Unrecognizable statement`

The statement cannot be identified as any FORTRAN statement. The statement is ignored.

113 `Statement contains unclosed literal constant`

A character or Hollerith constant is not closed at the end of the last continuation line of this statement. This can be caused by unbalanced apostrophes, missing delimiters (which can result in apostrophes not being treated as quote characters), a Hollerith field with an incorrect character count, or a missing continuation card. The statement is ignored.

114 `END statement is missing`

The end of the source input is encountered without finding an END statement. This can be caused by failure to place an END statement in the program, or by an error in the terminal statement label of a DELETE statement. The second case also causes error message 404. In either case, an END statement is assumed by the compiler.

115 `Characters` *`xx`* `follow logical end of statement`

When the syntax scan detects the end of the statement, the source line is not completely scanned. The statement is compiled as if it ends where the syntax scan indicates that the statement is complete.

116 `Target statement of logical IF is of wrong type`

The statement to be conditionally executed as part of a logical IF statement is not executable, or is a DO, block IF, ELSE IF, ELSE, END IF, END, or another logical IF statement. The logical IF and its target statement are ignored.

117 `Statement illegal in DEBUG packet`

A second DEBUG statement appears in the program unit. Since only one DEBUG statement is allowed per program unit, the second is deleted.

118 `Name` *`xx`* `is used incorrectly`

The name is previously defined or used incorrectly. If this is a warning message, the statement is processed as if no error occurred. If it is an error message, the statement is deleted.

119 `Statement illegal outside DEBUG packet`

TRACE ON, TRACE OFF, AT, or DISPLAY statement appears without a preceding DEBUG statement. This statement is deleted.

120 `Specification follows DEFINE or DATA statement`

This specification statement (type, DIMENSION, etc.) follows a statement function definition or a DATA statement. The specification statement is processed as if no error occurred.

121 `Specification follows executable statement`

This specification statement follows an executable statement. Specification statements following an ENTRY statement are flagged with a warning message indicating nonstandard usage. In all other cases, an error is issued and the erroneous specification statement is deleted.

122 `Program truncated at END statement`

The only lines that can follow an END statement are comments, START and STOP EDIT statements, and FUNCTION, SUBROUTINE, and BLOCK DATA statements, which also signal the start of the source of another program unit. If these rules are not followed, an error 122 is issued and all remaining source is ignored.

123 `AT statement missing in DEBUG packet`

An AT statement is required if executable statements follow the DEBUG statement. The executable statements after a DEBUG statement and before any AT statement are processed but can't be executed.

124 `xx field is repeated`

125 `Statement illegal in BLOCK DATA subprogram`

An executable statement or other statement not permitted in a BLOCK DATA subprogram is encountered. The statement is deleted.

126 `Statement illegal in function subprogram`

127 `Continued statement has no initial line`

The indicated statement begins with a continuation line. The initial line is assumed to be blank. If a label appears in columns 1 through 5 of the continuation line that begins the statement, it is taken as the statement label. This is the only situation where the label field of a continuation line is interpreted by the compiler.

128 `COMPILER statement option xx conflicts with previous options`

Certain options of the COMPILER statement are incompatible; for example, ARGCHK=ON is not compatible with ARGCHK=OFF. This error may also be received if the DATA=AUTO, DATA=REUSE, or PROGRAM=BIG options are misplaced in the source. Each of the three options should appear at the beginning of the source for the compilation.

129 `Empty string interpreted as string of one blank`

Null strings are not permitted. They are changed internally to a string of one blank.

130 `Syntax error in CALL statement`

There is a syntax error in the argument list to the subprogram being called in a CALL statement, or, there are nonblank characters following the argument list.

131 `Syntax error in CALL statement at` *xx*

Something other than a left parenthesis followed the subprogram name in a CALL statement.

132 `Variable name` *xx* `used incorrectly in TYPE statement`

The variable declared in the TYPE statement and being initialized has appeared also in an INTRINSIC statement. An attempt to initialize the variable conflicts with the definition of the INTRINSIC statement.

201 `IMPLICIT statement is out of correct order`

The IMPLICIT statement must precede any specification or executable statements. The IMPLICIT statement is accepted, but affects only variables that have not yet appeared in any statement.

202 `Length specification` *xx* `is incorrect`

The length specification is not a positive integer constant or is not permitted for the requested type. The optional length for the requested type is used. The length used for type CHARACTER is 1.

203 `Type specification` *xx* `is incorrect`

The type keyword in the IMPLICIT statement is not a recognized FORTRAN data type. The incorrect type specification is ignored.

204 `Letter specification` *xx* `is incorrect`

The indicated character is not a valid IMPLICIT range limit. The character is ignored. Thus REAL(I-*) is the same as REAL(I).

205 `Letter specification` *xx* `occurs previously in IMPLICIT`

An IMPLICIT type is specified previously for the indicated letter. The new type specification for the indicated letter is ignored.

206 `Statement order problem for` *xx*

A DATA statement intervenes between the type specification for the variable and its preceding dimension specification. The DATA statement doesn’t necessarily refer to the constant. The type specification for this variable is ignored. See 9.3.2 for a statement ordering description.

207 `VIRTUAL statement must precede specification statements`

The VIRTUAL statement must precede all specification statements (except possibly IMPLICIT or COMPILER) and executable statements in the program unit. If the VIRTUAL statement is out of order, then it is ignored.

PROGRAM, SUBROUTINE, FUNCTION, or BLOCK DATA statements can also precede the VIRTUAL statement.

208 `VIRTUAL statement must appear in first program unit`

If a VIRTUAL statement does not appear in the first program unit, then it cannot appear in any other program unit in the source. The VIRTUAL statement is ignored in this case.

209 `VIRTUAL statement option is misplaced`

If neither the ALL nor the ALLBUT option of the VIRTUAL statement appeared in the first program unit, then neither can appear in any other program unit in the source. The ALL or ALLBUT option is ignored in this error case.

210 `Common block` *`xx`* `not allowed in VIRTUAL stmt. in internal`

A VIRTUAL statement in an internal subprogram can only have local variable names in its list. Common block names are not allowed and are ignored.

211 `BANKED=ALL ignored when VIRTUAL(ALL) is specified`

If the ALL (or ALLBUT) option appears in a VIRTUAL statement in the first program unit, then the BANKED=ALL option of the COMPILER statement is not allowed (and is ignored). VIRTUAL(ALL) or VIRTUAL(ALLBUT *cblist*) takes precedence over COMPILER(BANKED=ALL).

```
.
.
.
END
SUBROUTINE SUB
VIRTUAL
COMMON/CBX/x,dp(29 999 999)
.
.
.
END
```

212 `Common block` *`xx`* `appears in both VIRTUAL and BANK statements`

A common block name cannot be in both virtual space (e.g., VIRTUAL /a/) and in a specified bank (e.g., BANK /b/ a). The BANK statement is ignored.

213 `Variable xx appears in both VIRTUAL and COMMON statements`

A variable name cannot be in both virtual space (for example, VIRTUAL a) and in common (e.g., COMMON /c/ a). The variable is treated as a common variable.

214 `Incorrect COMPILER statement option parameter xx`

The legal values for *n* in COMPILER(PAGESIZE=*nk*) are 4, 8, 16, 32, 64, 65, 128 and 131. The legal values for *n* in COMPILER(NBRPAGES=*n*) are 2 LE *n* LE 2048.

215 `Common Block xx cannot exist in both Virtual and non-Virtual space`

A common block was placed into virtual space in one program unit, but placed into non-virtual space in another. It cannot be in both.

Example:

```
VIRTUAL /CBX/
COMMON/CBX/x,dp(29 999 999)
.
.
.
END
SUBROUTINE SUB
VIRTUAL
COMMON/CBX/x,dp(29 999 999)
.
.
.
END
```

216 `All possible page sizes are bad for Virtual xx xx`

You declared a virtual array or scalar in a manner that no page size exists that would not cause a spanning problem on some array element(s). You must adjust the array offset in its common block so that spanning doesn't occur. If the item is a large character array with an element size that is not a power of two, you may need to increase the character size to the next power of two.

Example:

```
COMMON/CBX/ r,dp(999 999)

DOUBLE PRECISION dp
```

217 `Span error for selected page size on Virtual xx xx`

You declared a virtual array or scalar in such a manner that the selected (or default) page size causes a spanning problem on some array element(s).

218 `Large virtual array` *`xx`* `appears incorrectly in I/O statement`

A virtual array or virtual array argument whose size is known to be greater than 262K elements (or 262K words, if STD=66) at compile time cannot appear unsubscripted in an I/O statement, except as an I/O list item.

For example, such an array cannot appear unsubscripted in a FMT= clause in a WRITE statement.

219 `A member of common block` *`xx`* `incorrectly appears in a VIRTUAL statement`

Only local variables and common block names (in slashes) can be specified in a VIRTUAL statement. You specified a member of a common block in a VIRTUAL statement:

```
VIRTUAL a
COMMON /c1/b,x,y,z
EQUIVALENCE(a,b)
```

301 `PARAMETER name` *`xx`* `is previously defined`

The statement redefines the indicated symbolic name of a constant. The attempted redefinition is ignored.

302 `PARAMETER name` *`xx`* `is incorrect`

The symbol following the PARAMETER keyword is not a simple variable name. The PARAMETER definition is ignored. This message is also produced when a dummy argument name in a FUNCTION, ENTRY, or SUBROUTINE statement is incorrect (for example, a constant). The incorrect dummy argument is deleted.

303 `PARAMETER expression is not constant valued`

A PARAMETER expression can only contain constants, previously defined symbolic names of constants, and references to certain intrinsic functions. The PARAMETER definition doesn't satisfy this constraint and is ignored.

304 `PARAMETER expression may give questionable results`

Since symbolic names of constants have a type associated with them (FORTRAN 77), assigning a symbolic name of a constant as a typeless constant can produce unpredictable results when the symbolic name of a constant is used in an expression. This comes about because the internal representation of the typeless constant (a one-word string) may not conform to the internal representation expected by the operator.

401 `INCLUDE procedure cannot be found`

The procedure name specified in the INCLUDE statement can't be found. The INCLUDE statement is ignored. This message also occurs if an I/O error is encountered while searching for an INCLUDE procedure.

402 `Nested INCLUDE is illegal`

An INCLUDE statement is encountered in the text copied by an INCLUDE statement. This is not allowed. The nested INCLUDE is ignored.

403 `Input error` *`xx`* `on INCLUDE file`

The indicated I/O status is received on an ER IOW$ while reading or opening an INCLUDE procedure. If the error occurs while opening the procedure, the INCLUDE is ignored and error 401 is also generated. If the error occurs while reading the body of the procedure, the INCLUDE procedure is immediately terminated, as if the end-of-file sentinel is encountered. If, instead of an I/O status, the error read NOT-PF, then the file isn't a program file.

404 `Terminal label for DELETE cannot be found`

The end of the source input is reached without finding the statement label that terminates the active DELETE statement. The DELETE terminates automatically at the end of the source input.

405 `Correction image sequence error`

A source correction line is out of sequence or is incorrect. The correction line is printed in a previous message, for example, OUT OF SEQ -15.

406 `Procedure name` *`xx`* `is incorrect`

The indicated INCLUDE procedure name is invalid. The INCLUDE statement is ignored.

407 `INCLUDE option` *`xx`* `is incorrect`

The indicated characters were encountered where the LIST option was expected. The LIST option is assumed.

408 `EDIT option` *`xx`* `is incorrect`

The indicated option for a START EDIT or STOP EDIT statement must be either SOURCE, CODE, or PAGE. If the statement is START EDIT, the SOURCE option is assumed. If the statement is STOP EDIT, CODE is assumed.

409 `Error type` *`xx`* `on input`

The indicated I/O status is received from an ER IOW$ on input from the source input stream. The compilation terminates without producing an updated source element.

410 `Input error in open`

An I/O error occurs in opening the source input stream. The compilation stops immediately.

411 `PFI$ error` *xx* `closing file`

412 `INCLUDE file cannot be assigned`

The explicitly named INCLUDE file can't be assigned. The INCLUDE statement is deleted.

501 `Variable` *xx* `has been previously defined`

The indicated variable already has a type specified in a type statement. The new specification for this variable is ignored. This message also occurs in other cases where a name is defined twice.

502 `Length spec.` *xx* `is inconsistent or incorrect`

The specified type doesn't permit a length specification or doesn't permit the length that is specified. The length specification is ignored.

503 `Character *(*) not allowed for` *xx*

Dynamic character string arguments are not allowed for programs with old mode on (COMPILER(STD=66)), or on names that are not function names, argument names, or symbolic names of constants.

504 `Character *(*) not allowed with STD=66`

See the explanation for 503.

601 `Dimension or substring` *xx* `is not type integer`

The constant specified for the array's dimensions or the character item's substring start or end position is not type integer.

602 `Array` *xx* `has more than seven dimensions`

604 `Dimension` *xx* `is not a scalar, argument, or in common`

605 `Array` *xx* `is previously defined`

606 `Variable dimension is not permitted for` *xx*

The indicated variable is specified with variable dimensions, but isn't an argument of the subroutine.

607 `Asterisk not permitted as dimension bound for` *xx*

An asterisk is specified as a dimension in which the array isn't a dummy array or the asterisk isn't the upper dimension of the last dimension.

608 `Array` *xx* `is too large`

The number of array elements or the total word size of the array exceeds 262,143. You must reduce the array bounds.

609 `Dimension bound expression is not type integer for` *xx*

After calling the expression routine to evaluate a dimension the result type is not integer.

610 `Lower dimension bound greater than upper dimension bound for` *xx*

612 `Subscript or substring not integer constant expression for` *xx*

The array name specified in the DIMENSION or EQUIVALENCE statement contains subscript references that must be integer constant expressions or the character item contains substring references that must be integer constant expressions.

613 `Dimension value indeterminable at compile time for` *xx*

A program contains an array with a dimension expression that cannot be calculated at compile time because the ASCII FORTRAN compiler has been configured without the mathematical common banks (for example, 3**3 cannot be computed) or else the expression is out of range (for example, 200000,000000,000 ** 200000,000000,000).

614 `Dimension bound` *xx* `is not dummy argument or in common`

The indicated name appears in a dimension bound or in a dimension bound expression for an adjustable array declaration. The name cannot be a local variable. It must be a dummy argument, a variable appearing in a common block, or a symbolic name of a constant.

701 `Variable` *xx* `appears in previous COMMON or SAVE statement`

702 `Argument` *xx* `appears in COMMON, EQUIVALENCE, or VIRTUAL statement`

The indicated name is not permitted in a common block or in an EQUIVALENCE or VIRTUAL statement. The name is ignored.

703 *xx* `makes COMMON too large`

Inclusion of the indicated variable causes a common block to exceed 262,143 words. The assignment of addresses to this variable and those following it in the common block are incorrect.

704 `Maximum number of common blocks exceeded`

A maximum of 247 common blocks are specified in all program units of a FORTRAN compilation.

801 `EQUIVALENCE member` *`xx`* `links common blocks`

The indicated item is in a common block and is equivalenced to an item in another common block. The equivalence for the item is ignored.

802 `EQUIVALENCE member` *`xx`* `is the only member of a set`

803 `EQUIVALENCE member` *`xx`* `is inconsistent`

This item is equivalenced in a way incompatible with prior equivalences. The inconsistent equivalence is ignored.

804 `EQUIVALENCE member` *`xx`* `reorders a common block`

The indicated item is equivalenced in a way that conflicts with prior EQUIVALENCE and COMMON statements. The incorrect equivalence relationship is ignored.

805 `EQUIVALENCE member` *`xx`* `extends common incorrectly`

The indicated item causes a common block to be extended backward. The equivalence is ignored.

806 *`xx`* `subscript out of range in EQUIVALENCE`

A subscript for the indicated array is out of range. The variable is treated as a scalar.

807 *`xx`* `wrong number of subscripts in EQUIVALENCE`

The indicated array appears with more or fewer subscripts than it should have. Missing subscripts are assumed to be 1; excess subscripts are ignored.

808 `EQUIVALENCE member` *`xx`* `must begin on word boundary`

The item indicated is equivalenced to a character item that begins on a nonword boundary. The item is assigned storage on the previous word border.

809 *`xx`* `makes equivalence group too large`

The item makes an equivalence group exceed 262,143 words. Storage assignments for variables in the group are incorrect.

810 `Undimensioned name` *`xx`* `used in EQUIVALENCE`

The indicated name appears with subscripts in an EQUIVALENCE statement, but is not dimensioned. The subscripts are ignored.

901 `External name` *`xx`* `is not a subprogram name`

1001 `BANK member` *`xx`* `is previously defined`

1002 `BANK member` *`xx`* `is not COMMON or EXTERNAL name`

The indicated name appears in a BANK statement, but is neither a common block name nor an external subprogram name. The BANK specification is ignored for this item.

1003 `Generic function` *`xx`* `deintrinsified via BANK statement`

You named a generic function in a BANK statement. It is treated as if you also named it in an EXTERNAL statement. All automatic argument and result typing is lost because it isn't considered an intrinsic function.

1004 `The BANK statement on` *`xx`* `is misplaced`

You named a function/subroutine in a BANK statement in an internal subroutine, but the function/subroutine was first used in the outer external procedure. If you want to bank the function, put the statement in the outer external procedure.

1005 `Internal subprograms cannot be banked`

You tried to name an internal subroutine in a BANK statement. This is illegal.

1101 `NAMELIST name` *`xx`* `is previously defined`

The NAMELIST name already occurred as a variable, array, or NAMELIST name. The NAMELIST statement is deleted.

1102 `NAMELIST list reference` *`xx`* `is incorrect`

An item in the NAMELIST list is not a variable, array, or array element with constant subscripts, nor is a dummy argument of the subprogram. The NAMELIST statement is deleted.

1201 *`xx`* `may not be initialized`

The indicated identifier is an argument of the subprogram. Since initial value assignment is not permitted for dummy arguments, the DATA statement or initial value specification is ignored.

1202 `Constant does not match type of variable` *`xx`*

The initial value is of a type not permitted for the indicated variable. If the message is an error, the assignment is not done. If the message is a warning, the constant is converted to the type of the variable and the assignment is done.

1203 `Number of constants exceeds number of variables`

The initial value list contains more elements than the list of variables to be initialized. The excess constants are ignored.

1204 `Number of variables exceeds number of constants`

The list of variables is longer than the initial value list. The excess variables don't receive any initialization.

1205 `Implied DO is too complex - simplify`

The implied DO in a DATA statement is too complex, causing the interpreter stack to overflow. The rest of the current variable list is deleted.

1206 `Repeat count error at` *`xx`*

There is some kind of syntax error on a repeat count in the DATA statement.

1207 `Operand` *`xx`* `incorrect for DATA statement`

This message means one of two things has occurred:

1. The indicated operand appears in an implied DO in a DATA statement and is not permitted there. Only constants, symbolic names of constants, and DO-loop index variables are permitted in expressions in an implied DO in a DATA statement.
2. The operand is an argument to the subprogram and can't be initialized in a DATA statement.

1208 `Character string(s) truncated in DATA constant list`

The length of one or more character strings in a DATA statement constant list exceeds the storage limits for the variable in the corresponding data list. The character constant is truncated on the right.

1209 `Substring expression must be an integer constant expression`

Substring expressions must be integer constant expressions.

1210 `Object of substring must be a variable or array element`

When you use the object of a substring reference in a DATA statement, it must be a scalar character variable or character array element.

1211 `Substring reference` *`xx`* `not permitted here`

1212 `More than one initialization for` *`xx`*

A variable, array element or substring must not be initially defined more than once in an executable program. If two entities are associated, only one can be initially defined in a DATA statement in the same executable program. The initial value received by the entity is unpredictable.

1213 *`xx`* `in data list may result in initialization after use`

The data item listed in the warning message is part of a data list that initializes both nonvirtual common and virtual common or virtual static local data items. Due to the presence of virtual data items, the entire data list must be initialized with generated code. Initialization done by the Collector resulting from ROR packets is insensitive to program execution flow, whereas initialization done by generated code exactly follows the program execution flow and could result in an item being initialized after it has been used. To avoid this possible problem, segregate the nonvirtual common data items in a separate data list.

1214 *`xx`* `in data list prevents initialization`

Data items in the dynamic (automatic) storage class are initialized by generated code each time a subprogram is entered. Data items in the static storage class are initialized only once. If one or more data items in the data list are in virtual storage, the initialization is performed by generated code. The storage class (dynamic or static) of the first data item in a data list determines whether the entire data list is to be initialized once or each time a subprogram is entered. The appearance of the data item listed in the error message causes one of the mismatches listed below. The data list storage class mismatches which prevent initialization are:

- Virtual dynamic local with any of the following:
  - virtual common
  - virtual static local
  - nonvirtual common
  - nonvirtual static local
- Nonvirtual dynamic local with any of the following:
  - virtual common
  - virtual static local

1301 `BLOCK DATA statement is out of correct order`

1302 *`xx`* `is local to BLOCK DATA - ignored`

The indicated variable appears in a type, DIMENSION, or DATA statement in the BLOCK DATA subprogram, but is not in a common block. No storage is allocated for the variable, and any initial value assigned to it is ignored.

1303 `Virtual BLOCKDATA programs must appear with executable program units`

BLOCK DATA subprograms initializing virtual common must be located in a FORTRAN source element that holds at least one executable program unit (main program, subroutine, or function). At least one of the executable program units in the element must be entered before any of the virtual common blocks (being initialized by the BLOCK DATA program) are referenced in the program. The easiest way to ensure this is to place all BLOCK DATA programs into the element holding the virtual main program.

1401 `Argument` *`xx`* `is repeated in argument list`

A dummy argument name appears more than once in the argument list. Only the first appearance is processed.

1402 `Internal subprogram` *`xx`* `should not be used as argument`

This diagnostic appears when an internal subprogram is passed as an argument to an external subprogram and the DATA=AUTO or DATA=REUSE COMPILER statement options are present. Because of automatic storage stack conventions, an internal subprogram can only be called from the external program unit it is contained in or from another internal contained in the same external as itself.

1403 `Statement is used in main program`

An ENTRY statement is not permitted in a main program. The ENTRY statement is deleted.

1404 `ENTRY stmt. is within range of DO-loop or block IF structure`

An ENTRY statement is not allowed in the range of a DO loop or in the range of a block IF structure (that is, from an IF (*e*) THEN statement to the corresponding END IF statement). The ENTRY statement is deleted.

1405 `Character and non-character entry names not allowed`

A function subprogram contains a combination of character and noncharacter entry or function names. If the function name is type character all entry names must be type character. If an entry name is type character, the function name must be type character.

1406 `Subprogram or subprogram ref. has too many arguments`

There is a compiler limit of 250 subprogram arguments allowed in either a subprogram reference (that is, calling a subroutine or referencing a function in an expression) or a subprogram itself (total number or arguments in all SUBROUTINE, FUNCTION, or ENTRY statements in the subprogram). There is also a limit of 63 character subprogram arguments allowed.

1407 `ENTRY, FUNCTION, PROGRAM, or SUBROUTINE requires a name`

The statement requires a name and none was found. The statement is analyzed for further errors, then deleted. The routine is typed as a subroutine or function if the error is in a SUBROUTINE or FUNCTION statement.

1408 `FUNCTION name` *`xx`* `may not be assigned a value`

1409 `Functions with label arguments cannot be used in I/O statements`

According to FORTRAN 77, functions cannot have label arguments. ASCII FORTRAN *does* allow label arguments to have functions. However, there are implementation problems if functions with label arguments are coded in I/O statements. Therefore, they are disallowed in I/O statements.

1410 `A previous recursive call is in error`

1411 `Functions and arguments are inconsistent with COMPILER option DATA=REUSE`

The COMPILER statement option DATA=REUSE means that when the routine is entered, it reuses the automatic storage space of the calling program. This means that the program can't be returned to, since all register save areas are destroyed, so a function makes no sense. Also, arguments are passed in the automatic storage stack, and so passing arguments is not possible.

1412 `Substring expression must be type integer`

The expressions e_1 and e_2 in a substring reference (e_1:e_2) must both be type integer.

1413 `Incorrect substring reference`

A substring reference must have the format (e_1:e_2) immediately following a character scalar variable name or a character array element name, where e_1 and e_2 are integer expressions.

1414 `Constant used as substring expression out of range`

The values e_1 and e_2 in a substring reference (e_1:e_2) must satisfy the following relational expression: $1 \leq e_1 \leq e_2 \leq len$, where e_1 and e_2 are integer expressions, and *len* is the number of characters in the character scalar variable or character array element immediately preceding the format (e_1:e_2). When either e_1 or e_2 is a constant expression, checks are made at compilation time to see that the preceding relational expression is true.

1415 Substring value out of range for EQUIVALENCE item *xx*

The substring *start* and substring *end* values must satisfy the following relational expression: 1 ≤ *start* ≤ *end* ≤ *len*, where *len* is the declared character length of the item.

1416 Character substring reference invalid for EQUIVALENCE item *xx*

The substring reference for the item in equivalence must be on a character scalar or character array element.

1417 Text page size is too small for *xx* arguments

The compiler internal text page size is too small to hold all the information necessary for this many dummy arguments for this entry point to the subprogram. Either cut back the number of dummy arguments, or have your Systems Analyst rebuild the FTN compiler with a larger text page size.

1418 Text request of *xx* words cannot be filled

A request for internal text space in the compiler cannot be satisfied because the text space in the compiler cannot be satisfied because the text page size is too small. This is probably due to passing too many arguments to a function or subroutine. Either cut back on the number of arguments passed, or have your Systems Analyst rebuild the FTN compiler with a larger text page size.

1502 Statement function def. has too many arguments

The statement function definition has more than the compiler limit of 150 arguments. The statement is deleted.

1503 Stmt. function def. argument *xx* is incorrect

The indicated item appears where a dummy argument is expected, and either isn't a name or is the statement function name. The statement is deleted.

1504 Stmt. function name *xx* is previously defined

The statement function name was previously used in a way that precludes its use as a statement function name. The statement is deleted.

1505 Stmt. function param. *xx* is not used in def.

1601 Statement label is incorrect

Columns 1-5 of the following source line contain at least one character other than a space or a digit, and the line is not a comment line. The contents of columns 1-5 are ignored.

1602 Statement label *xx* is previously defined

Statement label already identifies a previous statement. This label is deleted.

1603 Labelled blank line treated as CONTINUE statement

1604 Columns 1-5 of continuation line should be blank

Nonblank characters appear in columns 1-5 of a continuation line. They are ignored.

1605 Statement should have a label

FORMAT statements and statements following a GO TO statement must be labeled. The statement is processed, but cannot be referenced or executed. This error may be caused by error 1602.

1606 Label *xx* referenced outside DEBUG packet

1608 FORMAT label *xx* is used incorrectly as a branch point

The label on the FORMAT statement was previously used incorrectly as the target of an IF or GO TO. Bad code may be generated for the statement.

1609 Label on this non-executable stmt. was previously referenced

Previous references to this label are in error. Any statement can be labeled, but only labeled executable statements (except for DEFINE FILE, ELSE, and ELSE IF) and FORMAT statements can be referenced by using statement labels. There is one exception: the DELETE statement can refer to any statement label.

1610 Stmt. references non-executable stmt. label *xx*

This statement is in error, since it refers to a label that can't be referenced. Any statement can be labeled, but only labeled executable statements (except for DEFINE FILE, ELSE, and ELSE IF) and FORMAT statements can be referenced by the use of statement labels. There is one exception: the DELETE statement can refer to any statement label.

1701 Wrong number of subscripts for array reference *xx*

A reference to the indicated array has the wrong number of subscripts. The statement is deleted.

1702 Subscripts required for array reference *xx*

The indicated array name appears in a context that requires a scalar reference. The statement is deleted.

1703 `Subscript expression is not constant`

Subscripted references in DISPLAY and NAMELIST statements must have constant subscripts. The statement is deleted.

1704 `Array` *`xx`* `is not dimensioned`

1705 `Subscript is out of range for array` *`xx`*

The constant subscript is out of range. The subscript is accepted as written except for subscript errors occurring in data initialization. In this case, the data initialization isn't done.

1706 `Subscripted reference` *`xx`* `not permitted here`

1707 `Subscript` *`xx`* `is of the wrong type`

A logical, complex, or character expression is used as a subscript. The statement is deleted.

1708 `Ref. to variable dimensioned array` *`xx`* `before ENTRY stmt.`

A reference is made to an array that is declared with variable dimensions but has not been described in a SUBROUTINE or ENTRY statement. This can result from omitting declarations for symbolic names of constants, then using these omitted constants as dimension specifications in a DIMENSION statement. Arrays so dimensioned appear to be variable-dimensioned.

1709 `Dimension bounds for` *`xx`* `cause compiler limit overflow`

When calculating the virtual origin value (location_counter_offset - sum_of_multipliers + byte_offset) for an array, the value exceeds the limit allowed by the compiler. You must change the dimensions for the array.

1710 `Relative address for` *`xx`* `exceeds limit`

This message is usually the result of an array reference such as ARR(70000) where the constant subscript value exceeds 65,535 words, the array is dimensioned less than 65,536 words, and the O option is not present on the @FTN control image. This message appears when an instruction is built by code generation and the address portion is truncated when filled into the 16- or 18-bit u-field of an instruction requiring relocation.

1801 `Expression is too complex - simplify`

The current expression is too complex for the expression scanner. It should be broken into two or more statements. This error can be caused by nesting array, function, and statement function references too deeply.

1802 Complex constant parts are not the same length

One of the components of a COMPLEX constant is REAL and the other is double precision. The shorter component is converted to double precision and the resulting constant is COMPLEX*16.

1803 Complex constant parts are not both real

1804 Constant exp. evaluation produces *xx* error

The combination of two constants has resulted in an arithmetic fault. The type of fault is indicated in the message. If the fault is UNDERFLOW, the result is set to zero. Otherwise, the constant valued expression is not evaluated.

1806 Subprogram ref. *xx* has wrong number of arguments

The indicated function or subroutine was first referenced with a different number of arguments. This statement is processed normally, unless the function is a MAX or MIN function, in which case the call is ignored. If the function is a FORTRAN-supplied function and referenced with no arguments, it is subsequently treated as a scalar. (See 7.3.2.2.)

1807 Stmt. function ref. *xx* has wrong number of arguments

The number of arguments in the statement function reference doesn't match the number in the definition. The statement is deleted.

1808 Subroutine name *xx* is used incorrectly

A subroutine is referenced as a function. The statement is deleted.

1809 Argument *xx* is of the wrong type

1810 Logical operand *xx* is of the wrong type

The expression contains a logical operator (.AND., .OR., .NOT.) with a nonlogical operand. The statement is deleted.

1811 Arithmetic operand *xx* is of the wrong type

The expression contains an arithmetic operator (+, -, *, /, **) with a nonarithmetic operand. The statement is deleted.

1812 Relational operand *xx* is of the wrong type

The expression contains a relational operator (.LT., .LE., .NE., .EQ., .GE., .GT.) with a logical or complex operand. The statement is deleted.

1813 Typeless operand *xx* is of the wrong type

You used a typeless operand in an expression with a complex or double-precision operand. The statement is deleted.

1814 Character operand *xx* is of the wrong type

The expression contains a concatenation operator with a noncharacter operand. The statement is deleted.

1815 Unary operator *xx* is used incorrectly

The indicated unary operator appears in a context that requires a binary operator, or it follows another unary operator. The statement is deleted.

1816 Binary operator *xx* is used as a unary operator

The indicated operator appears where an operand is required. The statement is deleted.

1817 *xx* is not a subroutine name

The name following the keyword CALL was previously used in a way that precludes it being a subroutine name. The CALL statement is deleted.

1818 Type conversion is incorrect

This message occurs if the target and expression in an assignment statement are of incompatible types (for example, character and integer). The assignment statement is deleted. This message also appears when the expression in a RETURN statement is not type integer. It is converted to an integer, if possible.

1819 Function ref. *xx* is inconsistent with compiler def.

1. A FORTRAN-supplied function name is referred to with the wrong number of arguments or with arguments of the wrong type. The function is assumed to be a user-supplied external function if this is its first occurrence. Error 1826 is also printed, informing you of this change.
2. This message can also appear when you use a generic function name without a local specification statement on the name.

1820 Character expression exceeds allowable length

A character string expression exceeds the compiler limit of 511 characters. The statement is deleted.

1821 *xx* argument is outside permitted range

An argument to the indicated FORTRAN-supplied function is incorrect.

1822 `xx argument is not normalized or outside allowable range`

An argument to the indicated FORTRAN-supplied function is incorrect.

1823 `Ref. to generic name xx may be inconsistent`

This message sometimes appears when you use a generic function name as a local variable with no specification statement on the name. The name is treated as a local scalar with default typing.

1825 `Division by zero detected in this statement`

Division by constant zero or a symbolic name of a constant whose value is zero is detected in an expression.

1826 `Function xx is now a user-supplied function`

The first occurrence of a compiler-defined function has the wrong number or type of arguments. The function is assumed to be user-supplied and loses any intrinsic properties.

1827 `Unary operator '(' follows constant in expression`

A left parenthesis cannot follow a constant in any expression syntax. This message can come out in a statement-function expansion. For example:

```
character*(*) sf
sf(arg) = arg(1:2)
print *, sf('1234')
```

This results in the expanded expression: '1234'(1:2) which isn't allowed (since only scalars and array elements can precede the (*e1*:*e2*) substring syntax).

1828 `Intrinsic name xx cannot be used as argument`

Certain intrinsic functions cannot pass as an argument to another routine because:

1. They are generated as inline functions and have no external entry point to pass and thus can't be called.
2. They have a calling sequence that is unique to their name, such as LOWERC and UPPERC, and thus cannot be called by a dummy subprogram name.

1829 `ACOB or PL/1 name xx cannot be a dummy argument`

You specified a dummy argument name as an external ACOB or PL/1 name. This is illegal. The dummy subprogram name is treated as an ASCII FORTRAN dummy subprogram name.

1830 `Intrinsic function name xx cannot appear in a CHARACTER type stmt`

The name UPPERC or LOWERC can't appear in a CHARACTER type statement when you use the name as an intrinsic function in the program unit. For example:

```
CHARACTER UPPERC          @defaults to CHARACTER*1
```

or

```
CHARACTER*8 UPPERC
```

followed by

```
xx = UPPERC(string)
```

causes UPPERC to be treated as a user-supplied function.

1831 `Main program name xx may not be the same as a subprogram name`

The name specified in the PROGRAM statement (that is, the program name) may not be the same as the name of an external user subprogram. The main program name cannot be called as a subprogram.

1832 `Possible precision loss - use xx function instead of xx`

The intrinsic functions REAL, SNGL, and FLOAT always return single-precision real (that is, REAL*4) results, no matter what type the argument is. You should specify the intrinsic function DBLE (in place of REAL, SNGL, or FLOAT) if you want a double-precision result.

For example, if D1 is type REAL*8 and CD1 is type COMPLEX*16, then the statement

```
D1 = REAL(CD1)
```

causes the real portion of CD1 to be converted to single-precision (because of the REAL function), and the assignment causes that single-precision result to be converted back to double-precision. In other words, there is a possible loss of precision because of converting from double- to single- to double-precision. The statement

```
D1 = DBLE (CD1)
```

causes no precision conversions and therefore no loss of precision.

Similarly, the intrinsic function CMPLX always returns a single-precision complex (that is, COMPLEX*8) result, no matter what type the argument(s) are. You should specify the intrinsic function DCMPLX (in place of CMPLX) if you want a double-precision complex result. For example, if D1 and D2 are type REAL*8 and CD1 is type COMPLEX*16, then the statement

```
CD1 = CMPLX (D1,D2)
```

causes D1 and D2 to convert to single-precision real to form the COMPLEX*8 result CMPLX (D1,D2), and the assignment causes that single-precision complex result to convert back to double-precision complex. The statement

```
CD1 = DCMPLX (D1,D2)
```

causes no precision conversions and therefore no loss of precision.

1901 `Assignment result` *`xx`* `is incorrect`

The item on the left-hand side of the assignment isn't permitted as the target of an assignment. The statement is deleted.

1902 `Assignment result` *`xx`* `is an active DO variable`

The indicated item is the index variable of an active DO loop. The statement is deleted.

1903 `Assignment result` *`xx`* `does not match expression`

1904 `Assignment result` *`xx`* `exceeds compiler table limit`

More than the compiler limit of 32 items appear in the left-hand side of a multiple assignment statement. The statement is deleted.

1906 `BITS pseudo function - illegal first argument`

The first argument of the BITS pseudo-function is a temporary. The result isn't inserted into the correct variable. This occurs when you attempt to take a BITS of a SUBSTR result.

2001 `Label reference` *`xx`* `is incorrect`

The statement label in an AT statement is missing, or it does not refer to a previous executable statement. The AT statement is deleted. This message can also occur if an *, $, or & in the actual argument list of a subroutine call isn't followed by a statement label, or if a statement label is passed as an actual argument in a function reference. In this case, the statement is deleted. This message is also generated if the statement label following END= or ERR= clause of an input/output statement appeared previously on a nonexecutable statement, in which case the statement is deleted.

2002 `Label reference expression is incorrect`

This message occurs for two reasons:

1. A statement of the form RETURN *i* is encountered in a subprogram that has no asterisks in its argument list.
2. The *i* specified in a RETURN *i* statement is either negative or too large for the asterisk list.

2003 `Assign variable` *`xx`* `is incorrect`

2004 `Label reference` *`xx`* `exceeds compiler table limit`

2005 `Statement label` *`xx`* `is not defined`

The indicated statement label is referenced, but never appears as the label of a statement. The reference may be deleted or it may be generated as a jump to an undefined address.

2006 `Keyword` *`xx`* `used as label reference in arithmetic IF stmt.`

The IF statement is recognized by the compiler as an arithmetic IF consisting of only the negative branch, which is a FORTRAN keyword. The warning is issued because you probably want a logical IF. For example:

```
IMPLICIT INTEGER (A-Z)
IF (LOGFCT) RETURN
```

2007 `Absence of list in Assigned GO TO inhibits global optimization`

The absence of the optional statement label list in an assigned GO TO inhibits global optimization and results in only local optimization being performed. An assumed list that contains every statement label that has been associated with the scalar integer variable in an ASSIGN or initialization statement could be constructed. However, it is impossible for the compiler to guarantee the completeness of such an assumed list because of the possible appearance of the scalar integer variable in EQUIVALENCE and/or assignment statements.

For example:

```
ASSIGN 10 TO LAB1
ASSIGN 20 TO LAB2
LAB1 = LAB2
GO TO LAB1
```

The assumed list for LAB1 would be 10. However, the complete list is 10,20.

Without a complete statement label list it is impossible to guarantee the correct analysis of program structure and control flow without which global optimization cannot produce valid results. A complete statement label list must be present on the assigned GO TO statement if global optimization is desired.

2008 `Presence of an Assigned GO TO inhibits global optimization`

The presence of an Assigned GO TO statement that transfers control to the head of a loop that is not a DO loop inhibits global optimization and results in only local optimization being performed. The backward movement and strength reduction optimizations performed on a loop require an area (commonly referred to as an initialization block) into which operations that are invariant in the loop can be moved. If transfers of control are made to the head of a non-DO loop from outside the loop, optimization must create an initialization block to be inserted immediately prior to the head of the non-DO loop. The transfers of control from outside the non-DO loop must then be modified to transfer control to the newly created initialization block. Since the point transferred to by an assigned GO TO is dependent on the contents of a variable that contains the address of any one of a list of labels, the needed modification cannot be made. Conversion of the assigned GO TO to a computed GO TO enables global optimization to be performed.

2104 `DEBUG facility and optimization are incompatible`

The combination of a DEBUG statement and an optimization option (V or Z) is not allowed, since bad code could be generated. Optimization is not equipped to handle debug code.

2105 `DEBUG SUBCHK not permitted for assumed-size array` *xx*

An assumed-size array (such as DIMENSION A(*)) can't be specified for subscript checking since the size of the array cannot be calculated.

2201 `DO variable is used in an outer DO`

The index variable of this DO statement is also the index variable of an earlier unclosed DO loop. The statement is accepted. The result of executing the DO loops is unpredictable.

2202 `Range of DO contains no executable statements`

There are no executable statements in the indicated DO loop. The DO statement is ignored.

2203 `DO loops are incorrectly nested`

The terminal statement of a DO statement was encountered, and the terminating statement for a later DO statement is not yet encountered. All unclosed DO loops encountered since the closing DO also close at this statement.

2204 `DO terminal statement number is incorrect`

The statement label that follows the keyword DO is incorrect. The DO statement is ignored.

2205 `DO terminal stmt. number is previously defined`

The DO terminal statement label must appear physically after the DO statement in the same compilation unit. The DO statement is ignored.

2207 `DO increment value is incorrect or inconsistent`

The increment value is not an integer, real, or double-precision expression or has constant value zero. This message can also occur if the initial, increment, and terminal values are all constants and the sign of the increment value is not consistent with the sign of the difference of the terminal and initial values. The DO statement is ignored.

2208 `DO terminal value is incorrect`

The DO terminal value is not an integer, real, or double-precision expression. The DO statement is ignored.

2209 `DO variable is subscripted`

The DO index variable can't be an array element. It must be a simple integer or real variable. The DO statement is ignored.

2210 `DO variable is of the wrong type`

The DO index is not an integer or real variable. The DO statement is ignored.

2211 `DO variable is incorrect`

There is an error in the DO index variable. The DO statement is ignored.

2212 `DO initial value is of the wrong type`

The initial value is not integer, real, or double precision. The DO statement is ignored.

2213 `DO increment value is of the wrong type`

The increment value is not integer, real, or double precision. The DO statement is ignored.

2214 `DO terminal value is of the wrong type`

The terminal value is not integer, real, or double precision. The DO statement is ignored.

2215 `DO terminal stmt. type prevents loop completion`

The terminal statement of the DO loop does not permit execution of the following statement. As a result, the increment and test parts of the DO loop are never executed.

2216 `I/O list contains redundant parentheses`

An element or sublist in an implied DO is enclosed in parentheses. The redundant parentheses are ignored.

2217 `Implied DO list contains no members`

The implied DO list is empty. The implied DO is ignored.

2218 `Implied DO list contains expression or constant`

The elements of an implied DO list should be variables, arrays, and array elements. For an input list, the implied DO is ignored. For an output list, the error is ignored.

2219 `Implied DO element must be subscripted array`

An element of an implied DO list used in data initialization is not an array element. The implied DO is ignored.

2220 `Syntax error in implied DO`

2221 `DO nesting exceeds compiler limit`

The level of DO and implied DO nesting exceeds the compiler limit of 25. The DO or implied DO is ignored.

2222 `DATA list contains expression or constant`

An element of a DATA statement variable list is not a variable, array element, or array. The list element is ignored.

2223 `Block IF nesting exceeds compiler limit`

The level of block IF statements exceeds the compiler limit of 25. The block IF statement is ignored. An END IF statement is required to terminate a block IF nesting level (which is started with a block IF statement, that is, IF (*e*) THEN).

2224 `END IF, ELSE, or ELSE IF stmt. is not in block IF structure`

The END IF, ELSE, or ELSE IF statement must have a matching block IF statement (that is, an IF (*e*) THEN statement at the same IF level) preceding it in the program unit. The END IF, ELSE, or ELSE IF statement is ignored. Each block IF statement must have exactly one corresponding END IF statement following it in the program unit (that is, one at the same IF level). Each block IF statement can also have zero or one corresponding ELSE statement and any number of corresponding ELSE IF statements following it in the program unit (at the same IF level).

2225 `Expression in ELSE IF stmt. is not type logical`

The expression *e* in the ELSE IF (*e*) THEN statement must be a logical expression. The ELSE IF statement is ignored.

2226 `A DO-loop and an IF-block are incorrectly nested`

When a DO statement appears in an IF block, ELSE IF block, or ELSE block, the range of the DO loop must be entirely within that block. Similarly, if a block IF statement (that is, IF (*e*) THEN) appears in the range of a DO loop, the corresponding END IF statement must also appear within the range of that DO loop.

2227 `ELSE or ELSE IF statement may not follow ELSE statement`

Once an ELSE statement appears at a given IF level, an END IF statement must appear before the next ELSE or ELSE IF statement at the same IF level. The ELSE or ELSE IF statement is ignored.

2228 `Block IF statement requires matching END IF statement`

Each block IF statement (that is, IF (*e*) THEN) must have exactly one corresponding END IF statement following it in the program unit. An implied END IF statement is inserted at the end of the program unit to match the block IF.

2229 `Statement cannot terminate a DO-loop`

The terminal statement of a DO-loop must not be a nonexecutable statement (except FORMAT) or one of the following executable statements: an unconditional GO TO, assigned GO TO, arithmetic IF, block IF, ELSE IF, ELSE, END IF, RETURN, STOP, END, or DO statement.

2230 `Implied DO list contains redundant parentheses`

2301 `FORMAT code` *`xx`* `is incorrect`

The indicated letter is not a valid edit descriptor for the FORMAT statement. The letter and all following characters are deleted up to the following non-period delimiter.

2302 `FORMAT code is not followed by an integer`

The indicated edit descriptor letter is not followed by a field-width designator. The edit descriptor letter is deleted.

2303 `FORMAT code integer component is too large`

The indicated integer is greater than 255. The value 255 is used instead of the large number.

2304 `FORMAT code` *`xx`* `fractional part is missing`

Either an edit descriptor that requires a *d* subfield (such as E*w.d* or F*w.d*) doesn't have the *d* subfield, or an edit descriptor for which the *d* subfield is optional (such as I*w*[.*d*]) has the period delimiter, but no value is specified for *d*. The value 0 is used for the missing subfield.

2305 `FORMAT code` *`xx`* `has zero field width`

The field width for the indicated edit descriptor has been specified as zero. The edit descriptor is deleted.

2306 `FORMAT has empty literal field`

The format statement contains a literal field containing zero characters. The empty literal field is deleted.

2307 `Repeat count is zero for` *xx*

The repeat count for the indicated edit descriptor or parenthesis is zero. The format is processed as though the zero repeat count is not specified.

2308 `Scale factor is not preceded by an integer`

A P field is not preceded by an integer. The value 0 is used.

2309 `FORMAT parenthesis nesting is too deep`

Parentheses are nested more than five deep, including the outermost pair that begin and end the statement. The format is translated as though no error has occurred, but group repetition is not done properly by the execution-time format scanner for groups that are too deep in the parenthesis nest.

2310 `Hollerith field ambiguity - list-directed FORMAT used`

The correct size of a Hollerith literal cannot be determined, because of a syntax error in specifying the field width. The FORMAT statement is replaced by an empty FORMAT statement, which forces list-directed input and output.

2311 `Hollerith field extends beyond end of FORMAT`

The length of a Hollerith literal is too long. It causes the remainder of the FORMAT statement to be absorbed into the literal.

For example:

```
FORMAT(1X,7HTitle)
```

The length (7) is too long. It should be 5. The FORMAT statement is replaced by an empty FORMAT statement, which forces list-directed input and output.

2401 `Arithmetic IF expression is of wrong type`

The parenthesized expression in the IF statement is not logical, so the IF is assumed to be arithmetic. However, the expression is not INTEGER, REAL, or DOUBLE PRECISION, the only types for which the required relations to zero are defined. The statement is ignored.

2402 `All three label positions of an arithmetic IF are void`

Since all three label positions of an arithmetic IF are void, control passes to the next statement. The arithmetic IF statement is no longer needed or a label position isn't filled in.

2501 `File reference number` *xx* `is incorrect`

In a DEFINE FILE statement, the unit specifier is not an integer, or has appeared as the unit specifier of a previous DEFINE FILE statement. The second DEFINE FILE statement is deleted. This message is also generated if the unit specifier of a BACKSPACE, REWIND, ENDFILE, FIND, READ, or WRITE statement isn't an integer expression or a character variable, character array name, or character array element where these are permitted. The statement is deleted.

2502 `Relative record number` *xx* `is incorrect`

The record number specification of a direct access I/O statement isn't an integer expression. The statement is deleted.

2503 `Input list member` *xx* `is an active DO variable`

The indicated variable appears in an input list. It is the index variable of an active DO or implied DO. The input list is ignored.

2504 `NAMELIST reference` *xx* `is incorrect`

The item encountered where the NAMELIST name is expected is not a valid name. The NAMELIST statement is deleted.

2505 `Associated variable` *xx* `is incorrect`

The associated variable of this DEFINE FILE or OPEN statement either is not an integer scalar or is used as the associated variable of a prior DEFINE FILE or OPEN statement. This DEFINE FILE or OPEN statement is deleted.

2506 `File format specification` *xx* `is incorrect`

The DEFINE FILE format specification is not one of the letters L, E, U, M, F, or V. The statement is deleted.

2507 `Number of records field` *xx* `is incorrect`

The *number-of-records* field of the DEFINE FILE statement is not an integer constant. The statement is deleted.

2508 `Size of record field` *xx* `is incorrect`

The *size-of-records* field of the DEFINE FILE statement requires a positive integer constant less than 262,144, which can't be found. The statement is deleted.

2509 `Character number` *`xx`* `is incorrect`

The record size specification of the ENCODE or DECODE statement isn't an integer constant, variable, or array element, or is negative. The statement is deleted.

2510 `Starting location` *`xx`* `is incorrect`

The location to which the ENCODE or DECODE data transfer is made is incorrectly specified. It must be an array, array element, or variable. The statement is deleted.

2511 `Characters processed name` *`xx`* `is incorrect`

The entity specified to receive the number of characters processed by the ENCODE or DECODE is not an integer variable or array element. The statement is deleted.

2512 `FORMAT reference` *`xx`* `is incorrect`

The FORMAT specification must be a FORMAT statement number, an integer variable containing a FORMAT statement label, or an array or character expression containing a run-time FORMAT. This message is generated if the FORMAT specification is a statement label but not a FORMAT statement label or is a variable but not of type integer or character. This message may be generated during the optimization process if it is determined that specified integer variable will not be assigned a FORMAT statement label in an ASSIGN statement when the referencing statement is executed. In any case, the statement is deleted.

2513 `Direct access or internal file I/O is list or NAMELIST directed`

The FORMAT specification in this direct access or internal file I/O statement is a NAMELIST name or an asterisk. Only formatted and unformatted I/O are permitted for direct access files, and only formatted sequential I/O is permitted for internal files.

2514 `Redundant 'ERR=' or 'END='`

A second ERR= or END= clause is encountered in the same statement. The repeated clause is ignored.

2515 `'END=' return is used incorrectly`

An END= clause is specified in a direct access READ or WRITE statement. The END= clause is ignored.

2516 `Input list member` *`xx`* `is an expression or constant`

The members of an input list must be variables, elements, or arrays. The input list is ignored.

2517 I/O list present with NAMELIST directed I/O

An I/O list is specified on a READ, WRITE, PRINT, or PUNCH statement that has a NAMELIST name instead of a FORMAT specification. The list is ignored.

2520 I/O list member *xx* is incorrect

The indicated item is not permitted to appear in an I/O list. The I/O list is ignored.

2521 DEFINE FILE option *xx* is incorrect

The indicated option is not valid for a sequential DEFINE FILE. The statement is deleted.

2522 File reference number is out of range

The unit specifier is negative or 0 or greater than 262,143. The statement is deleted.

2523 Internal file request is incorrect

Internal file I/O can be formatted and sequential only.

2524 IOSTAT field is incorrect

An input/output status specifier can only be an integer variable or an integer array element.

2525 Keyword *xx* is used incorrectly

A keyword used in an OPEN, CLOSE, or INQUIRE statement conflicts with another keyword, or is not permitted in this statement.

2526 Specification for *xx* is incorrect

The specification for this keyword is of the wrong type or is an expression in which this is not permitted.

2527 Only UNIT, IOSTAT, and ERR may be present with reread

2528 Keyword *xx* not permitted in CLOSE statement

2529 Keyword *xx* permitted only in INQUIRE statement

2531 Keyword *xx* not permitted in this statement

2532 UNIT or FILE specification must be present

2533 UNIT specification must be present

2602 Variable *xx* may be used before set to a value

The analysis of program flow determines that possible paths exist in the routine in which the value of the indicated variable can be referenced before any value has been assigned. It is your responsibility to ensure that the variable is initialized on all possible paths, or that the paths where the incorrect reference occurs can't be executed.

This warning message also appears when a character variable is only partially defined by an assignment statement before a use of the entire character variable. That is, only a substring of the character variable is defined by an assignment statement before the character variable is used. It is your responsibility to ensure that all byte positions of the character variable are defined before the use of the entire character variable.

2603 Arithmetic error in DO loop calculation

This error is issued in either of two cases:

1. When the evaluation of a constant-valued expression while calculating the iteration count causes an arithmetic fault; or
2. When the evaluation of a constant-valued expression, while performing the strength-reduction phase of global optimization, causes an arithmetic fault.

2604 Transfer of control into DO loop inhibits optimization

The branching structure of a DO-loop is too complex to be analyzed by the optimization phase. The code is correct, but not optimal. The complexity results from branches out of and into the DO-loop; most probably the branches are from the extended range of the DO-loop. You should consider simplifying the branching structure of the DO-loop to obtain optimal code from this compiler.

2605 Maximum number of reducible expressions in DO loop

The maximum number of reducible expressions for a DO-loop is encountered. Any further expressions in the loop are not considered for optimization.

2606 `Variable` *`xx`* `is referenced but is never assigned a value`

A variable name is referred to in the program but nowhere in the program is the variable assigned any value. A variable is assigned a value in a number of ways including an assignment statement, a DATA statement, and use of the variable as a subprogram argument. Ordinarily, reference to such a variable with no assigned value returns a zero value. However, the value referred to may be unknown, depending on the options used on the compilation and collection of the program.

2607 `Variable` *`xx`* `appears in a declaration but is never referenced`

The variable is specified in a type or DIMENSION statement but neither it nor any overlays is ever used in an executable statement. Check the program for misspellings, and consider eliminating the unnecessary variable.

2608 `Dummy argument` *`xx`* `is never referenced`

The dummy argument appears in the argument list of a FUNCTION, SUBROUTINE, or ENTRY statement, but is never used in an executable statement. Check the program for misspellings, and consider eliminating the unnecessary argument.

3100 `Stmt. function` *`xx`* `typed as CHARACTER*(*)`

The FORTRAN 77 standard requires a character statement function to have a length specification that is a constant integer expression. CHARACTER*(*) isn't allowed.

3101 `Stmt. function` *`xx`* `used as target of assign. stmt`

The FORTRAN 77 standard requires a statement function reference to appear in an expression. It does not allow a statement function reference to appear on the left-hand side of an assignment statement (at the outer level). ASCII FORTRAN allows this, and does not perform any type conversion (to convert the expanded statement function expression to the type of the statement function).

3102 *`xx`* `used as statement function actual arg`

The FORTRAN 77 standard requires a statement function actual argument to be an expression. It can't be a statement label, an array name, or a function name.

3103 `Char. function entry points have diff. lengths`

The FORTRAN 77 standard states that in a character function, all entry points must be typed with the same length specification; that is, all entry point names must have the same constant length (for example, CHARACTER*2), or all entry point names must be typed CHARACTER*(*).

3104 `Multiple assignment statement`

The FORTRAN 77 standard allows only one target variable in an assignment statement. ASCII FORTRAN allows the source item (on the right-hand side of the equal sign) to be stored to more than one target item (on the left-hand side of the equal sign, separated by commas).

3105 `Non-character item set to char. constant`

The FORTRAN 77 standard does not allow a noncharacter item to be the target (left-hand side of the equal sign) of an assignment statement, where the source item (right-hand side of the equal sign) is a character constant, since character-to-noncharacter conversion is not allowed. The standard has arithmetic assignment statements and character assignment statements, with no mixing allowed. ASCII FORTRAN allows this, with no conversion performed on the character constant (that is, it is simply stored to the target variable).

3106 `Dimension follows length for array` *xx*

The FORTRAN 77 standard states that the local length specification (if any) for a character type statement must follow the dimension information (if any). For example, the standard requires CHARACTER A(2)*3, while ASCII FORTRAN allows that syntax or CHARACTER A*3(2).

3107 `Data init. in spec. statement for` *xx*

The FORTRAN 77 standard allows data initialization only in the DATA statement. ASCII FORTRAN also allows it in explicit type and DIMENSION statements.

3108 `Length spec. *n appears for non-char. item`

The FORTRAN 77 standard allows a length specification of **n* (where *n* is an integer constant) or *(*e*) (where *e* is an integer constant expression) only for type character, in the explicit type, FUNCTION, and IMPLICIT statements. For example, the forms REAL*4, REAL*8, and COMPLEX*8 aren't allowed in the standard, but CHARACTER*2, CHARACTER*(2+3), and CHARACTER*(*) are allowed.

3109 `COMPLEX*16 data type is undefined`

The FORTRAN 77 standard defines the complex data type to be single precision (that is, composed of real and imaginary parts that are both single-precision real). ASCII FORTRAN allows single- precision complex (COMPLEX*8) and double-precision complex (COMPLEX*16).

3110 `COMPLEX*16 constant encountered`

The FORTRAN 77 standard has no double-precision complex data type (only single-precision complex). Therefore, a COMPLEX*16 constant is not allowed.

3111 `Format edit descriptor` *xx* `is undefined`

The repeatable edit descriptors J*w*, O*w*, R*w*, and E*w*.*d*D*e* are not defined in the FORTRAN 77 standard. The nonrepeatable edit descriptors +S, -S, and -*w*X are not defined in the standard. These edit descriptors are allowed in ASCII FORTRAN formats.

3112 *xx* `clause on I/O statement is undefined`

The listed clause is not allowed on the current input/output statement (OPEN, CLOSE, or INQUIRE) in the FORTRAN 77 standard.

3113 `END clause in WRITE statement`

The FORTRAN 77 standard does not allow an END=*s* clause on a WRITE statement.

3114 *u'r* `syntax used for direct access READ or WRITE`

The FORTRAN 77 standard does not allow an apostrophe (') to designate direct access I/O, as does ASCII FORTRAN. The standard has separate UNIT=*u* and REC=*r* clauses for the direct access READ and WRITE statements.

3115 `Non-char. array` *xx* `used as format specifier`

The FORTRAN 77 standard allows the following for a format specifier in I/O statements: statement label, integer variable, character expression, asterisk, or character array name. ASCII FORTRAN allows noncharacter array names, in addition to the standard list.

3116 `Namelist name` *xx* `used in I/O statement`

The FORTRAN 77 standard does not allow namelist I/O. The current statement is an I/O statement with one of the following formats:

```
READ(u,n,...)
READ n
WRITE(u,n,...)
PRINT n
PUNCH n
```

where *n* is a namelist name (declared in a NAMELIST statement).

3117 `Non-integer subscript used for array` *xx*

The FORTRAN 77 standard requires a subscript in an array reference to be an integer expression. ASCII FORTRAN allows real or double-precision expressions, as well as integer expressions.

3118 `CHARACTER*(*) item` *`xx`* `appears in concatenation`

The FORTRAN 77 standard allows a CHARACTER*(*) item (dummy scalar, dummy array element, or character function entry point) to appear as a concatenation operand only in a character assignment statement, and even then the concatenation may not appear in an argument (to either a function or statement function). ASCII FORTRAN allows a CHARACTER*(*) item to appear anywhere a character expression is allowed.

3119 `Non-character item compared to character constant`

The FORTRAN 77 standard allows a relational expression to compare the values of two arithmetic expressions or two character expressions. ASCII FORTRAN also allows a character constant to be compared for equality to an integer or real item if a P option is specified on the @FTN processor call.

3121 `Character length spec follows function name`

The FORTRAN 77 standard allows the syntax in the following example for a character function specification:

```
CHARACTER*5 FUNCTION C(arg)
```

ASCII FORTRAN also allows:

```
CHARACTER FUNCTION C*5(arg)
```

3122 `PARAMETER statement occurs among executable stmts.`

In the FORTRAN 77 standard, PARAMETER statements are only allowed to be mixed with IMPLICIT statements and specification statements. They can't occur after any DATA statements, statement function definitions, or executable statements.

3123 `Spec. stmt. occurs after DATA or stmt. function`

The FORTRAN 77 standard forces all specification statements to occur before any DATA statements or statement function definitions are encountered.

3124 `Internal subprograms are non-standard`

The FORTRAN 77 standard does not have the ASCII FORTRAN concept of internal subprograms, where a SUBROUTINE or FUNCTION statement occurring inside of another program unit with no intervening END statement causes the subprogram to be local to the containing external subprogram and to share its declarations.

3125 `Statement not allowed in BLOCKDATA subprogram`

The FORTRAN 77 standard does not allow the INTRINSIC or EXTERNAL statements to occur within a BLOCK DATA subprogram.

3126	`Statement is non-standard`

ASCII FORTRAN has the following statements that aren't in the FORTRAN 77 standard: FIND, PUNCH, ENCODE, DECODE, BANK, DEFINE, NAMELIST, DEBUG, AT, DEFINE FILE, START EDIT, STOP EDIT, DELETE, INCLUDE, DISPLAY, TRACE, COMPILER.

3127 `Over 19 continuation lines encountered`

The FORTRAN 77 standard allows up to 19 continuation lines. ASCII FORTRAN allows as many as necessary as long as the number of significant characters is under approximately 1,320.

3128 *xx* `equivalenced to a` *xx* `item`

The FORTRAN 77 standard does not allow character type items to be equivalenced to noncharacter type items.

3129 *xx* `common block contains character and noncharacter items`

The FORTRAN 77 standard doesn't allow character type items to be in the same common block as noncharacter type items.

3130 `Array` *xx* `equivalenced with wrong number of subscripts`

The FORTRAN 77 standard does not allow an array to appear in EQUIVALENCE with a number of subscripts that differs from the number of dimensions specified for the array. ASCII FORTRAN allows one subscript to appear on arrays in EQUIVALENCE even if the array was dimensioned with more than one dimension.

3131 `Parentheses missing on FUNCTION statement`

ASCII FORTRAN allows a FUNCTION statement without parentheses if the function has no arguments. The FORTRAN 77 standard requires them.

3132 `Slashes bracket dummy argument` *xx*

For compatibility reasons, ASCII FORTRAN lets you bracket a dummy argument with slashes:

```
   SUBROUTINE SUBX(a,/b/)
```

The FORTRAN 77 standard doesn't allow this.

3133 *xx* `used instead of asterisk to indicate a label`

ASCII FORTRAN allows a currency symbol ($) to indicate a label in a dummy argument list, and a currency symbol or ampersand (&) to precede a label when passing a label to a subprogram. The FORTRAN 77 standard uses an asterisk (*) only.

3134 `Function subprograms should not have label arguments`

The FORTRAN 77 standard only allows label arguments to be passed to and accepted by subroutines. ASCII FORTRAN also allows them with function subprograms.

3135 `xx used in EXTERNAL statement`

The FORTRAN 77 standard only allows a list of routine names on an EXTERNAL statement. ASCII FORTRAN allows the names to be preceded by & or * (& is ignored; * means it is a FORTRAN V subprogram). Also, ASCII FORTRAN allows the routine name to be followed by (*opt*), where *opt* is ACOB, PL1, or FOR to indicate the language of the routine (FOR is FORTRAN V).

3136 `Function subprogram has an alternate return`

The FORTRAN 77 standard doesn't allow labels to be passed to functions and therefore does not allow alternate returns. ASCII FORTRAN allows both.

3137 `Main program has a RETURN statement`

ASCII FORTRAN treats a RETURN statement in a main program as a STOP statement. The FORTRAN 77 standard doesn't allow a RETURN statement in a main program.

3138 `PARAMETER statement has missing parentheses`

According to the FORTRAN 77 standard, a PARAMETER statement needs opening and closing parentheses:

```
PARAMETER (I=2,j=3,char2='2')
```

ASCII FORTRAN also allows the form:

```
PARAMETER I=2,j=3,char2='2'
```

3139 `Intrinsic function xx used in constant expression`

The FORTRAN 77 standard states that only constant-valued expressions can be used in certain places such as the length specification in a PARAMETER statement. The standard defines constant-valued expressions as only having constants, other symbolic names of constants, and simple operators. No intrinsic functions such as SIN, REAL, and CHAR can be used, and only simple integer exponentiation can be used. ASCII FORTRAN allows the intrinsic functions and general exponentiation to be used wherever constant expressions are required (except for DIMENSION declarators). However, these are not available for use in constant-valued expressions if the compiler was generated such that the intrinsic functions are not available at compile time. This is rarely done, since the compiler as released is not generated in this manner. ASCII FORTRAN allows:

```
PARAMETER (sinpi1=1.0+sin(3.1416))
```

3140 `Label xx encountered in initialization list`

The FORTRAN 77 standard doesn't allow items to be initialized to label values in DATA statements, but ASCII FORTRAN does:

```
DATA i/&10/
```

3141 `DATA or FORMAT statement used to terminate DO-loop`

ASCII FORTRAN allows a DATA or FORMAT statement to be the terminator of a DO-loop; the FORTRAN 77 standard does not.

3142 `Octal constant encountered in initialization list`

ASCII FORTRAN allows octal values to be put into variables in DATA statements, and the FORTRAN 77 standard doesn't recognize the letter o for octal. For example:

```
DATA i/o040040/
```

3143 `Fieldata constant encountered in initialization list`

ASCII FORTRAN allows Fieldata values to be put into variables in DATA statements; the FORTRAN 77 standard has no Fieldata data type. For example:

```
DATA i/'abcdef'F/
```

3144 `Blank COMMON variable xx initialized`

The FORTRAN 77 standard prohibits blank common blocks from being initialized with DATA statements. ASCII FORTRAN allows it.

3145 `COMMON variable xx initialized in non-BLOCKDATA subprogram`

The FORTRAN 77 standard allows common blocks to be initialized only inside a BLOCK DATA subprogram. ASCII FORTRAN allows DATA statements for variables in common blocks in any kind of program unit.

3146 `Non-character item initialized to a character value`

The FORTRAN 77 standard doesn't allow noncharacter items to be initialized to character values in DATA statements. ASCII FORTRAN does allow this.

3147 `Digit string xx over 5 digits long`

The FORTRAN 77 standard only allows up to a 5-digit string on a PAUSE or STOP statement. ASCII FORTRAN allows more.

3148 `Hollerith literals are only allowed in FORMAT statements`

The FORTRAN 77 standard doesn't recognize Hollerith anywhere but in a FORMAT statement. ASCII FORTRAN allows Hollerith wherever a character constant can occur. When passed as arguments to a subprogram, Hollerith constants should *not* be passed if the dummy argument in the receiving program is type CHARACTER. If Hollerith is passed as an argument, the dummy argument must not be type character. (This seemingly strange situation is spelled out in the FORTRAN 77 standard, Appendix C, paragraph C7. It is done this way to allow existing programs that don't use character data type to continue to operate correctly under the new standard.)

3149 `PARAMETER constant` *`xx`* `used as part of complex constant`

The FORTRAN 77 standard does not allow the use of symbolic names of constants as the REAL or IMAGINARY portions of a complex constant. ASCII FORTRAN allows this. For example:

```
COMPLEX c
PARAMETER (p1=2.,p2=3.)
c = (p1,p2)
```

3150 `'$' used as a character in the name` *`xx`*

Use only the letters A-Z and the digits in a name under the FORTRAN 77 standard. ASCII FORTRAN also allows $ to be used, though it is discouraged since accidental conflicts with the run-time library entry points can occur if subprograms have $ in their names.

3151 `'&' used as concatenation operator`

The concatenation operator is the double slash (//). ASCII FORTRAN also allows the use of the ampersand (&).

```
CHARACTER*80 c80,c22*22,c2*2
c80 = c22//c2
c80 = c22&c2
```

3152 `Computed GOTO has empty label position(s)`

FORTRAN 77 does not allow missing positions in a computed GO TO. ASCII FORTRAN does allow void positions. If a void position is selected, the next statement in sequence is executed.

```
GO TO (10,20,,40) I+2
```

3153 `Integer variable` *`xx`* `used for a label`

FORTRAN 77 requires labels in the lists of arithmetic IF and computed GO TO statements. ASCII FORTRAN also allows integer variables, which must have received the value of some label via an ASSIGN or DATA statement.

```
        ASSIGN 10 TO ILAB
        GOTO (20,30,ILAB,40) I+3
   20   IF(J+2) 30,ILAB,40
```

3154 `Arithmetic IF has empty or missing label positions`

FORTRAN 77 does not allow empty or missing positions in an arithmetic IF. ASCII FORTRAN allows this, and if that position is selected, the next statement is executed. In the following example, the next statement is executed.

```
   I=2
   IF(I) 10,20
```

3155 `xx should be in an xx statement`

The FORTRAN 77 standard says that any subprogram name that is passed as an argument must be either in an EXTERNAL statement (if it is a user-supplied subprogram), or in an INTRINSIC statement (if it is a FORTRAN intrinsic function). ASCII FORTRAN also allows the names to be passed if they are used previously in the source program as a valid subprogram reference.

```
   X = SIN(A)
   Y = MYPROG(B)
   CALL SUB2(SIN,MYPROG)
```

The previous example is nonstandard, but ASCII FORTRAN lets it go through with no error.

3156 `xx inline comments encountered in xx`

Inline comments are nonstandard in FORTRAN 77. For example:

```
   x = 1.5        @ comments here
```

3157 `Function xx is not a standard intrinsic`

The function referred to is an intrinsic function in ASCII FORTRAN, but isn't included in FORTRAN 77.

3158 `Argument type incorrect for xx - generic function used`

ASCII FORTRAN allows any intrinsic function to be used as a generic name of a function; for example, DSIN(real) calls SIN automatically. This capability is limited to a small number of generic functions in FORTRAN 77.

3159 `Standard intrinsic function xx contains mixed argument types`

The FORTRAN 77 standard requires all arguments to be the same type for those intrinsic functions that have more than one argument. ASCII FORTRAN allows mixed argument types.

3160 `END statements cannot be continued`

The FORTRAN 77 standard says in section 11.14 that:

- An END statement can only be written in columns 1-72 of an initial line.
- An END statement cannot be continued via continuation lines.
- No other statement may have an initial line that appears to be an END statement.

This means that the following are illegal:

```
      END
     1 X = 22.

(I.E. ENDX = 22. )

      E
     xN
     xD
```

The first example is treated as an assignment statement, but gets this non-std-usage message. The second example is treated as an END statement, but gets this non-std-usage message.

3161 `Argument to xx converted to type xx`

The FORTRAN standard does not allow certain types of arguments for some intrinsic functions, but when ASCII FORTRAN can automatically convert them to the proper type, this non-standard usage message is issued.

3162 `Function xx has non-standard xx argument`

The FORTRAN standard does not allow certain types of arguments for some intrinsic functions, but when ASCII FORTRAN can automatically substitute another function for the generic one, this non-standard usage message is issued.

3163 `Argument number xx in stmt. function reference xx is of the wrong type`

The FORTRAN 77 standard requires a statement function's arguments to agree with the types of the corresponding dummy arguments. FTN accepts an argument of a different type, without conversion. This message is issued at the statement function reference if the statement function is not nested more than 10 deep.

3200 `Branch from line xx into an IF-block or DO-loop`

The FORTRAN 77 standard does not allow the transfer of control into an IF-block or DO-loop made from outside the block. This type of transfer is allowed by ASCII FORTRAN. In the case of DO-loops, the "Extended Range of a DO-loop" feature defines how the transfer of control should be done.

6301 `BANK and VIRTUAL statements ignored in checkout mode`

Your module, when compiled and executed with the checkout option (@FTN,C), must be self-contained. Since it is not collected into an absolute, the BANK statement has no meaning and is ignored.

The VIRTUAL statement is ignored in checkout mode since the virtual feature requires a different bank structure than an unbanked user checkout program.

6302 `COMPILER statement ignored in checkout mode`

The BANKED= options of the COMPILER statement aren’t allowed in checkout mode since your program cannot be banked. The LINK=IBJ$ and the DATA= options are allowed.

Appendix E
Conversion Table

Table E-1 shows if a specific data type can be converted to another desired data type. If conversion is possible, a brief description of the method of conversion is given.

Table E-1. Conversion Methods for Arithmetic Data

Target Type Desired ↓	**Present Type of Expression**								
	INTEGER	REAL	DOUBLE PRECISION	COMPLEX	**COMPLEX *16**	Character	Logical	**Octal**	Statement Number
INTEGER	Store.	Fix. Store.	Fix. Store.	Ignore imaginary part. Fix. Store.	**Ignore imaginary part. Fix. Store.**	**Not possible unless it's a char. literal or Hollerith constant, in which case, store.**	Not possible.	**Only in DATA stmt. (See 6.12 for details.)**	Store.
REAL	Float. Store.	Store.	Truncate to single. Store.	Ignore imaginary part. Store.	**Ignore imaginary part. Truncate to single. Store.**	**Not possible unless it's a char. literal or Hollerith constant, in which case, store.**	Not possible.	**Only in DATA stmt. (See 6.12 for details.)**	Not possible.
DOUBLE PRECISION	Double. Store.	Expand to double. Store.	Store.	Ignore imaginary part. Expand to double. Store.	**Ignore imaginary part. Store.**	**Not possible unless it's a char. literal or Hollerith constant, in which case, store.**	Not possible.	**Not possible .**	Not possible.
COMPLEX	Float. Store to real part. Store 0 to imaginary part.	Store to real part. Store 0 to imaginary part.	Truncate to single. Store to real part. Store 0 to imaginary part.	Store.	**Truncate both parts to single. Store.**	**Not possible unless it's a character literal or Hollerith constant, in which case, store.**	Not possible.	**Not possible .**	Not possible.
COMPLEX *16	**Double. Store to real part. Store 0 to imaginary part.**	**Expand to double. Store to real part. Store 0 to imaginary part.**	**Store to real part. Store 0 to imaginary part.**	**Expand both parts to double. Store.**	**Store.**	**Not possible unless it's a char. literal or Hollerith constant, in which case, store.**	Not possible.	**Not possible .**	Not possible.

continued

Key

Fix	is the same as applying the INT intrinsic function.
Float	is the same as applying the REAL intrinsic function.
Double	is the same as applying the DBLE intrinsic function .

Table E-1. Conversion Methods for Arithmetic Data (cont.)

Target Type Desired ↓	**Present Type of Expression**								
	INTEGER	REAL	DOUBLE PRECISION	COMPLEX	**COMPLEX *16**	Character	Logical	**Octal**	Statement Number
Character	Not possible.	Not possible.	Not possible.	Not possible.	Not possible.	Truncate or blank fill, if necessary, Store.	Not possible.	**Only in DATA stmt. (See 6.12 for details.)**	Not possible.
Logical	Not possible.	Not possible.	Not possible.	Not possible.	Not possible.	Not possible.	Store.	**Only in DATA stmt. (See 6.12 for details.)**	Not possible.

Key

Fix	is the same as applying the INT intrinsic function.
Float	is the same as applying the REAL intrinsic function.
Double	is the same as applying the DBLE intrinsic function .

Appendix F
Tables of FORTRAN Statements

Table F-1 and Table F-2 specify whether a FORTRAN statement is nonexecutable or executable, respectively.

Table F-1. Nonexecutable Statements

General Category	Statement
Specification statements	DIMENSION COMMON EQUIVALENCE **BANK** EXTERNAL **NAMELIST** PARAMETER INTRINSIC IMPLICIT Explicit type statements: INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, CHARACTER SAVE **VIRTUAL**
Data initialization statement	DATA
Format statement	FORMAT
Function defining statement	Statement function definition **(DEFINE)**
Program unit headings	PROGRAM FUNCTION SUBROUTINE ENTRY BLOCK DATA

continued

Table F-1. Nonexecutable Statements (cont.)

General Category	Statement
Program control statements	**INCLUDE** **DELETE** **EDIT (START EDIT, STOP EDIT)** **COMPILER**
Debug facility	**DEBUG** **AT**

Table F-2. Executable Statements

General Category	Statement
Assignment statements	Arithmetic assignment statement Logical assignment statement Character assignment statement ASSIGN
Control statements	Unconditional, assigned, and computed GO TO statements Arithmetic and logical IF statements Blocking statements (block IF, ELSE IF, ELSE, and END IF) CALL CONTINUE RETURN STOP PAUSE DO END

continued

Table F-2. Executable Statements (cont.)

General Category	Statement
I/O statements	READ WRITE PRINT **PUNCH** REWIND BACKSPACE ENDFILE **DEFINE FILE** **FIND** **ENCODE** **DECODE** OPEN CLOSE INQUIRE
Debug facility	**TRACE ON** **TRACE OFF** **DISPLAY**

Appendix G
ASCII FORTRAN Input/Output Guide

G.1. General

ASCII FORTRAN performs run-time input/output using the processor common I/O modules or the Shared File System (SFS). Processor common I/O consists of a number of common bank modules that access the various file types and a number of processor interface common bank modules for the various ASCII processors such as COBOL, FORTRAN, and PL/I.

Code compiled by ASCII FORTRAN references the ASCII FORTRAN I/O processor interface module, which references the required processor I/O common bank modules. The Processor Common I/O System (PCIOS) is used by the ASCII FORTRAN run-time system for I/O compatibility between the various language processors. See the *OS 2200 Processor Common Input/Output System (PCIOS) Administration and Programming Reference Manual*, *Level 6R2* (7831 0588).

All ASCII FORTRAN SDF and ANSI files are accessed through PCIOS. The specialized ASCII FORTRAN interface module handles all formatting and conversions, and all symbiont I/O (PRINT, PUNCH, READ). The actual I/O is done in PCIOS modules. However, because of a PCIOS restriction, I/O processing can't be done to word-addressable mass storage. I/O processing must be done to sector-formatted mass storage only.

You can access the following file formats:

- System Data Format (SDF)
- American National Standards Institute (ANSI) magnetic tape interchange format
- ASCII symbiont

G.2. System Data Format File

An SDF file produced by ASCII FORTRAN is explained in this subsection.

G.2.1. SDF File Description

An SDF file produced by ASCII FORTRAN contains a label record, data records or record segments, an end-of-file record, and possibly bypass records.

Each record or record segment has a control word in front of it. This control word has the form:

l	*p*	*f*	*s*

where:

l

is the length of the data record or record segment if bit 35 is set to 0. A data record or record segment length must be in the range 0 to 2,047 words. If bit 35 is not 0, the value of *l* can be:

05400	End of reel
05033	Label image
05100	Continuation
07700	End-of-file image
040x	Bypass images where x is the length of the bypass image with a range $0 \le x \le 63$

p

is the *l* field from the previous record or record segment. This field is used in backspacing.

f

is:

n

For unformatted records this field contains a count *n* of the number of bits used in the last word of the record.

077

Dummy record. Dummy records are produced when a direct-access file is skeletonized.

s

is the record segment indicator:

00	Record not segmented
01	First segment of record
02	Last segment of record
03	Not first or last segment of the record

SDF records are segmented to conserve main storage space when processing large unformatted records.

A direct-access SDF file has the same format as a sequential SDF file, except all records in a direct-access file are the same length. When a direct-access file is skeletonized, dummy records are written in the file.

G.2.1.1. SDF Labels

The SDF label contains the file's maximum record size, the block size the file is written with, the record segment size, the file type, and other information pertinent to processing this file. The label record is the first record of the file.

The format of a PCIOS SDF label record is:

0	050	033	035	ASCII flag
1	file			
2	name			
3	type	orig	level	recovery offset
4	block size			record size
5	date			
6	address of next file			
7	largest record key allowed			
8	largest record key written			
9	rcd\1\offset			segment size
10	ffc	skel flag	0	record size in characters
11	address of previous file			
12	0			
13	0			
14 .. 27	... (Words 14 through 27 are written as they appear in the label area at open time.)			

Word 0

Is the label control word in which a Fieldata X (035) identifies this SDF file as produced by the Processor Common Input/Output System (PCIOS) or the Shared File System (SFS). The ASCII flag has a value of 1 for ASCII files.

Words 1 and 2

Contain the internal file name used when creating the file.

Word 3

type

indicates the type of SDF file:

01	sequential
02	direct

orig

indicates the originator:

01	ASCII FORTRAN levels prior to 9R1
02	ASCII PL/I
03	ASCII FORTRAN
04	ASCII COBOL
05	Mixture of 1 with 2, 3, or 4

level

indicates that the C2DSDF/C2SSDF file is created or extended by PCIOS level 4R1 or higher.

recovery offset

is valid only if level is set; it is set by C2DSDF/C2SSDF and holds the word offset of record n+1 in the last file block. C2SSDF uses this for output extend recovery in the event of system failure during an extend operation.

Word 4

block size

specifies the block size (in words) of the file.

record size

specifies the record size (in words) of the file.

Word 5

Contains the file creation date.

Word 6

Contains the mass storage address of the next file for sequential stacked files on mass storage. This field is 0 for SDF direct files.

Word 7

Indicates the maximum relative record number that is allowed for an SDF direct file. This word is 0 for sequential files.

Word 8

Specifies the highest record number actually written in this SDF direct file. This word is 0 for sequential files.

Word 9

rcd\1*offset*

indicates the word offset for relative record number 1.

segment size

indicates the segment size (in words) used when creating this file.

Word 10

ffc

is the FORTRAN record format control code.

skel flag

is a skeletonization flag for SDF direct files.

record size in characters

is the record size in characters of the file.

Word 11

Contains the address of the previous file if files are stacked on mass storage. For the first file, this word is -0 since there is no previous file.

Words 12 and 13

Aren't currently used.

Words 14 through 27

Are written as they appear in the label area (PTLFMA) when the open output is called.

G.2.1.2. SDF Data Records/Record Segments

SDF data records can be character data or word-oriented. The length of an SDF data record is maintained in the file as words. An SDF record can be any length, but it is automatically segmented when the record is written if it exceeds the segment size for that particular file.

The segment size for a particular file is specified in the File Control Table (FCT) for that file. Specify the segment size for an SDF sequential output file in the SEG clause of the OPEN statement or in the *ss* field of the DEFINE FILE statement. The default segment size specified in the Storage Control Table (SCT) is used for output files when no OPEN or DEFINE FILE statement specifies the segment size (see G.8). When the file is an input file, the segment size field of the SDF label is used.

To minimize I/O overhead when processing sequential files, specify the segment size to be the smaller of: the size in words of the largest records written, or 2047 (the largest segment possible).

The maximum record size for the SDF file pertains to formatted records, since these records are not processed by segments like unformatted records are (although they are segmented when they are written in the file). Formatted records are processed by the FORTRAN editing routines as complete records. To ensure there is space for the complete record in the intermediate record area, the maximum size that may be encountered must be known.

The maximum record size for a particular file is maintained in the FCT for that file. You specify the maximum record size for an SDF sequential output file with the MRECL clause of the OPEN statement or the rs field of the DEFINE FILE statement. The default maximum record size specified in the SCT is used for output files when no OPEN or DEFINE FILE statement specifies the maximum record size (see G.8). When the file is an input file, the maximum record size field of the SDF label is used.

G.2.1.3. SDF Block Size

The block size for an SDF sequential data file is specified by the BLOCK clause of the OPEN statement or by the *bs* field of the sequential DEFINE FILE statement. If not specified, the default size is 224 words. The block size is independent of record size since records span the block when necessary. Record size can be larger than block size. When choosing a block size, be aware of the following:

- The block size field *bs* (or *bsize*) specifies the number of words in the block. To minimize I/O overhead, block size should be larger than segment size.
- Tape files must be processed with a block size at least as large as the block size they are created with.

- Mass storage files must be processed with the exact size they are created with if they are created with a block size that isn't an increment of 28 words.
- For disk mass storage files, block size should be a multiple of the prep factor (28, 56, or 112) to eliminate the Executive read-before-write.
- A number of systems processors (such as ED and DATA) and symbionts require a block size of 224 words.
- Mass storage files created with a block size that is an increment of 28 words can be read with a different block size. However, this block size must be an increment of 28 words.
- I/O overhead is reduced by choosing a block size which is greater than or equal to the record size (or a multiple of it) and which is a multiple of the prep factor.

G.2.1.4. SDF End-of-File Record

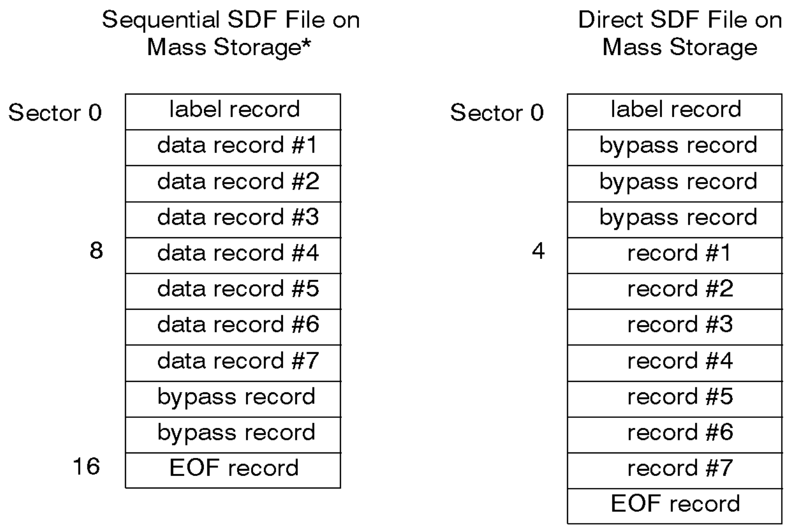
The SDF end-of-file record is written in the last word of the last block. It has the form:

07700	*p*	0	0

where *p* is T1 from the previous image control word.

G.2.1.5. SDF File Layout

The SDF file layout has one of two forms:



* *For SDF sequential tape files, the file can be blocked as shown with eight sectors per block.*

G.2.2. SDF File Processing

When the FORTRAN compiler encounters an I/O statement, it generates an I/O packet and a call to the FORTRAN I/O processor interface module (in C2F$ or in a relocatable library). The interface module does a lookup in the File Reference Table (FRT) using the unit specifier as an index.

The FRT entry has a pointer to the FCT if that unit is already open. This pointer is zero if the file is not open. When the pointer is zero, the interface module gets space from the free core area for the FCT and the buffers.

The interface module generates the FCT using the information given on the OPEN or DEFINE FILE statement, the input label, or default information. The interface module calls on PCIOS or SFS to open the file for input or output when required. The processor interface module also calls on PCIOS or SFS to read, write, backspace, and close (ENDFILE) when specified. Record editing takes place in the ASCII FORTRAN library.

G.2.2.1. Sequential Access

Records are input/output using a double buffering process. Records are generated in an intermediate record area on output and then buffered to the file device. On input, records are processed in the input buffer if they are in the buffer or moved to the intermediate record area if they are not. When records are buffered out to the file they are automatically segmented at the segment size specified in the FCT.

The intermediate record area must be large enough to handle the largest formatted (this includes list-directed and namelist) record it may encounter or the largest segment it may encounter for unformatted records. Initially this area is set to the default maximum record size or default segment size specified in the SCT (see G.8), whichever is larger. When an OPEN or DEFINE FILE statement is encountered with a larger maximum record size specification or a larger segment size specification, this area is freed and another area is obtained from the free main storage area.

Unformatted records are processed by segments. Therefore, the maximum record size field of the OPEN or DEFINE FILE statement doesn’t apply. The segment size specified in the FCT determines the size of the segments.

You must specify the maximum record field on the OPEN or DEFINE FILE statement to be able to write formatted records larger than the default record size specified in the SCT.

The ENDFILE statement writes the SDF end-of-file record in the file and writes the last block on the file device. Each block in an SDF file is the uniform block size specified for that particular file. This uniformity is accomplished by filling the last block with bypass records and writing the end-of-file record as the last word in the last block. After an ENDFILE statement, the file is positioned so that a WRITE statement begins another SDF file on the device. WRITE, REWIND, BACKSPACE, and ENDFILE (creates an empty file) are allowed after an ENDFILE.

The BACKSPACE statement backspaces over all segments of a record if the record is segmented.

G.2.2.2. Direct Access

Direct-access processing uses the record number to compute the location of the record in the file. The record is then read into a buffer unless the record already resides in the buffer due to a previous read.

The buffer size specified by the BUFR clause of the OPEN statement is used to determine the number of records read/written on an Exec I/O request. A larger buffer size reduces the number of I/O requests needed in cases where record access tends to be localized.

If skeletonization is specified when a direct-access file is initially created, dummy records are written to the file resulting in all necessary storage being acquired if not already acquired. The dummy records, if not written over by subsequent WRITE statements, are recognized as nonexistent records on subsequent READ statements. Detection of a dummy record on a direct-access read is indicated by error code 1053 (see G.9.3).

If skeletonization is not specified when a direct-access file is initially created, no dummy records are written to the file. Nonexistent records are not recognized on a direct-access read of a nonskeletonized file. In fact, if only a maximum file size is provided when assigning the file, storage for areas of the file not yet written to is not acquired until a write to that area takes place.

If nonexistent records must be recognized on a read of a direct- access file, the file must be skeletonized when initially created and when extended.

Reading areas of a nonskeletonized direct-access file before they are written to may result in an I/O error with a status of 5 if the area meant to hold the requested record is not yet acquired.

Direct-access records are segmented only when the records are longer than 2,047 words.

Direct-access SDF files can be read by using sequential I/O statements. Records are read in ascending record number order. Only records previously written are returned (that is, bypass and dummy records are ignored).

G.2.3. SDF Files Not Written by Processor Common I/O

SDF files not produced by the processor common I/O system, such as FORTRAN V SDF files, can be read sequentially with ASCII FORTRAN if the data in the record is compatible with ASCII FORTRAN. These files can only be read. Backspacing isn’t allowed. You can’t use OPEN options that imply changes to the file (EXTEND) with files that are not created by PCIOS.

No attempt is made to determine what type of record is used. If an unformatted read is specified, the record is treated as an unformatted record. If a formatted read is specified, the record is treated as a formatted record.

Formatted records are read into the intermediate record area and if the SDF label specifies Fieldata (bit 0 of the label control word = 0), the complete record is translated to ASCII before the record is edited. If the file has formatted records larger than the default maximum record size, an OPEN or DEFINE FILE statement must be used.

If the file has segmented records, it is assumed that they are segmented using the SDF continuation control image (051). For formatted reads, all segments of the record are read into the record area before the record is edited. For unformatted reads, the record is processed by segments. If unformatted records are read that are larger than the default segment size specified in the SCT, you must use the OPEN or DEFINE FILE statement to indicate this.

The files can reside on tape or mass storage. If mass storage contains stacked files (WRITE, ENDFILE, WRITE), only the first file is read. You can copy these files to tape with the B option. They are then processed as tape files.

When a PCIOS file is updated by a processor that does not use PCIOS, the PCIOS file becomes a file that was not created by PCIOS. FORTRAN programs should only read such a file.

G.3. ANSI Magnetic Tape Interchange Format

The ANSI file formats produced by ASCII FORTRAN comply with the American National Standards Institute (ANSI) magnetic tape interchange format as described in the American National Standards proposal X3L5/365T, dated September 27, 1973, and titled “Magnetic Tape Labels and File Structure for Information Interchange.”

G.3.1. ANSI File Description

An ANSI file can be labeled or unlabeled. If the J option is present on the assign of the tape, an unlabeled ANSI file is read or written. If the J option isn’t present, a labeled ANSI file is read or written.

A labeled ANSI file produced by ASCII FORTRAN contains a header label group, data blocks, and a trailer label group. Data blocks are separated from the header and trailer labels by a tape mark. Files on a tape are separated by a tape mark.

The header label group consists of the VOL1 label, the HDR1 label, and the HDR2 label. The trailer label group consists of the EOF1 label and the EOF2 label. If a file spans tape reels, the EOV1 and EOV2 labels terminate all file sections except the last file, which is terminated by the EOF1 and EOF2 labels.

Labels are written using the Executive Tape Labeling System (TLS). The volume labels are present only in the first header label group of each tape and are automatically processed by TLS.

Labeled ANSI magnetic tapes have the following structures. Asterisks (*) indicate tape marks.

- For a single file, single volume:

```
VOL 1 HDR1 HDR2* Data Block 1 Data Block 2 Data Block 3 * EOF1 EOF2**
```

- For a single file, multivolume:

```
VOL1 HDR1 HDR2* Data Block 1 Data Block 2 Data Block 3 * EOV1 EOV2**

VOL1 HDR1 HDR2* Data Block 4 * EOF1 EOF2**
```

- For a multifile, single volume:

```
VOL1 HDR1 HDR2* Data Block 11 Data Block 12 * EOF1 EOF2*

        HDR1 HDR2* Data Block 21 Data Block 22 * EOF1 EOF2**
```

- For a multifile, multivolume:

```
VOL1 HDR1 HDR2* Data Block 11 Data Block 12

        Data Block 13 * EOF1 EOF2** HDR1 HDR2* Data Block 21 * EOV1 EOV2**

VOL1 HDR1 HDR2* Data Block 22 Data Block 23 * EOV1 EOV2**

VOL1 HDR1 HDR2* Data Block 24 * EOF1 EOF2* HDR1 HDR2* Data Block 31* EOF1
EOF2**
```

An unlabeled ANSI file produced by ASCII FORTRAN allows the use of the ANSI record formats by sites that do not have TLS.

All of the ANSI record formats are available on output when creating an unlabeled ANSI tape with ASCII FORTRAN. Creation of a null or empty unlabeled ANSI tape file isn't permitted.

On input, all ANSI record formats can be read provided the unlabeled ANSI tape is created by ASCII FORTRAN or the data on the unlabeled ANSI tape meets the requirements of the ANSI standards. Reading data blocks from unlabeled foreign tapes is possible only when using the U, F, or FB record formats.

Since there are no labels, PCIOS/SFS is unable to compare record size, block size, buffer offset, and record format. You must ensure that the values passed to PCIOS/SFS (using the OPEN and DEFINE FILE statements) are correct for the file to be read.

The structure of an unlabeled ANSI magnetic tape is:

- For a single file, single volume:

```
Data Block 1 Data Block 2 Data Block 3**
```

- For a single file, multivolume:

```
Data Block 1 Data Block 2 Data Block 3* Swap Block** Data Block 4**
```

- For a multifile, single volume:

```
Data Block 11 Data Block 12* Data Block 21 Data Block 22**
```

- For a multifile, multivolume:

```
Data Block 11 Data Block 12 Data Block 13* Data Block 21*Swap Block**

Data Block 22 Data Block 23* Swap Block** Data Block 24*Data Block 31**
```

You can specify the following record formats using the OPEN statement described in 5.10.1 or the sequential DEFINE FILE statement described in 5.6.6. The data block format is dependent on the record format and blocking factor chosen.

Undefined

The record is written without control information appended to it. The records may be variable in length. Each data block contains only one record.

Fixed-unblocked

The record is written without control information appended to it. All records are of the same length. Each data block contains only one record.

Fixed-blocked

Basically the same as fixed-unblocked; however, each block contains one or more records. Each data block has the same number of records except possibly the last block. The last block is truncated if it doesn't contain the full number of records.

Variable-unblocked

Appended to each record are four ASCII characters of control information containing the total number of characters in the record (including the four control characters). The records can vary in length. The data block contains only one record.

Variable-blocked

The same as variable-unblocked except that the block contains as many complete records as possible.

Variable-unblocked-segmented

The record is segmented at the block boundary. Appended to each record segment are five characters of control information. The first character is the record segment indicator and the remaining four are the segment length. The data block contains only one segment of the record.

Variable-blocked-segmented

Similar to variable-unblocked-segmented; however, each block can contain more than one record segment but never more than one segment of the same record.

The segment indicator character has the following meaning:

0	Record begins and ends in this segment.
1	Record begins but doesn’t end in this segment.
2	Record neither begins nor ends in this segment.
3	Record ends but doesn’t begin in this segment.

ASCII FORTRAN I/O doesn’t produce any control information on the front of the data block; hence the buffer offset field of the HDR2 label is set to zero. Files with such information can be read with ASCII FORTRAN, but the OFF clause of the OPEN statement or the buffer offset field of the DEFINE FILE statement must indicate the size in characters of this control information. No attempt is made to interpret this control information. It is skipped over when reading the file.

ANSI files are treated as character data files and are processed internally in quarter-word mode. The actual format of the data written on the tape depends on whether or not the tape is written through a data transfer format.

To produce an ANSI file that is to be interchanged, a data transfer format must be available and the quarter-word byte channel data word format A must be specified in the format field of the ASG control command. You can specify the 6-bit packed format B or the 8-bit packed format C, but the tapes are not interchangeable because of the way the data is written to the tape. Specify the format you want in the format field of the ASG control command. The format specified when the tape is written must also be specified when the tape is read. If a data transfer format is not available, the default formats are 8-bit packed for 9-track tapes and 6-bit packed for 7-track tapes.

Block padding can be done on output, depending on the format with which the tape is written. The pad character is the ASCII circumflex.

Format	Result
Quarter-word	No padding is done, since the block is truncated at the last data character of the data block.
8-bit packed	The block can contain 0, 1, 2, or 3 pad characters, since an increment of words must be written.
6-bit packed	Blocks can contain 0, 1, 2, or 3 pad characters, since an increment of words must be written.

The block size for an ANSI file is specified by the BLOCK clause of the OPEN statement when using PCIOS. If not specified, the default size is 132 characters.

G.3.2. ANSI File Processing

ANSI files are processed much the same as SDF files with the following exceptions:

- ANSI records are always moved to the intermediate record area on input and generated in the intermediate record area on output. This means that unformatted records aren't processed by segments so the maximum record field of the OPEN or DEFINE FILE statement must reflect the largest record being read or written, whether it is formatted or unformatted.
- ANSI records are moved character by character to and from the buffers rather than by words.
- The BACKSPACE request must not be specified if the records are blocked.
- Unformatted records must not be written to a file using quarter-word data transfer format A unless all the variables are character type, since this format strips the ninth bit of each quarter word. Therefore, if this bit is not zero, the block is truncated.

G.3.3. ANSI Interchange Tapes from Other Systems

ANSI Interchange tapes written on other systems must be read through data transfer format in quarter-word mode. If read with the other formats (8-bit or 6-bit) the data would not be aligned correctly in the read buffer.

These ANSI files can have control information in the front of the data block. If so, this must be specified in the OFF clause of the OPEN statement or the buffer offset field of the DEFINE FILE statement. This control information is ignored and can be from 1 to 99 characters long.

These files can contain pad characters (ASCII circumflex) after the data in the data block, but the last character of a data record must not be an ASCII circumflex pad character.

Labels other than HDR1, HDR2, EOF1, EOF2, EOV1, or EOV2 are ignored.

G.4. ASCII Symbiont Files

To read symbiont input files or write symbiont output files, dedicate certain unit reference numbers to these files. Do this by generating the File Reference Table (FRT) (see G.6) or by using an OPEN or DEFINE FILE statement. The unit reference number dedicated to the particular type of symbiont file desired is referenced in input/output statements.

OPEN, READ, INQUIRE, and CLOSE are the only I/O statements allowed for an input symbiont file.

OPEN, WRITE, ENDFILE, INQUIRE, and CLOSE are the only I/O statements allowed for output symbiont files. ENDFILE is allowed only for the APUNCH$ and APNCHA$ files and causes an @EOF card to be punched.

Queue alternate print (APRNTA$) and punch (APNCHA$) files for symbiont processing as follows:

- If the alternate file is assigned to the run as a new file prior to execution of the FORTRAN program with an @ASG,C statement, the FORTRAN I/O does a @BRKPT when the file is closed. You do the @FREE and @SYM to have it printed or punched.
- If the alternate file is assigned to the run with the @ASG,A statement prior to the execution of the FORTRAN program, the FORTRAN I/O does a @BRKPT when the file is closed. You do the @SYM to have it printed or punched.
- If the alternate file is not assigned to the run when a batch FORTRAN program is executed, the FORTRAN I/O doesn't do an assignment but allows the Executive to assign the file when the first ER APRNTA$ or ER APNCHA$ is done. (See the *OS 2200 Exec System Software Executive Requests Programming Reference Manual*, 7830 7899). The FORTRAN I/O directs output to an alternate print file using @BRKPT when the file is closed. The Executive automatically queues the file for printing or punching when the breakpoint is done. In demand, you must assign the alternate file.

The default symbiont record sizes are as follows:

- APRINT-APRNTA-AREADA - 132 characters
- APUNCH-APNCHA - 80 characters
- AREAD - 80 characters

You can change these defaults with the OPEN or DEFINE FILE statement.

The run-time I/O library allows the use of SYMB$ in place of AREAD$ for a symbiont READ statement when the following two conditions are met:

1. Exec level 37R2D or higher is used.
2. An OPEN statement is present for the AREAD$ file. It indicates that the length of the record is greater than 80 characters.

Since SYMB$ is used with the truncation option, an attempt to read a record length that is longer than the specified record length for that AREAD$ file results in a warning message. The execution of the READ statement continues unless there is an ERR clause present. The record content is the truncated record.

The warning that appears is a change from the fatal error issued when AREAD$ reads a record that is larger than the specified record length for that AREAD$ file. This fatal error is necessary since AREAD$ doesn't truncate the record and overwrites whatever follows the record buffer.

The use of SYMB$ for a symbiont READ provides the capability of reading from a terminal with a line image length greater than 80 characters.

G.5. Unit Specifier and File Assignment

The FORTRAN language refers to files through a unit specifier. The OS 1100 Operating System maintains files under an external file name of the form:

```
[ [qualifier] *]file-name[ (F-cycle) ] [ / [read-key] [ / [write-key] ]
```

See the *OS 2200 Exec System Software Executive Requests Programming Reference Manual*, 7830 7899, for a description of the external file name.

The unit specifier must be linked to the external file name before the file can be referenced. The external file can't be word-addressable mass storage (equipment types 020-027) because of a PCIOS restriction. The unit specifier is linked to the external file name in one of three ways:

1. If the FILE clause is present in the OPEN statement, an @USE *unit specifier*, *file-name* is performed at execution time. The file name in the FILE clause is restricted to the form:

   ```
   [[qualifier]*]file-name[.]
   ```

2. Use the unit specifier as the *file-name* when the file is assigned:

   ```
   @ASG,T 4,U,BLANK
   ```

3. Use the unit specifier as an alternate or internal file name by linking it with the external file name using the USE command:

   ```
   @ASG,A FILEX
   @USE 4,FILEX
   ```

When a FORTRAN program that refers to a file is executed, that file may already be assigned to the run, cataloged, or nonexistent.

If the file is already assigned, the unit specifier must be linked as described previously.

If the file is cataloged and not assigned to the run, it is automatically assigned by the FORTRAN I/O module. The file must be cataloged with the unit specifier as the *file-name* or, if the FILE clause of the OPEN statement is not used, a USE command must be performed prior to execution linking the unit specifier with the cataloged file name.

If the file is a shared direct-access file, it is not assigned by the FORTRAN I/O module. You can use the FILE clause in the OPEN statement described in method 1 or use method 3 to attach the unit number to the file name. A qualifier is required for ITF files.

For sequential access files, if the file is neither assigned nor cataloged, a temporary mass storage file is assigned by the FORTRAN I/O module, with the default size determined by the Exec. For direct access files, the track size is determined from the OPEN statement clauses, the DEFINE FILE statement parameters, or the FORTRAN I/O module default (not Exec default) of 128 tracks. If the location in the file reference table designates this as a symbiont APRINT$, APUNCH$, or AREAD$ file, then no file assign is attempted since file space for these symbionts is handled by the Executive.

Once a unit specifier is associated with a file and that file is opened, the unit specifier can't be associated with another file until the first file is closed using either the CLOSE statement or the CLOSE service subprogram.

G.6. File Reference Table Element - F2FRT

The element F2FRT is the File Reference Table (FRT). The FRT is a variable length table of one-word entries that is used to link the unit specifier with the file control table for the physical file being processed. The unit specifier is used as an index into the FRT.

The entries in the FRT contain a pointer to the file control table if the file is open or 0 if the file is not open. In the latter case, an automatic open occurs when that particular unit is referenced with an I/O statement. The FRT entry can also contain a code in S2 designating the unit as a symbiont file as follows:

Code	File Indicated
051	APRINT$ symbiont
052	APUNCH$ symbiont
053	AREAD$ symbiont
054	APRNTA$ alternate symbiont
055	APNCHA$ alternate symbiont
056	AREADA$ alternate symbiont

The format of the file reference table is:

-3	0	053	a	f
-2	0	052	a	f
-1	0	051	a	f
F2FRT$ +0	0	s	a	f
1	0	s	a	f
2	0	s	a	f
3	0	s	a	f
4	0	s	a	f

..	...			
n	0	s	a	f

where:

f

is set to 0 until a reference is made to the unit. Then it contains the address of the file control table.

a

indicates the file status:

0	Never referenced
1	Open
2	Closed
3	Reread unit

s

is a symbiont code or 0.

S1 of each word in the file reference table is zero for units used with normal FORTRAN I/O; it is nonzero for units used by NTRAN$.

F2FRT -1, -2, -3 are used for those I/O statements that don't have a unit designated, implying one of the symbionts APRINT$, APUNCH$, or AREAD$.

An installation can generate a new file reference table by remaking the F2FRT element as follows:

```
@PDP,M FTNPROC,FTNPROC
@MASM,SI F2FRT,F2FRT
   F$FRT      f      . Procedure call
   PR         l      .
   PU         l      .
   CR         l      .
   APR        l      .
   APU        l      .
   ACR        l      .
   RR         l      .
   END
@EOF
```

where:

f

is the largest unit specifier to be accessed; PR, PU, CR, APR, APU, and ACR specify the symbionts APRINT$, APUNCH$, AREAD$, APRNTA$, APNCHA$, and AREADA$, respectively. RR specifies that the unit is a reread unit.

l

is a unit specifier list of the form u_1, u_2, . . . ,u_n. Each *u* must be in the following range: $0 \leq u \leq f$. The units specified are designated as symbiont files of the type specified in the operation field (PR, PU, etc) or as a REREAD unit. If a unit is specified in more than one operation, it gets the designation of the last operation field.

The operation fields PR, PU, and so on can appear in any order, but F$FRT must be the first operation.

FTNPROC can be found in the FORTRAN library (see 9.5.3).

When an entry in the file reference table is designated as one of the symbiont types above, it must be used for that type of symbiont file unless an OPEN statement is used to specify another file type.

F2FRT is released with the following designation:

```
@PDP,M FTNPROC, FTNPROC
@MASM,SI F2FRT,F2FRT
   F$FRT      30
   PR         6
   PU         1
   CR         5
   RR         0
   END
@EOF
```

Use the OPEN statement to override and modify the symbiont code field specified when the file reference table is built, provided the unit was never opened before or has a file status of closed before the OPEN statement. The new code (or 0 for SDF and ANSI) remains in effect for the duration of the run or until reset by another OPEN statement.

G.7. Free Core Area Element - F2FCA

Storage for record areas, buffers, file control tables, and floating-point conversion is obtained from the free core area when required. A scratch area must be added when using the sort/merge interface (see L.3 and the CORE= clause in L.4). The free core area is generated as follows:

```
@PDP,M FTNPROC,FTNPROC
@MASM,SI F2FCA,F2FCA
   F$FCA     s      . procedure call
   END
@EOF
```

where FTNPROC is an element found in the FORTRAN library (see 9.5.3) and *s* is a number indicating the amount of storage required. It can be 0 or void. The amount of space required is calculated by using the following formula:

$s = a + b_1 + b_2 \ . \ . \ . +b_n +c$

where:

a

$1+2*m_1 + (m_2+m_3)*(\textit{default- segment-size})$

where:

m_1 = *default-segment-size* or largest record size, whichever is larger.

m_2 = 2, if any DEBUG statements are present in the FORTRAN program; otherwise, $m_2 = 0$.

m_3 = 2, if any calls to PDUMP or FTNPMD or any F-option FTN compilations are present in the FORTRAN program; otherwise, $m_3 = 0$.

b

There is a *b* for every file active and the size of *b* depends on the type of file (SDF, ANSI, or symbiont).

For symbiont files:

b = 64 + 1

For SDF direct files:

b = 64 + 2*(*record-size +1*) + 2**prep-factor*

For SDF shared direct files:

b = 64 + 2* (*record-size +1*) + 448

For SDF sequential files:

b = 64 + (2*(*block-size-in-words*) +1) + 1

For ANSI files:

b = 64 + ((2*(*block-size-in-chars* + 3)) /4) + 1

c

Miscellaneous core requirements:

c = 25 plus the following, as applicable

1. the amount of any scratch area needed for the sort/merge interface (see Appendix L).
2. 1024 words if the MOVWD$ and/or MOVCH$ routines are used with banked arguments.
3. 200 words if FORMATs need to be interpreted at run time.
4. 24 words if OPEN, CLOSE, and/or INQUIRE statements are used.
5. enough room for local variables and subroutine linkage if DATA=AUTO or DATA=REUSE are used.

If *s* is void or 0, no space is reserved. Instead, ER MCORE$ is performed to add space to the end of the control D-bank, as required.

The default record size, block size, and segment size are given by parameter to the F2SCT element (see G.8).

The standard F2FCA is released with no space reserved, so the MCORE$ method is used. The library element FTNPROC is on the ASCII FORTRAN release tape in file 3.

When *s* is other than 0, or is void, the space reserved must be sufficient to fill a memory request or the program terminates in error.

If any subroutines not written in ASCII FORTRAN are called by an ASCII FORTRAN program, these subroutines should not allocate and deallocate storage. If any MCORE$ or LCORE$ is executed by these subroutines, generate element F2FCA with the size of storage needed by the ASCII FORTRAN program.

G.8. Storage Control Table Element - F2SCT

The element F2SCT is the storage control table that contains register save areas, working storage, and pointers to the file reference table and the free core area header. In addition, F2SCT contains a location that contains the default record size, the default segment size, and the default block size for SDF sequential files. Change these default fields at your installation by remaking the F2SCT element as follows:

```
@PDP,M FTNPROC,FTNPROC
@MASM,SI F2SCT,F2SCT
   F$SCT x,y,z . procedure call
   END
@EOF
```

where:

x

is the default SDF record size desired in words. The range is $1 \le x \le 262{,}000$ but a core area of size $2*x$ is reserved for the processing of records.

y

is the default SDF sequential segment size desired in words. The range is $1 \le y \le 2{,}047$.

z

is the default SDF sequential block size desired in words.

F2SCT is shipped with the following default sizes:

$x = 33$
$y = 111$
$z = 224$

FTNPROC is found in the FORTRAN library (see 9.5.3).

G.9. Input/Output Errors

When an input/output error is detected by the ASCII FORTRAN I/O handler, the action taken depends on whether or not an ERR clause (see 5.2.6) contingency is specified and whether or not an input/output status specifier (see 5.2.8) is present.

G.9.1. ERR Clause Specified

If an error specifier (ERR clause) is specified and an error or warning condition is encountered, the I/O status word PTIOE in the storage control table is set and transfer is made to the statement specified by the ERR clause. In addition, if the I/O status clause is present, the IOSTAT variable receives the contents of PTIOE prior to the transfer. An error message isn't printed.

The I/O status word is set as follows:

s	u	c

where:

c

indicates the cause of the error. Its value is the integer clause number specified on the error clauses listed in G.9.3.

s

indicates the substatus of the error if any.

If *c* is 1 then *s* is the I/O status coded from the I/O packet.

For ANSI files, if the error involves an ER to TLBL$, the ER TLBL$ error code is returned in the substatus field.

For those errors that have no substatus, *s* is 0.

u

is the integer unit specifier of the file in error:

$0 \le u \le f$	The unit number, where *f* is the largest unit specifier allowed (see G.5)
-1	The APRINT$ symbiont
-2	The APUNCH$ symbiont
-3	The AREAD$ symbiont
-0	The unit is undetermined

Test the particular fields of the I/O status word by using the functions:

IOC( ) cause
IOS( ) substatus
IOU( ) unit

In addition, if the input/output status specifier is present, the particular fields are available by using the IOSTAT variable.

G.9.2. ERR Clause Not Specified

When an error specifier isn't specified, but an I/O status clause is specified:

- the I/O status word PTIOE is set;
- the IOSTAT variable receives the contents of PTIOE; and
- control immediately transfers back to the program if an error is detected. If a warning condition is detected, execution of the current statement is continued.

If neither an ERR clause nor an I/O status clause is specified:

- an error or warning message is printed;
- for errors, all open files are closed and the program terminates; and
- for warnings, execution of the current statement continues.

The error message has the form:

```
FTN ERR ON UNIT u c
```

where:

u

may be a positive unit number, PRINT, PUNCH, READ, or ILLEGAL.

c

is one of the error clauses listed in G.9.3. For some error clauses, a second line is printed to provide additional information regarding the error.

G.9.3. Error Clause Listing

When the common storage management system detects error conditions, it prints one of the following error messages:

```
STORAGE FULL
BAD FREE ADR
BAD FREE LEN
BAD CHAIN
```

These error messages have no effect on PTIOE or the IOSTAT variable.

The following two lists are of the warning and error clauses applicable to the ASCII FORTRAN I/O error messages. The number listed is the number set in the clause field of the I/O status word PTIOE. The first list gives the errors detected by PCIOS; the second list gives the errors and warnings detected by FORTRAN library routines. The errors detected by the Shared File System for shared direct-access files are described in the *OS 1100 UDS Shared File System (UDS SFS 1100) Administration and Support Reference Manual*, *Level 2R2A*, 7831 0786.

1 `I/O STATUS CODE` *xx*

An ER in PCIOS received the nonzero I/O status code on completing an I/O request (see the *ER Programming Reference Manual*, 7830 7899).

6 `MAX REC SIZE NOT CONSISTENT WITH FILE`

The C2DSDF module returns a bad status while trying to open a file for direct input.

9 `EMPTY OR INVALID FILE STRUCTURE`

The C2SSDF module returns a bad status while trying to backspace a file opened for sequential access, open a file for output with extend option, or read a file with an I/O error. The file originator isn't PCIOS.

10 `INVALID DATA BLOCK STRUCTURE`

The C2SSDF module returns a bad status while trying to backspace a file opened for sequential access.

12 `FILE NOT SDF DIRECT`

The C2DSDF module returns a bad status while trying to open a file for direct input.

13 `FILE NOT CONSISTENT WITH DEFINE FILE`

The C2ANSI module returns the error while trying to open the file for input.

16 `FILE NOT SDF`

The C2SSDF module returns the error while trying to open the file.

37 `BLOCKSIZE NOT MULTIPLE OF FIXED RECSIZE`

The C2ANSI module returns the error while trying to read (write) from (to) the file.

39 `INCORRECT DATA BLOCK COUNT`

The C2ANSI module returns this error while executing a tape swap or a close on the file.

50 `ERROR ON TBL$ REQUEST`

The C2ANSI module returns this error while attempting an open, close, or tape swap on this file.

65 `MASS STORAGE FILE OVERFLOWED`

The C2SSDF module returns this error while attempting a write to the file.

66 `BLOCKSIZE INCONSISTENT WITH FILE`

The C2SSDF module returns this error while attempting to open the file for sequential access.

67 `INCORRECT VARIABLE FORMAT CONTENTS`

68 `FILE LABEL LACKS EXTENDED PARAMETERS`

69 `FILE STRUCTURE LACKS AN END-OF-FILE`

70 `MAXIMUM INTERCHANGE RECORD SIZE EXCEEDED`

71 `TAPE LABELING SYSTEM NOT AVAILABLE`

72 `LOSS OF POSITION ON THE TAPE UNIT`

73 `SKELETONIZATION INCONSISTENT WITH FILE`

74 `EXTENDED FILE SMALLER THAN ORIGINAL FILE`

C2DSDF returns the error when the size of the file to be extended is smaller than the size of the original file.

75 `FTN ERR ON UNIT` *u*

The following errors and warnings are detected by FORTRAN library routines. Asterisks (*) indicate warnings. The double asterisk (**) preceding clause 1015 indicates the mistake is fatal for nonformatted records and list-directed DECODE records, and is a warning for other formatted records.

1001 `INVALID RECORD NUMBER`

The record number specified in a direct access READ, WRITE, or FIND statement is not in the range 1 ≤ record number ≤ max/rcd/num, where max/rcd/num is specified in a direct access OPEN or DEFINE FILE statement.

1002 `INAPPROPRIATE UNIT`

The unit number specified in the I/O statement is bad for one of the following reasons:

- The unit number specified is not in the range 0 ≤ unit number ≤ largest unit number referenced. Refer to G.6 for a discussion on the largest unit number referenced.
- An attempt is made to read from a printer, write or punch to a card reader, and so on.
- The file reference table is overwritten.

1003 `FILE ASSIGNMENT FAILED`

One of the following file assignments failed:

- ASG,A of a file whose status is specified as OLD in an OPEN statement.
- ASG,T of a file whose status is specified as NEW, SCRATCH, or UNKNOWN in an OPEN statement or the file was being opened by a statement other than the OPEN statement.
- The order of entries in the PCT prevents access to the desired file. Putting an @USE name of the unit number on the file may resolve this problem.

1004 `ATTEMPTED OPEN RANDOM ON TAPE FILE`

An attempt is made to open a tape file as a direct access file.

*1005 `ILLEGAL FORMAT CHARS - TREATED AS BLANKS`

The warning message is given only once per statement. It is given for one of the following reasons:

- The format given to the run-time library routines in an array or character expression contains an unknown format type or an illegal combination of values. The illegal combination could be A*w.d* since *.d* can't be used with an A*w* format. The unknown format type is any alphabetic character not used as a format type now or it can mean a missing alphabetic character when one should occur, such as *w.d*. The scanning of the format array or expression continues following the warning unless too many errors are found.
- The values in a FORMAT statement may be encoded badly by the compiler. The encoded FORMAT contains an illegal code. The scanning of the FORMAT continues after the warning is given.

1006 `UNIT NOT OPEN FOR RANDOM INPUT/OUTPUT`

A direct access READ, WRITE, or FIND statement specifies a unit (other than one attached to a symbiont) that is open for sequential access.

1007 `UNIT NOT AVAILABLE FOR BUFFERED I/O`

One of the following statements attempts to access a symbiont:

- Unformatted READ or WRITE
- REWIND
- BACKSPACE
- Direct access READ, WRITE, or FIND

1008 `FILE ASG'D CAN'T HOLD ALL DIRECT RECORDS`

A user-assigned file isn't large enough to hold all of the records requested in the direct access OPEN or DEFINE FILE statement.

1009 `UNIT NOT AVAILABLE FOR SEQUENTIAL I/O`

One of the following statements attempts to manipulate a file open for direct access.

A sequential DEFINE FILE, READ, or WRITE

- ENDFILE
- REWIND
- BACKSPACE

1010 `READ TYPE AND RECORD TYPE DO NOT CONFORM`

The type (formatted, unformatted, list-directed, namelist) of a sequential or direct access read differs from that of the record to be read.

1011 `ATTEMPTED READ AFTER WRITE`

An attempt is made to read a record from a sequential file that is currently open for output.

1012 `ENDFILE ONLY LEGAL FOR PUNCH SYMBIONTS`

The only symbionts for which an ENDFILE can be executed are PUNCH$ and APUNCH$.

1013 `READ/WRITE TYPE INHIBITED FOR THIS FILE`

An attempt is made to execute a formatted (unformatted) read or write on a direct access file that contains only unformatted (formatted) records.

`1014` `ATTEMPTED TO READ PAST AN END-OF-FILE`

`**1015` `RECORDS EXCEEDING MAX LENGTH ARE FAULTY`

During a formatted read or write, the format can't read or write more than the record length. This warning message is given if the format requires a length greater than the given record length.

During an unformatted read or write of direct access files, the read or write attempts to require more length than the record contains. The message appears for this fatal error.

During an unformatted write of ANSI files, the write attempts to require more length than the record contains. The message is given for this fatal error.

During list-directed DECODE statements, the size of the DECODE block or internal storage area is exceeded.

`*1016` `FORMAT TYPE NOT SAME AS INTERNAL TYPE`

The type of the input/output list item is the internal type. The format type given requires a conversion to store the value to the list item on input or to write the list item value on output. A conversion is not done when the types don't match, that is, E*w.d* format with an integer list item. The run-time routine gives a warning message and assumes that it can try and use the given value with that particular format code.

`1017` `ABSOLUTE VALUE OF I/O ITEM TOO LARGE`

The absolute value of the exponent of a floating-point number is too large. The number of input digits for an integer exceeds the amount that can be used to hold the number internally. The fatal error message is given to prevent an overflow during conversion of the number during input or output routines.

`*1018` `INPUT DATA DOES NOT CORRESPOND TO TYPE`

The warning message appears when the data read isn't consistent with the type of the input list item. Up to 132 characters of the record in error are printed.

Floating-point

any characters other than numerals, plus, minus, and blank, or an incorrect form of a value.

Octal

any characters other than numerals or blanks.

Integer

any characters other than numerals or blanks.

Character

for list-directed input, a character string not enclosed in apostrophes.

1019 `NAMELIST NAME HAS MORE THAN SIX CHARS`

- The namelist name in the input data is longer than the legal six characters.
- The variable name in the input data in the present namelist is longer than six characters.

1020 `INCORRECT CHARACTER IN NAMELIST INPUT`

The error message appears when:

- the first nonblank character following the namelist name in the input isn't an alphabetic character for a variable name;
- the subscript of the variable name isn't followed by an equal sign;
- the length for a Hollerith input field is negative;
- the input data is Hollerith but the namelist variable isn't real or integer;
- the input data is literal but the namelist variable isn't of type character;
- the variable name is missing before the equal sign in the input data;
- the input data is octal but the list item is logical; or
- the input subscript contains something other than numerals, plus, minus (if lower bounds are present), left and right parentheses, commas, and blanks.

1021 `NAMELIST INPUT HAS TOO MANY SUBSCRIPTS`

The subscript of the input variable name contains more than seven dimensions.

*1022 `NAMELISTS DO NOT CONTAIN VARIABLE NAME`

The variable name in the input data is not part of the present namelist. This is a warning message. The namelist read skips this name and continues reading at the next variable name.

1023 `UNIT NOT CONSISTENT WITH FILE FORMAT`

The inconsistent file format arises in one of the following situations:

- A sequential ANSI OPEN or DEFINE FILE is attempted and the associated file is not a tape file.
- A sequential DEFINE FILE is attempted and either the DEFINE FILE statement or the Define File Block, if present, attempts to override a predefined (within the FRT) symbiont file format.

1024 `LIST NOT SATISFIED BY LAST FORMAT GROUP`

The FORMAT is not an empty format but does not contain any repeatable editing codes to handle the items in the input/output list. This is a fatal error message.

1025 `UNABLE TO FILL SPACE REQ FROM FREE CORE`

Insufficient free storage area exists to satisfy the dynamic storage requirements of a run-time routine. Refer to G.7.

1026 `INCORRECT VARIABLE FORMAT CONTENTS`

During ENCODE/DECODE before ASCII FORTRAN level 8R1, the message was produced for list-directed ENCODE/DECODE statements, which were not allowed.

If a format in an array or character expression gets more than 10 errors, the fatal diagnostic appears. If the size of *w*, *d*, *e*, or *p* is too large for the field width, this fatal diagnostic occurs.

1027 `FILE ALREADY OPEN WITH ANOTHER UNIT #`

An attempt has been made to open a file that is already associated with an open unit different from the one specified in the OPEN statement. A file can't be associated with two open units at the same time. A second line to the error message indicates the file involved.

```
   FILE INVOLVED IS -- filename
```

The *filename* is the file that your program tried to OPEN on two different units.

1028 `INVALID LITERAL IN OPEN/CLOSE CLAUSE`

A literal argument appearing in one of the clauses of an OPEN or CLOSE statement is either misspelled or invalid.

1029 `VALUE IN OPEN IS NEGATIVE OR TOO LARGE`

The value specified for one of the clauses in the OPEN statement is out of range. A second line to the error message indicates which clause is in error.

1030 `OPEN/CLOSE/INQ OPTION MISSING OR ILLEGAL`

One of the errors listed below was encountered in an OPEN, CLOSE, or INQUIRE statement. A second line to the error message indicates which error condition occurred.

- In an OPEN statement, if a file is being opened for unformatted I/O, the BLANK clause may not be present. For this case, the second line to the error is:

```
CLAUSE(S) IN ERROR -- BLANK
```

- The RECL clause is missing on an open of a direct access file. For this case, the second line to the error message is:

  ```
  CLAUSE(S) IN ERROR -- RECL
  ```

- Clauses that pertain only to a direct (sequential) open are present on a sequential (direct) open. See 5.10.1. For this case, the second line to the error message is:

  ```
  CLAUSE(S) IN ERROR -- DIRECT
  ```

  If sequential clauses were present on a direct open, the following message appears:

  ```
  CLAUSE(S) IN ERROR -- SEQUENTIAL
  ```

- The FORM and RFORM are both present in an OPEN statement. For this case, the second line to the error message is:

  ```
  CLAUSE(S) IN ERROR -- FORM & RFORM
  ```

- An open status of SCRATCH may not be present if the file clause is present. For this case, the second line to the error message is:

  ```
  CLAUSE(S) IN ERROR -- STATUS
  ```

- A zero length literal is used as an argument in an OPEN, CLOSE, or INQUIRE statement.

```
1031       OPEN STATUS OLD, FILE DOES NOT EXIST
```

An open status of OLD or EXTEND is specified in an OPEN statement; however, the file can’t be assigned to the run.

```
1032       ASG,CP ASSIGNMENT FAILED ON OPEN
```

An open status of NEW is specified in an OPEN statement; however, you didn’t preassign the file and one could not be assigned.

```
1033       FILE INCONSISTENT WITH DEFINE FILE/OPEN
```

The file format specified in an OPEN or DEFINE FILE of a file already open doesn’t match the file format of the opened file.

```
1034       SPECIFIED INPUT BEYOND END OF RECORD
```

The unformatted read requires more length from the record than is actually present. The record may or may not be segmented. All segments would be read if the record is segmented.

1035 `SCRATCH FILE NAME CONFLICT`

An open status of SCRATCH is specified in an OPEN statement. In attempting to assign a scratch file, it is determined that a file with a file name equal to the unit number already exists.

1036 `INFO$ REQUEST FAILED`

1037 `STMT OPTION INVALID FOR HVTS ENVIRONMENT`

One of the following occurs while processing an OPEN statement:

- The OFF clause is present.
- TYPE=ANSI, AREADA, APRNTA, APUNCH, or APNCHA
- The record size specified in a define file block is greater than the default size.

1038 `NOT ENOUGH TDFA SPACE OR FCT TOC FULL`

In an HVTS environment, there is no room for another file in the TDFA or the maximum number of files are currently in use (the file control table of contents is full).

*1039 `INQUIRE LITERAL TRUNCATED ON THE RIGHT`

The literal returned by the INQUIRE statement is longer than the receiving area and is truncated on the right.

1040 `INCORRECT NAMELIST SUBSCRIPT`

1041 `SIZE OF INTERNAL FILE EXCEEDED`

The formatted read of an internal file that is a character variable, character array element, or character substring tries to read beyond the end of that file (see 5.9.1).

1042 `INVALID STMT TYPE FIELD IN IO PACKET`

The I/O packet passed to the run-time system for an I/O statement is incorrect.

1043 `REREAD ILLEGAL FOR STMT TYPE OR SUBTYPE`

A reread unit can only be referenced by a formatted or list-directed READ statement.

1044 `ATTEMPTED TO REREAD BEYOND END OF BUFFER`

- The format used to reread the record tried to read beyond the reread record. The REREAD format can't contain slashes and can't be shorter than the input/output list.

- The list-directed reread tried to read more than one record.

1045 `OPENED UNIT MAY NOT BECOME A REREAD UNIT`

If a unit is open, it must be closed before attempting to open it as a reread unit.

1046 `INPUT BUFFER OVERWRITTEN`

The size of the record just read in exceeds the input buffer size. To create a larger input buffer use an OPEN or DEFINE FILE statement specifying the largest record size to be read.

1047 `ERROR ON CLOSE, UNIT NOT OPEN FOR REREAD`

Execution of a CLOSE statement with the REREAD clause present is attempted for a file open for regular input/output.

1048 `ERROR ON CLOSE, UNIT OPEN FOR REREAD`

Execution of a CLOSE statement without the REREAD clause present is attempted for a unit open for reread.

1049 `TARGET ADDR USED IN MORE THAN ONE CLAUSE`

A variable or array element that receives a value in an INQUIRE statement can't be referenced by more than one of the clauses in the same INQUIRE statement.

1050 `INVALID FILENAME FOR OPEN/INQUIRE`

Either an F-cycle, read/write keys, or an invalid character is encountered in a FILE clause of an OPEN or INQUIRE statement.

1051 `A SCRATCH FILE MAY NOT BE KEPT`

*1052 `RCD LARGER THAN MAX RCD SIZE TRUNCATED`

The size of the record just read via a symbiont read exceeds the maximum record size (default size if no OPEN or DEFINE FILE statement) for the file. The record is truncated to the maximum record size.

1053 `ATTEMPT TO READ A DUMMY RECORD`

A direct access read attempts to read a dummy record. The direct access record is not written to yet and does not contain useful information. Dummy records are only detected in skeletonized files.

1057 `ERROR ON TLBL$ REQUEST`

An error is encountered while doing an ER TLBL$. The substatus field of the I/O status word PTIOE is set to the ER TLBL$ error code. In addition, the ER TLBL$ error code will be provided by a second line to the error message. The second line has the format:

```
'ER TLBL$ ERROR CODE IS xxx'
```

1058 `EQUIPMENT TYPE NOT ALLOWED`

Because of a PCIOS restriction, I/O operations are prohibited for word-addressable mass storage with equipment codes 020-027.

1059 `INTERNAL FILE I/O IS INCORRECT`

An attempt is made to read an internal file using an empty FORMAT statement (FORMAT( )) when the input/output list is not empty.

*1060 `RECORD FORMAT ON OPEN IS INCONSISTENT`

At the opening of an existing direct-access file, the record format specified in the OPEN or DEFINE FILE doesn't agree with the record format with which the direct-access file was created.

1061 `ERROR MSG NOT DEFINED`

This error message number doesn't contain a valid error message at the current time.

1062 `RECURSIVE I/O CALLS NOT ALLOWED`

A function reference used in an input/output list attempts to perform I/O operations or refers to a subprogram that attempts to perform I/O operations. This is a fatal error and control does not return to the program.

1063 `DIRECT ACCESS BUFR TOO SMALL`

You must specify a larger buffer size for a direct access file in the OPEN statement. Refer to 5.10.1 for a discussion on the minimum buffer size.

1065 `ENCODE/DECODE RECORD CANNOT SPAN BANK`

The record being encoded or decoded is a Virtual item contained partially in one Virtual bank and partially in another. This is not allowed. See Appendix M for information on how Virtual items are stored in banks.

1066 `OPEN STATUS NOT ALLOWED WITH SHARED FILES`

An OPEN statement with STATUS='NEW' or STATUS='SCRATCH' attempted to create a shared direct access file. ASCII FORTRAN cannot create shared files.

```
1067        SEQUENTIAL FILE MAY NOT BE SHARED
```

The file is declared as SEQUENTIAL; only files declared as DIRECT ACCESS may be shared files.

```
1068        NTRAN UNIT MAY NOT BE USED FOR FTN I/O
```

Normal FORTRAN I/O statements (such as READ, WRITE, BACKSPACE, ENDFILE, OPEN, CLOSE, and INQUIRE) may not refer to a unit that is being used with the NTRAN$ subroutine (see 7.7.3.16).

```
1069        FILE IS READ OR WRITE INHIBITED
```

Read or write access to the file is not allowed. You may have omitted a read or write key, or access may be denied for some other reason.

```
1070        I/O PROCESSING NEEDS LARGER SCT
```

The library common bank uses a data storage area that is collected into the program. Your program was collected with an older library that contained a storage area that is too small for the current common bank. You must re-collect the program with a library that is at least as new as the level specified in the secondary message:

```
RE-MAP PROGRAM WITH AT LEAST FTN LEVEL level
```

```
1072        NAMELIST HOLLERITH INPUT SPANS BANK
```

An individual Hollerith value input during a NAMELIST READ cannot be transferred to a data area that spans from one Virtual bank to another (for example, the value 8Habcdefgh cannot be read into two consecutive elements of an integer array if the two elements happen to be allocated in two different Virtual banks).

G.10. FORTRAN Define File Block (DFB) Usage

The file description parameters given in the sequential OPEN or DEFINE FILE statements can also be specified in a Define File Block (DFB) external to the FORTRAN program. A DFB used by FORTRAN must exist in the file DFP$. Prior to execution of the FORTRAN program, a DFB is produced by the Define File Processor (DFP) as follows:

```
@DFP,E  fname.,DFP$.unit
```

where the E option indicates that DFB is to be produced. The variable *fname* specifies the external file name to which the DFB applies and is inserted in the DFB. DFP$.*unit* is the file name and element name into which the DFB is inserted. For FORTRAN, this must be the file DFP$ and the element name must be equivalent to a FORTRAN unit specifier. The variable *fname* must be followed by a period or the default file TPF$ is associated with the unit number.

When the FORTRAN input/output routines attempt to retrieve a DFB, the unit specifier from the sequential OPEN or DEFINE FILE statement is used as the element name of the DFB. You must assign the file DFP$ to the run prior to execution of the FORTRAN program.

If a DFB is found in DFP$, the input/output routines perform the following @USE statement on the file, providing it isn't a standard symbiont file:

```
@USE x,q*f
```

where *x* is the unit number of the file and *q*f* is the qualifier and file name from the DFB. This *q*f* is the same as *fname*, which is used by the DFP when the DFB is created. If the file is a buffered input/output file, the following assignment statement is also performed by the input/output routines:

```
@ASG x.
```

If the file type from the OPEN or DEFINE FILE statement matches the file type from the DFB, the parameters that exist in OPEN or DEFINE FILE statements and DFB override the file default parameters, respectively. If the file type from the OPEN or DEFINE FILE statement does not match the file type from the DFB, just the DFB parameters override the file default parameters.

The only DEFINE FILE statement or DFB parameter, other than *ft* (file type), that applies to symbiont files is *rs* (record size). Symbiont file unit members can be predefined in the file reference table (FRT). If a DEFINE FILE statement is executed for a predefined symbiont unit, the file type from the DEFINE FILE statement and the file type from the DFB (if a DFB exists) must match the file type in the FRT for the unit. If an OPEN or DEFINE FILE statement is executed for a symbiont unit that is not predefined in the FRT, the file type from the OPEN or DEFINE FILE statement, if not overridden by the DFB, is placed in the FRT and remains there for the duration of the run or until the unit is closed via the CLOSE statement and opened via the OPEN statement with a different file type.

The define file processor can't modify the parameters given in a direct access OPEN or DEFINE FILE statement; only sequential file parameters can be adjusted.

The following examples illustrate the DFP call and the keyword parameter lists applicable to the various FORTRAN file types.

- If *fname* is a sequential SDF file:

  ```
  @DFP,E fname.,DFP$.unit
  FILE = SDF
  RECORD = rs (record size)
  BLOCK = bs (block size)
  LINESIZE = ss (segment size)
  ```

  where:

  unit

  is the unit number.

rs

is a positive integer (optional).

bs

is an integer greater than 224 (optional).

ss

is a positive integer (optional).

- If *fname* is an ANSI file:

```
@DFP,E fname.,DFP$.unit
FILE = ANSI
RECORD = rs/rf (record size/record format)
BLOCK = bs/bo (block size/buffer offset)
```

where:

unit

is the unit number.

rs

is a positive integer (optional).

rf

is U, F, V, FB, VB, VS, or VBS (optional).

bs

is a positive integer (optional).

bo

is a positive integer less than *bs* (optional).

- If *fname* is a shared direct-access file:

```
@DFP,E fname., DFP$.unit
SHARE = a
```

where:

unit

is the unit number.

a

is the value NO or YES.

Note: *This clause overrides the S values of the RFORM clause in the OPEN statement, if both are present.*

- If *fname* is a print symbiont file:

```
@DFP,E  fname.,DFP$.unit
FILE = ft (file type)
RECORD = rs (record size)
```

where:

unit

is the unit number.

ft

is one of the following: APRINT, APRINT$, APRNTA, APRNTA$.

rs

is a positive integer less than or equal to 160.

Note: *The DFP requires the fname parameter, but fname is not used by the input/output routines for file assignment when the file type is for a standard symbiont file.*

Symbiont	Default Record Size (in Characters)	Allowable Range with DFP
051 APRINT	132	0 < *rs* ≤ 252
052 APUNCH	80	0 < *rs* ≤ 80
053 AREAD	80	0 < *rs* ≤ 131,071
054 APRNTA	132	0 < *rs* ≤ 252
055 APNCHA	80	0 < *rs* ≤ 80
056 AREADA	132	0 < *rs* ≤ 160

For example, consider the following program called TEST:

```
      COMMON I,DATA(90)
      DEFINE FILE 11(SDF,,10,224,112)
      CALL SUB
      DO 90 J = 1,I
90       WRITE(11,20) (DATA(K),K=1,J)
20       FORMAT (90(A1))
      END
```

Assume that something is going wrong with writing unit number 11. The following DFP-modified execution causes unit 11 output to go to the printer rather than to an SDF file:

```
@ASG,T   DFP$.
@DFP,E   DUM.,DFP$.11
 FILE = APRINT
 RECORD = 40
@XQT TEST
```

This use of DFP changes the maximum record size of all print units for this execution to 40 characters also.

G.11. Storage-Allocation Packet (Element MPKT)

A set of common storage allocation routines (which are used by the libraries of the ASCII FORTRAN, ASCII COBOL, and PL/I compilers) is used to control storage used by the I/O routines at run time (except in ASCII FORTRAN checkout mode). The packet used by the common routines contains values for the maximum address that the program can reach, the storage request increment size, and the storage free increment size.

The service routine MAXAD$ (see 7.7.3.19) can be called to change values in the common storage-allocation packet at run time.

Appendix H
Using Multibanking for Large Programs

H.1. Large Programs

The default address range for the instructions and data of a collected program, including all user subprograms and all referenced run-time library routines, is 65,535 decimal words. If the size of the collected program is larger than this 65K-word range, the collector produces truncation errors because it is trying to place an address that is greater than 65K into a 16-bit instruction u-field. Index registers can hold an 18-bit address. Therefore, ASCII FORTRAN generates code using index registers to hold addresses if the O option (the over-65K-address option) is used on the ASCII FORTRAN processor call. This raises the boundary to a 262K-word address range for your collected program before truncation problems again appear.

If the O option is used on the processor call and truncation errors do not occur during collection, your problem is solved. (Having the statement COMPILER(PROGRAM=BIG) in source programs is equivalent to using the O option when compiling them.)

However, if a program still gets truncation errors during collection, you must construct a multibanked program or place some large data items in virtual space. When the size problems are due to large user data objects, use either banked space or virtual space to solve the problem. Using virtual space is preferred since it is easier to set up and use (see Appendix M). When the size problem is due to a large amount of executable code rather than large data objects, you must construct a multibanked program to solve the problem. When the problem stems from both the size of the code and the size of the data objects, you must construct a multibanked program to solve the code size problem and use either banked or virtual space to solve the data size problem. (Multibanked programs can also use virtual space.)

H.2. Banking

Banking is a mechanism for sharing the address space between different pieces of code or data. For example, you can use the collector IBANK directive to direct the collector to construct a bank (an I-bank) holding subprogram X, and to construct another I-bank to hold subprogram Y. These two I-banks can be created with overlapping addresses. The same thing can be done with data. You can use the collector DBANK directive to place common block CB1 into one bank (a D-bank) and common block CB2 into another parallel D-bank using the same address space. These are called paged data banks. In this general manner, you create a banked program that needs less address space than the unbanked program. You can define almost any number of I-banks and D-banks; refer to the collector documentation for the exact limit (it is currently limited by the Exec to about 250). You must construct a collector symbolic (sometimes called a MAP symbolic) containing a sequence of collector directives that define the banking structure of your program.

For this mechanism to work, generated code must be able to move an address window from bank to bank. A Processor State Register (PSR) is a hardware register that defines two address windows for an executing program, an I-bank window and a D-bank window. An LIJ (load I-bank base and jump) instruction moves the I-bank window, and an LDJ (load D-bank base and jump) instruction moves the D-bank window from one bank to another.

ASCII FORTRAN is supported on two basic types of systems:

1. Older single-PSR systems, such as the 1106, 1108, 1100/10, and 1100/20 systems.
2. Newer dual-PSR systems and systems that are compatible with the dual-PSR structure. These include the 1110, 1100/40, 1100/60, 1100/70, 1100/80, 1100/90, and the 2200 systems.

On dual-PSR systems, the two PSRs are referred to as the main PSR (PSRM) and the utility PSR (PSRU). Systems that are compatible with dual-PSR systems use Base Descriptor Registers (BDRs) or Base Registers (B-registers). There is a one-for-one correspondence between BDRs or B-registers and the four windows defined by the two PSRs.

This appendix refers to only PSRs since there is a one-for-one equivalence to BDRs and B-registers.

When constructing a collector symbolic to create a multibanking structure for ASCII FORTRAN programs, you must follow certain conventions:

1. No address overlap should ever occur between I-banks and D-banks since results are unpredictable.
2. When multiple D-banks containing paged data are defined, they must start at the same address (FORTRAN’s mechanism to switch D-bank basing depends on this).
3. In any multibanked collection, one bank is defined as the control bank. The control bank is assumed to be always available and based since it contains any unbanked programs and data and also the unbanked portions of the run-time library routines. The control bank cannot be overlapped by any other bank.

H.2.1. General Banking Example (Dual-PSR System)

The following example consists of three separately compiled elements. MAIN1 is the main program and SUB1 and SUB2 are subroutines. The first statement in each sample routine is a directive to the compiler indicating that the final collected program is banked, and appropriate linkages (e.g., LIJ, LDJ, LBJ instructions) must be used to ensure that the correct banks are visible when necessary. The sizes of the code and data in the examples don't warrant the use of banking since these are simple examples for instruction only.

Example of a Main Program (MAIN1):

```
      COMPILER (BANKED=ALL)
      COMMON /cb1/ a,x /cb2/ b,y
      CHARACTER*1 a,b
      DATA a/'a'/, b/'b'/
      WRITE(6,100) 'reference to:', a
      WRITE(6,100) 'reference to:', b
      a = 'c'
      b = 'd'
      WRITE(6,100) 'reference to:', a
      WRITE(6,100) 'reference to:', b
100   FORMAT (1X, A13, 1X, A1)
      x = 2.
      y = sqrt(x)
      z = x + y
      WRITE(6,200) x, y
200   FORMAT (1X, 'sqrt of', F10.5, ' is', F10.5)
      CALL sub1 (a, b, x, y, z)
      END
```

Example 1 of a Subprogram (SUB1):

```
      COMPILER(BANKED=ALL)
      SUBROUTINE sub1 (a, b, x, y, z)
      CHARACTER*1 a, b
      WRITE(6,100) 'in subroutine sub1'
100   FORMAT(1X, A18)
      WRITE(6,200) a, b, x, y, z
200   FORMAT(1X, 'a=', A1, ' b = ', A1, ' x = ', F10.5, ' y = ',
      1F10.5, ' z = ', F10.5)
      CALL sub2(b, 2.0)
      END
```

Example 2 of a Subprogram (SUB2):

```
      COMPILER (BANKED=ALL)
      SUBROUTINE sub2(a, x)
      CHARACTER*(*) a
      CHARACTER*19 b3cell /'common block cb3'/
      COMMON /cb3/ b3cell
      CHARACTER*19 b4cell /'common block cb4'/
      COMMON /cb4/ b4cell
      PRINT *, 'a, len(a), x:', a, len(a), x
      PRINT *, 'common block cb3:', b3cell
      PRINT *, 'common block cb4:', b4cell
      END
```

H.2.1.1. Collection of the General Banking Example

The following collector symbolic can be used to collect the three sample program elements into a banked program. The LIB directive can be dropped if the ASCII FORTRAN library is in the system relocatable library, SYS$LIB$*FTN. This is the general form that you should follow for multibanking of ASCII FORTRAN programs on dual-PSR OS 1100 systems.

```
LIB SYS$LIB$*FTN.
IBANK,MRD    IBANKM          .  INITIALLY BASED, MAIN PSR
    IN MAIN1
IBANK,RD IBANK1,IBANKM
    IN SUB1
IBANK,RD IBANK2,IBANKM
    IN SUB2
DBANK,UD DBANK1,(040000,IBANKM,IBANK1,IBANK2)
    . INITIALLY BASED, UTILITY PSR
    IN F2ACTIV$($1)
    IN CB1
DBANK,D    DBANK2,DBANK1
    IN(MAIND) F2ACTIV$($3)
    IN CB2
DBANK,D    DBANK3,DBANK1
    IN(MAIND) F2ACTIV$($3)
    IN CB3
DBANK,CM MAIND,(DBANK1,DBANK2,DBANK3)
    . MAIND IS THE CONTROL BANK, ALWAYS BASED, ON MAIN PSR
    IN MAIN1
    IN SUB1
    IN SUB2
END
```

The collection of the sample program using this collector symbolic would result in the banking structure shown in Table H-1. The lines under the bank names are similar to the lines in a collector S-option listing in that they indicate length.

If the three FORTRAN relocatables are copied to file TPF$ and if the above collector symbolic is in element TPF$.BMAP, you can collect this program in a banked absolute and execute it with the following control images. (This assumes that there are no other relocatables in file TPF$.)

```
@MAP   BMAP,BABS
@XQT   BABS
```

The following control images do a default nonbanked collection of the program and give the same execution results.

```
@MAP,I MAP,ABS
IN MAIN1
LIB SYS$LIB$*FTN.
@XQT   ABS
```

Table H-1. Dual-PSR Banking Structure

I-Banks Based on Main I-Bank PSR (PSRM) (starts at 01000)	D-Banks Based on Utility D-Bank PSR (PSRU) (starts at 040000 or over)	Control D-Bank Based on Main D-Bank PSR (PSRM) (starts after largest of D-banks based on utility D-bank PSR and may reach 262K limit)
IBANKM MAIN1 ________________	DBANK1 CB1 ________	
IBANK1 SUB1 _____	DBANK2 CB2 ________________	Local data, library-MCORE$ area _________
IBANK2 SUB2 ______	DBANK3 CB3 ___________	
C2F$ I/O Common Bank _________		

Notes:

1. *Program code goes into I-banks.*
2. *Named common blocks go into D-banks (paged data banks).*
3. *Data local to subprograms, blank common, any programs or named common not placed in other banks and the run-time library routines, all go into the control bank.*
4. *The C2F$I/O common bank is not mentioned in the collection, though it is referenced at run time for all I/O activities.*
5. *The paged data banks must start at or after address 040000 to avoid address overlap with the hidden C2F$ I/O common bank.*
6. *The area after the control bank is open for the I/O complex to acquire buffer space. Executive Requests (ERs) to MCORE$ are made at run time to expand this area.*
7. *If any collected addresses go over 65K, use the O option or the COMPILER (PROGRAM=BIG) statement with all of the ASCII FORTRAN compilations.*

The execution of the banked or nonbanked absolute results in the following output:

```
reference to: a
reference to: b
reference to: c
reference to: d
sqrt of 2.00000 is 1.41421
in subroutine sub1
a= c b = d x = 2.00000 y = 1.41421 z = 3.41421
a, len(a), x:d         1 2.0000000
common block cb3:common block cb3
common block cb4:common block cb4
```

H.2.1.2. Analysis of the Collector Symbolic

This subsection describes the collector symbolic listed in H.2.1.

The main program MAIN1 is placed in an I-bank with the M option, which makes it initially based on the main PSR. (The main program must be in an initially based bank.)

The other two routines, SUB1 and SUB2, are placed into two other I-banks, each starting at the same address as the I-bank containing MAIN1. (They can also be put into the same I-bank as MAIN1 since they are so small.)

All I-banks have the R option on the IBANK directive to indicate they are read-only I-banks. As a read-only bank, there is less Executive swap file activity.

All D-banks except the control D-bank have the D option on the DBANK directive to indicate that they are dynamic banks. This means that they can be swapped out by the Executive if they are not currently based, saving on main storage usage (though possibly causing more Executive swap file activity).

The bank names given on the IBANK and DBANK directives (for example, IBANKM) are called Bank Descriptor Indexes, or BDIs. The collector gives them integer values that are used by the LIJ and LDJ bank-switching instructions.

The paged data banks contain named common blocks and must be based on the utility D-bank PSR. The paged data bank DBANK1 is chosen to be the one initially based. The U option on the DBANK directive for DBANK1 indicates it is initially based on the utility PSR. The other paged data banks holding named common are put at the same address as DBANK1.

The location counter one code ($1 code) of the run-time activate element F2ACTIV$ is put at the beginning of initially based paged data bank DBANK1. The same is done with $3 code of F2ACTIV$ for each of the other paged data banks. The F2ACTIV$ location counter ($1) contains information to make the bank in which it resides self-identifying. This location counter is often referred to as the bank's ID area. The code under location counter ($3) in F2ACTIV$ is an exact copy of location counter one. To work properly, location counters one and three must be collected at the same address in the banks. The IN directive of F2ACTIV$ ($3) has MAIND in parentheses. This is called local element inclusion and bypasses possible LOCAL-GLOBAL CONFLICT messages from the collector.

The control bank MAIND is the D-bank named in the DBANK directive with the C option, and the M option on it means it is also initially based on the main D-bank PSR. Only the main program MAIN1 is included through use of an IN statement in this bank, since the collector puts anything not specifically included in another bank in the control bank. The control bank MAIND is placed after the largest of the three paged data banks so that no address overlap occurs.

The area after the end of the control bank is used by the storage management complex to obtain buffer space for the ASCII FORTRAN run-time system. Executive Requests (ERs) to MCORE$ are made to acquire this space.

The three paged data banks must not be defined at less than an 040000 (octal) address and the paged data banks must start at an address higher than the highest I-bank address. This is because their address space would then overlap the C2F$ I/O common bank, and the results would be unpredictable.

The named common block CB4 is not given a home in any paged data bank. If the main program and any subprograms are explicitly included in an I-bank and a D-bank by an IN statement, CB4 falls in the control bank and, since the control bank is always based, it is not dynamically banked. (See the section on element placement in the *OS 1100 Collector Programming Reference Manual, Level 33R1*, 7830 9887.) This does not result in any problems; in fact, any subprograms not specifically included in a bank by an IN statement fall harmlessly in the control bank. As long as the control bank does not get so large as to cause collector truncation errors again, this is harmless. Any number of subprograms can be included in an I-bank, and any number of named common blocks can be included in a paged data bank. (Blank common and data local to subprograms must be in the control bank.) The criteria for the contents of a bank should be a function of the final collected size of the bank, and also a function of the locality of reference to the bank to try to minimize thrashing between banks. (You can minimize any problem caused by excessive Executive swap file activity by carefully making selected banks static by not putting the D option on their I-bank or D-bank statements.) A reasonable size for an I-bank or paged data bank is approximately 16,000 decimal words. This means that the control bank can be as large as about 32,000 words before truncation problems occur again.

H.2.1.3. Large Paged Data Banks

If reasonable I-bank and paged data bank sizes still result in truncation errors at collection time, or if large paged data banks (greater than approximately 30,000 words) are to be defined, the O option is needed on the ASCII FORTRAN processor calls to allow a 262K address range. In addition, the ordering of the paged data banks and the control bank must be inverted to prevent collector truncation errors on the run-time library routines in the control bank. This means that the control bank must be placed after the I-banks and before the paged data banks in the address space. Since the ASCII FORTRAN run-time system makes the control bank larger via ER MCORE$ to obtain buffer space, you must leave enough room between the control bank and the paged data banks for I/O main storage requirements. (Appendix G contains formulas for estimating I/O main storage requirements for a program.) Allowing 10,000 decimal words is usually sufficient. However, if an ER MCORE$ results in an address overlap of the control bank and paged data banks, incorrect results or error termination may occur. The F2FCA library element can also be reassembled with a sufficiently large local area in it (see G.7).

To collect your program with large D-banks, the collector symbolic (from H.2.1.1) must be changed. The DBANK directive for D-bank MAIND and the IN directive on MAIN1 are moved back to just before the DBANK1 definition. Then these statements are changed as follows:

```
   .
   .
   .
DBANK,CM MAIND, (040000, IBANKM, IBANK1, IBANK2)
   IN MAIN1
DBANK,UD DBANK1,(MAIND+10000)
   .
   .
   .
```

The rest of the collector symbolic remains unchanged.

The resulting collector symbolic is:

```
LIB SYS$LIB$*FTN.
IBANK,MR IBANKM . INITIALLY BASED, MAIN PSR
    IN MAIN1
IBANK,R IBANK1,IBANKM
    IN SUB1
IBANK,R IBANK2,IBANKM
    IN SUB2
DBANK,CM MAIND,(040000,IBANKM,IBANK1,IBANK2)
    .MAIND IS THE CONTROL BANK, ALWAYS BASED, ON MAIN PSR
    IN MAIN1
    IN SUB1
    IN SUB2
DBANK,UD DBANK1,(MAIND+10000) . INITIALLY BASED, UTILITY PSR
    IN F2ACTIV$($1)
    IN CB1
DBANK,D DBANK2,DBANK1
    IN(MAIND) F2ACTIV$($3)
    IN CB2
DBANK,D DBANK3,DBANK1
    IN(MAIND) F2ACTIV$($3)
    IN CB3
END
```

The collection then results in the banking structure of Table H-2. The lines under the bank names are similar to the lines in a collector S-option listing in that they indicate length.

When the common blocks don't fit in the paged D-banks (truncation errors appear), put them in virtual space (see Appendix M).

H.2.1.4. Variations on the Dual-PSR Structure

The generalized example shows multiple I-banks and multiple D-banks being used at the same time. If your program is large only in the amount of I-bank code, you can simply omit the definition of the paged data banks and define only I-banks and the control bank. If the program has large common blocks causing the size problem, then the Collector symbolic element can be cut back to defining only one I-bank, the control bank, and multiple-paged data banks.

The LIB directive tells the collector where to obtain the ASCII FORTRAN run-time library. The simple form of the LIB directive causes all run-time library routines, both code and data, to fall into the control bank MAIND since they are not explicitly included in any bank. The following form of the LIB directive directs the Collector to put anything taken from SYS$LIB$*FTN into I-bank IBANKM and D-bank MAIND in a normal $ODD/$EVEN I-bank/D-bank split:

```
LIB SYS$LIB$*FTN.(IBANKM/$ODD,MAIND/$EVEN)
```

This can minimize the size of the MAIND control bank.

Table H-2. Dual-PSR Banking Structure, Over 65K

I-Banks Based on Main I-Bank PSR (PSRM) (starts at 01000)	Control Bank Based on Main D-Bank PSR (PSRM) (starts at 040000)	D-Banks Based on Utility D-Bank PSR (PSRU) (starts after the control bank and goes up to a 262K limit)
IBANKM MAIN1 ________	MAIND Local data, Library-MCORE$ area ________	DBANK1 CB1 ________
IBANK1 SUB1 ________		DBANK2 CB2 ________
IBANK2 SUB2 ________		DBANK3 CB3 ________
C2F$ I/O common bank ________		

Notes:

1. *The paged data banks can extend out to the 262K address limit.*
2. *The 10K area between MAIND and the paged data banks is used by the I/O complex for buffers. If it is not sufficient, the separation must be increased or the library element F2FCA reassembled with a nonzero reserve.*
3. *I/O acquires storage in increments of eight storage blocks (4096 decimal words).*

H.2.2. Banking for Single-PSR 1100 Systems

The collector symbolics described in H.2.1 through H.2.1.4 use the utility PSR of dual-PSR systems to hold an address window for paged data. This utility PSR does not exist on single-PSR systems such as the 1106, 1108, 1100/10, and 1100/20. If your program needs only multiple I-banks and not multiple D-banks, the previously described collector symbolics can be used with the definitions of the paged data banks removed. If your program needs both multiple I-banks and multiple D-banks, it cannot be done on single-PSR hardware. However, you can define a banking structure for multiple D-banks for single-PSR systems. A single I-bank is defined, and it is also made the control bank to hold all unbanked code and data. The paged data banks are defined to come after the control bank, but enough room must be left between them for I/O buffers (which are dynamically acquired by ER MCORE$ at run time).

However, this type of collection has a problem resulting from the I-bank holding all unpaged data. Because of this, no common banks can be referred to at run time to do I/O, calls to the Common Mathematical Library (CML), etc. The run-time library used must have all run-time routines in relocatable form. (The ASCII FORTRAN library must be built as a type1 library, with the relocatable form of the PCIOS common I/O modules, and also the relocatable form of the CML modules.)

Another problem results from ASCII FORTRAN putting a SETMIN directive on all relocatable elements it generates, which ensures that code referencing array elements is correct. The collector emits a warning on each FORTRAN element. For example:

```
MAIN1 MINIMUM ADDRESS IGNORED-LCO NOT IN DBANK
```

These collector warnings can be ignored, but the I-bank must be started at address 040000 or higher to ensure that the FORTRAN-generated code works correctly.

You must also edit and reassemble the ASCII FORTRAN library element F2BDREQU$ when performing a multibanked collection for single-PSR systems. Adjust the values of three EQUs as follows:

Tag	Dual-PSR (default)	Single-PSR
CBDR$	2	0
VBDR$	1	1
BKBDR$	3	2

The example in H.2.1 using MAIN1, SUB1, and SUB2 can be collected using multiple D-banks for single-PSR systems with the following collector symbolic:

```
LIB FTN*RLIBX.   . SPECIAL (TYPE1) LIBRARY !
IBANK,MC     IBANKM,040000 . CONTROL BANK NOW
    IN MAIN1,SUB1,SUB2
DBANK,MD     DBANK1,(IBANKM+10000)
    IN F2ACTIV$($1)
    IN CB1
```

```
DBANK,D     DBANK2,DBANK1
    IN(IBANKM) F2ACTIV$($3)
    IN CB2
DBANK,D     DBANK3,DBANK1
    IN(IBANKM) F2ACTIV$($3)
    IN CB3
END
```

Collection results in the following collector diagnostics:

```
MAIN1   MINIMUM ADDRESS IGNORED - LC 0 NOT IN DBANK
SUB1    MINIMUM ADDRESS IGNORED - LC 0 NOT IN DBANK
SUB2    MINIMUM ADDRESS IGNORED - LC 0 NOT IN DBANK
CB4     MINIMUM ADDRESS IGNORED - LC 0 NOT IN DBANK
```

The collection results in the banking structure given in Table H-3 for the single-PSR multiple D-bank problem. The lines under the bank names in Table H-3 are similar to the lines in a collector S-option listing in that they indicate length.

Table H-3. Single-PSR Banking Structure

One I-Bank (the Control Bank) Based on the Main I-Bank PSR (PSRM) (starts at 040000)	D-Banks Based on the Main D-Bank PSR (PSRM) (starts after the control bank and may go up to 262K)
IBANKM Code, library, unbanked data-MCORE$ ────────────────────────────	DBANK1 CB1 ─────────
	DBANK2 CB2 ───────────────────────
	DBANK3 CB3 ───────────────────

Notes:

1. *There are unavoidable collector diagnostics.*
2. *A totally local library must be used; no run-time common banks can be referenced.*
3. *The 10K separation between IBANKM and the D-banks must be able to satisfy all buffer requests from the ASCII FORTRAN run-time system, or it must be increased, or the library element F2FCA must be reassembled with a nonzero reserve large enough to satisfy the buffer requests.*
4. *The I-bank must start at or after address 040000.*

5. *If the paged data banks extend beyond 65K, use the O option on all of the ASCII FORTRAN compilations.*
6. *IBANKM cannot extend beyond 65K in addressing.*

H.2.3. Banking, Efficiency, and Source Program Directives

The examples in H.2.1 through H.2.2 use a simple generalized directive to the ASCII FORTRAN compiler to indicate that banking is used in the final absolute program. In fact, this generalized statement, COMPILER(BANKED=ALL), means:

- Each subprogram referenced can be in a different I-bank, in the same I-bank, or in the control bank.
- Input arguments can be in paged data banks or in the control bank.
- Named common blocks can be in paged data banks or in the control bank.

Therefore, the actual banking structure is virtually unknown to the compiler, and yet it must create linkages to ensure that items are visible or based when they are referenced.

H.2.3.1. I-Bank Linkages

The ASCII FORTRAN compiler uses the LIJ instruction to link between I-banks. This linkage is fairly efficient since the bank switch is done and then the called subprogram is entered, usually for some period of time. The compiler generates a pseudo-linkage called the IBJ$ linkage. The collector replaces this linkage with an LMJ instruction if the destination is in the control bank, or the same I-bank, and with an LIJ instruction if the destination is in a different I-bank.

H.2.3.2. D-Bank Linkages

Each time the compiler generates code to reference data in a (possibly) paged data bank (which may be currently based), it must also generate an activate sequence. This activate code has several variations, but a typical sequence is two to four instructions long.

Example:

```
  Variables A, B, and C are in named common blocks CB1, CB2, and CB3.
```

The FORTRAN statement A = B+C is to be compiled.

If you haven't given any directives to the ASCII FORTRAN compiler indicating that multiple D-banks are being used, three machine instructions are generated for this statement: a LOAD, an ADD, and a STORE. If you indicate to the ASCII FORTRAN compiler (through BANK statements) that multiple D-banks are being used, there are also three activate sequences generated. This results in nine instructions instead of three for the unbanked program. In addition, each activate sequence contains LBJ instructions. The LBJ instruction is simulated in the Executive for older single-PSR systems and may cause a presence-bit interrupt if the bank is swapped out on any OS

1100 system. Program efficiency is extremely dependent on the contents of the paged data banks and on the organization of the ASCII FORTRAN code.

Because performance can be so dramatically affected, you can supply several directives, including the BANK statement (see 6.6) and several COMPILER statement options (see 8.5), to the ASCII FORTRAN compiler to help program efficiency.

H.2.3.3. Multiple I-Banks Only

If your collected program is constructed using multiple I-banks for code and does not define multiple paged data banks, use the LINK = IBJ$ option of the COMPILER statement rather than the more general BANKED=ALL option. The compiler then generates the efficient IBJ$ linkage for subprogram references and generates code assuming that data is not banked. The resulting program should be as efficient as an unbanked program.

H.2.3.4. Multiple Paged Data Banks

Once your multiple D-bank program is debugged and running, you might notice many activate code sequences in the generated code that are not necessary, since a given paged program bank may contain several named common blocks. Also, if you are often hopping between paged data banks in the generated code, you may wish for a much faster activate sequence.

H.2.3.4.1. The BANK Statement

The BANK statement associates a paged data bank BDI name with one or more named common blocks. Therefore, you can tell the compiler that common blocks CB1, CB2, and CB3 are in the same paged D-bank; therefore the compiler does not generate activate sequences when the program references them. Since the BANK statement tells the compiler that these items are definitely banked and the BDI is supplied, a more efficient bank switch can be done. The compiler generates a direct LBJ instruction to change the currently based D-bank rather than calling an F2ACTIV$ run-time routine.

Using a BANK statement to associate a BDI with a common block results in more efficient code to reference that common block. However, if the COMPILER (BANKED=ALL) statement is left in the program, the compiler still generates inefficient code sequences. These code sequences reference those named common blocks not specified in BANK statements. When you use BANK statements to associate BDIs with common block names for all common blocks residing in paged D-banks, you can replace the BANKED = ALL option of the COMPILER statement with the following three options:

```
(BANKED=ACTARG), (BANKED=DUMARG), (LINK=IBJ$)
```

In addition to more efficient code to reference nonbanked common blocks, the options ensure that:

- banked arguments are handled properly, and

- linkages to subprograms are correct.

Since BANK statements associate specific BDI names with common blocks, you must also change all the programs and recompile them if you change the banking structure.

The BANK statement and various COMPILER statement directives are described in 6.6 and 8.5.

H.2.3.4.2. Optimization and Program Organization

The ASCII FORTRAN compiler remembers which bank is currently based and does not generate unnecessary activate sequences if global optimization is used during program compilation. (Global optimization is called with the Z option on the ASCII FORTRAN processor call. See 9.6.2.) You should attempt to organize your code so that references to a given D-bank are grouped in areas of code. This is especially true for the inner loop of DO-loops. Try to have any inner loops refer to items in one paged data bank. (Unbanked data items can be referred to in any manner.)

Example:

```
      SUBROUTINE     SUBX (A,IA)
      COMPILER(BANKED=ALL)
      DIMENSION     A(IA)
      COMMON/C1/A1(1000),B1(1000)
      COMMON/C2/A2(1000),B2(1000)
      BANK/BNK1/C1,C2
      COMMON WORK(1000) @BLANK COMMON
      IL = IA-1
      DO 10 I=1,IL
   10     WORK(I)=A(I)/A(I+1)+.03
      DO 20 I=1,IL
          A1(I)=WORK(I)*B2(I)/B1(I)-A2(I)
          IF(A1(I).NE.0.0) A2(I)=1/A1(I)
   20     CONTINUE
      END
```

You have (possibly) banked arguments A and IA and two named common blocks that are known to be in D-bank BNK1.

Blank common can never be banked, so the program does some initial processing on the input array A and moves it to WORK in blank common. (A local array can also be used.) Since only one bank is referenced inside the first loop, the activate code sequence is moved out of the loop (if global optimization is used). The same thing is true for the second loop, and no activate sequences are done inside the loop. A single reference to an external routine inside either loop causes at least one set of activate code to be generated inside the loop since the external routine can possibly change which paged data bank is currently based.

If you had not supplied the BANK statement, the generated code would contain many unnecessary activate sequences since the compiler must assume the worst case.

H.2.4. Banking Summary

- Programs constructed using multiple I-banks and no multiple D-banks should use the COMPILER(LINK=IBJ$) statement to indicate banking to the ASCII FORTRAN compiler.
- The COMPILER(BANKED=ALL) statement stresses ease of use for multiple D-bank programs. However, CPU efficiency suffers when compared to the use of the BANK statement for common block names.
- Programs with multiple D-banks can reduce the number of activate code sequences generated, and can cause the direct generation of LBJ instructions by the selected use of BANK statements to associate named common blocks with specific paged data bank BDIs. Replace the COMPILER (BANKED=ALL) statement with COMPILER (BANKED=ACTARG), (BANKED=DUMARG), (LINK=IBJ$) to enhance efficiency.
- If BANK statements are used in a FORTRAN program to enhance efficiency, no error diagnostics occur if they are incorrect. Bad program results can occur.
- Use global optimization to dramatically reduce the number of generated activate sequences.
- Performance can be enhanced by judicious organization of program logic and careful definition of the contents of paged data banks.
- An address overlap must never occur between any I-bank and any D-bank, or between the control D-bank and any paged data banks. If the run-time library used is a normal common bank, the C2F$ I/O common bank can extend to address 037777 (octal). Therefore, no D-bank should start below address 040000.
- The control bank holds all of your unbanked data and routines, all run-time library D-bank, and any unbanked library routines. The control bank must be initially based and must never be unbased by an LIJ, LDJ, or LBJ instruction.
- If any addresses go beyond 65K in the collection, the O option or the statement COMPILER (PROGRAM=BIG) is needed on all ASCII FORTRAN compilations.
- Any element containing only BLOCK DATA subprograms must be included using an IN directive in the control bank in the collection, since the collector may otherwise ignore it. This also is true for nonbanked collections.
- Minimize control bank size by supplying a LIB statement to the collector that causes the code or I-bank portions of the run-time library to go into one of your I-banks. For example, LIB SYS$LIB$*FTN.(IBANKM/$ODD,MAIND/$EVEN).
- Multiple D-bank operation depends on copies of the activate code existing in each paged data bank at exactly the same relative address. The $1 and $3 F2ACTIV$ code segments are identical, and the only reason for the location counter split is to avoid local-global conflict messages during collection. The easiest way to ensure the same address for ACTIV$ code is to make each paged data bank start at the same address and to have the ACTIV$ code first in each D-bank.
- The local element inclusion of F2ACTIV$ code is also important. The $3 code (an exact copy of $1 code) has no tags, but refers to data in the control bank; therefore, it must be visible only to the control bank.

- Single-PSR 1100 systems can't have both multiple I-banks and multiple D-banks in the same collection.
- The TYPE BLOCKSIZE64 collector directive can save storage when the absolute element resulting from the collection has many banks.
- The single-PSR multiple D-bank setup needs a special library that contains a totally relocatable form of all run-time routines so that no common banks are referenced. The run-time element F2BDREQU$ must also be reassembled with some EQU values changed (see H.2.2).

Appendix I
Error Diagnostics in Checkout Mode

The following messages are associated with the checkout mode of the ASCII FORTRAN compiler (see 10.3):

Messages Occurring During Program Load

```
****CHECKOUT RELOCATION ERRORS****
```

If any errors are encountered while loading your program, this message is issued and the error messages are then printed.

`WARNING: NAME IS UNDEFINED:` *`name`*

You referenced a subprogram that is not defined in your compilation unit or the FTN library. The offending name is printed on the line following the message.

```
ERROR: USER PROGRAM TOO LARGE
```

An address generated while loading your program doesn't fit in an address field. Your program is too large. It may fit if the O option is used or if the Z option is omitted on the processor call image.

```
NO MAIN ENTRY POINT, NO EXECUTION POSSIBLE
```

This message is produced if your program does not contain a main program. Instead the program contains only subroutines, functions, and BLOCK DATA subprograms. Interactive debug mode is entered.

Messages Generated by Interactive Debugging

`BAD LINE NUMBER` *`n`*

The line number, *n* (specified in the BREAK command), is either out of range for your program or is on a nonexecutable statement.

```
BLOCK DATA PROGRAM NOT FOUND
```

The BLOCK DATA program specified in the *p* field of the PROG command or the *p* subfield of a command doesn't exist in the FORTRAN symbolic element.

```
COMMAND NOT ALLOWED
```

The GO command (no fields) or the WALKBACK command can't be executed because normal execution of the FORTRAN program is not possible; that is, there is no main program in the element, or the CALL command has executed a subprogram and returned.

```
COMMAND NOT ALLOWED BECAUSE OF CONTINGENCY
```

The GO command (no fields) can't be executed because a contingency is captured by the compiler. For example, if the FORTRAN program encounters a guard mode (IGDM) contingency, then normal execution of the program can't resume. A RESTORE command can bring back an original version of the program.

CONSTANT MUST BE TYPE *data-type*

The constant in the third field of the SET command isn't the same data type as the variable in the first field. The SET command doesn't perform conversions between data types.

```
ELEMENT HAS NO MAIN PROGRAM
```

The main program is specified in the *p* field of the PROG command or in the *p* subfield of a command, but the FORTRAN symbolic element doesn't contain a main program.

```
ENTIRE ASSUMED-SIZE ARRAY CANNOT BE DUMPED
```

The range of an assumed-size array is unknown. Only individual elements of an assumed-size array can be dumped.

```
ENTRY POINT NOT FOUND
```

The entry point specified in the *s* field of the CALL command or in the parameter list of the CALL command doesn't exist in the FORTRAN source.

```
ERROR: NO USER PROGRAM FOUND
```

You are using a RESTORE command but haven't previously done a SAVE on the desired version.

```
FTEMP$ STORAGE DESTROYED
```

A subprogram's temporary storage area (for saving registers and the parameter list) is destroyed because of an error in your program. The specified variable can't be dumped.

```
FUNCTION HAS NOT BEEN CALLED
```

A reference is made to a variable that is a character function entry point, but the function has not yet been called during execution of the FORTRAN program.

```
ILLEGAL COMMAND
```

An illegal debug command name is specified when input is solicited with ER ATREAD$, or the name specified in the *cmd* field of the HELP command isn't a debug command name.

```
ILLEGAL SYMBOLIC NAME
```

An illegal FORTRAN variable name is specified in the *v* subfield of a command, or an illegal subroutine or function name is specified in the *p* field of the PROG command or the *p* subfield of a command.

```
ILLEGAL SYNTAX
```

A general syntax error is found. This includes specifying a field for a command when none is required, or not specifying a field when one is required.

```
INCORRECT NUMBER OF SUBSCRIPTS FOR ARRAY *
```

The number of subscripts specified for the array in the *v* subfield of a command does not equal the number of dimensions declared for the array in the specified FORTRAN program unit.

```
IO ERROR ON LOADING USER PROGRAM, LOAD ABORTED
```

An I/O error occurs while accessing your program file during execution of the RESTORE command. The command is aborted. This may result in error termination also.

```
IO ERROR ON USER OUTPUT FILE, 'SAVE' COMMAND ABORTED
```

An I/O error occurs while accessing your program file during execution of the SAVE command. The command is aborted.

```
LABEL BREAK LIST IS FULL
```

An attempt is made to add an entry to the statement label break list with the command BREAK *n* L [/*p*], but the list already has eight entries.

```
LABEL UNDEFINED *
```

The statement label *n* in the BREAK *n*L [/*p*] command is not declared in the specified FORTRAN program unit.

```
LINE NUMBER BREAK LIST IS FULL
```

An attempt is made to add an entry to the line number break list with the command BREAK *n*, but the list already has eight entries.

```
NO BREAK SET FOR LABEL *
```

The statement label *n* (in the specified program unit) in the command CLEAR *n* L [/*p*] is not in the statement label break list.

```
NO BREAK SET FOR LINE NUMBER
```

The line number *n* in the command CLEAR *n* isn’t in the line number break list.

```
PARAMETER'S SUBPROGRAM HAS NOT BEEN CALLED
```

An attempt is made to reference a subroutine’s or function’s argument, but the subprogram has not yet been called during execution of the FORTRAN program.

```
PROGRAM UNIT NOT FOUND
```

The symbolic name specified in the *p* field of the PROG command or the *p* subfield of a command doesn’t exist in the FORTRAN symbolic element as a named program unit.

```
SETBP NOT ALLOWED ON COMPILER GENERATED FOR SINGLE-PSR MACHINE
```

The SETBP command can only be executed on an ASCII FORTRAN compiler generated for a dual-PSR machine. The nonreentrant ASCII FORTRAN absolute taken off the test file (file 2) of the ASCII FORTRAN release tape is a compiler generated for single-PSR machines.

```
SUBSCRIPT OUT OF RANGE FOR ARRAY
```

The constant subscripts specified for the array in subfield *v* of a command are too big or too small.

```
****UNDEFINED SUBROUTINE ENTRY****
```

During execution, your program calls a function or subroutine that is undefined. The name of the subprogram was previously printed out with the checkout relocation errors.

```
USER FILE REJECTED, NOT FASTRAND FORMATTED
```

You are attempting a SAVE or RESTORE command, but the file is not a program file. Something has happened, making it unusable. The file affected is the Relocatable Output (RO) file specified on the @FTN processor call command.

```
USER INPUT FILE CANNOT BE ASSIGNED
```

Your program file can’t be assigned to do the RESTORE command. Some other run must be using it.

```
USER OUTPUT FILE CANNOT BE ASSIGNED
```

Your program file can't be assigned to do the SAVE command. Some other run must be using it.

```
VARIABLE IS AN ARRAY *
```

The variable in subfield *v* of a command has no subscripts, but the variable is declared as an array in the specified program unit. An array element is required.

```
VARIABLE IS NOT AN ARRAY *
```

The variable in subfield *v* of a command has subscripts, but the variable is not declared as an array in the specified program unit.

```
VARIABLE NOT DEFINED *
```

The variable in subfield *v* of a command is not declared in the specified FORTRAN program unit.

```
WARNING: CHARACTER CONSTANT TRUNCATED
```

The character constant in the third field of the SET command has too many characters to fit in the character variable. It is truncated to the declared length of the variable.

An asterisk (*) indicates that the error message is followed by a second printed line. This line specifies the program unit (in the FORTRAN element) from which the variable or statement label (in the command image) comes. One of the following formats will appear:

```
IN MAIN PROGRAM
IN MAIN PROG n

IN BLOCK DATA n
IN BLOCK DATA PROGRAM m

IN SUBROUTINE n [:e ]

IN FUNCTION n [:e ]
```

where:

n

is a program unit name.

e

is an external program unit name.

m

is an unnamed BLOCK DATA subprogram sequence number.

The specified program unit is taken from the *p* subfield of the command, or from the PROG command default program unit, if *p* is not specified in the command.

Appendix J
Differences between ASCII FORTRAN Level 8R1 and Higher Levels

J.1. General

ASCII FORTRAN levels 9R1 and higher contain all the features of the ANSI FORTRAN standard, X3.9-1978 (called FORTRAN 77). ASCII FORTRAN level 8R1 doesn't have all these features. This appendix compares ASCII FORTRAN levels 9R1 and higher to level 8R1.

ASCII FORTRAN level 8R1 is missing the following six statements: PROGRAM (see 7.2.1), INTRINSIC (see 6.10), SAVE (see 6.11), OPEN (see 5.10.1), CLOSE (see 5.10.2), and INQUIRE (see 5.10.3). It is incompatible with ASCII FORTRAN levels 9R1 and higher in the storage allocation of character data, DO-loops, typing of symbolic names of constants and statement functions, and list-directed input/output. It doesn't contain the intrinsic functions: ICHAR, CHAR, LEN, INDEX, ANINT, DNINT, NINT, IDNINT, DPROD, LGE, LGT, LLE, LLT, UPPERC, LOWERC, and TRMLEN (see 7.7.1), or the logical operators, .EQV. and .NEQV.

An option, STD=66, has been added to the COMPILER statement (see 8.5 and 8.5.6) for levels 9R1 and higher. This option forces the ASCII FORTRAN compiler and library routines for levels 9R1 and higher to execute as previous levels of ASCII FORTRAN do in the areas of storage of character data, DO-loops, typing of statement functions and symbolic names of constants, and list-directed input and output.

This appendix is organized into subsections corresponding to the sections of the ANSI standard document, X3.9-1978. Each subsection of this appendix contains extensions in levels 9R1 and higher over ASCII FORTRAN level 8R1, and conflicts between ASCII FORTRAN levels 9R1 and higher and level 8R1. ASCII FORTRAN extensions to level 8R1 in levels 9R1 and higher are features that have been implemented to make ASCII FORTRAN conform to the standard completely. Conflicts occur where the same construct can have different meanings in the two levels. Thus, conflicts imply that a change has been made to a feature in ASCII FORTRAN level 8R1 to achieve compatibility with the standard.

J.2. FORTRAN Terms and Concepts

Extensions:

A main program can have a PROGRAM statement as its first statement. Level 8R1 has no PROGRAM statement (see 7.2.1).

Conflicts:

Character storage units for a datum are logically consecutive. Level 8R1 starts each character datum on a word boundary. The COMPILER statement provides an option, STD=66, to allow compatibility with previous levels on character data (see 8.5 and 8.5.6).

J.3. Characters, Lines, and Execution Sequence

Extensions:

PARAMETER statements can occur before and among IMPLICIT statements. Any specification statement that designates the type of a symbolic name of a constant must precede the PARAMETER statement that defines that particular symbolic name; the PARAMETER statement must precede all other statements containing the symbolic names of constants that are defined in the PARAMETER statement. Level 8R1 does not allow typing of symbolic names of constants (see 6.3 and 6.8). The COMPILER statement provides an option, STD=66, to allow compatibility with previous untyped symbolic names of constants.

Conflicts:

None.

J.4. Data Types and Constants

Extensions:

A complex constant can be written as a pair of integer constants or real constants. Level 8R1 allows only real constants (see 2.3.3).

Conflicts:

None.

J.5. Arrays and Substrings

Extensions:

You can use array names in a SAVE statement. Level 8R1 does not have a SAVE statement (see 6.11).

Conflicts:

None.

J.6. Expressions

Extensions:

- Complex operands are allowed in relational expressions with the .EQ. and .NE. operators unless one operand is double-precision. The comparison of a double-precision value and a complex value is not permitted. Level 8R1 does not allow complex operands in relational expressions (see 2.5.3).
- The logical operators .NEQV. and .EQV. with lowest precedence are allowed. These operators are not in level 8R1 (see 2.5.4.1).

Conflicts:

None.

J.7. Executable and Nonexecutable Statement Classification

Extensions:

None.

Conflicts:

None.

J.8. Specification Statements

Extensions:

- EQUIVALENCE statements can contain character substring names. Level 8R1 doesn't allow character substring names in EQUIVALENCE statement lists (see 6.4).
- Integer constant expressions are allowed for subscript and substring expressions in EQUIVALENCE statements. Level 8R1 does not allow substring expressions in EQUIVALENCE statements (see 6.4).
- In the COMMON statement, an optional comma is allowed before the slash that comes before the common block name, that is, [[,] / [*cb*] / *nlist*] . . . . No comma is allowed for level 8R1; an error occurs (see 6.5).
- A symbolic name of a constant can be typed in an IMPLICIT statement or in an explicit type statement (see 6.3 and 6.8). Level 8R1 does not allow typing of symbolic names of constants. The COMPILER statement provides an option, STD=66, to allow compatibility between levels 9R1 and higher and level 8R1 of ASCII FORTRAN for typing of symbolic names of constants (see 8.5 and 8.5.6).

- The name of a statement function can appear in an explicit type statement. The name of a statement function can be typed by an IMPLICIT statement. Level 8R1 does not type statement functions (see 6.3 and 7.3.1). The COMPILER statement provides an option, STD=66, to allow compatibility between levels 9R1 and higher and level 8R1 of ASCII FORTRAN for typing of statement functions (see 8.5 and 8.5.6).
- The length in a CHARACTER type statement can be an asterisk or an integer constant expression in parentheses such as (*) or (*exp*), or a constant with no parentheses, such as *const*. An entity in a CHARACTER statement must have an integer constant expression as a length specification unless that entity is an external function, a dummy argument of an external procedure, or a character constant that has a symbolic name. These exceptions can have a length specification of *asterisk*. The length specified for a character statement function or statement function dummy argument of type character must be an integer constant expression. Neither an asterisk nor an expression is allowed in the length specification in level 8R1 (see 6.3.2).
- The length for a CHARACTER array element can occur before and after the element (that is, *a*(*d*)**length*). In level 8R1, the length can come before the subscript but an error is issued if it appears after the subscript (see 6.3.2).
- The comma is optional in the character type statement in the form:

  ```
  CHARACTER[*len[,]] nam[,nam] . . .
  ```

  Level 8R1 issues an error message for the comma following *len* (see 6.3.2).
- In the IMPLICIT statement, the length for character entities can be an unsigned, nonzero integer constant, or an integer constant expression enclosed in parentheses that has a positive value. An unsigned, nonzero integer constant is allowed in level 8R1 but not an expression (see 6.3.1).
- A BLOCK DATA subprogram name can occur in an EXTERNAL statement. Level 8R1 does not allow the optional name for a BLOCK DATA subprogram (see 6.9 and 7.6.2).
- The INTRINSIC statement identifies a symbolic name as representing an intrinsic function. Level 8R1 does not have the INTRINSIC statement (see 6.10).
- The SAVE statement retains the definition status of an entity after execution of a RETURN or an END statement in a subprogram. Level 8R1 does not have a SAVE statement (see 6.11).

Conflicts:

- Character equivalencing is based on character storage units in levels 9R1 and higher and on words in level 8R1. This can give different results. This applies to both explicit EQUIVALENCE statements and to argument association and COMMON association. For example:

  ```
  CHARACTER A*4,B*4,C(2)*3
  EQUIVALENCE (A,C(1)),(B,C(2))
  ```

Levels 9R1 and Higher:

1	2	3	4	5	6	7	8
A							
			B				
C(1)			C(2)				

Level 8R1:

1	2	3	4	5	6	7	8
A				B			
C(1)				C(2)			

The COMPILER statement provides an option, STD=66, to provide for compatibility with lower levels of ASCII FORTRAN (see 8.5 and 8.5.6).

- The PARAMETER statement gives a constant a symbolic name. If the type of the name is not implied by the name rule (see 2.4.2.1), the type must be specified by an explicit type statement or by an IMPLICIT statement prior to the appearance of the name in a PARAMETER statement. Symbolic names of constants defined in PARAMETER statements have no type in level 8R1. Assignment of the value of the expression is done in levels 9R1 and higher as in an assignment statement (that is, with type conversion, if necessary). The syntax of the PARAMETER statement is different in level 8R1 in that no parentheses can be used. Both forms of the PARAMETER statement syntax are allowed in levels 9R1 and higher (see 6.8). The STD=66 option of the COMPILER statement provides for compatibility on previous levels of ASCII FORTRAN for typing of symbolic names of constants (see 8.5 and 8.5.6).

J.9. DATA Statement

Extensions:

- The comma before the variable list is optional, that is, [[,]*nlist/clist/*] . . . . The comma is required in level 8R1. A warning is given if the comma is omitted (see 6.12.1).
- Substring names are allowed in the variable list. Level 8R1 doesn't allow substring names in a DATA statement variable list (see 6.12.1).

Conflicts:

A symbolic name of a constant (defined in a PARAMETER statement) that begins with the letter O, and appears in the constant list of a DATA statement is interpreted by level 8R1 as an octal constant and by levels 9R1 and higher as the symbolic name of the constant (see 6.12.1).

For example:

```
PARAMETER (O2=5.)
DATA A/O2/
```

J.10. Assignment Statements

Extensions:

None.

Conflicts:

None.

J.11. Control Statements

Extensions:

The DO-variable and the DO-statement parameters can be real, double precision, or integer. Level 8R1 generates an error for noninteger DO-variables (see 4.5).

Conflicts:

In the standard, a DO-loop need not be executed. The iteration count is given by MAX(INT((*m2*-*m1*+*m3*)/*m3*),0). The DO-loop is not executed if *m1* > *m2* and *m3* > 0 or if *m1* < *m2* and *m3* < 0. In level 8R1, a DO-loop is always executed. The iteration count is given by MAX *e2*-*e1*)/*e3*+1),1). If *e1* > *e2* and *e3* is omitted, level 8R1 assumes *e3*=-1, and levels 9R1 and higher assume *e3*=+1 (see 4.5.4.1). The STD=66 option of the COMPILER statement provides for compatibility with previous levels of ASCII FORTRAN for DO-loops (see 8.5 and 8.5.6).

J.12. Input/Output Statements

Extensions:

- The internal unit identifier is a character variable or character array or character array element or character substring that specifies an internal file. Level 8R1 does not allow a character substring for an internal unit identifier (see 5.9.1 and 5.9.3).
- An empty I/O list is allowed on reads or writes to skip a record or to write an empty record. A level 8R1 compiler requires a nonempty I/O list on list-directed reads, unformatted sequential-access writes, list-directed write or print, and unformatted direct-access writes. Errors are issued when the list is missing (see 5.6.1.4, 5.6.2.2, 5.6.2.4, and 5.7.3).
- Character substring names are allowed in I/O lists. Level 8R1 allows them only in output lists (see 5.2.3).
- Character constants produced by list-directed output are not delimited by apostrophes, are not preceded or followed by a blank or comma, and do not have internal apostrophes represented by two apostrophes.
- The implied-DO list parameters are the same as the new DO-loop parameters (that is, more types, zero iterations possible). Level 8R1 doesn't allow the new parameters (see 4.5 and 5.2.3).
- The OPEN statement is not in level 8R1 (see 5.10.1).
- The CLOSE statement is not in level 8R1 (see 5.10.2).
- The INQUIRE statement is not in level 8R1 (see 5.10.3).

Conflicts:

Character variables are only blank-filled to the declared size of the variable during assignment statements and I/O. Prelevel 9R1 compilers and I/O systems performed blank-fill to word boundaries even though the character variable was not a multiple of four characters. The STD=66 option does not change this incompatibility. Prelevel 9R1 absolute elements that use the FORTRAN common bank C2F$ will not execute as before. That is, the old compiler will blank-fill to a word boundary, but formatted I/O will not blank-fill to a word boundary. Character comparisons of the data will not find any equal conditions.

J.13. Format Specification

Extensions:

If the output format is G*w* .*d* E*e* and the value of the variable fits an F format, the format used is F(*w*- (*e*+2)).*d* -*i*, (*e*+2)('*b*'), where *b* is a blank. The format is F(*w*-4).*d* -*i*, 4('*b*') in level 8R1 (see 5.3.1).

Conflicts:

- During list-directed input, if the first record read in a read operation has no characters preceding the first value separator, this indicates a null field. In level 8R1, it does not indicate a null field but is handled the same as any other record. If you use the STD=66 option of the COMPILER statement, execution chooses level 8R1 and earlier methods of input.
- During list-directed output, character constants always have the PRINT format (that is, no apostrophes around character output). In level 8R1, PRINT and WRITE have different formatting in that apostrophes are used during the WRITE. Also, on list-directed output in levels 9R1 and higher, a complex constant must be written on one record if it fits, by itself, on a record. Level 8R1 breaks it up without checking to see if it fits on one line. If you use the STD=66 option of the COMPILER statement, execution proceeds with level 8R1 and earlier methods of output.

J.14. Main Program

Extensions:

The PROGRAM statement to name a main program must be the first statement in the main program if it occurs. The PROGRAM statement is not implemented in level 8R1 (see 7.2.1).

Conflicts:

None.

J.15. Functions and Subroutines

Extensions:

- The following intrinsic functions are not in level 8R1: ICHAR, CHAR, LEN, INDEX, ANINT, DNINT, NINT, IDNINT, DPROD, LGE, LGT, LLE, LLT, UPPERC, LOWERC, and TRMLEN (see 7.7.1).
- Extended conversion intrinsic functions are the following: INT, REAL, DBLE, and CMPLX for all argument types. Level 8R1 gives warnings for use of complex type with INT and proceeds to flag further uses of INT as a user function. Levels 9R1 and higher allow integer, real, complex, and double-precision types as arguments for INT. Level 8R1 gives warnings for any use of REAL with variables other than complex and proceeds to flag further uses of REAL as a user function. Levels 9R1 and higher allow integer, real, and complex types as arguments for REAL. Level 8R1 gives warnings for any use of DBLE with complex variables, and interprets further calls of DBLE as a user function. Levels 9R1 and higher allow integer, real, double-precision, and complex types as arguments for DBLE. Level 8R1 gives warnings for any complex variables used as arguments of CMPLX and interprets further calls of CMPLX as a user function. Levels 9R1 and higher allow integer, real, double-precision, and complex types arguments for CMPLX (see 7.7.1).

- The FUNCTION statement specifies the length for a character function as CHARACTER[**length*]C(A). Level 8R1 allows the length after the function name, that is, CHARACTER FUNCTION C*3(A) (see 7.3.2.1). Both forms are allowed in level 9R1 and higher.
- Empty parentheses are allowed on the SUBROUTINE statement. The level 9R1 and higher form is SUBROUTINE *sub* [([*d*[,*d*]. . .])]; the level 8R1 form is SUBROUTINE *sub* [(*d*[,*d*] . . .)] . The forms *sub* and *sub*() are equivalent (see 7.3.3.1).
- An actual argument for a subroutine call can be **s*, where *s* is a statement number. Level 8R1 uses currency signs ($) or ampersands (&) for the statement number (see 7.3.2.2 and 7.3.3.2).
- A dummy argument array name can be associated with an actual argument that is an array element substring as well as an array or array element. Level 8R1 passes a temporary for an array element substring.
- A dummy argument that becomes defined can be associated with a substring as an actual argument. Level 8R1 associates it with a variable, an array element, or an expression.
- Empty parentheses are allowed in the ENTRY statement. The level 9R1 and higher form is ENTRY *en*[([*d*[,*d*] . . .])] while the level 8R1 form is ENTRY *en*[(*d*[,*d*] . . .)] . Levels 9R1 and higher require that the function be specified with the form *en*() even if the entry statement did not have the empty set of parentheses (see 7.5).

Conflicts:

- Statement functions are typeless in level 8R1, but are typed (just like other functions) in levels 9R1 and higher. This can cause different results because of implied type conversions (see 6.3 and 7.3.1). The STD=66 option of the COMPILER statement provides for compatibility of levels 9R1 and higher with previous levels of ASCII FORTRAN for typing of statement functions (see 8.5 and 8.5.6).
- Register A1 contains a function packet address for character function references (see K.4.6). For ASCII FORTRAN level 8R1, register A1 contains a result address for character function references. This is an incompatibility between level 8R1 and all higher levels. If a program compiled with level 9R1 or higher refers to a character function program compiled by level 8R1, the STD=66 option of the COMPILER statement must be present in the level 9R1 or higher program.
- If the value of *e* in RETURN[*e*] is less than one or greater than the number of asterisks in the subroutine entry, control in level 9R1 returns to the CALL statement that initiates the subprogram reference. In level 8R1, an error is issued and the program continues with unknown results (see 7.4).

J.16. BLOCK DATA Subprogram

Extensions:

An optional global name can be specified for a BLOCK DATA subprogram, that is, BLOCK DATA [*name*]. Level 8R1 doesn't allow the optional name.

Conflicts:

None.

Appendix K
Interlanguage Communication

K.1. ASCII FORTRAN (FTN) to FORTRAN V

The FORTRAN V subprogram must be declared as:

```
EXTERNAL a(FOR)
```

or:

```
EXTERNAL *a
```

where *a* represents the FORTRAN V subprogram name.

The syntax of a call to a FORTRAN V subprogram is the same as a call to an ASCII FORTRAN subprogram.

Restrictions and Considerations:

- If both the ASCII FORTRAN and FORTRAN V programs perform I/O operations, you must reassemble the ASCII FORTRAN library element F2FCA. Buffer area allocation and release is not common between the two FORTRANs, and I/O operations may fail. This problem is resolved by reassembling element F2FCA with the required amount of main storage. To determine the required amount, refer to G.7.
- FORTRAN V Series E subprograms are restricted to symbiont types of I/O; the tag CLOST$ will be undefined at collection time. This can be ignored since CLOST$ is defined in FORTRAN V Series T.
- Any files opened by FORTRAN V Series T must be explicitly closed using a CALL CLOSE statement in a FORTRAN V subprogram if they are to be usable after program termination.
- The FORTRAN V subprogram cannot call the EXIT service subprogram.
- The FORTRAN V subprogram name can't appear in a BANK statement.
- FORTRAN V subprogram arguments and function names should not be type character or double-precision complex, because FORTRAN V supports neither data type.
- The walkback mechanism doesn't work if a walkback is attempted from a FORTRAN V subprogram to an ASCII FORTRAN program.
- Common blocks or local variables shared with or passed to FORTRAN V routines should not be in virtual or banked space.

K.2. ASCII FORTRAN to PL/I

The PL/I external procedure must be declared as:

```
EXTERNAL a(PL1)
```

where *a* represents the PL/I procedure name.

The syntax of a call to a PL/I procedure is the same as a call to an ASCII FORTRAN subprogram.

K.2.1. Restrictions and Considerations

- Level 8R1 PL/I or higher must be used.
- Any file opened by a PL/I procedure must be closed by a PL/I procedure. Files can be shared between ASCII FORTRAN and PL/I, but they must be closed by the language that opened them before they can be accessed by the other language. A file that is opened by a given language does not have to be closed before switching to another language as long as the called language routine does not access the file.
- The PL/I procedure name can't be a dummy argument name.
- An argument to a PL/I procedure cannot be a label, subprogram name, or array name. An array element can be passed from ASCII FORTRAN to PL/I, but PL/I must declare its counterpart as a single data item; structures or cross-sections of arrays are not allowed.
- You can use common blocks to pass an array name from ASCII FORTRAN to PL/I. The PL/I procedure must use the EXTERNAL attribute on the declaration, and the ASCII FORTRAN program must specify the array in a named common block. The PL/I common block name is the variable name with the EXTERNAL attribute. ASCII FORTRAN stores arrays in column-major order while PL/I stores them in row-major order. This means that either the ASCII FORTRAN program must transpose the array so that it is in row-major order when the PL/I procedure is called or the PL/I procedure must refer to the array with the subscripts in reverse order and the array dimensioned in reverse order.

  For example:

```
ASCII FORTRAN                PL/I

EXTERNAL PL1SUB(PL1)         PL1SUB: PROC;
INTEGER ARR(2,4,6)           DCL 1 BLK1 EXTERNAL ALIGNED,
COMMON/BLK1/ARR                   2  ARR(6,4,2) FIXED DECIMAL(10,0);
ARR(2,3,4) = 234             PUT SKIP ('WANT 234 :', ARR(4,3,2));
CALL PL1SUB                  PUT SKIP;
END                          END;
```

- If execution is stopped by the PL/I procedure using the STOP statement, any files opened by ASCII FORTRAN are not properly closed unless the ASCII FORTRAN program explicitly closes them via the CLOSE statement.
- Common blocks or local variables shared with or passed to PL/I should not be in virtual or banked space.

K.2.2. PL/I Argument Counterparts

PL/I has the following argument counterparts to ASCII FORTRAN.

ASCII FORTRAN	PL/I	Comment
INTEGER	FIXED BINARY (*p*,*q*) FIXED DECIMAL (*p*,*q*)	*p* can range 1-35, *q* must be 0. *p* can range 1-10, *q* must be 0.
REAL	FLOAT BINARY (*p*) FLOAT DECIMAL (*p*)	*p* can range 1-27. *p* can range 1-8.
DOUBLE PRECISION	FLOAT BINARY (*p*) FLOAT DECIMAL (*p*)	*p* can range 28-60. *p* can range 9-18.
COMPLEX	FLOAT BINARY COMPLEX (*p*) FLOAT DECIMAL COMPLEX (*p*)	*p* can range 1-27. *p* can range 1-8.
COMPLEX*16	FLOAT BINARY COMPLEX (*p*) FLOAT DECIMAL COMPLEX (*p*)	*p* can range 28-60. *p* can range 9-18.
LOGICAL	BIT (36) ALIGNED	Only the rightmost bit is used by ASCII FORTRAN. The PL/I string may not be of varying length.
CHARACTER**n*	CHARACTER (*n*)	The PL/I string may not be of varying length.

K.3. ASCII FORTRAN to ASCII COBOL (ACOB)

The ASCII COBOL (ACOB) subprogram must be declared as:

```
EXTERNAL a(ACOB)
```

where *a* represents the ASCII COBOL subprogram name.

The syntax of a call to an ACOB subprogram is the same as a call to an ASCII FORTRAN subprogram.

K.3.1. Restrictions and Considerations

- ACOB level 4R2 or higher must be used.
- Any file opened by an ACOB subprogram must be closed by an ACOB subprogram. Files can be shared between ASCII FORTRAN and ACOB, but they must be closed by the language that opened them before they can be accessed by the other language. A file that is opened by a given language doesn't have to be closed before

switching to another language as long as the called language routine doesn't access the file.

- The ACOB subprogram name can't be a dummy argument name.
- It is your responsibility to ensure the data alignment is the same for an ASCII FORTRAN argument and its ACOB counterpart. Special care must be taken when passing character type data to ACOB. If the ASCII FORTRAN argument is not word-aligned, the ACOB argument declaration must reflect the offset via the use of a structure. To ensure that an ASCII FORTRAN character scalar or array is word-aligned, place it as the first item in COMMON or equivalence the character item to an integer variable.
- The ACOB subprogram name must not be a function name since ACOB doesn't support functions.
- An argument to an ACOB subprogram must not be a label or subprogram name since ACOB has no argument counterpart.
- An array name can be passed from ASCII FORTRAN to ACOB as an argument. However, ASCII FORTRAN stores arrays in column-major order while ACOB stores them in row-major order. This means either the ASCII FORTRAN program must transpose the array so that it will be effectively in row-major order when the ACOB subprogram is called, or the ACOB procedure must reference the array with the subscripts in reverse order and the array dimensioned in reverse order. Beware of ASCII FORTRAN and ACOB alignment conventions.

  Example:

```
ASCII FORTRAN                           ACOB

EXTERNAL C(ACOB)                        LINKAGE SECTION.
CHARACTER*5 ARR(2,4,6)                  01   BUFF.
EQUIVALENCE (ARR(1,1,1),IDUM)                02 BUFA OCCURS 6 TIMES.
ARR(2,3,4) = 'ABCD'                          03 BUFB OCCURS 4 TIMES.
CALL C(ARR)                                  04 BUFC PIC X(5) OCCURS 2
TIMES.
PRINT *, 'WANT EFG:', ARR(2,3,4)        PROCEDURE DIVISION USING BUFF.
END                                     C.
                                             DISPLAY 'WANT ABDC:',
                                              BUFC (4, 3, 2).
                                             MOVE 'EFG' TO BUFC (4, 3, 2).
                                             EXIT PROGRAM.
```

- If execution is stopped by the ACOB subprogram, any files opened by ASCII FORTRAN don't close properly unless the ASCII FORTRAN program explicitly closes them via the CLOSE statement.
- If ASCII COBOL passes a group item with no explicit type to an ASCII FORTRAN program, the corresponding ASCII FORTRAN argument can be type INTEGER, REAL, DOUBLE PRECISION, or LOGICAL. CHARACTER type is not allowed. ASCII FORTRAN can only access that portion of a COBOL group item that is declared, either explicitly or implicitly, by the ASCII FORTRAN program.
- Common blocks or local variables shared with or passed to ACOB should not be in virtual or banked space.

K.3.2. ASCII COBOL Argument Counterparts

ASCII COBOL (ACOB) has the following argument counterparts to ASCII FORTRAN.

ASCII FORTRAN	ACOB	Comment
INTEGER	PIC S9(10) COMP SYNC	Ensure that the ACOB item is word-aligned.
REAL	COMP-1	
DOUBLE PRECISION	COMP-2	
COMPLEX	No ACOB counterpart	
LOGICAL	PIC 1 (36) SYNC	Only the rightmost bit is used by ASCII FORTRAN. Ensure that the ACOB item is word-aligned.
CHARACTER*(n)	PIC X(n)	Ensure that the alignment is the same for ASCII FORTRAN and ACOB.
TYPELESS ‡	PIC 1 (36) SYNC	Ensure that the ACOB item is word-aligned.

‡ A typeless argument results from a typeless function, see 2.5.5.1.

K.4. ASCII FORTRAN and MASM Interfaces

This subsection provides information needed when writing Assembler routines that call or are called by ASCII FORTRAN routines.

K.4.1. Arguments

For procedure calls with one or more arguments, ASCII FORTRAN requires an argument list. The address of the argument list is in H2 of register A0. A0 also contains the number of character arguments in S1 and the total number of arguments in Q2. Bit number 9 of A0 (leftmost bit is bit 1) specifies when argument type checking is desired by the caller.

Format of register A0 for calling an ASCII FORTRAN program:

A		B	C	D

where:

A

(S1 of A0) is the number of character arguments (maximum of 63).

B

(Bit 9 of A0; assume bits are numbered 1-36 and the leftmost bit is 1) is the argument type checking bit. If set to 1 by the caller, argument type checking will not be done. If set to 0, argument type checking will be done unless the called subprogram has disabled type checking. The called subprogram can disable type checking by using the COMPILER statement option ARGCHK=OFF or by compiling the called subprogram with optimization (Z or V option).

C

(Q2 of A0) is the total number of arguments (maximum is 250).

D

(H2 of A0) is the address of the argument list descriptor words that are explained in the following discussion.

Addressing Words:

BDI of argument	Address of argument

Character Descriptor Words:

offset	length	0

Argument Type Words

type	type	type	type	type	type

The addressing words follow one another consecutively in storage. There is one addressing word for each argument. The Bank Descriptor Index (BDI) is required if the ASCII FORTRAN subprogram being called expects banked arguments and the argument is not in the control D-bank. It is also required if the argument is an external subprogram name and the ASCII FORTRAN subprogram being called has the LINK=IBJ$ or BANKED=ALL COMPILER statement options present. Otherwise, the BDI is zero. If an ASCII FORTRAN program calls a MASM routine with arguments that have a BDI associated with them (that is, the actual data passed resides in a banked common block), the MASM routine is responsible for basing the argument's data bank. All D-bank basing must be done by an ASCII FORTRAN library routine in element F2ACTIV$. In other words, no LDJ or LBJ instructions should appear in your assembly code to switch the utility D-bank basing. The interface to the activate routine is:

```
LA       A0, addreswd       .  GET BDI and address in A0
LMJ      X11,VACTIV$        .  base D-bank
```

A0 now contains the item's absolute address, and its bank is based. All registers except A0 and X11 are preserved. For details on ASCII FORTRAN banked programs, see Appendix H.

The character descriptor words follow one another consecutively in storage and follow the addressing words. A character function name or a Hollerith string passed as an argument does not have a character descriptor word. All other character types of arguments have a character descriptor word. The offset is the byte offset of the start of the character item within the word and has the value 0, 1, 2, or 3. If the character item begins on a word boundary, the offset is zero. A character item beginning on Q2 of the word has an offset of 1, an item beginning on Q3 has an offset of 2, and an item beginning on Q4 has an offset of 3. The character length, represented in number of characters, is in Q2 of the character descriptor word. If a character array is passed as an argument, the length passed is the size of an array element. If a character substring is passed as an argument, the length passed is the length of the substring.

If argument type checking is desired, bit 9 of A0 (assume bits numbered from 1 to 36 and the leftmost bit is 1) must be zero and the argument type words must be present. The argument type words follow one another consecutively in storage and follow the character descriptor words. There is one type word for each six arguments. The following is a list of allowable types:

0	Subprogram
1	Integer
2	Real
3	Double-Precision Real
4	Complex
5	Double-Precision Complex
6	Character
7	Logical
8	Label
9	Hollerith

A type 0 subprogram matches any other type. Type 9, Hollerith, matches all types except character and label. All other types must match exactly or else a run-time diagnostic message is issued when type checking is enabled. The argument type words are optional. If they are not present, either the caller must specify so by setting bit 9 of A0 to 1 or the callee must disable type checking by using the COMPILER statement option ARGCHK=OFF or compiling with optimization.

If an Assembler routine refers to an ASCII FORTRAN subprogram that contains a COMPILER statement STD=66 option, or is compiled by an ASCII FORTRAN compiler lower than level 9R1, the contents of A0 and the packet format differ. S1 and bit 9 of A0 are ignored. The character descriptor words and the argument type words are not required. In addition, any character item passed to the ASCII FORTRAN subprogram must begin on a word boundary.

K.4.2. ASCII FORTRAN Register Usage

An ASCII FORTRAN subprogram saves and restores all registers that it uses except for the volatile set X11, A0-A5, and R1-R3. An ASCII FORTRAN subprogram requires register R15 to contain the address of a storage control table (F2SCT), which is used on I/O operations and by several of the ASCII FORTRAN library routines. Register R15 is loaded with the F2SCT address as part of the initialization performed by an ASCII FORTRAN main program. Once R15 has been initialized, it is the responsibility of any routine outside the ASCII FORTRAN environment to preserve its contents upon reentry to an ASCII FORTRAN subprogram.

K.4.3. Initializing the ASCII FORTRAN Environment

Under normal conditions, the ASCII FORTRAN environment is initialized by a call from the ASCII FORTRAN main program to the ASCII FORTRAN initialization routine. If an ASCII FORTRAN subprogram is called from an Assembler routine and there is not an ASCII FORTRAN main program, it is the responsibility of the Assembler routine to call the ASCII FORTRAN initialization routine. One of two ASCII FORTRAN initialization routines must be called by the Assembler routine before the ASCII FORTRAN subprogram is called. The ASCII FORTRAN initialization need be called only once during the program execution. The initialization routines use only the volatile set of registers. Both initialization routines acquire buffer space and initialize tables that are used by I/O and ASCII FORTRAN library routines. This space is acquired by the common storage management system. If the Assembler routine or controlling program has its own storage management system, the ASCII FORTRAN library element F2FCA can be modified and reassembled to avoid calls to ER MCORE$. See G.7 for details.

One of the initialization routines also registers a contingency routine for capturing contingency interrupts, which is required for the proper execution of some ASCII FORTRAN programs. However, since some applications prefer to capture their own contingencies, a second routine that does not register contingencies is provided. The ASCII FORTRAN service subprograms UNDSET, OVFSET, and DIVSET register contingencies and should not be called if an application depends on another contingency registration.

The following call initializes the ASCII FORTRAN environment but doesn't register the ASCII FORTRAN contingency routine.

```
LMJ A2,FINT$
```

The following call initializes the ASCII FORTRAN environment and registers the ASCII FORTRAN contingency routine.

```
LMJ X11,FINT2$
```

On return from either of the initialization routines, register R15 contains the address of the ASCII FORTRAN storage control table (F2SCT). It is the responsibility of the Assembler routine to ensure that R15 contains the F2SCT address when calling an ASCII FORTRAN subprogram.

K.4.4. Terminating the ASCII FORTRAN Environment

Under normal conditions, the ASCII FORTRAN environment is terminated by a call from the main program to the ASCII FORTRAN termination routine. The function of the termination routine is to output buffered I/O to the appropriate files and close all opened files. An ER EXIT$ is then performed which terminates the program. If an Assembler routine calls an ASCII FORTRAN subprogram and control never reaches an ASCII FORTRAN main program for normal program termination, it is the responsibility of the Assembler routine to close all opened I/O files. Files can best be closed by using the CLOSE statement in the ASCII FORTRAN subprogram. If the ASCII FORTRAN termination routine is called, files are closed but control is not returned to the caller. The following is the Assembler call to the ASCII FORTRAN termination routine.

```
LMJ X11,FEXIT$
```

K.4.5. Calling an ASCII FORTRAN Subprogram

A call to an ASCII FORTRAN subprogram takes one of several forms, depending on whether the subprogram is banked or not. For a nonbanked ASCII FORTRAN subprogram call, the following Assembler linkage can be used:

```
LXI,U  X11,0
LMJ    X11,entry-point
```

ASCII FORTRAN returns to the caller via:

```
J  0,X11
```

For a banked ASCII FORTRAN subprogram call (that is, the ASCII FORTRAN subprogram has the COMPILER statement option BANKED=ALL, BANKED=RETURN, or LINK=IBJ$), use one of two calling sequences:

```
LXI,U  X11,bdi-of-the-subprograms-bank
LIJ    X11,entry-point
```

or:

```
LXI,U  X11,BDICALL$+entry-point
IBJ$   X11,entry-point
```

For a description of IBJ$ and BDICALL$, see the *Series 1100 Collector, Programmer Reference*, UP-8721.

An ASCII FORTRAN banked subprogram returns by:

```
LA,H1  A4,X11-save-location
JZ     A4,0,X11
LIJ    X11,0,X11
```

K.4.6. ASCII FORTRAN Function References

If an ASCII FORTRAN (level 9R1 or higher) character function is referred to, register A1 must be set up by the calling routine to contain:

0	*fctn-pkt-addr*

where *fctn-pkt* has the form:

0		*result-addr*
0	*char-length*	0

The *result-addr* field in H2 of the first word points to the caller's storage area where the result of the function will be stored. This storage area must be in the control bank or be visible to the function. It is the caller's responsibility to ensure that the function result area is large enough to hold the function result.

The *char-length* field in Q2 of the second word of the packet is the length of the function result expressed in number of characters.

For ASCII FORTRAN levels 8R1 and lower, or whenever the COMPILER statement option STD=66 is used in the function being called, register A1 must contain:

0	*result-addr*

The *result-addr* field description is the same as for levels 9R1 and higher. A *fctn-pkt* is not required for levels lower than 9R1.

An ASCII FORTRAN character function places the function result in the caller's storage area pointed to by *result-addr*. If the value of a function isn't a character string, it is returned in registers A0, A0 through A1, or A0 through A3, depending on the function type.

K.4.7. Example

The following is an example of an Assembler routine that passes three arguments to the ASCII FORTRAN subroutine FTEST.

```
          AXR$
$(1)
MATH*     LMJ       A2,FINT$           . initialize FTN
.
          L,U       R4,4               . loop initialization count
FTNREF    LA        A0,(020003,ARGS)   . list pointer
          LXI,U     X11,0              .
          LMJ       X11,FTEST          . call nonbanked FTN rtn
```

```
9.          LA       A4,REALNO          . bump 2nd arg by 1.0
10.         FA       A4,(1.0)           .
11.         SA       A4,REALNO          .
12.         JGD      R4,FTNREF          . repeat call to FTN rtn
13. .
14.         LMJ      X11,FEXIT$         . terminate program
15. .
16. $(0)
17. CHARD   FORM     9,9,18             .
18. .
19. ARGS    +        STRING             . address of 1st arg
20.         +        REALNO             . address of 2nd arg
21.         +        STRING             . address of 3rd arg
22.         CHARD    0,3,0              . 'ASM' offset=0 len=3
23.         CHARD    3,3,0              . 'B17' offset=3 len=3
24.         +        6,2,6,0,0,0        . char real char type
25. .
26.         ASCII
27. STRING  'ASMB17'                    .
28. REALNO  +        2.5                .
29.         END      MATH               .
```

The following is the ASCII FORTRAN subroutine FTEST that is called from the preceding Assembler routine:

```
1.      SUBROUTINE FTEST (CALTYP, X, CALID)
2.      CHARACTER CALTYP*(*), CALID*3
3.  *
4.      PRINT *, 'CALLER ID ', CALID, ' TYPE ', CALTYP
5.      PRINT *, 'SIN OF', X, ' = ', SIN(X)
6.      PRINT *, 'COS OF', X, ' = ', COS(X)
7.      RETURN
8.      END
```

The execution of the program is:

```
CALLER ID B17 TYPE ASM
SIN OF 2.5000000  =  .59847214
COS OF 2.5000000  = -.80114362
CALLER ID B17 TYPE ASM
SIN OF 3.5000000  = -.35078323
COS OF 3.5000000  = -.93645669
CALLER ID B17 TYPE ASM
SIN OF 4.5000000  = -.97753011
COS OF 4.5000000  = -.21079580
CALLER ID B17 TYPE ASM
SIN OF 5.5000000  = -.70554033
COS OF 5.5000000  =  .70866977
CALLER ID B17 TYPE ASM
SIN OF 6.5000000  =  .21511999
COS OF 6.5000000  =  .97658762
```

At line 3 of the Assembler routine, a call is made to initialize the ASCII FORTRAN environment. The ASCII FORTRAN initialization routine acquires I/O buffer storage via the common storage management system. If the Assembler routine has its own storage management system, the ASCII FORTRAN library element F2FCA needs modifications and reassembling. See Appendix G for details.

At line 6, register A0 is loaded with a literal that specifies:

1. Two character arguments (S1)

2. Argument type checking (bit 9 of the literal is 0)
3. Three total arguments (Q2).

The second half of A0 contains the address of the argument list.

Lines 19 through 21 contain the argument addressing words. Lines 22 through 23 contain the character descriptor words for the first and third arguments, respectively, which are character types. The first argument passed is the character string ASM, the second argument passed is the real number 2.5 (which is modified after each call by the Assembler routine), and the third argument is the character string B17.

Line 24 contains the argument type word that the ASCII FORTRAN subprogram requires for argument type checking. The first and third arguments are character types as indicated by the 6, and the second argument is a real type as indicated by the 2.

At line 14, the ASCII FORTRAN termination routine is called. This closes any opened I/O file and then terminates program execution using an ER EXIT$.

Appendix L
ASCII FORTRAN Sort/Merge Interface

L.1. General

A sort/merge interface is available from ASCII FORTRAN to the sort/merge package. The sort/merge package is described in the *OS 2200 Sort/Merge Programming Guide, Level 17R1* (7831 0687).

L.2. Sort/Merge Features Available Through ASCII FORTRAN

Enter the sort/merge interface using the CALL statement in ASCII FORTRAN. The sort/merge interface provides the following functions:

CALL FSORT

Perform a sort.

CALL FMERGE

Perform a merge.

CALL FSCOPY

Specify an Assembler sort parameter table to be copied. This table can be used in subsequent sorts or merges. The use of such a table can also be inhibited.

CALL FSSEQ

Specify the collating sequence that is used in subsequent sorts or merges. The use of such a collating sequence can also be inhibited.

CALL FSGIVE

Deliver an input record to sort without leaving your input subroutine.

CALL FSTAKE

Receive an output record from sort without leaving your output subroutine.

L.3. Sort/Merge Interface Restrictions and Concerns

The following are sort/merge interface restrictions.

1. The data passed to the sort/merge interface must not be banked. The scratch area used by the sort/merge interface must not be banked.
2. The sort/merge interface is restricted to reading files that can be read by an ASCII FORTRAN READ statement. Files such as those created by NTRAN$ are not allowed.
3. You can't use an asterisk as a length specification for the dummy character arguments for the following user-specified subroutines:
   - Input
   - Output
   - Comparison
   - Data reduction

   The length specification must be an unsigned, nonzero integer constant, or an integer constant expression enclosed in parentheses that has a positive value.
4. When the file R$CORE is not assigned to the run (for FSORT only), the sort/merge interface attempts to get storage space from the ASCII FORTRAN library Common Storage Management System (CSMS). If you supply a version of ASCII FORTRAN library element F2FCA so that the CSMS routines are not used, you must make element F2FCA large enough to accommodate the storage area needed by the sort/merge interface routines, the sort/merge package, and the FORTRAN library (see G.7).
5. After a call to FSORT or FMERGE, the file pointers for the input and output files are positioned to their initial point.

L.4. CALL Statement for a Sort (FSORT)

The form of the CALL statement for a sort is:

```
CALL FSORT (infost,inpt,outpt [,comprt] [,datrd] )
```

where:

infost

is the information string, a character variable or literal that describes various parameters to the sort/merge interface, such as record sizes, key fields, and scratch facilities for FSORT. A key field (or your comparison routine) and a record size must be specified in *infost.*

infost contains items of information separated by commas. Blanks in *infost* are ignored. No distinction is made between uppercase and lowercase alphabetic characters. *infost* is scanned from left to right and must be terminated by some character that is an illegal ASCII FORTRAN character, such as an exclamation point (!). An asterisk (*) must not terminate *infost.*

infost can contain several clauses. The mnemonics used are truncated by the sort/merge interface to the first four characters. The following items can be used in *infost* for the call to FSORT:

1. RSZ=*rlch*

   rlch is the record length in ASCII characters. This record size must be specified when a sort is to be done. The RSZ clause can appear only once in *infost*. A record size clause can appear only once in *infost*; for example, the RSZ and VRSZ clauses can't appear in the same *infost*.

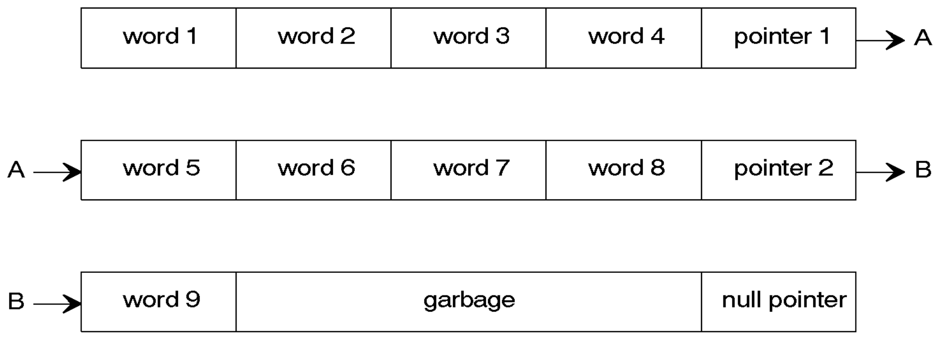
2. VRSZ=*mrlch/lnkszch*

   mrlch is the maximum record size in ASCII characters for variable length records. *lnkszch* is an optional parameter indicating link size in ASCII characters. When *lnkszch* is omitted, the slash (/) can also be omitted. *lnkszch* must be large enough to accommodate all keys. For example, if the keys are specified by:

   ```
   KEY=(11/15,1/10/d/s)
   ```

   the last character in any key field for this KEY specification is the 25th character. Therefore, the link size must be at least 25 characters long. The link size should be specified only when a comparison routine has been specified. The VRSZ clause can appear only once in *infost*. A record size clause can appear only once in *infost*; for example, the RSZ clause and the VRSZ clause can't both be used in the same *infost*.

   When sorting variable-length records, the records are separated into smaller parts (links) of equal size that are joined by pointers. As an example, consider a record of nine words with the link size four words. Schematically, the record is stored as:



   When the link size is given in the VRSZ clause, consider the following rules:

   - The link size should not be too small. For example, if the link size is given as one word, the core (main storage) required for each record is exactly twice the record size. This means that the sort uses many more resources (main storage, mass storage, and tapes) than necessary.
   - The link size should not be too large, since this may mean that much of the area remains unused in the last link. This causes problems because of poor use of main storage.

   Note: *The two rules are in conflict. The choice of a link size requires a compromise between these two rules.*

It is frequently advantageous to sort short variable-length records as if these records were fixed-length records because of the resources used by the sort/merge package. If these records are treated as fixed length, you must keep track of the record length.

3. CONS

   CONS indicates that the closing messages from the sort/merge package are to be sent to the system console. If CONS is not present, the SORT CONSOLE configurable runtime parameter determines if messages are sent to the system console. The opening messages give the block sizes on mass storage and may be used to check the efficiency of the sort/merge usage of the scratch area. The closing messages give the input and output record counts and the bias of the data. The CONS specification can appear only once in *infost.*

4. DELL

   DELL is used to indicate that the opening and closing messages from the sort/merge package should not be sent to the system log. If DELL is not present, the SORT LOG configurable runtime parameter determines if messages are sent to the system log. This clause can appear only once in *infost.*

5. KEY=*keysp*

   or:

   KEY=*keyspn*

   keysp is a single key specification; *keyspn* is a multiple key specification of the form (*keysp*$_1$, *keysp*$_2$, . . .). The single key specification form, KEY=*keysp*, can occur a maximum of 40 times in *infost.* There can be a maximum of 40 *keysp* specifications within the *keyspn.* More than one KEY=*keyspn* clause can occur in *infost*, but only 40 keys are allowed for each call to FSORT. The limit of 40 keys includes any keys given in the sort parameter tables copied through the COPY clause (see L.7). The key specification can indicate a character key, that is, a key that begins and ends on a character boundary, or a bit key that either starts or ends outside a character boundary. The form of the character key is:

   ```
   charpos/length/seq/type
   ```

   where:

   charpos

   is the position within the record of the most significant character of the key. Character positions are counted from left to right beginning with position 1.

   length

   is an optional field that specifies the length of the key in characters. The default for *length* is 1.

   seq

   is an optional field that specifies the sequencing order of the key. The value A is used for ascending order; D is used for descending. The default value is A.

   type

   is an optional field that specifies the type of the key. The value of this field is B, Q, R, S, T, U, or V; U is the default value. The values for this field indicate:

B

The key field contains a signed number in an OS 1100 system internal representation.

Q

The key field contains a signed decimal number in 9-bit ISO character representation with a sign overpunched on the last digit.

R

The key field contains 9-bit ISO characters with a leading sign, that is, a plus, minus, or blank. Any character in the sign position that is not a plus, a minus, or a blank is set to a blank.

S

The key field contains 9-bit characters. See L.7.

T

The key field contains 9-bit ISO characters with a sign in the last character position, that is, a plus, minus, or blank. Any character in the sign position that is not a plus, minus, or blank is set to a blank.

U

The key field contains an unsigned number in the OS 1100 system internal representation.

V

The key field contains a signed decimal in 9-bit ISO characters with a sign overpunched on the first character.

The form of the bit key is:

```
BIT/wordpos/bitpos/length/seq/type
```

where:

wordpos

is the position within the record of the word that contains the most significant bit of the record. Words within the record are numbered from 1.

bitpos

is the position of the first bit of the key in the first word of that key. Bits are numbered from left to right beginning at 1.

length

is an optional field that specifies the length of the key in bits.

seq

is an optional field that specifies the sequencing order of the key: A for ascending or D for descending. The default is A.

type

is an optional field that specifies the type of the key. The values for *type* can be A, B, D, G, L, M, P, or U; the default is U. The values for *type* indicate:

A

The key field contains 6-bit characters. All A key fields must start and end on 6-bit byte boundaries.

B

The key field contains a signed number in the OS 1100 system internal representation.

D

The key field contains 6-bit Fieldata characters with a leading sign, that is, a plus, minus, or blank. Any character in the sign position that is not a plus, minus, or blank is set to a blank. All D key fields must start and end on 6-bit byte boundaries.

G

The key field contains 6-bit Fieldata characters with a sign in the last character, that is, a plus, minus, or blank. Any character in the sign position that is not a plus, minus, or blank is set to a blank. The key field must begin and end on a 6-bit byte boundary.

L

The key field contains a number in 6-bit Fieldata characters with a sign overpunched on the first digit. The key field must start and end on a 6-bit byte boundary.

M

The key field contains a number in signed magnitude representation. This means that the first bit is the sign (that is, a 1 for negative and a 0 for positive), and the rest of the field is the absolute value of the number.

P

The key field contains a signed decimal number in 6-bit Fieldata characters with a sign overpunched on the last digit. The key field must begin and end on a 6-bit byte boundary.

U

The key field contains an unsigned number in OS 1100 system internal representation.

6. COMP

   COMP indicates the use of your comparison routine. This indicates the presence of *comprt* in the call to FSORT. The COMP clause can appear only once in *infost.* See L.8.2.

7. COPY

   COPY indicates that an Assembler sort parameter table is to be copied. The COPY clause can appear only once in *infost.* See L.6.

8. DATA

   DATA indicates that your data reduction routine is present. This indicates the presence of *datrd* in the call to FSORT. This option can only be specified when fixed length records are sorted. The DATA clause can appear only once in *infost.* See L.8.3.

9. SELE=*recno1*

   or:

   SELE=*recno1*/*recno2*

   or:

   SELE=(*recno1*[/*recno2*] ,*recno3*[/*recno4*] , . . .)

   recno1 through *recno4* are record numbers. The SELE, or select, clause indicates which records are given to the sort/merge package. If the first form is used, only the record specified by *recno1* is given to the sort. If the second form is used with *recno2*, all records from *recno1* through *recno2* are given to the sort. If the third form of the SELE clause is used, the records between each pair of record numbers are given to the sort individually. All records are read, but only those records specified in the SELE clause are given to the sort. Only 10 record number pairs can be used in the third form. For each pair, *recno1* must be less than or equal to *recno2*, and the last number of each pair must be less than the first number of the next pair. If *recno1* appears without *recno2*, or *recno3* appears without *recno4*, only *recno1* or *recno3* are given to the sort. This clause can appear only once in *infost.*

10. CORE=*corsz*

    corsz is the size in words of the scratch area to be used by the sort. At least 3,000 words must be used. In general, the sort runs faster if the scratch area given to sort is expanded. This clause can appear only once in *infost.* See L.9.2.

11. FILE=*file-name*

    or:

    FILE=(*file-name*, *file-name*, . . .)

    file-name is an internal file name. The second form of the FILE clause permits the specification of more than one file name within the clause. See L.9.3.1 for restrictions that apply to scratch files.

12. NOCH=*chksm*

    chksm is any combination of the letters D, F, K, and T. The letter T refers to tape and D, F, and K refer to mass storage. The nocheck clause is used to omit a checksum. When the sort uses one or two mass storage scratch files, D refers to the smaller of the two (one must be at least twice the size of the other) and F refers to the larger of the two, if both files are present.

    If the sort uses three or more mass storage scratch files, K refers to the checksum on all the files.

    If K is specified for a sort with fewer than three mass storage files, D and F are assumed. If D or F is specified for a sort with more than two mass storage scratch files, K is assumed. This clause can appear only once in *infost.* See L.9.3.2.

13. MESH=*meshsz*/*device*

meshsz is the mesh size and *device* is any combination of the letters D, F, K, and T. If *meshsz* is not given, the value 5 is assumed. If *device* is not present, the mesh size is assumed to apply to all device types. The letter T indicates the use of tape scratch files; D, F, and K indicate the use of mass storage scratch files. D and F are used if one or two mass storage scratch files are used. D refers to the smaller of the two mass storage scratch files and F refers to the larger of these two files. The letter K indicates the checksum of three or more mass storage scratch files. The MESH= specification can appear only once in *infost.* The letters D, F, K, and T can be used only once each in the MESH specification. See L.9.3.2.

14. BIAS=*biasno*

    biasno is the average number of records in sorted subsequences present in the input file. For example, *biasno* is 1 if the input is in exactly reverse order. For random data, *biasno* is 2. If the input file is in an almost sorted order, the bias value is higher. The BIAS clause can appear only once in *infost.* Giving the bias value, if known, improves the performance of the sort substantially. See L.9.1.

An example of an information string *infost* to sort variable-length records with a maximum length of 200 characters with four key fields is:

```
'VRSZ=200,KEY=(1/10//s,11/10/d/q,21/10,31/10/d),CONS!'
```

The key fields in the record are defined as:

1. The first key starts in the first character position of the record, is 10 characters long, and is sorted in ascending order with a user-specified collating sequence, if that sequence is present.
2. The second key starts in character position 11, is 10 characters long, and is sorted in descending order with a sign overpunched on the last digit.
3. The third key starts in character position 21, has a length of 10 characters, and is sorted in ascending order.
4. The fourth key starts in character position 31, has a length of 10 characters, and is sorted in descending order.

The CONS clause is present so that all messages are sent from the sort/merge package to the console.

To sort record images of 80 characters with a key that starts in character position 1, that has a length of 5 characters, and that is sorted in ascending order, the information string *infost* can be:

```
'RSZ=80,KEY=1/5!'
```

infost must be the first parameter in the call to FSORT. The other parameters follow *infost.*

inpt

is either a logical unit number or the name of an input subroutine. If *inpt* is the name of an input subroutine, the subroutine must be declared in an EXTERNAL statement in the program unit containing the call to FSORT. See L.8.1.

outpt

is either a logical unit number or the name of an output subroutine. If *outpt* is the name of an output subroutine, the subroutine must be declared in an EXTERNAL statement in the program unit containing the call to FSORT. See L.8.4.

comprt

is the name of a comparison subroutine supplied by you. The subroutine is called whenever two records are to be compared. The name of the comparison subroutine must be declared in an EXTERNAL statement. This parameter must not be present if you have not provided a comparison subroutine. This parameter must be present if the COMP clause occurs in *infost*. See L.8.2.

datred

is the name of a data reduction subroutine. The name of *datred* must be declared in an EXTERNAL statement. This subroutine is called whenever two records with equal keys are found. It decides whether the two records are merged into one record or are not merged. This feature sorts fixed length records only. *Datred* must not be present if you have not specified the DATA clause in *infost*; it must be present if the DATA clause occurs in *infost*. See L.8.3.

At compile time, the ASCII FORTRAN compiler prints out a warning each time FSORT is called from the same program unit with a different number of arguments than was specified on the first call in the program unit. You can ignore these warnings.

L.4.1. Examples of Sort with Logical Unit Numbers

The following runstream contains a call to FSORT with a simple information string *infost* that contains a KEY clause and an RSZ clause. The RSZ clause gives a record size of 80 characters. The KEY clause states that the key begins in the first character position of the record, has a length of five characters, and is sorted in ascending order. *infost* ends with an exclamation point (!). The input and output parameters are simply unit numbers 5 and 6. The sort/merge interface does formatted reads on unit 5 until all the input data is read. The interface then calls the sort/merge package to sort the data, and does formatted writes on unit 6 of the data from the sort/merge package.

```
@RUN
@FTN,SI

      CALL FSORT('key=1/5,RSZ=80!',5,6)
      END

@XQT
. . . data images to be sorted . . .
@FIN
```

Note: *TheRSZ clause should not be greater than 1,024 characters when you use FSORT with logical unit numbers. If the RSZ clause is greater than 1,024 characters, the output file is larger than the input file. Utilize user-supplied input and output routines to handle larger record sizes so that the input and output files can be the same size.*

Another example of a simple sort appears in the following runstream. This program assumes that the source input from file IN*PUT is written to the file OUT*PUT. The records are 80 characters long with keys starting in character positions 1 and 6. Each key is five characters long. The first key is sorted in ascending order, and the second key in descending order.

```
@RUN
@FTN,SI

     CALL FSORT('rsz=80,key=(1/5,6/5/d)!',9,10)
     END

@ASG,A  IN*PUT
@ASG,C  OUT*PUT
@USE  9,IN*PUT
@USE  10,OUT*PUT
@XQT
@FIN
```

L.4.2. Examples of Sort with User Subroutines

The following example is a simple variation of the first example in L.4.1. The RSZ clause declares the record size to be 80 characters. The KEY clause indicates that the key starts in the first character position of the record, has a length of five characters, and is sorted in ascending order. The *inpt* and *outpt* parameters are user-supplied input and output subroutines that are declared in an EXTERNAL statement in the program. The subroutines contain formatted I/O statements to read from unit 5 and write to unit 6.

```
@RUN
@FTN,SI

   EXTERNAL IN,OUT
   CALL FSORT('key=1/5,rsz=80!',IN,OUT)
   END

@FTN,SI IN

   SUBROUTINE IN(RECORD,LENGTH,IEOF)
   CHARACTER*4 RECORD(20)
   READ(5,1,END=2) RECORD
   LENGTH=80
   IEOF=0
   RETURN
1  FORMAT(20A4)
2  IEOF=1
   RETURN
   END

@FTN,SI OUT

   SUBROUTINE OUT(RECORD,LENGTH)
   CHARACTER*4 RECORD(20)
   IF (LENGTH.GE.0) WRITE(6,1) RECORD
1  FORMAT(1X,20A4)
   RETURN
   END

@XQT
. . . data images to be sorted . . .
@FIN
```

Another example of a sort with user I/O routines appears in the following runstream:

```
@RUN
@FTN,SI  EXTERNAL IN,OUT
         CALL FSORT('rsz=80,key=(1/5,6/5/d),core=20000!',IN,OUT)
         END

@FTN,SI    IN

         SUBROUTINE IN(RECORD,LENGTH,IEOF)
         CHARACTER*4 RECORD(20)
         READ(9,1,END=2) RECORD
1        FORMAT(20A4)
         LENGTH=80
         IEOF=0
         RETURN
2        IEOF=1
         RETURN
         END

@FTN,SI  OUT

         SUBROUTINE OUT(RECORD,LENGTH)
         CHARACTER*4 RECORD(20)
         IF (LENGTH.GT.0) THEN
           WRITE(10,1) RECORD
1          FORMAT(20A4)
           RETURN
         END IF
         ENDFILE 10
         RETURN
         END

@ASG,A  IN*PUT
@ASG,C  OUT*PUT
@USE  9,IN*PUT
@USE  10,OUT*PUT
@XQT
@FIN
```

L.5. CALL Statement for a Merge (FMERGE)

The form of the CALL statement for a merge is:

```
CALL FMERGE (infost, inpts, outpt [,comprt] )
```

where:

infost

is the information string, a character variable or literal that describes various parameters to the sort/merge interface, such as record sizes, key fields, and scratch facilities for FMERGE. A key field (or your comparison routine) must be specified in *infost.*

infost contains items of information separated by commas. Blanks in *infost* are ignored. No distinction is made between uppercase and lowercase alphabetic characters. *infost* is scanned from left to right. *infost* must be terminated by some character that is an illegal ASCII FORTRAN character, such as an exclamation point (!), but don't use an asterisk (*).

infost can contain several clauses. The mnemonics are truncated by the sort/merge interface to the first four characters. The following items can be used in *infost* for the call to FMERGE:

1. RSZ=*rlch*

   rlch is the record length in ASCII characters. This clause is not required. If the RSZ clause is not given, the sort/merge interface assumes that the maximum record length is 1,000 words. This wastes some main storage. If the RSZ clause is present, the interface checks that the records from each input source are in sequence. The RSZ clause can appear only once in *infost.* Only one record size clause can appear in *infost* at a time. Thus, the VRSZ clause can't be used if the RSZ clause is used in *infost.*

2. VRSZ=*mrlch*/*lnkszch*

   mrlch is the maximum record size in ASCII characters for variable-length records. *lnkszch* is an optional parameter indicating link size in ASCII characters. If *lnkszch* is omitted, the slash (/) is also omitted. *lnkszch* must be large enough to accommodate all keys. For example, if the keys are specified by:

   ```
   KEY=(11/15,1/10/d/s)
   ```

   the last character in any key field for this KEY specification is the 25th character. Therefore, the link size must be at least 25 characters long. Specify the link size only when a comparison routine is specified. The sort/merge checks to see if the keys fit in the link size but otherwise ignores the link size for a merge. The VRSZ clause can appear only once in *infost* and only one record size clause can be specified in *infost* at a time. Thus, the RSZ clause can't be used if the VRSZ clause appears in *infost.*

3. KEY=*keysp*

   or:

   KEY=*keyspn*

   keysp is a single key specification and *keyspn* is a multiple key specification of the form (*keysp*$_1$, *keysp*$_2$, . . .). The single key specification form KEY=*keysp* can occur a maximum of 40 times in *infost.* There can be a maximum of 40 *keysp* specifications in *keyspn.* More than one KEY=*keyspn* clause can occur in *infost*, but only 40 keys are allowed for each call to FMERGE, including any keys copied through the COPY clause (see L.7). The key specification can indicate a character key, that is, a key that begins and ends on a character boundary, or a bit key that either starts or ends outside a character boundary. The form of the character key is:

   ```
   charpos/length/seq/type
   ```

   where each field is the same as for FSORT. See L.4.

   The form of the bit key is:

   ```
   BIT/wordpos/bitpos/length/seq/type
   ```

   where each field is the same as for FSORT. See L.4.

4. COMP

COMP indicates the use of your comparison routine. This requires the presence of *comprt* in the call to FMERGE. The COMP clause can appear only once in *infost*. See L.8.2.

5. COPY

   COPY indicates that an Assembler sort parameter table is to be copied. This clause can appear only once in *infost*. See L.6.

6. INPU=*inptsor*

   inptsor is an integer constant from 2 through 24 that indicates how many input sources are given in the parameter *inpts*. The INPU clause must appear only once in *infost*.

   An example of an information string *infost* that merges two files containing variable length records with a maximum length of 200 characters with four key fields is:

```
'VRSZ=200,KEY=(1/10//s,11/10/d/q,21/10,31/10/d), INPUT=2!'
```

   The key fields in the record are defined as:

   - The first key starts in the first character position of the record, is 10 characters long, and is sorted in ascending order with a user-specified collating sequence if that sequence is present.
   - The second key starts in character position 11, is 10 characters long, and is sorted in descending order with an overpunched sign in the last digit.
   - The third key starts in character position 21, has a length of 10 characters, and is sorted in ascending order.
   - The fourth key starts in character position 31, has a length of 10 characters, and is sorted in descending order.

   To merge three files that:

   - contain records of 80 characters,
   - have a key starting in character position 1,
   - have a length of five characters, and
   - are sorted in ascending order,

   the information string *infost* may be:

```
'RSZ=80, INPUT=3, KEY=1/5!'
```

   infost must be the first parameter in the call to FMERGE. The other parameters follow *infost*.

inpts

is two or more logical unit numbers, the names of two or more input subroutines, or a combination of logical unit numbers and input subroutines. The parameter *inpts* can contain from 2 through 24 input sources. When *inpts* contains the names of input subroutines, the subroutine names must be declared in an EXTERNAL statement in the program unit containing the call to FMERGE. See L.8.1.

outpt

is either a logical unit number or the name of an output subroutine. If *outpt* is the name of an output subroutine, the subroutine must be declared in an EXTERNAL statement in the program unit containing the call to FMERGE. See L.8.4.

comprt

is the name of a comparison subroutine that you supply. The subroutine is called when two records are to be compared. The name of the comparison subroutine must be declared in an EXTERNAL statement. This parameter must not be present if you don't provide a comparison subroutine. This parameter must be present if you use the COMP clause in *infost.* See L.8.2.

At compile time, the ASCII FORTRAN compiler prints out a warning each time FMERGE is called from the same program unit with a different number of arguments than was specified on the first call in the program unit. You can ignore these warnings.

L.5.1. Examples of CALL Statements to Merge

The following runstream contains a call to FMERGE with a simple information string *infost* that contains a KEY clause and an RSZ clause. The RSZ clause declares a record size of 80 characters. The KEY clause indicates that the key starts in the first character position of the record, has a length of five characters, and is sorted in ascending order. *infost* contains the clause INPUT=2 to indicate that there are two input sources contained in the input parameter *inpts.* The *inpts* parameters are the logical unit number 8 and the user-supplied input subroutine name IN. The output parameter *outpt* is the logical unit number 9. *infost* ends with an exclamation point (!).

```
@RUN
@FTN,SI      MAIN

     EXTERNAL IN
     CALL FMERGE('rsz=80,key=1/5,input=2!',8,IN,9)
     END

@FTN,SI IN

     SUBROUTINE IN(RECORD,LENGTH,IEOF)
     CHARACTER*4 RECORD(20)
     READ(5,1,END=2) RECORD
     LENGTH=80
     IEOF=0
     RETURN
1    FORMAT(20A4)
2    IEOF=1
     RETURN
     END

@ASG,A IN
@ASG,C OUT
@USE 8,IN
@USE 9,OUT
@XQT
. . . the secofd input file on data images . . .
@FIN
```

The example that follows merges two input image files (IN*1 and IN*2) that were sorted in ascending order on columns 1 through 10 and the merged data is written to file OUT*PUT. The input subroutine ignores all records in the input file IN*2 that have a 1 in column 11.

```
@RUN
@FTN,SI

    EXTERNAL IN
    CALL FMERGE('rsz=80,key=1/10,input=2!',8,IN,10)
    END

@FTN,SI   IN

    SUBROUTINE IN(RECORD,LENGTH,IEOF)
    CHARACTER RECORD*80,I,ONE/ '1'/
1   FORMAT(A)
2   FORMAT(10X,A1)
3   READ(9,1,END=4) RECORD
    DECODE(2,RECORD) I
    IF (I.EQ.ONE) GO TO 3
    LENGTH=80
    IEOF=0
    RETURN
4   IEOF=1
    END

@ASG,A IN*1
@ASG,A IN*2
@ASG,C OUT*PUT
@USE 8,IN*1
@USE 9,IN*2
@USE 10,OUT*PUT
@XQT
@FIN
```

L.6. CALL Statement to Copy an External Sort Parameter Table (FSCOPY)

This facility provides access to the Assembler procedure R$FILE.

The form of the CALL statement to copy an external Assembler sort parameter table is:

```
CALL FSCOPY[([table])]
```

where *table* is the name of an externalized entry point. *table* must be declared in an EXTERNAL statement. The CALL statement to FSCOPY with one external argument must occur before a call to FSORT or FMERGE with the COPY clause in its information string *infost*. (See L.4 and L.5.) The call of FSCOPY establishes which sort parameter table is copied. The subroutine argument is the first word of the sort parameter table to be copied. The subroutine argument is not an ASCII FORTRAN subroutine but is an Assembler entry point.

Only one sort parameter table can be copied at one time. Each call on FSCOPY deletes the previous table that was copied. If FSCOPY is called without any arguments, a new table is not copied and the previous table is deleted.

The optional *field-number* field in a KEY parameter table entry must not be present.

L.6.1. Record Size When FSCOPY Is Used

Key positions, record lengths, and link sizes in Assembler sort parameters must be given as if there were an extra word in front of the record. (See L.4 and L.5.)

L.6.2. Example of CALL Statement to FSCOPY

The following runstream contains an Assembler sort parameter table and program that calls FSORT and FSCOPY. The program sorts the source input from character positions 1 through 6 in ascending order, from character positions 7 through 12 in descending order, and from character positions 13 through 16 (Fieldata character positions 19 through 24) in ascending Fieldata order, with a special collating sequence such that all Bs precede all As. The information for character positions 13 through 16 comes from the Assembler sort parameter table. The source input is on logical unit 5 and the output is placed on logical unit number 6. Note the extra word or six characters in the starting character position (19+6). This is described in L.6.1.

If SORT/MERGE has been installed in a file other than SYS$*RLIB$, the following example will need the $INCLUDE statement for the MASM processor to find the R$FILE proc. In the example below, assume that SORT/MERGE has been installed in the file SYS$LIB$*SORT. If SORT/MERGE has been installed in SYS$*RLIB$, no $INCLUDE statement is necessary. See the *OS 2200 Sort/Merge Programming Guide, Level 17R1* (7831 0687) for more information on R$FILE.

```
@RUN
@MASM,SI   COPIED
           $INCLUDE   'SYS$LIB$*SORT.RPROC$/'

COPIED*    R$FILE     'KEY',19+6,6,'A','A' ; Extra word!
                      'SEQ','@','UPTO',' ',
                      'B','A','ALL'
           END

@FTN,SI
           INTEGER CORE(21000)
           EXTERNAL COPIED
           CALL FSCOPY(COPIED)
           CALL FSORT('key=(1/6,7/6/d),copy,rsz=80,core=20000!',
         1   5,6,CORE)
           END

@XQT
. . . data images . . .
@FIN
```

L.7. CALL Statement to Provide a User-Specified Collating Sequence (FSSEQ)

A user-specified collating sequence can't be explicitly declared in the information string *infost* of a call to FSORT or FMERGE. The use of a nonstandard collating sequence is specified only in the KEY clause field *type* in a character key with the value S. (See L.4

and L.5.) The user-specified collating sequence is set up through a call to FSSEQ with a single argument.

The form of a CALL statement to FSSEQ is:

```
CALL FSSEQ [ ([seqtbl]) ]
```

where *seqtbl* is an argument containing a character string that is 256 characters long. *Seqtbl* contains the user-defined collating sequence of the ISO character set. If *seqtbl* is not present, the previous user-defined collating sequence is deleted. Only one user-defined ISO collating sequence is in use at any one time. A second CALL statement to FSSEQ causes the previous collating sequence to be replaced with the new user-defined collating sequence.

L.7.1. Example of CALL Statement to FSSEQ

The following runstream contains two calls to FSORT and two calls to FSSEQ. The first call to FSSEQ contains a user-defined collating sequence in array SEQTAB. The collating sequence is the same as the standard ISO collating sequence except that the letters A and B (uppercase and lowercase) are interchanged. The first call to FSORT uses the user-defined collating sequence. The input is read from unit 5. The input is sorted according to:

1. The first key that starts in character position 1 of the record, has a length of 10 characters, and is sorted in ascending order.
2. The second key that starts in character position 11 of the record, has a length of five characters, and is sorted in descending order according to the user-defined collating sequence specified in the call to FSSEQ.
3. The third key that starts in character position 16 of the record, has a length of five characters, and is sorted in ascending order.

The result is written by the user output routine OUT.

The second call to FSSEQ deletes the previous user-defined collating sequence and does not set up another sequence. This means that the normal ISO collating sequence is used when sorting. Note the use of the CORE= clause in *infost* in both calls to FSORT.

```
@RUN
@FTN,SI

      EXTERNAL OUT
      INTEGER SEQTAB(64),POSN
C           set up collating sequence.
      POSN(I)=BITS(SEQTAB(1+I/4),1+9*MOD(I,4),9)

      DO 1 I=0,255
1        POSN(I)=I
      POSN(65)=POSN(65)+1
      POSN(66)=POSN(66)-1
      POSN(97)=POSN(97)+1
      POSN(98)=POSN(98)-1
C          give collating sequence to sort.
      CALL FSSEQ(SEQTAB)
C          do the first sort
```

```
    CALL FSORT('key=(1/10,11/5/d/s,16/5),rsz=80,
   1     core=20000!',5,OUT)
C        remove collating sequence.
    CALL FSSEQ
C        do the second sort.
    CALL FSORT('key=(1/10,11/5/d/s,16/5),rsz=80,
   1    core=20000!',10,6)
    END

@FTN,SI  OUT

    SUBROUTINE OUT(RECORD,LENGTH)
    CHARACTER*80 RECORD
    IF (LENGTH.LE.0) RETURN
    WRITE(10,1) RECORD
    PRINT 2,RECORD
    RETURN
1   FORMAT(A)
2   FORMAT(1X,A)
    END

@ASG,T TEMP
@USE 10,TEMP
@XQT
. . . data images . . .
@FIN
```

L.8. User-Specified Subroutines

The sort/merge interface lets you provide subroutines to:

- Read records
- Compare records
- Examine fixed-length records with equal keys and optionally merge the records
- Write records

You are not required to supply any of these subroutines. The sort/merge interface and package handles all these areas when you don't want to supply any subroutines.

L.8.1. User-Specified Input Subroutine

An input subroutine can be supplied to be called by the sort/merge package to read the records. (See L.4 and L.5.) The input subroutine is called with three arguments. The first argument is an array that contains the input record to be returned to the sort/merge package. The second argument is an integer that contains the length of the input record in characters. The third argument is an integer that indicates when the last record is delivered.

The input subroutine can do the following:

1. Read a record and return the record to the sort.
2. Return the null string with a record length of zero and the third argument set to a one to indicate the end of the input file.

3. Read a record and call FSGIVE with that record as an argument. The input subroutine can enter FSGIVE several times before returning an input record or an end-of-file mark to the sort/merge package.

The end of the file can be signaled through FSGIVE. Control is not returned to the instruction following the call to FSGIVE in the input subroutine. Control returns to the sort/merge package.

L.8.1.1. Example of a User-Specified Input Subroutine

The following input subroutine reads the source input from unit 5 and returns a record length of 80 characters. The third argument is set to 0 if the end of the file is not reached and set to 1 if the end of the file is reached in the input file.

```
      SUBROUTINE IN(RECORD,LENGTH,IEOF)
      CHARACTER*4 RECORD(20)
      READ(5,1,END=2) RECORD
      LENGTH=80
      IEOF=0
      RETURN
 1    FORMAT(20A4)
 2    IEOF=1
      RETURN
      END
```

L.8.1.2. CALL Statement to FSGIVE

The call to FSGIVE provides the capability of giving a record to the sort without leaving the user-specified input subroutine. Call FSGIVE with three arguments:

1. the input record for sort,
2. the length of the record given to the sort, and
3. the flag given to sort to indicate that the end of the file is reached.

When the flag is 0, the end of the file was not reached, while a nonzero flag indicates that the end of the file is found.

These arguments are similar to the arguments for the input subroutine.

L.8.1.3. Example of User-Specified Input Subroutine with FSGIVE

The following input subroutine reads characters separated into words by blanks or by the end of the line from input unit 5. Any character except a space can be part of a word.

The input subroutine reads from unit 5 when first entered. The subroutine moves each word that it finds to the record area and then calls FSGIVE for each word that is not the last word on a data image. The input subroutine IN is reentered each time a new data image is needed from the input file. The end of the input file is signaled by the input subroutine IN. The end of the input file can also be indicated by setting the third argument to FSGIVE to a nonzero value. The calls to FSGIVE are intermixed with calls to the input subroutine IN.

```
     SUBROUTINE IN(RECORD,LENGTH,IEOF)
     CHARACTER RECORD*80,CARD(80),BLANK/ ' '/
1    READ(5,2,END=99) CARD
2    FORMAT(80A1)
C             find last nonblank character
     DO 3 IMAX = 80,1,-1
3       IF (CARD(IMAX) .NE. BLANK) GO TO 4
C            The input record was blank, so read a new record.
     GO TO 1
4    I = 1
C            Find first blank separator.
5    DO 6 J = I,IMAX
6       IF (CARD(J) .EQ. BLANK) GO TO 8
C            Record has no more blanks - deliver.
     ENCODE(80,2,RECORD) (CARD(J),J = I,IMAX)
     LENGTH = 80
     IEOF = 0
     RETURN
C            At least one blank was found.
8    IF (J .EQ. I) THEN
C            It was a leading blank - ignore it.
         I = I + 1
     ELSE
C            A word was found - deliver.
         ENCODE(80,2,RECORD) (CARD(K),K = I,J - 1)
         CALL FSGIVE(RECORD,80,0)
         I = J + 1
     END IF
     GO TO 5
C           This is end of input - tell the sort.
99   IEOF = 1
     RETURN
     END
```

L.8.2. User Comparison Routine

If a comparison routine is present, it is called whenever the sort/merge package must compare two records. The COMP clause must be present in the information string of the call to FSORT or FMERGE. The parameter *comprt* must also be specified in the call to FSORT or FMERGE. (See L.4 and L.5.)

The compare subroutine is called with three arguments. The first two arguments are the two records compared when the records are fixed-length records or the first links of the two records compared when the records are variable length. The third argument is an integer whose value informs the sort of the result of the comparison done by the comparison subroutine. The result can be:

- The value 1 if the first record precedes the second record
- The value 2 if the order of the records is immaterial
- The value 3 if the second record precedes the first record

Care is necessary when using a comparison subroutine together with keys specified in the information string because the sort/merge package translates the key fields according to certain rules. The *OS 2200 Sort/Merge Programming Guide, Level 17R1* (7831 0687), contains a description of the translation rules.

L.8.2.1. Example of a User Comparison Subroutine

For the following example, assume the first five characters of each record contain a signed, nonzero number between -49999 and 49999. A negative number X is represented by 50000-X. If key translation is not used, the following comparison subroutine can be used:

```
      SUBROUTINE COMP(FIRST,SECOND,CODE)
      INTEGER FIRST,SECOND,CODE,F,S
      DECODE(1,FIRST) F
      DECODE(1,SECOND) S
  1   FORMAT(I5)
      IF (F.GE.50000) F=50000-F
      IF (S.GE.50000) S=50000-S
      IF (F-S) 2,3,4
  2   CODE=1
      RETURN
  3   CODE=2
      RETURN
  4   CODE=3
      RETURN
      END
```

L.8.2.2. Example of a Runstream with a Comparison Subroutine

In the following example of a comparison subroutine, the first two characters of the records given to the comparison subroutine contain an integer that indicates the starting position of the key within the record. The key is five characters long and contains a right-justified integer value. A complete runstream for using this comparison subroutine is:

```
@RUN
@FTN,SI

      EXTERNAL COMP
      CALL FSORT('comp,rsz=80,core=20000!',9,10,COMP)
      END

@FTN,SI COMP

      SUBROUTINE COMP(R1,R2,CODE)
      INTEGER CODE,P1,P2
      CHARACTER R1*80,R2*80,F1*8,F2*8
C               Compute the key values.
      DECODE(4,R1) P1
      DECODE(4,R2) P2
      ENCODE(8,5,F1) P1-1
      ENCODE(8,5,F2) P2-1
      DECODE(F1,R1) P1
      DECODE(F2,R2) P2
C               Do the comparisons.
      IF (P1-P2) 1,2,3
  1   CODE=1
      RETURN
  2   CODE=2
      RETURN
```

```
3   CODE=3
    RETURN
4   FORMAT(I2)
5   FORMAT('(',I2,'X,I5)')
    END

@ASG,A  IN*PUT
@ASG,C  OUT*PUT
@USE    9,IN*PUT
@USE    10,OUT*PUT
@XQT
@FIN
```

L.8.3. User Data Reduction Subroutine

The data reduction subroutine is called by the sort/merge package whenever the sort/merge package finds two records whose order is immaterial. The data reduction subroutine may or may not merge the two records into the first record. The data reduction subroutine can be specified only when sorting fixed-length records. The DATA clause must be specified in the information string in the call to FSORT. The *datred* parameter must be present in the call to FSORT. Two restrictions must be remembered:

1. The records, if merged, must always be merged into the first record (that is, the first argument).
2. The data reduction routine can't change key fields.

The data reduction subroutine is called with three arguments. The first two arguments are the two records. The third argument is an integer result assigned by the data reduction subroutine with the following possible values:

- The value 1 indicates that the two records are merged.
- The value 2 indicates that the two records are not merged.

The sort/merge subroutines translate the key fields according to certain rules. Exercise care when using a data reduction subroutine together with keys specified in the information string in the call to FSORT. The translation rules are described in the *OS 2200 Sort/Merge Programming Guide*, *Level 17R1* (7831 0687).

L.8.3.1. Simple Example of a Data Reduction Subroutine

The following data reduction subroutine assumes that any input record that contains a 1 in character position 6 is chosen over any other record. If both records contain a 1 in character position 6, the first record is chosen over the second record.

```
    SUBROUTINE DATA(FIRST,SECOND,CODE)
    INTEGER CODE
    CHARACTER FIRST*80,SECOND*80,TEST,ONE/'1'/
    DECODE(1,FIRST) TEST
1   FORMAT(5X,A1)
    IF (TEST.EQ.ONE) THEN
       CODE=1
       RETURN
    END IF
```

```
      DECODE(1,SECOND) TEST
      IF (TEST.NE.ONE) THEN
         CODE=2
         RETURN
      END IF
      FIRST=SECOND
      CODE=1
      END
```

L.8.3.2. Example of a Runstream with a Data Reduction Subroutine

This example with a data reduction subroutine assumes that two records with equal keys are merged if character position 11 of at least one of the records is blank. The record with the blank in character position 11 is retained. If both records have character position 11 blank, the first record is retained.

This runstream chooses the decision field outside the key fields to avoid any problems with key field translation.

```
@RUN
@FTN,SI

      EXTERNAL DATA
      CALL FSORT('key=(1/5,6/5/d),rsz=80,data reduction
     1      user code,core=20000!',9,10,DATA)
      END

@FTN,SI DATA

      SUBROUTINE DATA(R1,R2,CODE)
      INTEGER CODE,BLANK/' '/
      CHARACTER*80 R1,R2
      DECODE(1,R1) IB
1     FORMAT(10X,A1)
      IF (IB.EQ.BLANK) THEN
         CODE=1
         RETURN
      END IF
      DECODE(1,R2) IB
      IF (IB.NE.BLANK) THEN
         CODE=2
         RETURN
      END IF
      R1=R2
      CODE=1
      END

@ASG,A    IN*PUT
@ASG,C    OUT*PUT
@USE      9,IN*PUT
@USE      10,OUT*PUT
@XQT
@FIN
```

L.8.4. User-Specified Output Subroutine

The output subroutine is called by the sort/merge package when a record is written. The output subroutine is called with two arguments. The first argument is the record to be written and the second argument is the length in characters of the record to be written.

The second argument is also a flag to your output subroutine to indicate when the sort delivers the last record to be written. The length is normally a positive number indicating the size of the record in characters. If the length is negative or zero, the last record has already been delivered to the output subroutine.

L.8.4.1. Simple Example of a User-Specified Output Subroutine

The following output subroutine outputs records to unit 6 through a formatted write:

```
      SUBROUTINE OUT(RECORD,LENGTH)
      CHARACTER*80 RECORD
      IF (LENGTH.GT.0) PRINT 1,RECORD
1     FORMAT(1X,A)
      RETURN
      END
```

L.8.4.2. CALL Statement to FSTAKE

Your output routine can indicate to the sort/merge package when the output routine needs a new output record. This is done by a CALL statement to FSTAKE with two arguments. The first argument is the record received from the sort/merge package. The second argument is the length in characters of the new record. If the length argument is negative after returning from FSTAKE, the last record has already been delivered from the sort/merge package.

L.8.4.3. Example of FSTAKE in an Output Subroutine

The following example moves records containing one word each into card images with exactly one space between the words, and then writes the record when the card image becomes full. The sorted records are assumed to contain 80 characters.

```
      SUBROUTINE OUT(RECORD,LENGTH)
      INTEGER POS/1/
      CHARACTER CARD(80),CR(80),BLANK/ ' '/,RECORD*80
      IF (LENGTH.LT.0) GO TO 99
C                Blank the output record.
      DO 1 I=1,80
1          CARD(I)=BLANK
C                Place each character of the record in a word.
2     DECODE(80,3,RECORD) CR
3     FORMAT(80A1)
C                Find actual length of record.
      DO 4 IL=80,1,-1
4          IF (CR(IL).NE.BLANK) GO TO 5
C                This is a blank record--ignore the record.
      GO TO 10
5     IF (POS+IL.GT.81) THEN
C                The card image to print is full--print it.
           PRINT 6,CARD
6          FORMAT(1X,80A1)
           DO 7 I=1,80
7                CARD(I)=BLANK
           POS=1
      END IF
      DO 9 I=1,IL
9          CARD(POS-1+I)=CR(I)
```

```
      POS=POS+IL+1
C               Get next record to get next word.
10    CALL FSTAKE(RECORD,LENGTH)
      IF (LENGTH.GE.0) GO TO 2
99    IF (POS.GT.1) PRINT 6,CARD
      RETURN
      END
```

L.9. Optimizing Sorts

An understanding of this subsection is not required to do a sort. This information is provided for those who need to sort larger data sets than the standard scratch assignments (main storage and mass storage) allow. This information also helps those who need to minimize the resources used in a sort. You also need to use this information if the sort/merge package error B5 is given for a sort (see the *OS 2200 Sort/Merge Programming Guide*, *Level 17R1*, 7831 0687).

The standard scratch file assignments are:

- 19,000 words of main storage
- Six disk files of 512 tracks each (initial reserve 0)

This amount of storage should be sufficient to sort some 200,000 to 250,000 80-character records. If a very large sort (that is, a multiple cycle sort requiring operator intervention) is necessary, you should consult the *OS 2200 Sort/Merge Programming Guide*, *Level 17R1* (7831 0687). The performance of a sort is mainly determined by the following three factors:

1. The bias of the input data
2. The size of the main storage scratch area
3. The scratch files used

The CPU time used by the sort is decreased slightly by inhibiting the checksum on the sort's scratch files or by increasing the size of the checksum mesh.

L.9.1. Bias of the Input Data

The bias is the average number of records in sorted subsequences present in the input file. (See BIAS=*biasno* in L.4.) For example, if the input file is exactly in reverse order, the bias is 1. Also, random data has a bias of 2. Generally, the bias is greater if the input file is almost sorted; that is, the more nearly sorted the input file, the greater the bias.

If a bias is specified, the sort is able to use available resources optimally so that more data can be sorted using the same amount of scratch storage. You should specify the bias whenever it is known and when the bias is less than 1.4 or greater than 3.

The bias is specified by the form:

```
BIAS=biasno
```

L.9.2. Size of the Main Storage Scratch Area

The size of the main storage scratch area is specified two ways:

1. Assign the file R$CORE with a suitable maximum granule value.
2. Specify the size of the main storage scratch area in the information string for the sort or merge.

If the size of the main storage scratch area is given by both methods, the R$CORE value overrides the size of the main storage scratch area given in the information string. (See L.4.)

L.9.2.1. Using R$CORE

The size of the main storage scratch area in words is specified at run time by assigning the file R$CORE with a suitable maximum size. For example, if 20,000 words of main storage scratch area are desired, the following control statement guarantees that 20,000 words of storage are available to the sort/merge interface and package:

```
@ASG,T R$CORE,///20
```

L.9.2.2. Using the CORE Clause in the Information String

The amount of main storage scratch area for the sort is specified by the following CORE clause in the information of the call to FSORT:

```
CORE=corsz
```

where *corsz* is the size of the main storage scratch area in words. The sort/merge interface rejects any size that is less than 3,000 words. The sort generally executes faster when it is given more main storage.

When you use this clause in the information string, don't assign the file R$CORE.

L.9.3. Using Scratch Files and Checksum

Avoid the use of tape scratch files when possible. Tape sorts are slower and require operator intervention. The sort/merge package distinguishes two cases for mass storage files:

1. One or two mass storage files
2. More than two mass storage files

The first case is more suitable when only one or two mass storage units are available to the sort. However, this case requires a careful assignment of main storage and mass storage scratch resources. The optimal amount of main storage will depend on how much mass storage is available to the sort. A suitable assignment of facilities appears in the *OS 2200 Sort/Merge Programming Guide, Level 17R1* (7831 0687). Different mass storage scratch files should be kept on separate mass storage units if possible.

L.9.3.1. Naming Scratch Files in the Information String

Scratch files are specified by the FILE clause in the information string. (See L.4.) The following restrictions apply when the FILE clause is used:

1. All scratch files must be assigned when the sort is started.
2. A maximum of 26 scratch files can be specified.
3. At least three tape scratch files must be used if any tape scratch files are used.
4. If tape scratch files are used, a maximum of two mass storage files is used for the sort.

L.9.3.2. Checksum and the Sort

A checksum is normally done on all tape and mass storage files. You can omit the checksum on one or more device types (mass storage or tape). You can also specify a checksum mesh size for each device type. For example, if a mesh size of 5 is given, only every fifth word of each block written to tape or mass storage is included in the checksum.

You can omit the checksum by specifying the nocheck clause (NOCH) in the information string for the call to FSORT. You provide a mesh size by specifying the MESH clause in the information string. These clauses are described in the CALL statement to FSORT. (See L.4.)

L.10. Sorting Very Large Amounts of Data

When it is not practical to assign enough scratch storage to hold all of the data to be sorted, a multicycle sort must be done. For that case, the ASCII FORTRAN program must be executed with the P option (@XQT,P) and some sort/merge package parameter data images must be prepared. These data images are fully described in the *OS 2200 Sort/Merge Programming Guide, Level 17R1* (7831 0687).

The SMRG parameter data image format is:

```
'SMRG','outptprefx', nbrrecds, nbrreel
```

where:

outptprefx

is a string two characters long that identifies the intermediate output tapes.

nbrrecds

specifies the number of records sorted in each cycle. It is optional.

nbrreel

specifies the number of reels to be produced in each cycle and is optional. If the number given in *nbrrecds* specifies more records than the assigned hardware can hold, *nbrrecds* is ignored.

The parameter data images are read after the call to FSORT but before the first input record is read or before your input routine is first called.

You must use the following control image after the last sort/merge parameter data image:

```
@EOF A
```

If you wish to rerun interrupted multicycle sorts, refer to the *OS 2200 Sort/Merge Programming Guide*, *Level 17R1* (7831 0687).

L.10.1. Example of a Large Single-Cycle Sort

The following runstream contains a sort that must run as efficiently as possible. The records are in nearly reverse order (the bias is about 1.2). A checksum is not done. About 400,000 records must be sorted, so the standard scratch assignments cannot be used. Ample amounts of main storage and mass storage are available for the sort.

The first step is to calculate the sort volume. This is the record size times the number of records times a safety factor:

```
20 * 400000 * (1 + .1)
```

which equals about 9 million words.

For a big sort, use six equal-size files. This makes each file about 1.5 million words, or about 850 tracks.

A suitable amount of main storage scratch area is about 50K words.

The scratch files must be assigned before the sort begins. The runstream for the sort can be:

```
@RUN
@FTN,SIO

     CALL FSORT('rsz=80,key=(1/5,6/5/d),core=50000,
   1      bias=1.2,files=(M1,M2),nocheck=dft,
   1      files=(m3,m4,m5,m6)!',9,10)
     END

@ASG,A   IN*PUT
@ASG,T   OUT*PUT,T,REELNO
@ASG,T   M1,///850
@ASG,T   M2,///850
@ASG,T   M3,///850
@ASG,T   M4,///850
@ASG,T   M5,///850
@ASG,T   M6,///850
@USE     9,IN*PUT
```

```
@USE     10,OUT*PUT
@XQT
@FIN
```

L.10.2. Example of a Multicycle Sort

The following runstream is used for a multicycle sort. The information is much the same as the large single-cycle sort except that there are about 20 million records sorted, using the same amount of main storage and mass storage. In addition, four scratch tape files are used.

```
@RUN
@FTN,SIO

     CALL FSORT('rsz=80,key=(1/5,6/5/d),core=50000,
    1      bias=1.2,file=(M1,M2,T1,T2,T3,T4),noch=dft,
    1      files=(m3,m4,m5,m6)!',9,10)
     END

@ASG,A    IN*PUT
@ASG,TV   OUT*PUT,U9V/2,REEL1/REEL2/REEL3
@ASG,T    M1,///1117
@ASG,T    M2,//POS/715
@ASG,T    T1,T
@ASG,T    T2,T
@ASG,T    T3,T
@ASG,T    T4,T
@USE      9,IN*PUT
@USE      10,OUT*PUT
@XQT,P
'SMRG','EX'
@EOF A
@FIN
```

L.11. Error Messages from a Sort or a Merge

Two different types of error messages can be produced during a sort or a merge. The first type is written to the console and its form is:

```
XXXX ERROR CODE YZ
```

where *XXXX* is SORT or MERGE, *Y* is a letter, and *Z* is a digit. This message is immediately followed by an ER ERR$ exit. This type of message is produced by the sort/merge subroutines and is described in the *Series 1100 Sort/Merge*, *Programmer Reference*, UP-7621.

The second type of message is produced by the sort/merge interface with the form:

```
FTN SORT/MERGE ERROR CODE NN strg
```

where *NN* is a 2-digit error code. The error codes and an explanation for each follow. *Strg* is a four-character string that provides further information about the error.

01

The mnemonic in the information string whose first four characters are given in *strg* is not known to the routine called (for example, SELE is not allowed for merges and UNKNOWN is not allowed for sorts or for merges).

02

The routine specified in *strg* is called with the wrong number of arguments.

03

The character position of the most significant character of a key is negative or too large.

04

A key length is negative or too large.

05

An erroneous key type (such as A for a character key) is specified.

06

The sorting sequence is not A, D, or a null string.

07

The word position of the most significant bit of a bit key is negative or too large.

08

The bit position of the most significant bit of a bit key is incorrect (0 or greater than 36).

09

The translation table in FSSEQ does not contain the full ISO set. *Strg* contains the octal code for the first character found that cannot be translated.

10

The maximum record size (RSZ) given in the information string is negative or greater than 65K.

11

No record size (RSZ) is specified in the information string for a sort.

12

An impossible link size (0 or greater than the maximum record size) is specified.

13

No keys and no user comparison routine are given in the information string.

14

An error exists in the collating sequence (FSSEQ). The given string is less than 256 characters.

15

The auxiliary main storage area is full. For remedial action, please submit a Software User Report (SUR).

16

An overflow occurs in the sort parameter table. For remedial action, please submit a Software User Report (SUR).

17

At least one key extends beyond the record.

18

A bit key of type A, D, G, L, or P does not start on a 6-bit byte boundary.

19

The length in bits of a bit key of type A, D, G, L, or P is not divisible by 6.

20

An erroneous return code is given on exit from your comparison routine.

21

An erroneous record length is given on exit from your input routine.

22

The link size is not specified for variable length records and no key specifications are given.

23

A forbidden character was found in the information string.

24

The output routine/file or your comparison routine is not in the argument list.

25

The COPY specification is given in the information string, but FSCOPY is not called (or the most recent call has no arguments).

26

For a sort, an input file/routine is not in the argument list. For a merge, either too few (less than two) or too many (more than 26) input files/routines are given in the argument list.

27

The bias is given as less than 1.

28

The mnemonic whose first four characters are in *strg* appears more than once in the information string. If *strg* is RSZ, VRSZ may have appeared before (and vice versa).

29

The size of the main storage scratch area is given as less than 3,000 words or greater than 262,141 words.

30

An illegal character is given in the NOCH specification (only D, F, K, and T are accepted to the right of the equals sign) in the information string.

31

Your data reduction routine is not in the argument list.

32

A given scratch file is not on mass storage. The most common reason is that the file is not assigned to the run.

34

More than 26 scratch files are specified in the information string.

35

The first member of a select pair (SELE clause) is not greater than the previous pair's second member.

36

The second member of a select pair is less than the first member.

37

An erroneous return code is given on exit from your data reduction routine.

38

A facility reject status is generated in the attempt to assign one of the sort scratch files. *Strg* specifies which of the six standard files can't be assigned. The next line gives the FAC REJECT code.

39

Some keys overlap. *Strg* gives the number of the major key of the pair that overlaps (the most major key is number 1, the next number 2, etc).

40

An illegal sign appears in an arithmetic field. *Strg* gives the first four characters found after (and including) the one in error.

41

An illegal character appears in a numeric field. *Strg* gives the first four significant digits.

42

The field in *strg* is not followed by an equals sign.

43

A numeric field given in *strg* appears when an alphabetic field is expected.

44

Nonblank characters appear between an equals sign and a left parenthesis.

45

The first field of a *keyspn* in a KEY clause is alphabetic and is not BIT.

46

An alphabetic field in *strg* appears when a numeric field is expected.

47

An integer field contains a decimal point.

48

An invalid delimiter in *strg* is found.

49

The name of a scratch file contains more than 12 characters.

50

The first instruction of your routine is illegal. You may also have passed a banked argument in the position indicated by *strg*.

51

Too many parameters in the call to FSORT or FMERGE exist.

52

No main storage scratch parameter argument is given and an OWN clause is in the information string.

53

The data reduction routine is specified for a sort of variable-length records.

55

The first word of the user-provided sort parameter table is wrong.

63

The mesh size is previously specified for a device given in *strg* (*strg* has a value D, F, K, or T).

64

An impossible mesh size is specified.

65

An illegal device type in *strg* is used in a MESH specification.

66

An illegal delimiter is used in a MESH specification.

67

Banked or virtual data arguments are not allowed. *Strg* indicates the argument with the error.

68

Only the first argument to FSORT, FMERGE, FSGIVE, or FSTAKE can be of type CHARACTER.

69

The logical unit number for input or output is a reread unit or outside the defined range of logical units.

70

A bit key of the type A, D, G, L, or P is not allowed when input is from a logical unit number, because such input is always formatted input, and FTN converts Fieldata records to ASCII during formatted input. You must use an input subroutine rather than a logical unit number.

Appendix M
Virtual FORTRAN

M.1. General

When a collected FORTRAN program doesn't fit in the traditional 65,535 words of main storage (or 262,143 words if the FORTRAN programs were compiled with the O option) and collector truncation diagnostics result, consider putting large objects, such as common blocks or local arrays, in virtual space.

Putting a large object in virtual space reduces the main storage requirements of the collected absolute element. For example, a subroutine that processes two REAL argument arrays of extent NxN needs six local arrays of the same size (or larger) as working storage during its processing. Since a local array can't be dimensioned as NxN, a reasonable maximum size must be picked and the subroutine must be compiled with that size. But, the following problems exist:

- When NxN is 150x150, this takes up 22K words of main storage per local array or 132K words total in local storage for this one routine.
- When a typical N in an execution is only 50, most of the storage space just mentioned goes to waste.
- When using an N over 150, the element must be recompiled with a larger size to handle it.
- When using an N over 180, the main storage requirements are too large to fit in the 262,000-word address range limit of the OS 1100 architecture.

But when these six arrays are put in virtual space, the following occurs:

- The collected size drops by about 132K words.
- The local arrays are dynamically allocated in virtual space on subprogram entry, and freed on subprogram exit.
- During execution, main storage requirements are only marginally larger than the collected size.

- Virtual space is exempt from the 262,000-word limit of the current OS 1100 architecture, and you can pick a value for N that is rarely exceeded (thereby forcing recompilation) with minimal extra overhead.

The following example shows the same subroutine written with and without virtual space:

Program before change:

```
SUBROUTINE SUB(N,AR1,AR2)
PARAMETER (M=150)             @ marginal size
REAL L1(M,M),L2(M,M),L3(M,M)
REAL L4(M,M),L5(M,M),L6(M,M)
REAL AR1(N,N),AR2(N,N)
```

Program after change:

```
SUBROUTINE SUB(N,AR1,AR2)
VIRTUAL L1,L2,L3,L4,L5,L6
PARAMETER (M=300)             @ comfortable size
REAL L1(M,M),L2(M,M),L3(M,M)
REAL L4(M,M),L5(M,M),L6(M,M)
REAL AR1(N,N),AR2(N,N)
```

Named common can also be put into virtual space as shown below.

Program before change:

```
PARAMETER (M=100)             @ marginal size
COMMON/C1/A(M,M),B(M,M)
COMMON/C2/D(6000),F(99000),G(M)
COMMON/C3/PIVOT(M)
```

Program after change:

```
VIRTUAL /C1/,/C2/
PARAMETER (M=400)             @ comfortable size
COMMON/C1/A(M,M),B(M,M)
COMMON/C2/D(6000),F(99000),G(M)
COMMON/C3/PIVOT(M)
```

A maximum virtual address range of 32 million words is currently available using virtual space. The default size is 6.5 million words. A big difference exists between theoretical limits and practical limits. We recommend that a casual user keep within two or three times the real memory size to keep thrashing levels acceptable.

Compiler-generated code that references a virtual object is generally less efficient than that for references to objects in nonvirtual space. (An object refers to a scalar variable or an array in common, or local to, the subprogram.) Only the larger common blocks and local arrays that cause size problems should be put into virtual space. For example, when arrays dimensioned as NxM are put in virtual space, smaller single-dimension arrays dimensioned by N or M to hold a row or column should be left in normal nonvirtual space.

Several routines are available to you that enhance CPU performance of virtual or banked programs. For more information, see M.17.

M.2. Method

Multiple D-banks are used in the implementation of virtual space. Each D-bank contains a page of virtual space. A virtual object from your program is dynamically allocated by the ASCII FORTRAN run-time system and gets as many pages as are necessary to hold the object. Since the Executive currently allows a maximum of 251 banks in a collected absolute, this puts an upper limit on the number of D-banks used for virtual pages. The amount of virtual space available is the maximum number of banks (pages) times the virtual page size. You can select the maximum number of pages allocated for virtual space and their size. Defaults are 200 pages and 32K words, giving a default of 6.5 million words of virtual space. (K means 1,024 words.) Page sizes are a power of 2 and can be 4K, 8K, 16K, 32K (default), 64K, or 128K words. A total of 246 banks of size 128K words gives a range of 32 million words. Large page sizes give a large virtual address space, but smaller page sizes minimize main storage requirements and dramatically reduce potential thrashing problems of a large page size.

The D-bank pages used for virtual space are defined in the FORTRAN main program's relocatable by use of a collector INFO-11 directive. Use of these directives in the generated relocatable defines a set of initially void D-banks to the collector. COMPILER statement options can be used in the main program to change the size or the number of these D-banks from the default values. When the main program does not contain a VIRTUAL statement or is not written in FORTRAN, virtual space can still be used. The run-time library element VSPACE$ then supplies the INFO-11 directives defining virtual space. Default values can be altered by changing tags in the procedure VIRTPROC$ in the MASM procedure element FTNPROC (see 9.5.3), reprocessing FTNPROC with PDP, and then reassembling elements VSPACE$ and VFTNEQU$.

Virtual space is allocated and initialized dynamically during execution of your program. When an external subprogram is entered for the first time, it calls the virtual storage allocator to allocate space for the static virtual objects belonging to it or to any of its internal subprograms. These static virtual objects can be named common blocks and selected large local arrays. Generated code saves the virtual addresses of these objects returned by the virtual storage allocator. After allocation, execution of code performs any initialization needed resulting from DATA statements on the virtual items. On subsequent entries to the subprogram, the above process is skipped for the allocation and initialization of static virtual space; it is done only once per external program unit. When a named common block has already been allocated by a previous program unit, the virtual allocation routine simply returns its virtual address.

Each D-bank holding a page of virtual space has a 64-word ID area at the beginning of the bank. This area makes the bank self-identifying, and is used by generated code to speed up execution time.

Virtual local arrays in the automatic storage class (those that do not have SAVE statements specifying them) are allocated and initialized on each entry to an external subprogram, and are freed on each return from the subprogram. Local virtual space defaults to the automatic storage class and local nonvirtual space defaults to the static storage class.

Expect the following when virtual space is allocated for a virtual object:

- The object is allocated on a 64-word boundary in a virtual page.
- The first 2,048 words of a virtual object are guaranteed to be in one virtual page. This permits more efficient code to be generated to reference virtual objects when they are in the first 2K of their allocation.

M.3. Restrictions for Declaration Matching

Since special code sequences are required to reference virtual objects, all FORTRAN routines that reference a virtual object must declare the object as a virtual object. However, arguments don't need to be specified in a VIRTUAL statement. A VIRTUAL statement in a FORTRAN element guarantees that arguments are correctly referenced when they reside in normal, banked, or virtual space. For more information on banking, see Appendix H.

Example

One element:

```
VIRTUAL L1,/C1/
COMMON/C1/C1(999)
COMMON/C2/C2(10000)
REAL L1(9999)
CALL X(L1(I),1.0,1)
END
```

Separate element:

```
SUBROUTINE X(A1,XL,1)
VIRTUAL/C1/
COMMON/C1/C1(999)
REAL A1(*)
C1(I)=A1(I)
    .
    .
    .
END
```

When subprogram X doesn't contain a VIRTUAL statement or when common block C1 is not named in its VIRTUAL statement, fatal run-time diagnostics result, indicating storage mismatch.

In current FORTRAN library files holding FORTRAN relocatables, often a simple COMPILER(BANKED=ALL) statement is put in each FORTRAN element in case any program using the library has multiple D-banking present. Obtain the same results by inserting a VIRTUAL statement into each element in case any arguments are in virtual space. (This also handles banked arguments.) When any named common blocks are in virtual space, they must be named in VIRTUAL statements in any subprogram that declares them.

M.4. Initialization of Virtual Objects

Initialization for objects in virtual space is needed when they have initial values specified in DATA, DIMENSION, or type statements. This initialization is accomplished by code that executes after the allocation calls. A DATA statement consisting of a mismatch of storage classes presents a problem, as the following example shows:

```
VIRTUAL /CBVIRT/,LOCALD,LOCALS
COMMON /CB1/A(1000)        @ standard common
COMMON /CBVIRT/V(1000000)  @ virtual common block
DIMENSION LOCALD(1000000)  @ virtual dynamic local
DIMENSION LOCALS(1000000)  @ virtual static local
SAVE LOCALS                @ put LOCALS in static class
DATA A(1),V(1),LOCALD(1),LOCALS(1)/1.,2.,3,4/    @ mismatch
```

The DATA statement in the above example calls for:

- Initialization of nonvirtual common block CB1. (This is normally done by ROR packets, that is, code is normally not executed to initialize items in nonvirtual common.)
- Initialization of virtual common block CBVIRT. (This must be done only once by generated code after allocation.)
- Initialization of the static local array LOCALS. (This must be done only once by generated code after allocation.)
- Initialization of the dynamic local virtual array LOCALD. (This must be done by generated code after allocation, on each entry to the subprogram.)

The ASCII FORTRAN compiler can't separate the initializations of these various types so that they can be done in their respective manners. Therefore, when any item in any initialization statement is in virtual space, the entire statement is accomplished by generated code. When any item in an initialization is in the virtual automatic storage class, all items must be in the automatic storage class or the statement is flagged in error. (The preceding example is in error.) When an initialization statement has a virtual/nonvirtual mismatch on static class items, a warning is issued at compile time, and code is generated to accomplish the entire statement at execution time. The warning is issued since dynamically initializing nonvirtual common can lead to incorrect program results. Initialization done by the collector resulting from ROR packets is insensitive to program execution flow, whereas initialization done by generated code exactly follows the program execution flow and can result in an item being used before it is initialized.

To prevent problems in an initialization statement, you should ensure that:

- A mixture of static storage class virtual items and automatic storage class virtual items is not contained in an initialization statement
- A mixture of virtual items and nonvirtual items is not contained in an initialization statement

The following shows the compiler action on combinations of various storage classes in a single initialization statement:

		VIRTUAL			NONVIRTUAL		
		common	*static local*	*automatic local*	*common*	*static local*	*automatic local*
VIRTUAL	*common*	X					
	static local	OK	X				
	automatic local	E	E	X			
NONVIRTUAL	*common*	W	W	E	X		
	static local	OK	OK	E	OK	X	
	automatic local	E	E	OK	OK	OK	X

Note: *E means an error diagnostic; W means a warning diagnostic.*

M.5. BLOCK DATA Subprograms

BLOCK DATA subprograms initializing common blocks in virtual space must appear in an element holding one or more executable program units. The first program unit entered in the element also causes the BLOCK DATA initializations to be done dynamically. Due to this, we recommend that you place the BLOCK DATA subprograms in the element holding the main program. Always place BLOCK DATA subprograms in an element holding executable code because the collector ignores the BLOCK DATA element unless the element is part of the collection.

M.6. COMPILER(DATA=AUTO) Statement

The COMPILER(DATA=AUTO) statement activates the ASCII FORTRAN automatic storage feature. Programs using this feature can also use the VIRTUAL statement. However, one or more noncommon static D-bank cells are allocated per program unit, and the prolog code is slightly longer.

M.7. Error Detection

M.7.1. Insufficient Space

A request for virtual space that cannot be met results in a run-time diagnostic and error termination. You must either limit your storage requirements or increase the virtual address space available by putting COMPILER statements in your main program that specify larger banks or more banks for virtual space.

Once the Executive is changed to allow the loading of a program containing over 251 banks, the full 2,047 banks allowed by the collector can be used, and the maximum virtual address range rises to 262 million words.

Most run-time diagnostics are accompanied by a walkback to aid in debugging. The walkback is available only when one or more of the compilations uses the F option.

M.7.2. Bad Allocation or Initialization

Because of the dynamic allocation and initialization of named common blocks in virtual space, the first program unit that calls the virtual storage allocator for a given common block must have the same or larger size for the common block of any other program unit and must be the only program unit with DATA initialization on it. An exception to this is when there are BLOCK DATA program units in the same element that are executed first. A program unit presenting a request to the virtual storage allocator for an allocated common block results in a run-time diagnostic when either:

- The requested size is greater than that originally allocated and the allocation can't be expanded
- Initialization is to be performed by this program unit

The virtual storage allocator keeps track of nonvirtual common blocks. When one subprogram indicates a common block is a virtual object and another says it is not, a fatal run-time diagnostic occurs.

When a nonvirtual common block has dynamic initialization due to a DATA statement in a given program unit and the common block is already referenced by another program unit, a run-time diagnostic also occurs.

M.7.3. Page Spanning

A major restriction of virtual FORTRAN is:

> *No portion of a scalar or array element may span from one virtual page to the next.*

The compiler is not aware of the page size to use unless a main program is present in the compilation unit. Thus, for many instances it can't diagnose a scalar or array element that illegally spans a page boundary. So, the generated code calls a span-check routine for each referenced variable that has a potential spanning problem. A run-time diagnostic occurs when a problem exists.

Example:

```
VIRTUAL /C1/
COMMON /C1/ R,DP(400000),R2(100000)
COMPILER(PAGESIZE=4K)
REAL R,R2
DOUBLE PRECISION DP
    .
    .
    .
```

This program is in error. The double-precision array DP will have spanning problems on page sizes from 4K to 64K, and can also span on a 128K page size when the common block is allocated far enough into the virtual page. Change the program to place the double-precision array DP on a double word boundary. One method is:

```
COMMON /C1/ R,TRASH,DP(400000),R2(100000)
```

Insertion of the variable TRASH puts array DP on a double-word boundary. Run-time diagnostics appear when bank spanning is detected.

Arguments also pose a problem. Bank spanning can occur when the size or type declared for the dummy argument isn't the same as that declared for the actual argument passed. Examples include:

- passing a REAL array, but declaring it as DOUBLE PRECISION or COMPLEX in the called routine
- a mismatch on size declarations on actual and dummy character arguments (these can be either scalars or arrays)

To detect a bank-spanning problem on arguments from any of the above causes, a span-checking routine is called on entry to a subprogram to check its arguments. A run-time diagnostic occurs for each argument with a problem.

This span-checking routine is not called for arguments when optimization is used or when a COMPILER(ARGCHK=OFF) statement is in the called routine. It is handled like the normal ASCII FORTRAN argument type-checking routine (for being called or not being called).

Example:

```
      VIRTUAL /C1/,/C2/
      COMMON /C1/C4(9000)
      COMMON /C2/R(9000)
      CHARACTER*4 C4
      REAL R
C
C         Arrays C4 and R can't incorrectly span banks; they start on
C         acceptable boundaries.
C
      CALL SUBR(C4(1),R(2))
             .
```

```
                  .
                  .
         SUBROUTINE SUBR(C,DP)
         VIRTUAL COMPILER(ARGCHK=OFF)
         CHARACTER*11 C(4000)
         DOUBLE PRECISION DP(3000)
   C
   C        These actual to dummy argument type-
   C        mismatches may cause spanning problems.
   C
                  .
                  .
                  .
```

Program execution errors can result because bank spanning can occur on some array elements, depending on virtual page size, where the common blocks are allocated in them, and which array elements are referenced. When dummy argument arrays have an asterisk (*) as the last dimension, the extent of the array is unknown and no span-checking function takes place.

Example:

```
   SUBROUTINE INTX(C,X,Y,N,COK)
   VIRTUAL
   CHARACTER*(*) C(*),COK(9991,N)
   DOUBLE PRECISION X(N,*)
   COMPLEX*16 Y(100,N,*)
```

Only array COK can be span-checked in this example.

M.8. Character Arrays

Code generated to reference a character array uses a compiler-generated variable referred to as a virtual origin variable. The virtual origin variable contains the address of the array manipulated to simplify index calculations. The virtual origin of a character array whose element size is not a multiple of four characters (including all CHARACTER*(*) arrays) is kept in character address form (for example, base address times four). When the array is in virtual space, its base address is a virtual address. When the run-time library element F2BDREQU$ (see M.10) is changed to use BDR3 for referencing virtual space, the D-bit of the BDR field is shifted up to make the virtual origin value overflow and become negative when using 128K banks for virtual space. All other virtual bank sizes don't cause a problem. Therefore, the following restriction is placed on the use of virtual FORTRAN:

> A program defining a virtual or dummy character array whose element size is not a multiple of four characters results in a run-time diagnostic when the collected bank size is 128K and BDR3 is used for virtual space.

There is no problem with 128K banks when the default BDR1 references virtual space. The span-check routines issue the diagnostics for this nonmultiple-of-four character array problem for 128K banks.

M.9. Banking and BDR Use

The D-banks used for virtual pages start near the end of the standard 262,143-word address space: 262,144 minus page size minus 512. Don't overlap this address space with the control bank or other banks.

M.10. Hidden User Banks

When on dual-PSR machines, you can use the utility PSR that is not used for the virtual banks, as long as its use is hidden from ASCII FORTRAN. You must define the banks in the collector symbolic and manage all basing of banks under that window. ASCII FORTRAN treats items in these user banks as if they are in the control bank. BDIs should never be passed for arguments in user banks; instead, a BDI field of zero should be passed so that ASCII FORTRAN treats the items as unbanked (that is in the control bank). This happens automatically for FORTRAN programs when nonvirtual common blocks are included by the IN directive in the user banks.

When hidden user banks are used, the following rules apply:

- When a named common block is in a hidden bank, it may not be referenced from a FORTRAN subprogram with a COMPILER(BANKED=ALL) statement in it.
- User banks can't overlap in address space with any other program banks (I-banks, control bank, or virtual banks).
- Two BDRs simultaneously basing banks with overlapping address space are not allowed.
- A virtual bank must not be based under a BDR used for user banks.
- A request for storage by an MCORE$ statement on the control bank must not overlap address space with user banks.
- No FORTRAN-generated code (location counters 0, 1, and 4 in an ASCII FORTRAN compilation) should be placed in hidden user banks.

The utility I-bank BDR (BDR1) is used to base the virtual banks. When you are using this BDR for FORTRAN banked space, or for your own use hidden to FORTRAN, the utility D-bank BDR can be used instead for virtual space. The run-time library element F2BDREQU$ holds EQU values for the BDRs, as follows:

VBDR$	BDR used for virtual space
BKBDR$	BDR used for banked space
CBDR$	BDR used for the control bank

When your program is not collected according to these conventions, you must edit F2BDREQU$ and reassemble it.

M.11. Performance

Virtual FORTRAN adds the ability to define and use large user objects such as scalar variables and arrays to the ASCII FORTRAN system. Referencing a virtual object requires more code than referencing a nonvirtual object, so only put larger objects in virtual space (an object here means an array or scalar variable). When a large common block has mostly large arrays, you can place the larger arrays in a new common block that is then placed in virtual space. This leaves the smaller items in nonvirtual space, which is more efficient to reference.

The software paging of virtual FORTRAN is similar to the virtual memory architectures of other manufacturer's machines. Improper use of a large user address space in these machines often means disaster for you. This is also true for the ASCII FORTRAN virtual system. CPU performance and page traffic depend on what you do and don't place in virtual space, as well as on how you use it. It is possible to cause dramatic CPU performance and page-traffic changes by a simple reordering of key loops in some programs. A program that goes through a large amount of virtual space referencing one item per page doesn't execute in a reasonable amount of time.

A program that uses a large amount of virtual space takes some setup time. This is because even static virtual space is dynamically allocated and initialized, and each bank must have its correct size allocated by an ER to MCORE$. The overhead involved with allocation of virtual space is incurred only once per program. For a program with a large amount of virtual space, this overhead is not unreasonable.

M.12. Thrashing

On virtual storage machines, your user address space is broken into pages for purposes of swapping. A reference to something in a nonresident page automatically brings that page into real memory. Resident pages not currently being referenced are swapped out by the operating system when storage becomes scarce.

In the ASCII FORTRAN virtual system, the pages reside in the D-banks defined by the main program's INFO-11 directives. The allowable page sizes are 4K, 8K, 16K, 32K, 64K, and 128K words. You select the page sizes in your main program. Large page sizes can cause thrashing when your reference pattern doesn't result in a reasonably sized working set of pages. Avoid large page sizes unless necessary for a larger virtual address range. Programs that need a very large address space (i.e., 4 million words or more) should be run when the system is lightly loaded.

When a program takes excessive wall clock time to execute in comparison to the SUP times accumulated, thrashing is occurring. Look for loops in your program that reference virtual objects inefficiently.

Example 1:

The following program defines and references its data in an inefficient manner:

```
SUBROUTINE COMP
VIRTUAL /CB1/,VEC
COMMON /CB1/A1(2000,2000),B(2000,2000)
```

```
      REAL VEC(2000)
      DO 5 K = 1,2000
5             VEC(K) = 0.0
      DO 10 J=1,2000
      DO 10 I=1,2000
10            VEC(I) = A1(J,I) - SIN(B(J,I))
               .
               .
               .
```

The inner loop of this program references three separate virtual pages on each iteration, and two of these pages are referenced for only a very few iterations before a bank change occurs. The local vector VEC isn't that large (2,000 words) and should not be placed in virtual space when used in this manner.

Example 2:

The following variation on the above program segment is essentially equivalent, and greatly reduces page traffic and thrashing:

```
      SUBROUTINE COMP
      VIRTUAL /CB1/
      COMMON /CB1/A1(2000,2000),B(2000,2000)
      REAL VEC(2000)
      DO 5 K = 1,2000
5                  VEC(K) = 0.0
      DO 10 J=1,2000
      DO 10 I=1,2000
10                 VEC(I) = A1(J,I) - SIN(B(J,I))
                     .
                     .
                     .
```

In this code sequence, there are many references to a page before it is no longer needed. The working set of pages referenced by the program changes very slowly with time. In addition, page traffic and bank swapping by the operating system are substantially reduced.

The Dynamic Allocator (DA) in the Executive controls storage allocation and swapping; thus, its operation directly influences the thrashing potential of a program. An executing program must have the ability to accumulate a reasonable working set of resident pages or thrashing occurs. DA modifications improve its operation in this area. The changes are called the PEF-1 package. This package is available starting with Exec level 38R1. This DA enhancement is needed for a program that heavily uses a large amount of virtual address space.

M.13. CPU Performance

M.13.1. Generated Code

When you make a reference to a virtual object, the ASCII FORTRAN compiler must generate a decomposition code sequence. This can add three to seven machine instructions to a simple reference of an array element. One of these instructions is either an LBJ, which is a very slow instruction, or a link to an activation run-time routine. There are several variations of code sequences that reference items in virtual space. Some of these sequences have a test around the LBJ instruction or around the link to an activation run-time routine. These test sequences are used when the probability of a bank change occurring is low. For example:

```
A (I) = A (I+100) - A (I-1)
```

The decomposition code sequences are generated when referencing an array that is known to be in virtual space by its declaration, or is a dummy argument. It is not known whether an argument is in virtual space or not, so a special virtual code sequence must be used when a VIRTUAL statement is in the subprogram.

Scalar arguments or scalars in virtual space must also be activated. No decomposition sequence needs to be done, but an LBJ (and possibly a preceding test instruction) must be executed to reference the scalar. Since scalars take up little space, we strongly recommend that you place as few scalars as possible in virtual space. For heavy use of scalar dummy arguments in a subprogram, it is faster to move them to local variables in the subprogram to cut down on unnecessary LBJ instructions that can cause unwanted page traffic to occur.

A scalar in virtual space that resides wholly under a relative of address 2K in its location counter has better code generated to reference it. A full decomposition sequence need not be generated, though the LBJ activation is still needed. The LBJ is expensive and can cause page traffic, so we again recommend that you keep scalars and small arrays out of virtual space.

ASCII FORTRAN-generated code uses index registers to reference all data when the O option (over 65K addressing option) is used. This is normally needed when the last address of the collected program is over 65K. Use of this option may be needed in a virtual FORTRAN program when the control bank that holds all nonvirtual data is too large, and truncation errors result during collection. However, the location and size of the banks defining virtual space do not affect use of the O option. The code generated to reference nonvirtual data is faster without the O option. Therefore, the O option is not assumed by default.

M.13.2. Input/Output

M.13.2.1. Striping and Implied-DOs

A system that needs a large amount of virtual space may also need to do a large amount of I/O on these large arrays for input and results. Unformatted I/O on a whole large array does striping to avoid heavy addressing arithmetic and LBJ usage.

Example:

```
VIRTUAL A
REAL A(2000,1000)
READ(10,END=20,ERR=22) A
```

An implied DO in the I/O statement causes CPU usage on the statement to increase dramatically, because the compiler generates codes for each element reference and a control transfer occurs between this code and the I/O complex for each array element.

Example:

```
VIRTUAL A
REAL A(2000,1000)
READ(10,END=20,ERR=22)((A(I,J),I=1,2000),J=1,1000)
```

You often want to do a large block of I/O from a contiguous area of an array but cannot simply use the whole array name as a list item because of one of the following:

- The starting point is a variable.
- The end point is a variable.
- You only want to do one column of a multidimensional array.

Thus, you can use only the inefficient implied DO in the I/O list.

Example:

```
SUBROUTINE X(NSTRT,NEDN,A,INX)
COMMON/CX/C(100000),C2(4000,20)
DIMENSION A(2000,1000)
READ(10) (A(I),I=NSTRT,NEND)
READ(11) (C(I),I=NSTRT,NEND)
WRITE(12)(C2(I,INX),I=1,4000)
END
```

You can change the previous example to do more efficient whole-array I/O by the method shown below:

```
SUBROUTINE X(NSTRT,NEND,A,INX)
COMMON/CS/C(100000),C2(4000,20)
CALL ARYIO(.TRUE.,10,A(NSTRT),NEND-NSTRT+1)
CALL ARYIO(.TRUE.,11,C(NSTRT),NEND-NSTRT+1)
CALL ARYIO(.FALSE.,12,C2(1,INX),4000)

SUBROUTINE ARYIO(READ,UNIT,ARY,SIZE)
LOGICAL READ
INTEGER UNIT,SIZE
REAL ARY(SIZE)
IF (READ) THEN
   READ(UNIT) ARY
ELSE
   WRITE(UNIT) ARY
```

```
ENDIF
END
```

Note: *This method applies to FORTRAN programs that either do or do not use virtual space.*

M.13.2.2. Buffer Sizes

When you do unformatted I/O on large arrays, the OPEN statement should specify a reasonable block size and segment size. The defaults of 111 words for segment size and 224 words for block size are not appropriate when transferring millions of words of data (see Appendix G and 5.10.1). Control bank space is used for I/O buffers. You cannot define very large buffer sizes such as 300K words.

M.13.2.3. NTRAN$

The NTRAN$ service subprogram can do very efficient, primitive I/O on any size virtual object. (See 7.7.3.16.1 and 7.7.3.16.2.)

M.13.3. Intrinsic Functions

The ASCII FORTRAN compiler moves banked or virtual array elements to local storage when these are passed as arguments to selected mathematical intrinsic functions.

M.14. Timings

The following tests show the effect of various addressing decomposition code sequences on execution times. The results are also compared to nonvirtual FORTRAN timings. The basic test is a simulation of the following FORTRAN program segment:

```
      REAL A(1048576),B(1048576),C(1048576),D(1048576)
      DO 10 I = 1,1048576
10    A(I) = B(I)*C(I) + D(I)
```

Several different decomposition code sequences are run (called A, B, C, D, E, etc., in Table M-1), some having tests around the LBJ instruction or activation call.

This program shows ASCII FORTRAN nonvirtual code-generation capabilities in the best light, since all array references are strength-reduced and a free auto-increment can be done on X-registers for each array reference. It is an inefficient example for the virtual approach since each array reference requires a full decomposition sequence and LBJ. Real programs are never nearly as optimizable for ASCII FORTRAN as this one, and are hopefully not as inefficient as this one is for virtual FORTRAN.

Test setups:

- TEST1:

  Four simulated one-million-word arrays, each getting eight 128K banks. (The arrays are 1,048,576 words each, octal 04000000.)

- TEST2:

  Four 32K arrays all in one bank. An outer loop brings the number of iterations up to be the same as the four one-million-word arrays. The purpose of this test is to see what effect the test instructions skipping around the LBJ or activation calls has.

The following control tests are done for comparison purposes using ASCII FORTRAN level 10R1 with compiler options O and Z. An outer loop brings the number of iterations up to one million.

- TEST5:

  No banking.

- TEST6:

  Banking, all arrays in one bank, a BANK statement is supplied to the compiler.

- TEST7:

  Banking, each array in a different bank, four BANK statements are supplied to the compiler.

- TEST8:

  Banking, all arrays in one bank, no BANK statements, but BANKED=ALL used to indicate banking.

- TEST9:

  Banking, each array in a different bank, no BANK statements, but BANKED=ALL used to indicate banking.

In Table M-1, the VA/addr column gives the registers that the virtual address is taken from, and the register that the absolute address resides in after activation. A*x* means a single A*x*- register such as A0, A1, A2, A3. Pair means an even-numbered non-A*x*-register pair, such as A4-A5, A6-A7.

Example:

```
A/Ax
```

The VA is in an A-register, and the absolute address is in an A*x*-register after activation.

Table M-1. Timings (in seconds)

	Code Sequence	Added Instr. /Ref.	VA/Addr.	TEST1 Time	TEST2 Time	Activate Link
Nontest Sequences	C	5	Ax/Ax	20.93	20.93	LBJ
	EAO	6	Pair/Ax	21.15	21.15	LBJ
Nonargument Test Sequences	TLEABS1	4	Ax/Ax	27.71	10.44	LMJ
	FAO	6	Pair/Ax	25.16	10.87	LMJ
Argument Sequences	TLARGB	5	Ax/Ax	29.82	12.60	SLJ
	TLARGU	6	Pair/Ax	31.11	15.09	SLJ

Control Tests (FTN 10R1):

Test	Time	Comments
TEST5	2.663	Control test, no banking done
TEST6	2.647	One LBJ per 32K iterations
TEST7	14.991	Four LBJs per iteration
TEST8	6.748	One SLJ, total
TEST9	33.902	Four SLJs per iteration

These tests were done on an 1100/80 system using a UNISCOPE 200 display terminal. We obtained CPU timings by executing a TIME program before and after each execution.

M.15. Efficiency Suggestions

- Avoid using the ALL option of the VIRTUAL statement because it places all named common blocks into virtual space. Separate out the largest arrays, and put those arrays into one or more common blocks in virtual space. (Large local arrays can also be named in the VIRTUAL statement.)
- Keep scalers and small arrays out of virtual space as much as possible. When heavy use of scalar arguments occurs (especially in loops), move them to local variables after subprogram entry and use the local copies in the subprogram. (The compiler often does this itself, when it is safe to do so.)
- When a subprogram has many array arguments that you know are usually in nonvirtual space, drop some of these arguments and use nonvirtual common instead.

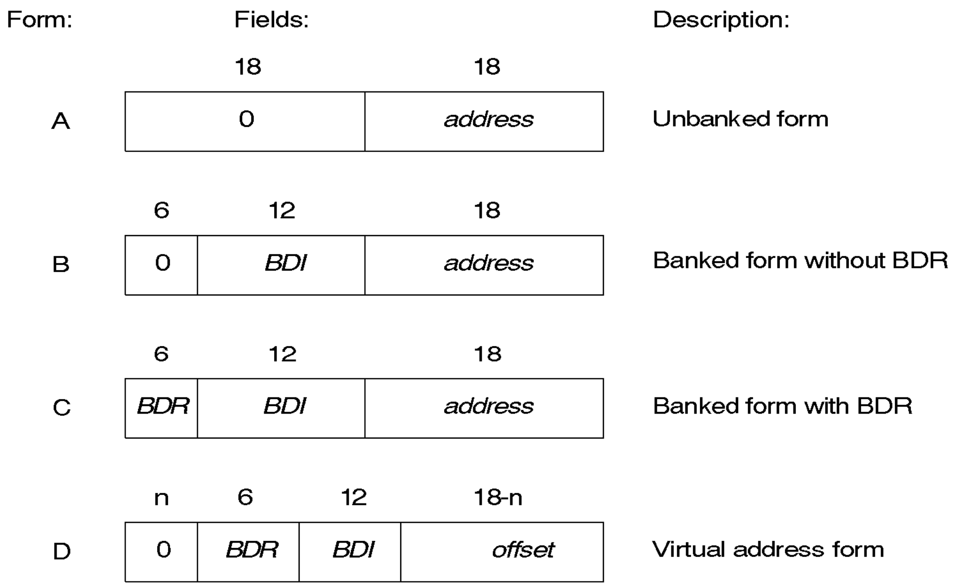
- When possible, organize your code so that references to items in virtual space iterate in small increments, rather than randomly or in large increments. This can minimize bank switching and thrashing.

  Note: *Using the MOVWD$ service subprogram to move data from one array to another often aids efficiency.*

- Increase the default buffer sizes for files having heavy unformatted I/O on large arrays. Use whole arrays for I/O list items when possible. See the method shown in M.13.2.1.
- Use the smallest bank size possible for virtual space to help minimize thrashing potential.

M.16. Argument Forms

When arguments pass to a FORTRAN program, a packet is generated that contains one address word for each argument. There are four address forms that can pass for an argument in ASCII FORTRAN.



where 0 is size *n* and *offset* is 18- *n*; see Form D explanation below.

The 6-bit BDR field in forms C and D is set as follows (the bits are numbered from left to right, starting at 1 by the FORTRAN convention):

- Bit 1: 0
- Bits 2 - 3: BDR number.

BDR1	Utility I-bank
BDR2	Main D-bank
BDR3	Utility D-bank

- Bits 4 - 6: 0

The different forms are used for the following purposes:

- Form A is passed for arguments known to be in the control bank at compile time. This includes nonvirtual and nonbanked data (local and common), constants, and temporaries (expression results).
- Form B is passed for arguments that are banked items declared with the standard banking mechanism (the BANKED=ALL COMPILER statement option or a BANK statement with a common block list). Form B is also used in I/O packets and packets for character run-time routine calls. In addition, it is used internally by generated code for certain character array arguments when VIRTUAL is on. Normally, a form B address converts to a form C address for internal use by compiled code.
- Form C is used when passing a dummy argument as an actual argument when BANKED=DUMARG is on, and for passing a dummy scalar argument as an actual argument when VIRTUAL is on.
- Form D, the virtual address form, is passed for variables in virtual space. The BDR can be 1 or 3 (it is defined in element F2BDREQU$). It cannot be 0 or 2 because then a virtual address is not unique (that is, forms C and D look identical). Form D is used for virtual items only. The bit size *n* is defined according to the virtual bank size as follows:

Virtual Bank Size	**Value of *n***
4K	6
8K	5
16K	4
32K	3
64K	2
128K	1

Note: *K mean 1,024 words.*

There is no ambiguity between the four forms because of the BDR field used in forms C and D. This field is used by subprogram prolog code and run-time routines to distinguish between the forms.

To get this uniqueness of forms, the following assumptions are made concerning the maximum number of pages in the program:

500	when banking with no virtual
500	when virtual with bank size 4K
1,000	when virtual with bank size 8K
2,000	when virtual with bank size 16K, 32K, 64K, or 128K

Note: *The defaults are 200 pages of 32K words each.*

Nonvirtual FORTRAN with standard banking handles forms A, B, and C. Virtual FORTRAN handles all four argument forms.

M.16.1. Processing Dummy Scalars

In a virtual FORTRAN subprogram-generated code, dummy scalar references use a banking sequence that involves an LBJ (or test and LBJ), but no virtual address decomposition. Therefore, dummy scalars must have their argument words converted to form C by a run-time call in the subprogram prolog code. In nonvirtual FORTRAN with standard banking, a banking sequence and form C are used for all dummy scalars and dummy arrays.

M.16.2. Processing Dummy Arrays

A large number of decomposition code sequences can be used on a virtual address. Arguments pose a special problem since it cannot be determined at compilation time that they are in virtual space. When these dummy arguments are not in virtual space, they need not be activated by an LBJ since they are always visible. A coding sequence manipulating fields of a virtual address has a problem when used on an array argument that is in nonvirtual space. Once the array is indexed past the collected virtual page size, a BDI switch occurs. This would mean that a program using 4K-sized virtual pages cannot safely pass a control bank (nonvirtual) array larger than 4K words in size as an argument. This would be a poor restriction. There are other activation sequences that do not do a simple decomposition into fields.

Assume that two words in each ID area (RANGEABS and RANGEABS+1) give the absolute address range for that virtual page. Another ID area word (MAGICCELL) is structured such that the virtual address of any word in that bank added to the word gives the absolute address of that word. Assuming that register X10 points at the virtual ID area, the following range test code sequence is possible for items declared in virtual space.

Sequence TLEABS1:

```
  .
  .                      .VA ->  A0, VA means virtual address
```

```
.
A A0,MAGICCELL,X10        .Make absolute address if visible
TLE  A0,RANGEABS+1,X10    .Address too big?
TLE  A0,RANGEABS,X10      .No. Too small?
LMJ  X11,ACTA0            .Wrong bank visible, fix it.
.
.                         .Use address in A0
.
```

An argument may or may not be in virtual space, so the canned X10 pointer to the virtual ID area is insufficient; a control bank argument would always cause an out-of-line activation call. So, each argument needs an associated ID area pointer to be used by a variation of the above sequence. (Even the control bank has an ID area.)

Sequence TLARGB:

```
.
.                         .VA ->  A0, VA of elt of X to A0
.
L X11,ARGXIDPTR           .Ptr to ID area for arg X
A A0,MAGICCELL,X11        .
TLE  A0,RANGEABS+1,X11    .
TLE  A0,RANGEABS,X11      .
SLJ  X11,ACTA0X           .
.
.                         .Use address in A0
.
```

The control bank ID area defines its absolute address range in cells RANGEABS and RANGEABS+1 as FIRST$ to 0777777. The control bank ID area has a MAGICCELL of zero and argument prolog code uses an absolute address for control bank array arguments rather than a virtual address when creating the virtual origin variable for a dummy array. Use of range test sequences like the above on array arguments means that:

- the code sequence works correctly for both virtual and nonvirtual arguments
- no bank needlessly activates when already based, even control bank arguments
- artificial size restrictions do not have to be placed on nonvirtual arrays when they are passed as arguments (even when they are in banked space)

These test sequences are used to reference dummy array arguments.

M.16.3. Example of Argument Forms Passed

The following example has items in virtual space, banked space, and the control bank:

```
SUBROUTINE x(e,b)
VIRTUAL /C1/
COMPILER(BANKED=ALL)
COMMON/C2/c(1000)
DIMENSION e(100000)
COMMON /C1/d(10000000)
CALL y( 1, b, c(i), d(i), e(i) )
END
```

The following list shows the address forms in the parameter packet passed to subroutine y from subroutine x:

Argument	Description of Argument	Address Form
1	Constant (control bank)	A
b	Dummy scalar	C
c (i)	Element of banked array	B
d (i)	Element of virtual array	D
e (i)	Dummy array element	A, C, or D

When dummy array **e** is a control bank array, form A is passed; when **e** is in virtual space, form D is passed; and when **e** is an array in banked space, form C is passed.

M.17. Library Utility Routines

M.17.1. VACTIV$

Purpose:

Use VACTIV$ to base an item's bank and return its absolute address. Only experienced programmers should use this routine.

Form:

```
L      A0,address
LMJ    X11,VACTIV$
```

Description:

Use the VACTIV$ linkage protocol from MASM routines only. It is called from the FORTRAN library, including I/O, and user MASM routines to base an item's bank (if it has not already been based) and returns its absolute address. It replaces the old ACTIV$ routine that could not handle virtual addresses. The linkage is LMJ X11,VACTIV$. VACTIV$ is in the control bank, so an IBJ$ or LIJ need not be done. All registers are restored except X11 and A0.

Example:

Input:

A0 has an address in one of the four forms: A, B, C, or D.

Output:

H2 of A0 contains the item's absolute address. Also, the item's bank is based.

Note: *H1 of A0 may be nonzero on the return.*

M.17.2. Service Routines

Several routines aid you in enhancing CPU performance of virtual or banked programs. With the exception of the MOVWD$ and MOVCH$ routines, the following routines should be used only by programmers skilled in the use of assembly language and familiar with multibanking and the ASCII FORTRAN subprogram linkage conventions. The MOVWD$ and MOVCH$ routines, on the other hand, are easy to use, and their use in a few key places enhances performance. All of these routines can be called from FORTRAN programs.

Several examples can be found in M.17.4, following the presentation of all of the routines.

M.17.2.1. LOCV$

Purpose:

LOCV$ is an extension of the ASCII FORTRAN LOC service subprogram that handles virtual addresses.

Form:

```
I = LOCV$(arg)
```

where *arg* is a FORTRAN variable. The argument can be in the control bank, in banked space, or in virtual space.

Description:

The address of the passed-argument is returned. The form A address word is returned when the argument is in nonvirtual space. A form D address word is returned if the argument is in virtual space.

Note: *When the item is type character, its offset is ignored. The address word passed for the argument can be in one of the forms A, B, C, or D. An item in virtual space can only have a form C or D address word.*

M.17.2.2. CVVA$F

Purpose:

CVVA$F returns information about the bank in which the argument resides.

Form:

```
DP = CVVA$F(arg)
```

where *arg* is a FORTRAN variable.

Description:

This routine returns information in registers A0 and A1. The address returned in register A0 is in one of three forms: A, C, or D. The form A address word is returned for a control bank argument, form C is returned for an argument in banked space, and form D is returned for an argument in virtual space. Register A1 holds the ID area pointer for the bank in which the argument resides.

When you want both results, type the function as DOUBLE PRECISION. Otherwise, type the function as REAL or INTEGER.

Note: *An ID area pointer for a bank has the BDR+BDI in H1, and H2 is the address of the 64-word ID area for that bank.*

M.17.2.3. CVBK$F

Purpose:

CVBK$F returns information about the bank in which the argument resides.

Form:

```
DP = CVBK$F(arg)
```

where *arg* is a FORTRAN variable.

Description:

This routine returns information in registers A0 and A1. The address returned in register A0 can be one of two forms: A or C. The form A address word is returned for a control bank argument, and form C is returned for an argument in banked or in virtual space. Register A1 holds the ID area pointer for the bank in which the argument resides.

When you want both results, type the function as DOUBLE PRECISION. Otherwise, type the function as REAL or INTEGER.

M.17.2.4. CVBK$I

Purpose:

CVBK$I returns information on the BDI portion of the address contained in its argument.

Form:

```
DP = CVBK$I(arg)
```

where *arg* is a FORTRAN variable holding an address.

Description:

CVBK$I is an indirect version of CVBK$F. This routine returns information in registers A0 and A1. The address returned in register A0 is in one of two forms: A or C. The form A address word is returned for a control bank address, and form C is returned for an address representing banked or virtual space. Register A1 holds the ID area pointer for the bank that the address represents.

When you want both results, type the function as DOUBLE PRECISION. Otherwise, type the function as REAL or INTEGER.

M.17.2.5. LBJ$IT

Purpose:

LBJ$IT performs an LBJ instruction to base a desired bank.

Form:

```
CALL LBJ$IT(arg)
```

where *arg* is a one-word FORTRAN variable or array element. It holds a form C address word for a bank that is to be based. The address can be for virtual space, banked space, or the control bank.

Description:

An LBJ instruction is done on the passed argument to base the desired bank. Then a return takes place to the caller.

Note: *The argument is not checked for validity. H2 of the argument (the address) is not used and need not be a valid address in the bank.*

M.17.2.6. FTNWB$

Purpose:

FTNWB$ performs a full walkback trace.

Form:

```
CALL FTNWB$
```

Note: *There are no arguments.*

Description:

This walkback call doesn't pull the FTNPMD complex into your collected program. When the FTNPMD complex is collected in your program, this call activates a full walkback trace from the point of call; then normal execution resumes. When the FTNPMD complex is not collected in your program, the call has no effect.

Note: *The FTNPMD complex is brought into a collection when one or more FORTRAN relocatables are compiled with the F option, when you include element FTNPMD1 in your collection with the INCLUDE directive, or when you call the FTNPMD complex entry points FTNPMD or FTNWB in the FORTRAN program.*

M.17.2.7. MINE$F

Purpose:

The MINE$F routine is a strip-mining routine that is callable by a FORTRAN program. It is meant to be used as a primitive in strip-mining basic operations on virtual banks.

Form:

```
I = MINE$F(present,direct,next)
```

where (the arguments on input to MINE$F are):

present

is a variable holding an address in one of the forms A, B, C, or D. You want to know the number of words remaining in the bank indicated by this address.

direct

is an integer variable that indicates the direction taken to check the amount remaining in the bank.

When *direct* .GE. zero, movement is forward.

When *direct* .LT. zero, movement is backwards.

The values that the arguments to the MINE$F routine contain when the function returns are:

present

Unchanged.

direct

contains a signed integer value that indicates the remaining number of words in a virtual bank. When the *present* argument is not for virtual space, the value returned is a positive or negative 0777776, depending on the direction as specified by the *direct* argument on input.

next

contains the form D address of the start of the next virtual bank when the flag in *direct* is positive. When the flag is negative, the virtual address points to the end of the previous virtual bank. When the *present* argument does not hold an address for virtual space, the address returned in *next* is 0.

I

The function value returned is the original item address as held on entry in *present*, but possibly changed to another form. Two forms of output for I are:

1. The function result is a form A address when *present* holds a control bank address.
2. The function result is a form C address when *present* holds an address for banked or virtual space.

Description:

The caller wants to know the number of words remaining in a virtual bank from some present point (*present*). The FORTRAN variable *present* holds an address in one of four forms: A, B, C, or D. The direction can be forward or backward as indicated by *direct* being positive or negative. The function result can be used in an LBJ instruction to base the bank where the stripe resides.

M.17.2.8. MOVWD$

Purpose:

The MOVWD$ routine moves word-oriented items efficiently. It can also broadcast a scalar item to an array. Either *source* or *target* or both items can be banked or virtual.

Form:

```
CALL MOVWD$(source,srceincr,target,trgtincr,precision,itercount)
```

where:

source

is the item that is moved to a target.

srceincr

is an integer expression indicating the increment in elements to find the next source item to be moved. This increment can be positive, negative, or zero. When the increment is zero, the item is a scalar and the scalar is broadcast in the target array.

target

is the destination item that is filled from the source.

trgtincr

is an integer expression indicating the increment in elements to find the next target item to be stored to. This increment can be positive, negative, or zero. When the increment is zero, the item is a scalar.

precision

is an integer expression indicating the number of words of storage per array element of the source and target items. The precision is assumed to be the same for both source and target. The values for *precision* are: one, two, or four words.

itercount

is an integer expression indicating the number of iterations or loops of this move. For zero or negative values of *itercount*, no moves are done. This form of a zero-trip DO-loop allows you to easily replace simple loops that move data with CALL statements to MOVWD$.

Note: *Character items can be used for source and target items only when the character items have no offset and the size of their array elements is one, two, or four words.*

Restriction:

If *source* and *target* overlap, and the *target* address is greater than the *source* address, the move must be set up to start at the end of the *source* and *target*, and copy towards the beginning. A negative increment for both the *source* and *target* would be used to do this.

M.17.2.9. MOVCH$

Purpose:

Use the MOVCH$ routine to move character items efficiently. It can also broadcast a scalar item to an array. Either *source* or *target* or both items can be banked or virtual.

Form:

```
CALL MOVCH$(source,srceincr,target,trgtincr,itercount)
```

where:

source

is the character item to be moved to a target.

srceincr

is an integer expression indicating the increment in elements to find the next source item that is moved. This increment can be positive, negative, or zero. When the increment is zero, the item is a scalar and the scalar is broadcast in the target array.

target

is the destination character item that is filled from the source.

trgtincr

is an integer expression indicating the increment in elements to find the next target item to be stored to. This increment can be positive, negative, or zero. When the increment is zero, the item is a scalar.

itercount

is an integer expression indicating the number of iterations or loops of this move. For zero or negative values of *itercount*, no moves are done. This form of a zero-trip DO-loop allows you to easily replace simple loops that move data with CALLs to MOVCH$.

Note: *The target and source items do not have to be the same character lengths. The normal rules of truncation or filling with blanks for character assignment statements are followed by the MOVCH$ routine.*

Restriction:

If *source* and *target* overlap, and the *target* address is greater than the *source* address, the move must be set up to start at the end of the *source* and *target*, and copy towards the beginning. A negative increment for both the *source* and *target* would be used to do this.

M.17.3. Virtual Storage Allocator

There are three routines to allocate virtual storage. Two allocate static virtual storage for virtual common blocks and static local virtual variables. One allocates and frees dynamic virtual space for local virtual storage in the automatic class. These routines are called by generated code, or explicitly called by FORTRAN programs.

Note: *The virtual storage allocators acquire about 500 words of control D-bank storage by use of the MCORF$ routine. Other utilities that you use can also acquire storage in a similar manner.*

M.17.3.1. SALC$P

Purpose:

The SALC$P routine allocates static virtual storage in a packed manner.

Form:

```
VA = SALC$P(cbnam,size,flag,base)
```

where:

cbnam

is a one-word code (or name) used as a tag on this allocation. It must be unique for all allocations. The compiler uses the common block name in Fieldata for virtual common blocks. It uses a created name for local static virtual space.

size

is the size in words of this virtual allocation.

flag

is either .TRUE. or .FALSE. When .FALSE., this allocation is dynamically initialized by the caller to simulate DATA statement operation. It prints run-time diagnostics.

base

is set to the form C BDR-BDI/address of the first word of this allocation by the virtual storage allocator.

Description:

The function result is the form D virtual address of the first word of this allocation. The virtual storage allocator SALC$P allocates *size* words of virtual space to the static virtual object of name *cbnam*. When the object already exists in virtual space, the existing allocation is used. When *size* is greater than the existing size on a previously allocated object, an attempt is made to expand it. Expansion is possible when the object is the last one allocated in static virtual space or when there is room between the end of its current allocation and the next allocation.

When expansion is not possible, a nonfatal run-time diagnostic occurs. When the object is already allocated and *flag* is .FALSE., indicating that the object undergoes dynamic initialization on the return, a nonfatal run-time diagnostic occurs.

When the object is a common block and another program unit has already used the common block but did not indicate it as being a virtual object, error termination occurs. When not enough virtual space is available for the allocation, error termination also occurs.

Note: *The size and number of D-banks that virtual space uses is defined by the main program or by the library element VSPACE$ when the main program is not FORTRAN or does not contain a VIRTUAL statement.*

The routine SALC$P allocates virtual space in a packed manner and is the compiler default. This means that each allocation starts where the last allocation left off instead of at the beginning of a new D-bank. However, all virtual allocations start on a 64-word boundary in virtual space, and the first 2,048 words of an allocation are in one D-bank. (The 64-word boundary minimizes potential bank-spanning problems, and the first 2,048 words being in one bank allows efficient coding to reference scalars in the first 2,048 words of a virtual common block.)

If a virtual array is to be expanded, it should be declared to be one element long. This tells the compiler to not make any assumptions about the size of the array. Therefore, the code that is generated can handle any size array that can fit in virtual memory.

M.17.3.2. SALOC$

Purpose:

The routine SALOC$ allocates static virtual space in an unpacked manner.

Form:

```
VA = SALOC$(cbnam,size,flag,base)
```

where:

cbnam

is a one-word code (or name) used as a tag on this allocation. It must be unique for all allocations. The compiler uses the common block name in Fieldata for virtual common blocks. It uses a created name for local static virtual space.

size

is the size in words of this virtual allocation.

flag

is either .TRUE. or .FALSE. When .FALSE., this allocation is dynamically initialized by the caller to simulate DATA statement operation. It prints run-time diagnostics. When .TRUE., this allocation is not dynamically initialized.

base

is set to the form C BDR-BDI/address of the first word of this allocation by the virtual storage allocator.

Description:

SALOC$ operates identically to SALC$P, except that it allocates virtual storage in an unpacked manner, starting a new allocation at the beginning of a new D-bank.

M.17.3.3. DALC$P

Purpose:

The routine DALC$P allocates and frees dynamic local virtual space.

Form:

```
VA = DALC$P(size,base)
```

where:

size

is the size in words of this virtual allocation. When negative, it is a deallocation call.

base

is set to the form C BDR-BDI/address of the first word of this allocation by the virtual storage allocator (allocation call only).

Description:

The function result is the form D virtual address of the first word of this allocation. Dynamic virtual space is allocated in a LIFO (last-in-first-out) manner. A deallocation call *size* must be the negative of the allocation size, or error termination occurs. Dynamic virtual space is allocated from the last virtual D-bank backwards, and static virtual space is allocated from the first virtual D-bank forwards. This separates the two forms of allocation so that fragmentation of virtual space does not occur. When not enough virtual space is available for the allocation, error termination occurs. On a deallocation call the *base* argument isn't set, and no function result is returned.

M.17.4. Examples Using the Virtual Feature

Example 1:

The following example shows changes that can be done to enhance performance of programs that use virtual or banked space heavily. This program does a simple sum reduction of a REAL array. The code is:

```
      PARAMETER(N=<size>)
      REAL A(N), SUM
      SUM=0
      DO 10 I=1,N
10    SUM=SUM + A(I)
      PRINT*,SUM
```

Now let's define a service routine that performs a sum reduction, and replace the loop with a call to it:

```
PARAMETER(N=<size>)
REAL A(N),SUM,SUMRED
SUM=SUMRED(A,N)
PRINT*,SUM
```

The SUMRED service routine does a sum reduction by strip-mining portions of its input array, basing the stripe to make it visible, and performing several adds in its inner loop to hide the costs of storing the sum and the JGD instruction. It references the source array using base-offset type of referencing, which means it can access the source array in virtual space, banked space, or the control bank.

```
      REAL FUNCTION SUMRED(SOURCE,ITER)
      VIRTUAL
      COMPILER(PROGRAM=BIG),(U1110=OPT)
      IMPLICIT INTEGER(A-Z)
      REAL SOURCE(*),DUM(1),DUMY
      DEFINE DUMY(i)=DUM(i+OFF)

      SUMRED=0
      REMAIN=ITER
      NOWDS=1                        @ go forwards in source
      START=CVVA$F(SOURCE)            @ starting point in input array
      LOCDUM=LOCV$(DUM)               @ abs. addr. of local array DUM
C
C  Calculate a stripe size and address, set up for
C  base-offset referencing using DUMY, base the stripe,
C  do a sum reduction on the stripe.
C
1     START=MINE$F(START,NOWDS,NEXT)
      IF (NOWDS.EQ.0) RETURN
      IF (BITS(START,1,18).NE.0)CALL LBJ$IT(START)
      OFF=BITS(START,19,18)-LOCDUM    @ offset to stripe

      NUM=MIN(REMAIN,NOWDS)           @ # words in this stripe
      LITTLE=MOD(NUM,4)
      DO 10 I=1,LITTLE                @ do remains of MOD 4 first
10    SUMRED=SUMRED+DUMY(I)
      DO 20 I=LITTLE+1,NUM,4          @ do most of the stripe
20    SUMRED=SUMRED+DUMY(I)+DUMY(I+1)+DUMY(I+2)+DUMY(I+3)

      REMAIN=REMAIN-NOWDS
      IF(REMAIN.LE.0)RETURN
      START=NEXT
      GO TO 1
      END
```

This SUMRED example can be extended to have increments other than one. Similar routines can handle primitives other than sum reductions.

This program was executed on an 1100/80 system with an array size of 67,000 words. It was timed with the array in virtual space and in the control bank. The original program executed once with the simple loop and once with the modification to call SUMRED.

The following shows the 1100/80 system CPU timings in seconds:

	Nonvirtual Array (in Control Bank)	**Virtual Array (32K Banks)**
Original Program	0.1141	0.3118
Program Modified to Call SUMRED	0.0637	0.0638

There are 67,000 floating-point adds performed. Using the 0.0638- second time, that is over one million floating-point operations per second (one MFLOP).

Even the nonvirtual sum reduction has its CPU time cut approximately in half by calling SUMRED. This is due to the SUMRED routine stringing out the inner loop by doing four operations per iteration. The original program can also do this.

Example 2:

The following example shows the use of the MOVWD$ call to enhance performance on an inner loop:

```
          SUBROUTINE SUB(ARG,N)
          VIRTUAL/CX/
          COMMON/CX/C(2000,2000)
          REAL LOCAL(2000),ARG(2000)
          DO 10 I=1,2000

          LOCAL(I)=C(I,N)
    10    ARG(I)=C(I,N+1)
             .
             .
             .
```

The loop can be replaced by:

```
    CALL MOVWD$(C(1,N),1,LOCAL,1,1,2000)
    CALL MOVWD$(C(1,N+1),1,ARG,1,1,2000)
```

The subprogram SUB is called 100 times from a driver, where the N passed as the second argument is the loop index. Timings are done on the original program, the program modified to call MOVWD$, with variations of the argument array being in the control bank or in virtual space. The default page size (32K words) is used for virtual space.

The following shows the 1100/80 system CPU times in seconds:

	ARG Array in Control Bank	**ARG Array in Virtual Space**
Original Program With a Loop	1.508	3.226
Program Modified to Call MOVWD$	0.106	0.157

When MOVWD$ calls can replace a loop, it is usually much more efficient than the original loop. When the target or source items are banked objects, virtual objects, banked dummy arguments, or virtual dummy arguments, a speed-up factor of 10 to 20 can result. The number of words to be moved must be large enough to cover the setup cost. A call to MOVWD$ to move 40 words or less is of marginal benefit over regular compiled code.

Example 3:

MOVWD$ can also be used to broadcast a scalar in an array as the following example shows:

Original Program:

```
         REAL A(10000)
         COMPLEX C(20000),CX
         DO 10 I=1,10000
         A(I)=2.0
         C(I*2-1)=(1.0,0.)
    10   C(I*2)=CX
           .
```

```
        .
        .
```

This program can be modified as follows:

```
REAL A(10000)
COMPLEX C(20000),CX
CALL MOVWD$(2.0,0,A,1,1,10000)
CALL MOVWD$((1.0,0),0,C(1),2,2,10000)
CALL MOVWD$(CX,0,C(2),2,2,10000)
       .
       .
       .
```

This program segment executed 20 times in both the original and modified form.

The following shows the 1100/80 system CPU times in seconds:

	Nonvirtual (Arrays in Control Bank)	Virtual Arrays
Original Program	1.294	2.787
Modified Program to Call MOVWD$	0.410	0.414

M.17.5. User-Controlled Dynamic Storage Allocation Examples

The FORTRAN language has only static and dynamic data classes for user variables. Common is always in the static class, and local variables can be placed selectively into either the static or dynamic class. Often a program needs to obtain space in a more controlled manner than this, like PL/I's BASED storage. You can simply allocate portions of a larger array as space is needed.

Method 1:

```
      SUBROUTINE GETDAT(RSIZ,DATAI)
C
C          This routine is entered periodically to acquire a buffer of
C          size rsiz and initialize it.  Later, all of the buffers acquired
C          are processed elsewhere as a group.
C
      COMMON /BUFFRS/ BUFRS(40 000)
      INTEGER RSIZ,RECEND,NORECS,RECINX,RECSIZ
      REAL BUFRS,RECORD,DATAI
      COMMON /BUFDSC/ RECEND,NORECS,RECINX(500),RECSIZ(500)
      DEFINE RECORD(I) = BUFRS(I+RECINX(NORECS))
      DATA NORECS,RECEND/0,0/
      NORECS = NORECS + 1
      IF (NORECS .GT. 500 .OR. RECEND+RECSIZ .GT. 40000) THEN
         STOP 'Allocation overflow'
      ENDIF
      RECINX(NORECS) = RECEND            @ Remember record location
      RECSIZ(NORECS) = RSIZ              @ and its size.
      RECEND = RECEND + RSIZ             @ New start point for next time
      DO 10 I = 1,RSIZ                  @ Initialize new record
10    RECORD(I) = AMOD(FLOAT(I),DATAI)
      END
```

Other program units can access the various records in the buffer pool by using statement functions similar to RECORD in the preceding example.

If the 40,000-word area in the preceding example is not sufficient, the buffer area can be expanded and placed into virtual space as follows:

```
SUBROUTINE GETDAT(RSIZ,DATAI)
VIRTUAL /BUFFRS/
COMMON /BUFFRS/ BUFRS(4 000 000)          @ Note, 4 million words
     .
     .
     .
```

This method uses a large virtual object as a buffer source and has an unfortunate side effect. The D-banks used for virtual space are dynamically based and space is acquired dynamically by an MCORE$ Executive Request by the first program unit to reference the virtual object. When the amount of buffer space needed is quite variable (thereby forcing a large maximum size), but usually only a small amount of space, the additional virtual space acquired is wasted. The full declared size is allocated, resulting in unnecessary additional start-up time and an unnecessarily large allocation of system swap file. A method using the virtual storage allocation routines can be used that does not have these undesirable side effects. It is almost identical to the method using MCORF$ for acquiring control D-bank space dynamically.

Method 2:

```
      SUBROUTINE GETDAT(RSIZ,DATAI)
C
C           This routine is entered periodically to acquire a buffer of
C           size rsiz and initialize it.  Later, all of the buffers acquired
C           are processed elsewhere as a group.
C
      VIRTUAL /BUFFRS/
      COMMON /BUFFRS/ BUFRS(2050)
      INTEGER RSIZ,NORECS,RECINX,RECSIZ
      INTEGER LOCV$,SALC$P
      REAL BUFRS,RECORD,DATAI
      COMMON /BUFDSC/ NORECS,RECINX(500),RECSIZ(500)
      DEFINE RECORD(I) = BUFRS(I+RECINX(NORECS))
      DATA NORECS/0/
      NORECS = NORECS + 1
      IF (NORECS .GT. 500) THEN
           STOP 'Allocation overflow'
      ENDIF
C
C          Note:  We use the record index as the name of our
C          virtual allocation for this record.
C
      I = RSIZ
      RECINX(NORECS)=SALC$P(NORECS,I,.TRUE.,TRASH)-LOCV$(BUFRS)
C
C          “I” passed to SALC$P instead of RSIZ, since RSIZ could be
C          in virtual or banked space.
C
      RECSIZ(NORECS) = RSIZ
      DO 10 I = 1,RSIZ             @ Initialize new record
```

```
10    RECORD(I) = AMOD(FLOAT(I),DATAI)
      END
```

Note: *The various arguments passed to the virtual storage allocator must not be in virtual or in banked space.*

This program acquires a portion of virtual space dynamically for each record. There is no wasted space, no undesirable start-up overhead, and no unnecessary use of the system swap file. The sections of virtual space acquired can be separated from one another by allocations for virtual common (or static local) initiated by other subprograms.

One difference of the preceding method is that you can't easily do a collection process to compact your buffer space. When buffers must be capable of being released, it is better to use the first method of directly allocating out of an array or to use the modification described next.

There is a side effect of ASCII FORTRAN's virtual allocation mechanism that can eliminate the overhead of unused virtual space. When the virtual storage allocator is asked to allocate an object that is already in static virtual space, it attempts to expand the existing allocation when the requested size is greater than the existing size. A small amount of expansion is possible when there is unused space between the end of the current allocation and the start of the allocation of the next virtual object. Multiple open-ended expansions are possible for an existing virtual allocation when it is the last object that is allocated in virtual space. Your allocation routine then must simply ensure that the virtual object used for dynamic allocation of buffers has a size last presented to the virtual storage allocator that is sufficient to materialize the needed amount of space. To do this, use modifications of either of the two allocation methods previously described. The following example is an expansion of the second method.

Method 3:

```
      SUBROUTINE GETDAT(RSIZ,DATAI)
C
C           This routine is entered periodically to acquire a buffer of
C           size rsiz and initialize it.  Later, all of the buffers acquired
C           are processed elsewhere as a group.
C           This routine expands an existing virtual allocation.  No
C           static virtual space can be allocated by any other subprogram
C           once this subroutine is entered.
C
      VIRTUAL /BUFFRS/
      COMMON /BUFFRS/ BUFRS(2050)
      INTEGER RSIZ,RECEND,RECMAX,NORECS,RECINX,RECSIZ,OFFSET
      INTEGER LOCV$,SALC$P
      REAL BUFRS,RECORD,DATAI
      COMMON/BUFDSC/RECEND,RECMAX,NORECS,RECINX(500),RECSIZ(500)
      DEFINE RECORD(I) = BUFRS(I+RECINX(NORECS))
      DATA NORECS,RECEND,RECMAX/0,0,-33 000 000/
      NORECS = NORECS + 1
      IF (NORECS .GT. 500) THEN
           STOP 'Allocation overflow'
      ENDIF
     IF (RECEND+RSIZ .GE. RECMAX) THEN
      RECMAX = RECEND + RSIZ          @ Expanded size of $MINE$
      OFFSET = SALC$P(FDCBNM,RECMAX,.TRUE.,TRASH)-LOCV$(BUFRS)
      DATA FDCBNM/'$MINE$'F/          @ Fieldata name of our buffer
      IF (RECEND .EQ. 0) THEN
```

```
C
C       First entry: Set RECEND to get at the beginning of our $MINE$
C       buffer area by base-offset referencing of BUFFRS in common block
C       BUFFRS.  We could have expanded the BUFFRS common
C       block allocation also, passing BUFFRS in Fieldata to SALC$P.
C       Then the variable OFFSET would not be needed.  However, other
C       subprograms may have already allocated the common block BUFFRS
C       and they may also have caused subsequent allocations in virtual
C       space, thus ensuring that BUFFRS could not be expanded when
C       needed.
C
          RECEND = OFFSET
      ENDIF
   ENDIF
   RECINX(NORECS) = RECEND            @ Remember record location
   RECSIZ(NORECS) = RSIZ              @ and its size
   RECEND = RECEND + RSIZ             @ Start point for next request
   DO 10 I = 1,RSIZ                   @ Initialize new record
10 RECORD(I) = AMOD(FLOAT(I),DATAI)
   END
```

This method has no extra start-up overhead and causes no unnecessary swap file load. You can easily compact your buffer pool by supplying simple buffer release and buffer compaction routines.

Bibliography

OS 2200 Exec System Software Installation and Configuration Guide (7830 7915). Unisys Corporation.

OS 2200 Exec System Software Executive Requests Programming Reference Manual (7830 7899). Unisys Corporation.

OS 2200 Sort/Merge Programming Guide, Level 17R1 (7831 0687). Unisys Corporation.

Index

Index

Index

Index