HP 3000 Computer Systems

FORTRAN / 3000

Reference Manual

HEWLETT
PACKARD

5303 STEVENS CREEK BLVD., SANTA CLARA, CALIFORNIA, 95050

Part No. 30000-90040
Product No. 32102B

Printed in U.S.A. 6/76
Update No. 3 Incorporated 5/79

LIST OF EFFECTIVE PAGES

The List of Effective Pages gives the date of the current edition and of any pages changed in updates to that edition. Within the manual, any page changed since the last edition is indicated by printing the date the changes were made on the bottom of the page. Changes are marked with a vertical bar in the margin. If an update is incorporated when an edition is reprinted, these bars are removed but the dates remain. No information is incorporated into a reprinting unless it appears as a prior update.

First Edition. . . . . Jun 1976

Changed Pages	Effective Date	Changed Pages	Effective Date
title to iv	May 1979	7-26	Feb 1977
ix to x	Apr 1978	7-30	Feb 1977
1-3	Feb 1977	7-31	May 1979
1-5	Feb 1977	7-34	Feb 1977
1-6	Apr 1978	8-7	Feb 1977
2-1 to 2-2	Apr 1978	8-10	May 1979
2-3 to 2-4	Feb 1977	9-1	Feb 1977
2-7	Feb 1977	9-2 to 9-3	Apr 1978
2-8	May 1979	9-4	May 1979
3-1	Feb 1977	9-4a	Apr 1978
3-2 to 3-5	Apr 1978	9-7 to 9-8	May 1979
3-6	May 1979	9-9	Apr 1978
3-7 to 3-9	Apr 1978	10-1	Feb 1977
4-4	Feb 1977	10-2	May 1979
4-14	May 1979	10-5 to 10-6	Feb 1977
4-15	Feb 1977	11-3 to 11-4	Feb 1977
4-19	Apr 1978	11-6 to 11-6a	Apr 1978
4-21	Feb 1977	11-8 to 11-10	Feb 1977
5-1 to 5-1a	May 1979	11-11	Apr 1978
5-3	Feb 1977	12-16	Apr 1978
5-6 to 5-8	Apr 1978	A-2	Feb 1977
5-10 to 5-12	May 1979	A-10	Feb 1977
5-13	Apr 1978	B-2 to B-3	Apr 1978
5-14	May 1979	D-7	Feb 1977
5-16 to 5-17	Apr 1978	D-11 to D-11a	Apr 1978
6-1	Feb 1977	D-16	Feb 1977
6-10	Feb 1977	D-17 to D-18	Apr 1978
6-17	Feb 1977	F-1 to F-8	Apr 1978
6-18	May 1979	I-1 to I-5	May 1979
7-1	May 1979	I-6	Apr 1978
7-3	Feb 1977	I-7	May 1979
7-7	May 1979	I-8	Apr 1978
7-17 to 7-18	Feb 1977		

PRINTING HISTORY

New editions are complete revisions of the manual. Update packages, which are issued between editions, contain additional and replacement pages to be merged into the manual by the customer. The date on the title page and back cover of the manual changes only when a new edition is published. When an edition is reprinted, all the prior updates to the edition are incorporated. No information is incorporated into a reprinting unless it appears as a prior update. The edition does not change.

The software product part number printed alongside the date indicates the version and update level of the software product at the time the manual edition or update was issued. Many product updates and fixes do not require manual changes, and conversely, manual corrections may be done without accompanying product changes. Therefore, do not expect a one to one correspondence between product updates and manual updates.

First Edition	Jun 1976	32102B.00
Update Package No. 1	Feb 1977	32102B.00
Update No. 1 Incorporated	May 1977	32102B.00
Update Package No. 2	Apr 1978	32102B.00
Update No. 3 Incorporated	May 1979	32102B.00

PREFACE

This publication is the reference manual for the HP 3000 Computer System FORTRAN programming language (FORTRAN/3000).

The wide range of users of publications such as this was considered during its preparation and it is hoped that the needs of all, from the very experienced programmer to those with little or no experience, have been met. The cross-reference index at the back of the manual should allow the experienced user to look up the syntax of the various statements very quickly. The syntax of all statements is highlighted throughout the manual by grey shading which, when the referenced page is found, eliminates the need to search for the necessary information. Simple (but complete and operational) programs are used to demonstrate usage of FORTRAN/3000 statements and concepts.

This publication contains the following sections:

Section I -	is an introduction to FORTRAN/3000. The various types of statements are discussed and simple source programs are shown in "fixed-field" and "free-field" formats.
Section II -	describes FORTRAN/3000 data storage formats and the concepts of constants and variables.
Section III -	discusses expressions (arithmetic, character, and logical) and assignment statements in FORTRAN/3000.
Section IV -	describes *control* statements in FORTRAN/3000, including GO TO, IF, DO, CONTINUE, STOP, END, PAUSE, CALL, and RETURN statements. It also describes trap handling.
Section V -	describes *declaration* statements. Type, PARAMETER, DIMENSION, EQUIVALENCE, COMMON, IMPLICIT, EXTERNAL, and DATA statements are discussed. In addition, *statement functions are described.*
Section VI -	discusses *input/output* statements and covers READ, WRITE, ACCEPT, and DISPLAY input and output statements; DO-implied lists; and the auxiliary input/output statements REWIND, BACKSPACE, and ENDFILE.
Section VII -	covers FORMAT statements and the use of the FORTRAN/3000 Formatter.
Section VIII -	discusses the MPE file system and provides example programs showing the use of the file system intrinsics FOPEN, FCLOSE, and FREAD. Also described are the FORTRAN Logical Unit Table (FLUT), and the HP 3000 Compiler Library procedures FSET, FNUM, and UNITCONTROL.
Section IX -	describes the FORTRAN/3000 compiler commands $CONTROL, $PAGE, $INTEGER*4, $TITLE, $SET, $IF, $EDIT, and $TRACE.
Section X -	discusses FORTRAN/3000 intrinsic functions, basic external functions, and generic functions.
Section XI -	discusses FORTRAN/3000 main programs; dummy and actual argument characteristics; and subroutine, function, and block data subprograms.
Section XII -	demonstrates how to compile, prepare, and execute FORTRAN/3000 source programs. The MPE :JOB, :HELLO, :FILE, :BUILD, :PURGE, :CONTINUE, :FORTRAN, :FORTPREP, :FORTGO, :PREP, :PREPRUN, :RUN, :EOD, :EOJ, and :BYE commands are discussed. This section also shows how to create and maintain relocatable libraries (RL's) and segmented libraries (SL's), and how to call procedures in these libraries from FORtran/3000 programs.

Additionally, six appendices and one index are provided, as follows:

Appendix A -	discusses non-FORTRAN/3000 program units and MPE system intrinsics.
Appendix B -	outlines differences between FORTRAN/3000 and the American National Standard Institute (ANSI) FORTRAN IV.
Appendix C -	lists differences between FORTRAN/3000 and HP 2100 FORTRAN.
Appendix D -	lists and describes error and warning messages.

Appendix E -	contains the American Standard Code for Information Interchange (ASCII) character set.
Appendix F -	contains various techniques to optimize the use of the FORTRAN compiler.
Index -	is a cross-reference index of all topics covered in this manual.

Other publications which should be available for reference when using this manual are:

MPE Commands Reference Manual (30000-90009)

MPE Intrinsics Reference Manual (30000-90010)

MPE Segmenter Reference Manual (30000-90011)

Compiler Library Reference Manual (03000-90028)

TRACE/3000 Reference Manual (03000-90015)

EDIT/3000 Reference Manual (03000-90012)

Systems Programming Language Reference Manual (SPL) (30000-90024)

CONTENTS

CONTENTS (continued)

ILLUSTRATIONS

TABLES

INTRODUCING FORTRAN/3000

SECTION I

1-1. WHAT IS FORTRAN/3000?

Hewlett-Packard FORTRAN/3000 language is based on the American National Standard Institute's Standard FORTRAN (X3.9 - 1966) and is used with the HP 3000 Computer System.

To provide a more powerful programming language, FORTRAN/3000 extends beyond the Standard FORTRAN. Some of these extensions are:

- Program entry can occur in a free-field format as well as a fixed-field format.
- Symbolic names can consist of as many as 15 characters instead of just six.
- Subprograms written in other programming languages, notably SPL/3000 (Systems Programming Language for HP 3000), can be called by FORTRAN/3000 statements.

A complete list of FORTRAN/3000 extensions beyond Standard FORTRAN (and some minor restrictions to conform with the HP 3000 Computer System architecture) is presented in Appendix B of this manual.

1-2. A FORTRAN/3000 SOURCE PROGRAM

FORTRAN/3000 code which is entered into the computer from, for example, a card reader or a terminal,is the *source* program.

The source program is translated into internal form (*object* program) by the HP 3000 Computer System FORTRAN/3000 *compiler* and this object program is stored on disc. The *segmenter* subsystem then *prepares* the object program into linked code segments, the *loader* binds the segments from the program file to referenced external segments from a library and *creates a process* to run the program, and, finally, the *dispatcher* schedules the *execution* of the program.

A short FORTRAN/3000 source program is shown in figure 1-1 in coded form preparatory to being entered into the computer. The program increments a value from 1.0 to 10.0 and computes and prints the square root and the reciprocal of the value.

The four lines following PROGRAM FIXED are *comments*. Comments are non-executable and are inserted in a program to assist in understanding the purpose of the program. The next two lines (10 and 20) are *FORMAT* statements which specify the format of the data to be printed out. The *WRITE* statement prints out the headings NUMBER, SQUARE ROOT, and RECIPROCAL in accordance with the format defined in FORMAT statement number 10. (The other number (6) in parentheses in the *WRITE* statement specifies the FORTRAN/3000 logical unit to be used for the printout.)

The *DO* statement (DO 30 I = 1,10) causes all executable statements succeeding it (through statement number 30) to be executed 10 times. The statement, ROOT = SQRT(A), calls the external function SQRT to compute the square root of A. The first time the DO loop is executed, A = 1.0; and A is incremented each time the DO loop is repeated until, after 10 iterations, A = 10.0. The statement, RCPL = 1/A, computes the reciprocal of A by dividing it into 1. The *WRITE* statement prints the value of A and the square root and the reciprocal of the value. Statement number 30 increments A by 1.0 and the DO loop then is repeated.

The *STOP* statement causes termination of program execution and the *END* statement informs the compiler that there are no more lines of code in this program unit.

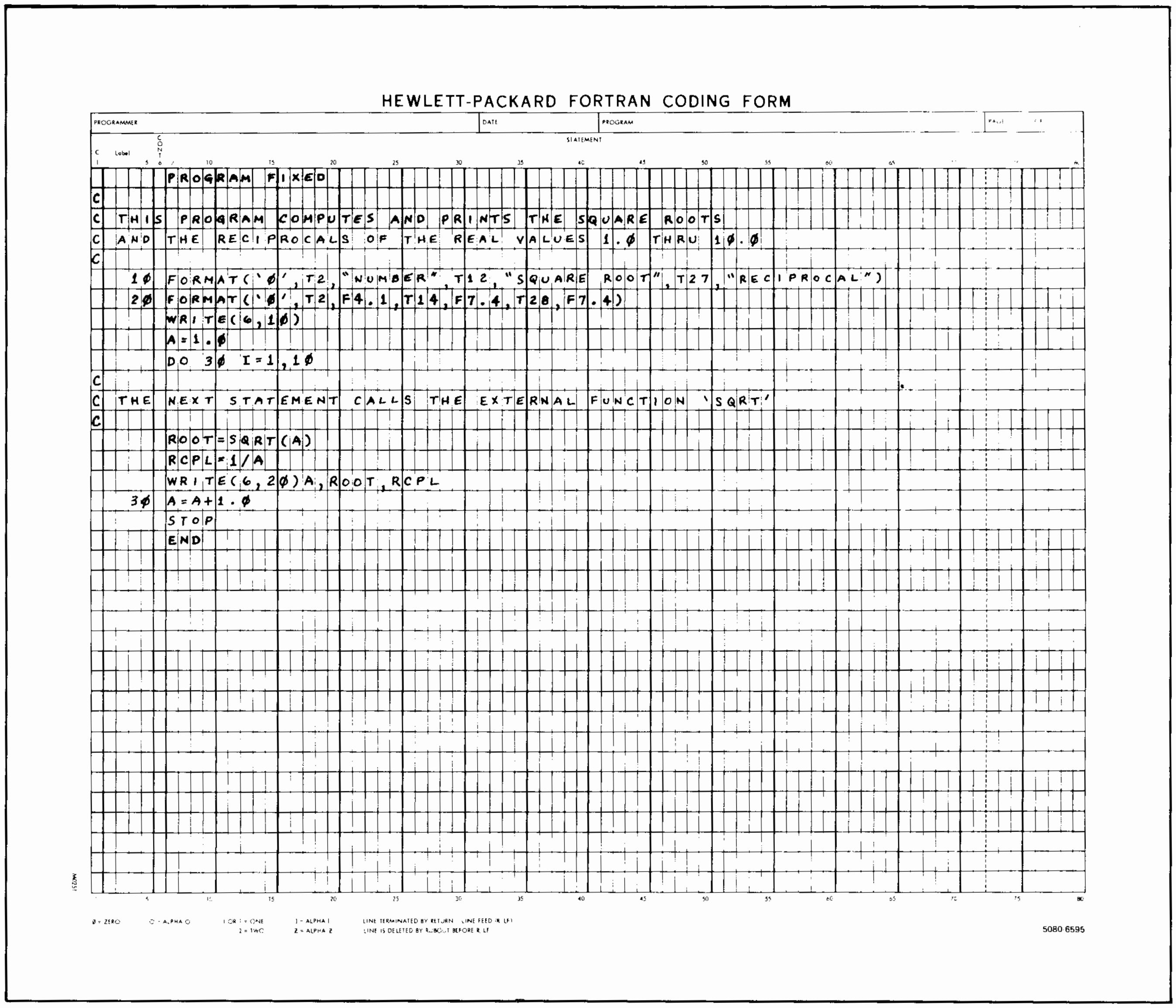
1-3. FORTRAN/3000 CHARACTER SET

A FORTRAN/3000 source program is written using alphabetic characters A through Z, numeric characters 0 through 9, and any other printing ASCII character. See Appendix E for a listing of the complete ASCII character set.

Blanks may be used anywhere within the body of a FORTRAN/3000 statement. They are ignored by the compiler except in Hollerith and string constants, where they represent blank characters. Those cases where blanks have special meanings when used outside the body of a FORTRAN/3000 statement are explained in this manual where applicable.

1-4. SOURCE PROGRAM CONTENT

FORTRAN/3000 source programs are composed of lines of code signifying *statements*, *comments*, or *compiler commands*. The lines of code are arranged into a *main program*, and (as necessary) *subroutine subprograms, function subprograms,* and *block data subprograms.* A *main program* is a set of FORTRAN/3000 statements and comments not containing a FUNCTION or SUBROUTINE statement. A *subroutine subprogram* is defined by FORTRAN/3000 statements and headed by a SUBROUTINE statement. A *function subprogram* is defined by



HEWLETT-PACKARD FORTRAN CODING FORM

```
      PROGRAM FIXED
C
C THIS PROGRAM COMPUTES AND PRINTS THE SQUARE ROOTS
C AND THE RECIPROCALS OF THE REAL VALUES 1.Ø THRU 1Ø.Ø
C
   1Ø FORMAT('Ø',T2,"NUMBER",T12,"SQUARE ROOT",T27,"RECIPROCAL")
   2Ø FORMAT('Ø',T2,F4.1,T14,F7.4,T28,F7.4)
      WRITE(6,1Ø)
      A=1.Ø
      DO 3Ø I=1,1Ø
C
C THE NEXT STATEMENT CALLS THE EXTERNAL FUNCTION 'SQRT'
C
      ROOT=SQRT(A)
      RCPL=1/A
      WRITE(6,2Ø)A,ROOT,RCPL
   3Ø A=A+1.Ø
      STOP
      END
```

5080 6595

Figure 1-1. A FORTRAN/3000 Source Program

FORTRAN/3000 statements and headed by a FUNCTION statement. A *block data subprogram* consists of a BLOCK DATA statement, and, as necessary, IMPLICIT, COMMON, DIMENSION, EQUIVALENCE, Type, and DATA statements.

The general order of a FORTRAN/3000 source program, shown punched on cards in figure 1-2, is as follows:

1. *Compiler commands* which direct compiler action.

2. *Declaration* (or specification) statements which define the characteristics of data used in the program. Declaration statements are non-executable and,if used, must appear before the first executable statement in each program unit.

3. *Executable* statements. Some types of executable statements are:

   ARITHMETIC statements which specify numeric calculations.
   LOGICAL statements which specify logical calculations.
   GO TO statements which pass control from one statement to another statement in the same program unit.
   IF statements which evaluate an expression and transfer control to other statements depending upon the result of the evaluation.
   DO statements which cause statements to be repeated a specific number of times.
   INPUT/OUTPUT statements which transfer information between memory and external files, or between different locations in memory.

FORMAT statements which, although not executable, are included in a source program to specify the format of data to be transferred.

4. *Comments,* which are non-executable and merely aid in interpreting the purpose of the program. Comments may be inserted anywhere in a source program.

5. *Subprograms,* which are self-contained computational procedures that require activation by the main program or another subprogram, are included in source programs to perform special functions such as solving a mathematical problem or performing a sort.

6. *END* statements. All separately compiled program units (main programs and subprograms) must be terminated by an END statement.

1-5. SOURCE PROGRAM FORMAT

A line of FORTRAN/3000 code can contain as many as 80 characters (including blanks). The character positions (columns) are numbered from 1 to 80. Characters placed in columns 73 through 80 (fixed-field format, see paragraph 1-6) or the characters up through the first blank (free-field format, see paragraph 1-12) are not part of the program body (text), but are used by the compiler for sequencing information.

1-6. FIXED-FIELD FORMAT

In fixed-field format, the characters in FORTRAN/3000 lines of code must be placed in specific column order. (Refer to the source program shown in figure 1-1.) The purpose of the columns is explained in the following paragraphs.

1-7. STATEMENTS. A FORTRAN/3000 statement can be of two types: *executable* and *non-executable.* Executable statements specify action that the program is to take; non-executable statements contain information such as the characteristics of operands, types of data, and format specifications for input/output information.

The body (text) of a FORTRAN/3000 statement in fixed-field format must start in column 7 and must not extend beyond column 72. (Columns 73 through 80 can be used for sequencing information but are not part of the statement text.) A single statement may consist of from one to 20 lines. The first line of a statement must contain a zero or blank in column 6. Continuation lines must contain any character other than blank or zero in column 6. As with the first line of a statement, the text for statement continuation lines must start at column 7 and cannot extend beyond column 72.

1-8. Statement Labels. A statement can be labeled so that it can be referred to by other statements in the program unit. (See the fourth line of the first example in figure 1-3.) A label consists of a number in the range of from 1 to 99999 in columns 1 through 5. Embedded blanks and leading zeros are ignored by the compiler. For example, 00070, 070, 0b070, and 70 can be referenced elsewhere in the program unit as 00070, 070, or 70.

Note: Where necessary for clarity, blank characters are denoted by the letter *b* throughout this manual.

Only the first line of a multi-line statement is labeled. If a statement is not labeled, the first five columns of the line must remain blank.

1-9. COMMENTS. Comments are inserted in a FORTRAN/3000 source program to aid in interpreting the purpose of the program. Comments can be included between statements or compiler commands but not between lines of multi-line statements or compiler commands. A comment line must contain a C in column 1; positions 2 through 72 contain the text of the comment. Each line of a multi-line comment must contain a C in column 1. The text of comment lines cannot extend beyond column 72.

1-10. COMPILER COMMANDS. Compiler commands are not part of the program proper, but are included with the source program to indicate compiler options, such as suppressing listings or diagnostic messages. A compiler command starts with a $ in column 1 and columns 2 through 72 are available for the various compiler commands. Commas (plus blanks if desired) separate each individual option. If a compiler command requires more than one line, the initial line and each line which must be continued must end with an ampersand (&) and each line must begin with a $ in column 1.

1-11. SEQUENCING INFORMATION. Columns 73 through 80 can be used for sequencing information and may contain any alphanumeric characters. Sequencing information allows the programmer to keep track of the correct order of the FORTRAN/3000 source program and is especially helpful when the source program is on cards. Sequencing information is ignored by the compiler unless the $EDIT command is inserted in the program as a compiler command. When the $EDIT command is used, the following editing capabilities are available.

- Merging a correction program with an old program to produce a new source program for compilation.
- Checking source program sequence for ascending order.
- Bypassing sections of a source program.

A detailed description of the $EDIT command is contained in Section IX.

1-12. FREE-FIELD FORMAT

When entering a FORTRAN/3000 source program from a terminal such as a teleprinter, the fixed-field format often is inconvenient because of the difficulty in determining column positions. To overcome this problem, FORTRAN/3000

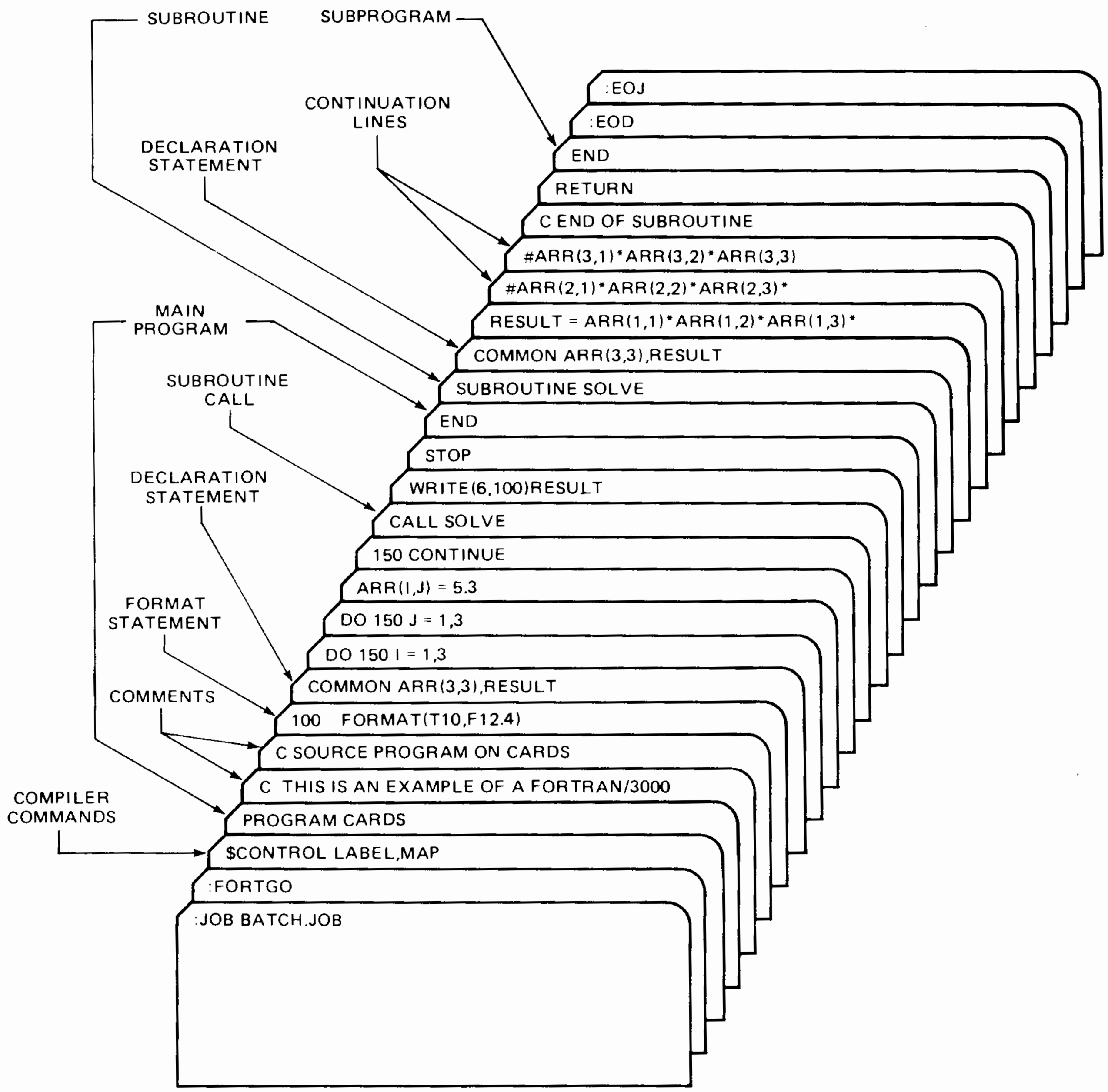
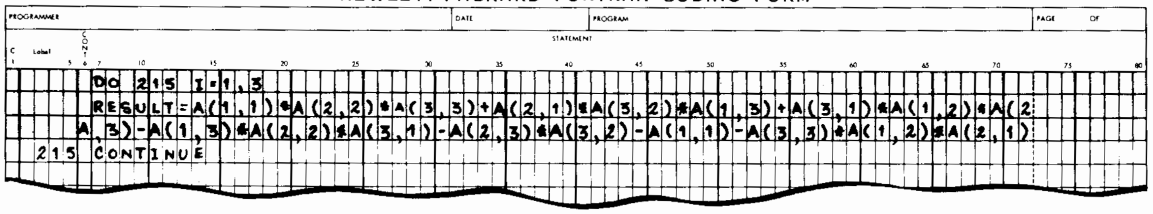


Figure 1-2. FORTRAN/3000 Source Program Punched on Cards

STATEMENTS

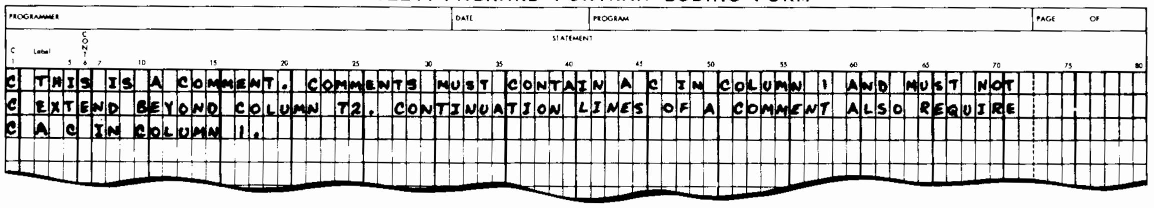


HEWLETT-PACKARD FORTRAN CODING FORM

PROGRAMMER | DATE | PROGRAM | PAGE OF

```
      DO 215 I=1,3
      RESULT=A(1,1)*A(2,2)*A(3,3)+A(2,1)*A(3,2)*A(1,3)+A(3,1)*A(1,2)*A(2
     A,3)-A(1,3)*A(2,2)*A(3,1)-A(2,3)*A(3,2)-A(1,1)-A(3,3)*A(1,2)*A(2,1)
  215 CONTINUE
```

COMMENTS

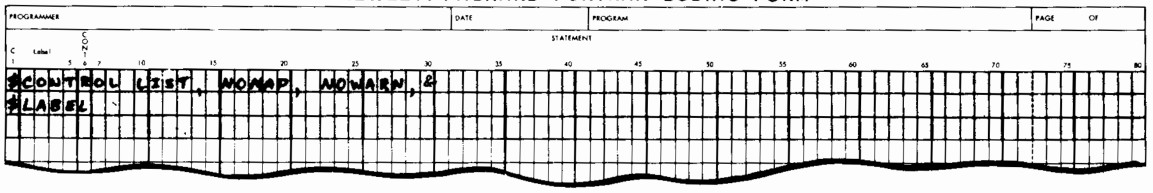


HEWLETT-PACKARD FORTRAN CODING FORM

PROGRAMMER | DATE | PROGRAM | PAGE OF

```
C THIS IS A COMMENT. COMMENTS MUST CONTAIN A C IN COLUMN 1 AND MUST NOT
C EXTEND BEYOND COLUMN 72. CONTINUATION LINES OF A COMMENT ALSO REQUIRE
C A C IN COLUMN 1.
```

COMPILER COMMANDS

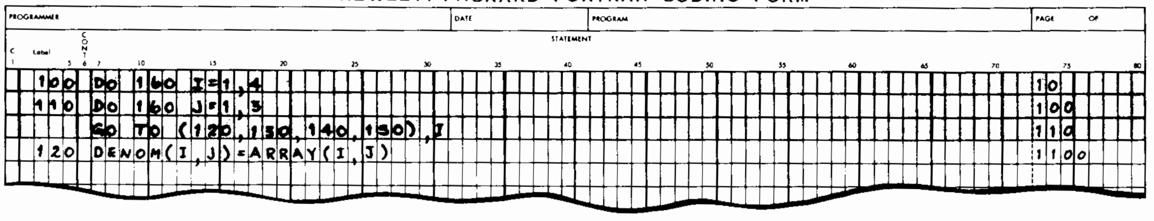


HEWLETT-PACKARD FORTRAN CODING FORM

PROGRAMMER | DATE | PROGRAM | PAGE OF

```
$CONTROL LIST, NOMAP, NOWARN, &
$LABEL
```

SEQUENCING INFORMATION



HEWLETT-PACKARD FORTRAN CODING FORM

PROGRAMMER | DATE | PROGRAM | PAGE OF

```
  100 DO 160 I=1,4                                                      10
  110 DO 160 J=1,3                                                      100
      GO TO (120,150,140,150),I                                         110
  120 DENOM(I,J)=ARRAY(I,J)                                             1100
```

Figure 1-3. Coding Examples

source programs can be entered into the computer in free-field format. Each line of a source program entered in free-field format consists of up to 80 characters. Each line begins with a sequence field (corresponding to positions 73 through 80 in fixed-field format). The sequence field extends up to (but does not include) the first blank in the line and cannot exceed eight characters. Sequence fields must start in position 1. (A blank in position 1 causes the compiler to treat the entire sequence field as blank.) A sequence string of less than eight characters is treated as being right justified by the compiler. A sequence field longer than eight characters is truncated from the left.

The remaining positions (starting with the first position after the blank terminating the sequence string) make up a FORTRAN/3000 line in free-field format. If position 1 is blank, indicating that the sequence field is not used, lines can start at position 2; however, no more than 71 characters may be used for such lines because the compiler treats the line as if it were started in position 10.

Comment lines are indicated by a hatch mark (#) in the first position following the sequence field and blank. Comment continuation lines also must contain a # in the first position following the sequence field and blank. Compiler commands are indicated by a $ as the first character in the first position following the sequence field and blank.

Note: The only exception is the compiler command $CONTROL FREE, which starts in position 1.

If compiler commands or FORTRAN/3000 statements require continuation lines, each line except the last (i.e., each line to be continued) must end with an &. In addition, each compiler command continuation line must begin with a $ following the sequence field and blank.

FORTRAN/3000 statements can be labeled in the free-field format. If the first character following the sequence field and blank is a numeric character, then this character begins a statement label. The label can be more than five characters long (including leading zeros and embedded blanks) but cannot exceed 99999 in value. A blank may be used between the label and the first character of the statement text. The first non-blank, non-numeric character following the sequence field starts the text of the FORTRAN/3000 statement.

The source program shown earlier in figure 1-1 is shown in free-field format in figure 1-4. Note that the first statement ($CONTROL FREE) is output starting in column 10 by the compiler. This is merely the way the compiler outputs this command, the command started in position 1 in the source program input.

```
:FORTGO FTRAN1

PAGE ØØØ1   HP321Ø2B.ØØ.Ø


              $CONTROL FREE
 #
 # FREE-FIELD FORMAT EXAMPLE
 #
 # THIS PROGRAM COMPUTES AND PRINTS THE SQUARE ROOTS
 # AND THE RECIPROCALS OF THE REAL VALUES 1.Ø THRU 1Ø.Ø
 #
 1Ø FORMAT('Ø',T2,"NUMBER",T12,"SQUARE ROOT",T27,"RECIPROCAL"//)
 2Ø FORMAT(T2,F4.1,T14,F7.4,T28,F7.4)
 WRITE(6,1Ø)
 A=1.Ø
 DO 3Ø I=1,1Ø
 #
 # THE NEXT STATEMENT CALLS THE EXTERNAL FUNCTION 'SQRT'
 #
 ROOT=SQRT(A)
 RCPL=1/A
 WRITE(6,2Ø)A,ROOT,RCPL
 3Ø A=A+1.Ø
 STOP
 END



 ****       GLOBAL STATISTICS        ****
 ****    NO ERRORS,    NO WARNINGS   ****
 TOTAL COMPILATION TIME  Ø:ØØ:Ø1
 TOTAL ELAPSED TIME      Ø:ØØ:Ø6

 END OF COMPILE

 END OF PREPARE



NUMBER     SQUARE ROOT    RECIPROCAL


 1.Ø           1.ØØØØ         1.ØØØØ
 2.Ø           1.4142          .5ØØØ
 3.Ø           1.7321          .3333
 4.Ø           2.ØØØØ          .25ØØ
 5.Ø           2.2361          .2ØØØ
 6.Ø           2.4495          .1667
 7.Ø           2.6458          .1429
 8.Ø           2.8284          .125Ø
 9.Ø           3.ØØØØ          .1111
1Ø.Ø           3.1623          .1ØØØ
 END OF PROGRAM
```

Figure 1-4. FORTRAN/3000 Source Program in Free-Field Format

ELEMENTS OF FORTRAN/3000

SECTION II

2-1. DATA STORAGE FORMATS

FORTRAN/3000 processes seven types of data-integer, double integer, real (floating point), double precision real, logical, complex, and character.

Each of the seven data types differs in the way it is represented in memory. The following paragraphs describe the data types and discuss the manner in which they are stored in memory.

2-2. INTEGER FORMAT

Integers are whole numbers containing no fractional part. Integer values are stored in one 16-bit computer word. The leftmost bit (bit 0) represents the arithmetic sign of the number (1 = negative, 0 = positive). The other 15 bits represent the binary value of the number. Integer numbers are represented in two's complement form and the *range* of integer numbers is from -32768 to +32767.

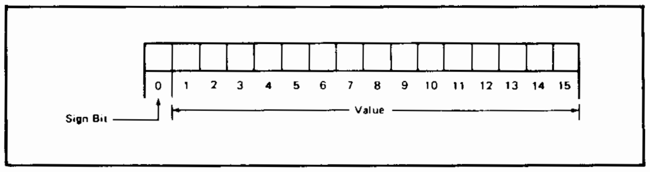


Figure 2-1. Internal Representation of Integer Values

2-3. DOUBLE INTEGER FORMAT

Double integers are represented in memory by 32 bits (two consecutive 16-bit words). The first one is called the most significant word (MSW) and the second one is called the least significant word (LSW). Representation is in two's complement form. The *range* of a double integer number is from -2147483647 to +2,147,483,646.

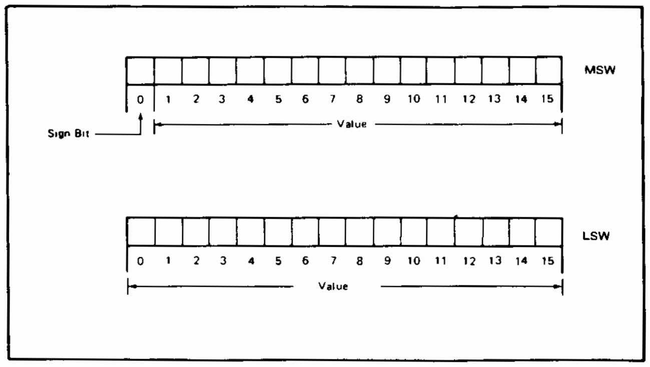


Figure 2-2. Internal Representation of Double Integer Values.

2-4. REAL (FLOATING-POINT) FORMAT

Real numbers are represented in memory by 32 bits (two consecutive 16-bit words) with three fields. These fields are the sign, the exponent, and the mantissa. The format is that known as excess 256. Thus, a real number consists of:

Sign (S)
Bit 0 for the first word. Positive = 0, negative = 1. A value X and its negative, -X, differ only in the sign bit.

Exponent (E)
Bits 1 through 9 of the first word. The exponent ranges from 0 to 777 octal (511 decimal). This number (E) represents a binary exponent, biased by 400 octal (256 decimal). The true exponent, therefore is E - 256; it ranges from -256 to +255.

Fraction (F)
A binary number of the form 1.*xxx*, where *xxx* is 22 bits, stored in bits 10 through 15 of the first word and all of the bits of the second word. Note that the 1.0 is not actually stored, there is an assumed 1 to the left of the binary point. Floating-point zero is the only exception. It is represented by all 32 bits being zero.

The *range* of non-zero real values is from 0.863617 x 10^{-77} to .1157920x 10^{78}. The formula for computing the decimal value of a floating-point representation is: Decimal value = $(-1)^S * 2^{(E-256)} * F$.

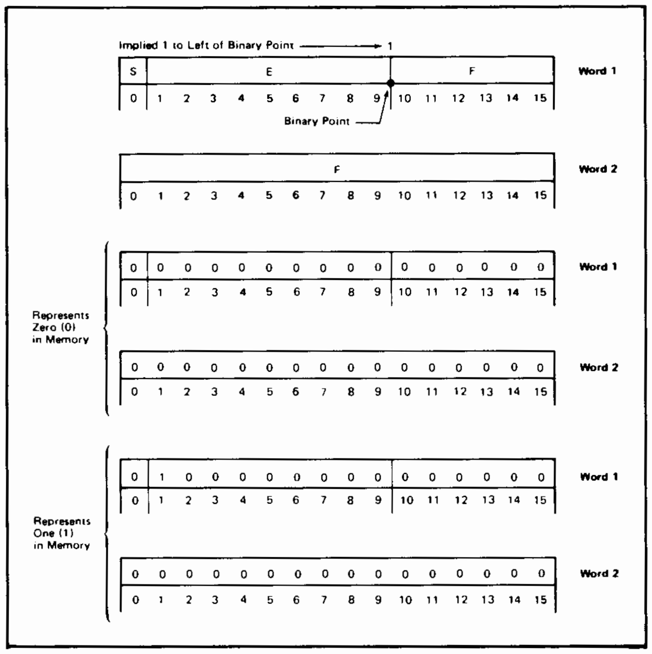


Figure 2-3. Internal Representation of Real Values

2-5. DOUBLE PRECISION FORMAT

Double precision values are stored in four consecutive 16-bit computer words and are identical to real (single precision) values except that the fractional part (mantissa) of the number is extended from 22 to 54 binary bits. The format is that known as excess 256. The *range* of non-zero double precision values is from .8636168555094445x10^{-77} to 0.1157920892373161x10^{78}.

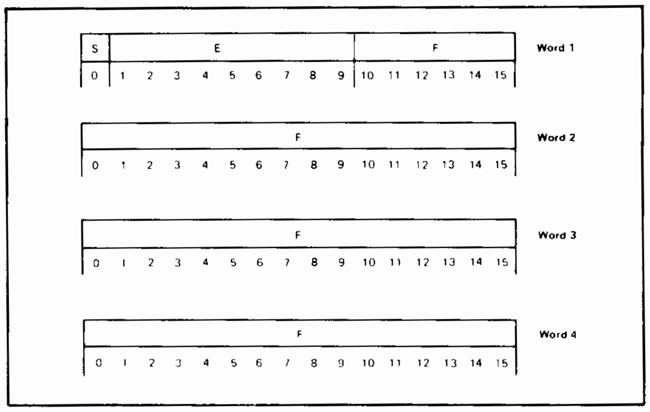


Figure 2-4. Internal Representation of Double Precision Values

2-6. LOGICAL FORMAT

Logical values are stored in one 16-bit computer word. Normally, only two values are used for logical constants: .TRUE. and .FALSE. .TRUE. is represented by all 16 bits being equal to 1; .FALSE. is represented by all 16 bits being equal to 0. Testing a logical value for .TRUE. or .FALSE., however, is performed by examining only bit 15. Any other pattern of bits (other than all 1's or 0's) can be used with logical operators to perform masking operations (see Section III).

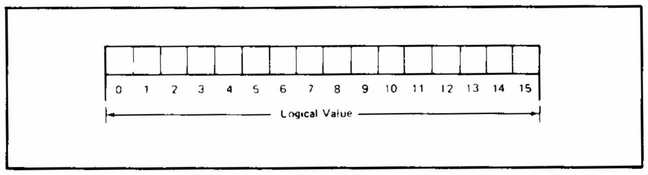


Figure 2-5. Internal Representation of Logical Values

2-7. COMPLEX FORMAT

A complex value consists of an ordered pair of real numbers, specifying the real and imaginary parts. Complex numbers are stored in four consecutive 16-bit computer words; the first two words are for the real part, the second two words are for the real number specifying the imaginary part. The two numbers specifying the complex value must be real.

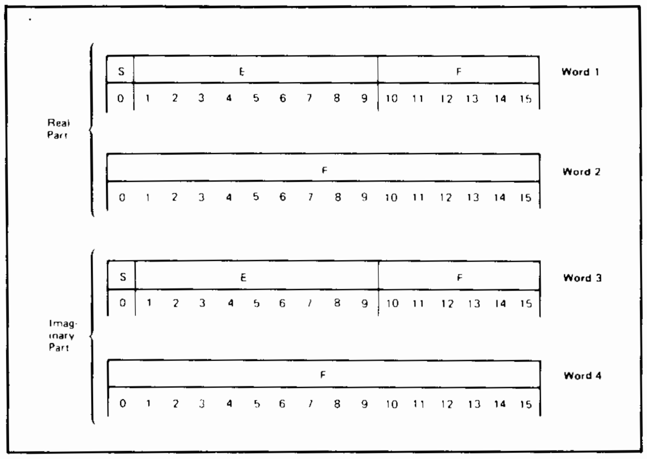


Figure 2-6. Internal Representation of Complex Values

2-8. CHARACTER FORMAT

Character values represent ASCII character strings which are manipulated using character expressions. (Input/ output statements frequently use character values. See Section VI.)

Character values are represented by 8-bit ASCII codes, two characters packed in one 16-bit computer word. The number of words used to represent a character value depends on the actual number of characters in the source representation of the value.

Character values can be specified by an octal number representing a character bit pattern. The octal number is followed by the letter C. For example,

%101C represents the character A
%102C represents character B (the % sign signifies an octal value).

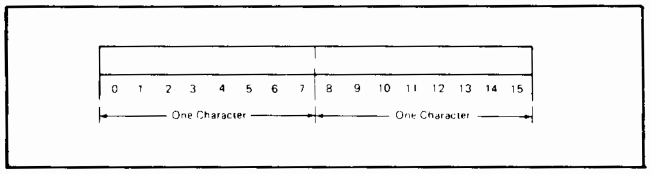


Figure 2-7. Internal Representation of Character Values

2-9. CONSTANTS

A constant is a data element representing one specific value which remains unchanged throughout the program. Thus, in the expression A + 4, the numerical quantity 4 is a constant. (A is a variable and is described in paragraph 2-19.)

A constant can be of several types: integer, double integer, real (floating point), double precision real, complex, logical, or character.

2-10. INTEGER CONSTANTS

Integers are signed whole numbers containing no fractional part. Integers may be specified in four ways: decimal, octal, ASCII, and composite. Decimal integer constants use the decimal digits 0 through 9. They can contain a leading plus (+) or minus (−) sign (a number with no leading sign is positive). The range of a decimal integer constant is from −32767 to +32767. The decimal constant −32768 can be represented in the machine, but can not be directly represented in the FORTRAN source code.

For example,

```
0
+45
-365
4012
```

Octal integer constants are denoted by the % character. They may contain up to six octal digits and an optional leading sign. The number ranges from -100000_8 to $+77777_8$. For example,

```
%4777
+%605
-%17
%177777 (This is -1)
%100000 (this is -32768₁₀)
```

ASCII integer constants are used to write one or two ASCII bit patterns into one 16-bit computer word. ASCII integer constants are written with the % character followed by the ASCII characters enclosed in quote marks (also called string bracket characters) or apostrophes.

For example,

```
+%"AB"
-%"U"
%"HI"
%"L"
-%'XY'
+%'A'
```

If only one ASCII character is specified, the bit pattern representing that character is placed in the computer word right-justified, and the left half of the word is filled with leading zeros. Leading plus and minus signs can be used. Integer constants also can be specified in composite form. See paragraph 2-17.

Note: For FORTRAN programs which currently run on other machines requiring 32 bit integers, an aid has been implemented. Namely the $INTEGER*4 command which will cause the compiler to treat explicit INTEGER types as 32 bit integers; in addition, all decimal integer constants will be treated as 32 bit constants. For a more complete description of this facility, see Section IX.

2-11. DOUBLE INTEGER CONSTANTS

Double integers are signed whole numbers containing no fractional part. Double integers may be specified in four ways: decimal, octal, ASCII, and composite. Decimal double integer constants use the decimal digits 0 through 9. They can contain a leading plus (+) or minus (−) sign (a number with no leading sign is positive). The range of a double integer constant is from −2147483647 to +2147483647. The decimal constant −2147483648 can be represented in the machine, but cannot be directly represented in the FORTRAN source code. Except as noted in Section IX, paragraph 9-23, double integer constants must be followed by a J.

For example,

```
1J
-467J
+23456J
8967125J
```

Octal double integer constants are denoted by the % character. They may contain up to eleven octal digits and an optional leading sign. The number ranges from -20000000000_8 to $+17777777777_8$.

For example,

```
%47J
+%567J
-%3472J
%4000012J
%20000000000J (this is -2147483648₁₀)
%37777777777J (this is -1)
```

ASCII double integer constants are used to write one to four ASCII bit patterns into one 32-bit computer word. ASCII double integer constants are written with the % character followed by the ASCII characters enclosed in quote marks (also called string bracket characters) or apostrophes.

For example,

```
+%"ABCD"J
-%"EF"J
%"DEC"J
%"M"J
-%"HIP"J
```

If only one ASCII character is specified, the bit pattern representing that character is placed in the computer word right-justified, and the left part of the word is filled with leading zeros. Leading plus and minus signs can be used. Integer*4 constants also can be specified in composite form. See paragraph 2-17.

2-12. REAL (FLOATING-POINT) CONSTANTS

Real constants are represented by an integer part, a decimal point, and a decimal fraction (mantissa) part. A leading sign may be used. The constant can contain a scale factor (which represents a power of ten by which the constant is multiplied).

The eleven forms of a real constant are:

n	(20.)
.*n*	(.2)
n.*n*	(20.5)
n.E+*e*	(2.E+2)
.*n*E+*e*	(.2E+3)
n.*n*E±*e*	(2.5E+2 or 2.5E−2)
*n*E±*e*	(2E+2 or 2E−2)
*n*E*e*	(2E2)
.*n*E*e*	(.2E2)
n.E*e*	(2.E2)
n.*n*E*e*	(2.2E2)

The *range* of *e* is from −77 to +78.

The letter *n* is a decimal integer. The construct E ±*e* stands for $10^{\pm e}$ where *e* is the power of 10 which is multiplied by the other part of the number (*n*., −*n*,*n*.*n*, etc.). The construct E*e* is equivalent to E + *e*. Examples,

3.4 E $-4 = \text{x}\ 10^{-4} = .00034$
-3.4 E $4 = -3.4 \text{ x } 10^{4} = -34000$

A real constant may be written any number of digits in length, but the internal representation in memory only allows six or seven significant decimal digits.

Real constants also can be specified as octal numbers, followed by the letter R. The bit pattern specified by the octal number is loaded (right justified) into two consecutive words in memory and is treated as a floating-point number.

For example,

```
%3775R
```

ASCII real (right-justified) constants also are allowed. From one to four 8-bit ASCII patterns are stored in the two 16-bit words.

Examples:

```
%"ABCD"R
-%"DEF"R
%"V"R
+%"SXZ"R
```

Composite numbers followed by the letter R also can specify real numbers. See paragraph 2-17.

2-13. DOUBLE PRECISION REAL CONSTANTS

Double precision real constants are similar to real (single precision) constants. Substituting the letter D for the letter E in the scale factor of a real constant gives a double precision real constant with 16 or 17 significant decimal digits as opposed to the 6 or 7 significant digits in the single precision real constant. Double precision constants can start with an optional sign.

The eight representations of double precision constants are:

n D±*e*	(2D +2 or 2D −2)
.*n* D±*e*	(.2D +2 or .2D −2)
n. D±*e*	(2.D +2 or 2.D −2)
n.*n* D±*e*	(2.2D +2 or 2.2D −2)
*n*D*e*	*(2D2)*
n.D*e*	*(2.D2)*
.*n*D*e*	*(.2D2)*
n.*n*D*e*	(2.2D2)

The *range* of *e* is from −77 to +78.

The real constant forms .*n*, *n*., or *n*.*n* (those without the scale factor) are not allowed for double precision constants, as FORTRAN/3000 has no way of knowing whether the number should be stored in single or double precision format.

Double precision numbers can be represented in octal format. When written, double precision octal numbers are preceded by % and followed by the letter D.

For example,

```
-%3776125D
%64333D
%45D
```

ASCII double precision real constants also are allowed. From one to eight 8-bit ASCII patterns are stored in the four 16-bit words.

Examples:

```
%"ABCDEF"D
-%"A"D
%"TR"D
+%"DFG"D
%"LKJH"D
+%"KLMNOPQS"
```

Composite double precision real constants also are allowed. See paragraph 2-17.

2-14. COMPLEX CONSTANTS

Complex constants are represented by an ordered pair of real constants enclosed in parentheses and separated by a comma. The first number represents the real part and second number represents the imaginary part of the complex number.

The real constants of each ordered pair can be represented as integers, decimal fractions (with or without a scale factor), octal numbers, or composite numbers.

Double precision constants cannot be used to represent either the real or the imaginary part of a complex constant.

Examples of complex constants are:

```
(3.0, -2.5E3)
(%376R, %736R)
```

2-15. LOGICAL CONSTANTS

Normally, only two values are used for logical constants: .TRUE. and .FALSE. .TRUE. is represented by all 16 bits of the computer word being equal to 1. .FALSE. is represented by all 16 bits equal to 0. Any other pattern of 16 bits can be used with logical operators, however, to perform masking operations (see Section III).

The actual bit pattern of a mask is specified by an octal constant, an ASCII character string of up to two ASCII characters or a composite number followed by an L.

For example,

```
%177777L
%"AB"L
%1006L
```

2-16. CHARACTER CONSTANTS

Character constants represent ASCII character strings which can be manipulated using character expressions and input/output statements. String constants are character values bound by quote marks or apostrophes, called string bracket characters. All printing ASCII characters can be used. Blanks are significant characters within a character string.

For example,

```
"THIS IS A CHARACTER STRING"
"NOW IS THE TIME"
'ANOTHER FORM USING APOSTROPHES'
'HE SAID, "HELLO" '
```

If a quote mark (") must be included within a string bracketed by quotes, or if an apostrophe (') must be included within a string bracketed by apostrophes, write the quotes or apostrophes twice in a row to distinguish them from the string bracket characters. Apostrophes in strings bracketed by quotes and quotes in strings bracketed by apostrophes need only be written once.

Examples:

```
'AB"CD'       "AB'CD"
"ABC""XYZ"    "4XC" "D"
'ABG' 'HG'    'TYU' 'V'
```

To indicate a null string (a string with no value), write a pair of string bracket characters with no intervening characters or blanks.
For example,

"" or ''

Note: A character string written with one or more blanks between the string bracket characters is not the null string, but represents a string of ASCII blanks.

Hollerith constants are a special format for character strings. They consist of a decimal integer specifying the number of characters, followed by the letter H and the characters (character value).
For example,

HOLLERITH	ASCII	STRING
19HBLANKS ARE INCLUDED	BLANKS ARE INCLUDED	BLANKS ARE INCLUDED
7HAB CDFG	AB CDFG	AB CDFG

Characters can be specified by an octal number representing a character bit pattern. The octal number is followed by the letter C. For example,
%101C represents the character A.
%15C represents a carriage return.

The above form is useful for representing non-printing characters (such as carriage returns) in source programs. Octal representations of characters are shown in Appendix E.

2-17. COMPOSITE NUMBERS

Composite numbers are a convenient way of representing specific bit patterns for any type constant except character. A composite number takes the form

% [$A_1/N_1, A_2/N_2, \ldots A_n/N_n$] *letter*

For example,

%[3/7,4/12,2/1] L

A_1 through A_n (the numerals 3, 4, and 2 in the example) are decimal integers which represent the number of bits in the bit pattern subfield. N_1 through N_n are the octal or decimal values set right-justified into the subfields. Unspecified leading bits are set to zeros. Extra leading bits are truncated.

For example, 3/7 creates a subfield 3 bits long with the binary value 111_2 set into it. The numerals 4/7 create a subfield 4 bits long. The value $7_{10} = 111_2$ is right-justified into the 4-bit field and the unspecified leading bit is set to zero. The resulting subfield is 0111_2. The data storage word format for the type of constant is indicated by *letter*. Integer composite constants do not have a letter suffix. Logical is indicated by L, double integer by J, real by R, and double precision by D. Complex constants consist of an ordered pair of real numbers, either or both of which can be real composite numbers. The bit pattern specified in each subfield is concatenated from left to right and the result is stored right-justified in the storage space for the constant type indicated by the *letter*. Unspecified leading bits are set to zero. Examples of the different constant types and bit patterns are shown in table 2-1. The number of bits in a composite number must be less than or equal to the number of bits for the data type.

The bit pattern is determined as follows:

$\%[3/7,4/7]\text{L} = 000167_8$

where L indicates a logical constant.

The two subfields $3/7 = 111_2$ and $4/7 = 0111_2$ are concatenated left to right to form the bit pattern 1110111_2.

This value is placed right-justified in a 16-bit word, with unspecified leading bits set to zero. The resulting logical value bit pattern is

0000000001110111

2-18. NUMBER RANGES

Numbers represented by FORTRAN/3000 constants and variables have specific positive and negative ranges which limit the size of the number represented. Table 2-2 shows the various number types and their associated ranges. Complex numbers are not shown; they are represented by an ordered pair of real numbers.

2-19. VARIABLES

A quantity which appears in a FORTRAN/3000 statement in numeral form is a constant, while a quantity which appears in the form of a name is called a *variable*. A variable is a symbolic name of from *one to 15 alphanumeric characters* which *represent* a *value*. The first character must be a letter. Each variable can represent one type of value only: integer, double integer, real (floating point), double precision real, complex, logical, or character.

The type of value represented by a variable can be established through the use of a Type or IMPLICIT statement (see Section V). If the variable is not specified in a Type or IMPLICIT statement, the variable type is conveyed to the compiler *implicitly* as type integer or real by the first letter of the variable name. Names starting with the letters I, J, K, L, M, or N are implied type integer. Variable names starting with any other letter are implied type real.

A variable is given a value through an assignment statement (see Section III), a READ statement (see Section VI), or a DATA statement (see Section V). The specific value represented by a variable can be changed during execution of a program.

Table 2-1. Composite Numbers

TYPE	EXAMPLE	BIT PATTERN
Integer	%[4/15,6/%13,2/1] (no *letter*)	0 000 111 100 101 101
Double Integer	%[5/12,4/%15,10/192]J	0 000 000 000 000 010 1 011 010 011 000 000
Logical	%[3/7,4/7]L	0 000 000 001 110 111
Real	%[4/9,16/%1245,10/49]R	0 010 010 000 001 010 1 001 010 000 110 001

Double Precision Real	%[35/4775,10/%777]D	0 000 000 000 000 000
		0 000 000 000 000 000
		0 000 000 001 001 010
		1 001 110 111 111 111

Complex	(%[15/777]R,%[12/%4444]R)	0 000 000 000 000 000
		0 000 001 100 001 001
		0 000 000 000 000 000
		0 000 100 100 100 100

Table 2-2. Number Ranges in FORTRAN/3000

TYPE	BITS	RANGE AND ACCURACY
Integer	16	−32,768 to +32,767 (-2^{15} to $2^{15}-1$)
Double Integer	32	−2147483648 to +2147483647 (-2^{31} to $2^{31}-1$)
Real	32	0.863617 x 10^{-77} to 1.157921 x 10^{77} $(1+2^{-22})2^{-256}$ to $(2-2^{-22})2^{255}$ Accuracy is 6.9 significant decimal digits. Largest accurate integer value expressed to the units digit is $2^{23}-1$ = 8,388,607
Double Precision	64	8.636168555094445 x 10^{-78} to 1.157920892373162 x 10^{77} $(1+2^{-55})2^{-256}$ to $(2-2^{-55})2^{255}$ Accuracy is 16.5 significant decimal digits. Largest accurate integer value expressed to the units digit is $2^{55}-1$ = 36,028,797,018,963,967

2-20. SIMPLE VARIABLES

A simple variable is a symbolic name which has only one value at a time. Examples of simple variables are:

Integer	Real
I	ABLE
J	A
JACK123	ZERO
MAN	Q45
NOW	ZEBRA789
KNLQ1234	BETA

2-21. ARRAYS AND ARRAY NAMES

An array is a collection of several values of the same type, all of which are represented by an array name. An array name is a symbolic name and represents all values, or *elements*, of the array. In order to designate exactly one data value, or element, of the array, the symbolic name is suffixed by *subscript.*

A group of values arranged in a single dimension is a one-dimensional array. The elements of such an array are identified by a single subscript. For example, A(I) refers to the "Ith" element of array A, and A(1) refers to element 1 of array A.

If two subscripts are used to identify an element of an array, then this array is two-dimensional. A chess board, for example, is two-dimensional consisting as it does of 8 horizontal rows, beginning at 1 at the top, and 8 vertical columns, beginning at 1 on the left.

An array to represent the chess board could be dimensioned as CHESS (8,8). With this dimension statement, the subscripted variable, CHESS (6,4) could represent the square at the 6th *row*, 4th *column* of the chess board.

The type of data represented by an array name can be determined through the use of a Type or IMPLICIT statement (see Section V). If the array name is not mentioned in a Type or IMPLICIT statement, the array type is determined by the first letter of the variable name. Names starting with I, J, K, L, M, or N are type integer. Array names starting with any other letter are type real. An array, its dimensions, and the number of elements per dimension (called *bounds*) must be defined in a DIMENSION, Type, or COMMON statement (see Section V).

Subscripts can consist of constants, variables, or expressions of any type except complex or character (i.e., *linear* expressions. See Section III for a discussion of expressions.) Subscript values should not be negative or zero.

As mentioned, subscripts designate a specific element of an array. An array variable name must contain as many subscripts as there are dimensions in the array. (The maximum number of dimensions allowed in FORTRAN/3000 is 255.) For example, a one-dimensional array name is of the form A(1), A(2), A(I), etc; a two-dimensional array name is A(1,1), A(1,2), A(I,J), etc. In addition, the upper limit of the subscript value should match the number of elements per dimension in the array. Thus, a variable for a one-dimensional array of three elements could have the form A(1), A(2), A(3) to represent all the elements of the array. Examples of array variables and subscripts are as follows:

ARR(1,2)	*Represents the element 1, 2 of the array ARR. Array ARR is implied type real because the name begins with a letter other than I, J, K, L, M, or N.*
CHESS(I,J)	*Subscripts I and J are variables which represent different elements of array CHESS, depending on the value of I and J.*

ARR(I+4,J-2)	*Subscripts I+4 and J-2 are expressions which when evaluated, represent specific elements of array ARR.*
I((3x+1)/4)	*If X* = 3.6 (real), the expression evaluates to 2.9 (real). Nearest integer less than the expression is 2, so the subscripted variable equals I(2).

Note: If an actual subscript value is less than 1 or greater than the maximum value for elements (bounds) specified in the DIMENSION statement, an element outside the array will be referenced. You must guard against this possibility, since the program does not check for subscripts to be within bounds. (See the compiler command, $CONTROL BOUNDS, in Section IX.)

WRITING EXPRESSIONS AND ASSIGNMENT STATEMENTS IN FORTRAN/3000

SECTION III

3-1. EXPRESSIONS

In FORTRAN/3000, an expression can be a single constant, a single simple or array variable, or a single function name; or combinations of the foregoing joined by arithmetic, logical, and relational *operators*. All of the following are valid expressions:

6	Integer constant
I	Integer variable
ARR(I,J)	Array variable
3.456	Real constant
"STRING"	Character constant
SQRT(A)	Function
A + B +6	The sum of the variables of A, B, and 6, which are variables and a constant joined by arithmetic operators (plus sign).
(A + B) + (B/C)	In this expression, (A + B) and (B/C) are subexpressions.

Subexpressions may not be type character, thus ("STRING") + ("CHARACTER") is not a valid expression.

Expressions are of three main types: arithmetic, logical and character. Arithmetic expressions return a single value of type integer, double integer, real, double precision, or complex. Logical expressions evaluate to either .TRUE. or .FALSE., or to a 16-bit mask which can be used in later manipulations (depending on the context of the expression). Character expressions manipulate character variables and constants and return character values.

3-2. ARITHMETIC EXPRESSIONS

Arithmetic expressions perform arithmetic operations. An arithmetic expression may consist of a single constant, variable, or function name or it may consist of two or more constants, variables, or function names joined by the following arithmetic operators:

+	Addition
−	Subtraction
*	Multiplication
/	Division
**	Exponentiation

The following are valid arithmetic expressions:

```
A          A**2
3.14       (C**4) * (C * 9)
A + 3      A + B
SQRT (A)
```

Thus, A + B is an arithmetic expression, consisting as it does of two *operands* (A, B) which are used to evaluate A plus B.

In the expression (C**4) * (C * 9), the subexpressions (C**4) and (C * 9) are operands and, once evaluated, are used to compute the value of the complete expression (C**4) * (C * 9).

The hierarchy of arithmetic operations is:

**	Exponentiation
*	Multiplication and / Division
+	Addition and − Subtraction

Exponentiation precedes all arithmetic operations within an expression, and multiplication and division occur before addition and subtraction. For example,

A ** B + C * D + 6
is evaluated in the following order:

1. A ** B is evaluated to form the operand OP1.
2. C * D is evaluated to form the operand OP2.

Note: OP1 and OP2 are *intermediate* results.

3. OP1 + OP2 + 6 is evaluated to determine the value of the expression.

The order for evaluating expressions with operators of the same level of hierarchy (* and / or + and -) is determined by the data type of the operands. Integer operands are combined first and then converted to the next higher type to which they are connected by an operator. (See paragraph 3.3). If the associative and commutative laws of mathematics apply to an expression being evaluated, the compiler may use them. Since they do not apply to integer expressions involving division, such expressions are evaluated from left to right. Therefore, the expression I * J/K does not yield the same result as J/K * I. However, the real expressions, R * X/Y and X/Y * R, yield the same result.

When using integer division, it is also important to recall that integer division provides no remainder. Thus, 2/3 yields the quotient zero; 7/3 yields the quotient 2. Consequently, the value of an integer expression is dependent on the order of computation, and it is therefore evaluated from left to right.

Parentheses may be used to control the order of evaluation of expressions. For example,

A + B + C
is evaluated according to the types assigned to A, B, C.

A + (B + C)
evaluates (B + C) first, and then adds it to A while
(A + B) + C
evaluates (A + B) first and then adds it to C.

(A +B +C) − ((C + D) + X + Y)

In this expression, the evaluation of (A + B + C) occurs according to the variable types. In ((C + D) + X + Y), the subexpression (C + D) is evaluated first and then added to either X or Y depending on the type of X and Y. Finally, the evaluated result of ((C + D) + X + Y) is subtracted from the evaluated result of (A + B + C).

Two arithmetic operators cannot appear in a row unless one of the operators is enclosed in parentheses. For example, A**−3 is not allowed by FORTRAN/3000 but A**(−3) is allowed.

3-3. ARITHMETIC EXPRESSION TYPE.

Integer, double integer, real, double precision, and complex operands may be intermixed freely in an arithmetic expression. Before an arithmetic operation is performed, the lower type operand is converted to the higher type. The expression takes on the type of the highest type operand in the expression. Operand types rank from lowest to highest in the following order:

Integer	*Lowest*
Double Integer	
Real	
Double Precision Real	
Complex	*Highest*

Note: Some accuracy could be lost in the conversion from double integer to real, since real values have only 22 bits of accuracy compared to the 31 bits of the double integer. Type conversion from double integer to double precision real can be forced by using the compiler library routine DFLOAT.

3-4. CHARACTER EXPRESSIONS

Character expressions define character strings and consist of character constants, variables, or function references; or character variables or function references followed by substring designators (see paragraph 3-8). For example,

"THIS IS A STRING"	Character constant.
CHAR	Character variable. Note that CHAR has to be declared type character in a Type or IMPLICIT statement (see Section V).
A(I) = FORMS (M)	Function reference. The function FORMS computes a character value for the variable M and assigns this value to element I of character array A.
VAR[3:7]	Character variable followed by substring designator.
VAR[3:7] = ENDLINE(X[3:7])	Function reference (ENDLINE), followed by substring designator.

No arithmetic operators are used in character expressions; however, character expressions can be used with relational operators to form *logical* expressions. (See paragraph 3-5 for a discussion of relational operators.) For example,

IF(NAME.EQ."HARRY")GOTO 100
Character expression "HARRY" used with character variable NAME and *relational operator* to form logical IF statement.

Character expressions are compared in one of the following ways, depending on whether they are equal or unequal in length.

- *Expressions of Equal Length:* The two expressions are compared character by character, starting from the left. The comparison continues until a pair of unequal characters is encountered or all the character-pairs have been compared. Thus, if the first characters are equal, the second characters are compared. If the second characters are equal, the third characters are compared, and so forth. When unequal characters are encountered, the greater of the two expressions is determined by these characters.

  For example, when the two expressions "HARRY" and "HANDS" are compared, the first expression is considered greater than the second. This is determined by the third character, R, which is greater than N in the ASCII collating sequence.

- *Expressions of Unequal Length:* The comparison continues until a pair of unequal characters is encountered or all the characters in the shorter expression are found to be equal to the corresponding characters in the longer expression. In the latter case, the longer expression is considered greater.

  For example, "ZOO" is considered greater than "ZERO" since O is greater that E in the second pair of characters compared. For the same reason, "APE" is considered smaller than "APOCALYPSE". On the other hand, when expressions such as "APPLEJUICE" and "APPLE" are compared, all the characters of the latter expression are equal to the corresponding characters of the former. Thus, "APPLEJUICE" is greater than "APPLE".

Note: The *length* of a character variable (denoting the number of characters) must be defined when the variable is declared as type character. For example, in the expression A = "HARRY", the variable A must have been declared as type character with a length attribute of at least 5. See Section V for a further discussion of the length attribute.

Strings of characters and each character in the string can be manipulated using substring designators (see paragraph 3-8).

3-5. LOGICAL EXPRESSIONS

Logical expressions are similar to arithmetic expressions, but, when evaluated, return a single logical value - either true or false.

The form of a logical expression is

operand **(a logical variable)**

or

operand relational operator operand

where

operand
is a constant, variable,function reference, subexpression, or an arithmetic or character expression.

relational operator
the relational operators are: .EQ., .NE., .LT., .LE., .GE., and .GT. (see below).

A simple logical expression consists of one logical variable, or two arithmetic or character expressions joined by a *relational operator*. These simple logical expressions can be joined by *logical operators* (see below) to form more complicated logical expressions.

The *relational operators* are

.EQ.	Equality
.NE.	Non-equality
.LT.	Less than
.LE.	Less than or equal
.GT.	Greater than
.GE.	Greater than or equal

These operators combine with arithmetic expressions or character expressions to form relations. Each relation is evaluated and assigned the logical value .TRUE. or .FALSE. depending on whether the relation between the two operands is satisfied (.TRUE.) or not (.FALSE.). Expressions used as operands in a logical relation may be integer, double integer, real, or double precision (linear), complex, or character, but not logical. Types integer, real, and double precision can be mixed in one logical relation.

Note: A *linear* expression is an expression of type integer, double integer, real, or double precision; a *complex* expression is type complex; and a *character* expression is type character.

Complex expressions can be used as operands with .EQ. and .NE. relational operators only. The concept of "less than" or "greater than" is not defined for complex numbers. The following are valid simple logical expressions:

L (where L is type logical)
A .LT. I (A real, I integer)
R .GT. 5 (R is type real)
A .EQ. B } (A and B are real)
A .LT. B }
CHAR .EQ. "END" (CHAR is type character)
(X + Y) .GT. VAL
I(7,4) .NE. 45
IF (L)GOTO 50

The last example shows a logical expression (L) used in an IF statement. If L evaluates to .TRUE. (bit 15 = 1), the statement GOTO 50 is executed and control is passed to statement 50. See Section IV for a discussion of IF statements.

Simple logical expressions can be joined by logical operators to form more complex expressions. The logical operators are

.NOT.	Complement (.NOT. is a *unary operator*, that is, it operates on only one operand)
.AND.	AND
.XOR.	Exclusive OR
.OR.	Inclusive OR

An example of simple logical expressions joined by logical operators is

IF (A .EQ. B .AND. C .EQ. D) GOTO 100
(If A equals B and C equals D, control is passed to statement 100)

The unary operator .NOT. takes the complement of the logical value of the operand immediately following the .NOT. operator.

The .AND. operator returns a value of .TRUE. if, and only if, the logical operands on both sides of the .AND. operator evaluate as .TRUE.

The .XOR. operator (exclusive OR) returns a value of .TRUE. if, and only if, one (but not both) of the logical operands on either side of the .XOR. operator is .TRUE.

The .OR. operator (inclusive OR) returns a value of .TRUE. if one or both of the logical operands on either side of the .OR. operator are .TRUE. A truth table for the logical operators is shown in table 3-1.

Table 3-1. Truth Table for Logical Operators

A	B	.NOT. A	.NOT. B	A .AND. B	A .XOR. B	A. OR. B
.TRUE.	.TRUE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.
.TRUE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.	.TRUE.
.FALSE.	.TRUE.	.TRUE.	.FALSE.	.FALSE.	.TRUE.	.TRUE.
.FALSE.	.FALSE.	.TRUE.	.TRUE.	.FALSE.	.FALSE.	.FALSE.

The hierarchy of logical operations is:

.NOT.	Complement
.AND.	AND
.XOR.	Exclusive OR
.OR.	Inclusive OR

Thus, .NOT. operations are performed before all other operations and .OR. operations are performed after all other operations.

For example,

Given the expression .NOT. A .AND. B, with A and B both .TRUE., .NOT. A is evaluated first (if A is .TRUE., then .NOT. A is .FALSE.), now the expression would be .FALSE. .AND. B, which would evaluate .FALSE. .AND. TRUE. (B is .TRUE.). The result is .FALSE. If A is .TRUE. and B is .FALSE., then the expression .NOT. B .XOR. A .AND. B is evaluated .NOT. B (= .TRUE.) first, then A .AND. B (= .FALSE.) then .TRUE. .XOR. .FALSE. (= .TRUE.).

Parentheses can be used to direct the order of evaluation of a logical expression. For example,

If A and B are both .TRUE., then the expression .NOT. (A .AND. B) evaluates A .AND. B (= .TRUE.) first, then evaluates .NOT. .TRUE. (= .FALSE.).
Without parentheses, the .NOT. operation would be performed first. If A = .TRUE., B = .TRUE., and C = .FALSE., then the expression (A .OR. B) .AND. C .OR. B evaluates A .OR. B (= .TRUE.) first, then evaluates .TRUE. .AND. C (= .FALSE.) then evaluates .FALSE. .OR. B (= .TRUE.).

3-6. MASKING OPERATIONS. Besides returning values which can be evaluated .TRUE. or .FALSE. by examining bit 15, logical operators can perform bit-by-bit operations on 16-bit logical values.

The .NOT. operator complements a 16-bit value. The .AND. operator performs a logical AND on two 16-bit logical values; the .XOR. operator performs a logical exclusive OR on two 16-bit logical values; and the .OR. operator performs a logical OR on two 16-bit values.

Table 3-2 shows the results of bit-by-bit operations on the 16-bit logical values L1 and L2.

Table 3-2. Logical Operations on 16-bit Logical Values

L1	0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1
L2	0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1
.NOT. L1	1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0
.NOT. L2	1 1 1 1 0 0 0 0 1 1 1 1 0 0 0 0
L1 .AND. L2	0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1
L1 .XOR. L2	0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0
L1 .OR. L2	0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1

Partial-word designators can be used in logical expressions. For complete details, see paragraphs 3-7.

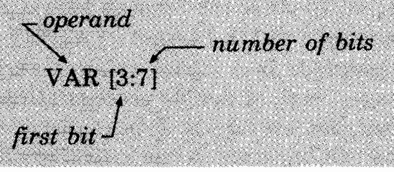
3-7. PARTIAL-WORD DESIGNATORS

A partial-word designator acts as a unary operator (i.e., operates on only one operand) which indicates a specific pattern of bits of the operand it suffixes. A partial-word designator can be used to extract a bit pattern from the operand it suffixes and right-justify these bits to form a new value of the same type. (The operand itself remains unchanged; a *new* operand is formed.) Partial-word designators can be used with integer operands in arithmetic expressions, and with logical operands in logical expressions.

The partial-word designator form is

***operand** [**first bit:number of bits**]*

For example,



Note: Brackets [ ] must be used (instead of parentheses) with partial-word designators.

where

operand
is an integer or logical constant, variable, function reference or subexpression.

first bit
is an integer constant specifying the beginning bit position of the bit pattern. The leftmost bit is number 0, the rightmost bit is number 15.

number of bits
only applies to 1-word quantities and is an integer constant specifying the length of the bit pattern (cannot exceed 15). If the *number of bits* is not specified, the length is equal to 16 minus *first bit.*

If *number of bits* is greater than (16 - *first bit*), the bit pattern wraps around (takes bits from 0, 1, etc.). See figure 3-1 for examples.

In the example, VAR [15:3], wrap-around occurred when *number of bits* was greater than (16 - *first bit*). In the example, K[4], the number of bits was left out so the default value (16 - *first bit*) was used. Bits 5 through 15 were extracted from the original value and right-justified automatically (since bit 15 is the last bit.

3-8. SUBSTRING DESIGNATORS

A substring designator is a unary operator which extracts specified substrings of characters from a character value and creates a new value from the extracted substring.

The substring designator form is
name[first character: number of characters]

3-9. ASSIGNMENT STATEMENTS

Assignment statements use the replacement operator " = ", which means, *"is replaced by the value of,"* to assign values to variable or statement function names.

An assignment statement has the form

name = expression

For example,

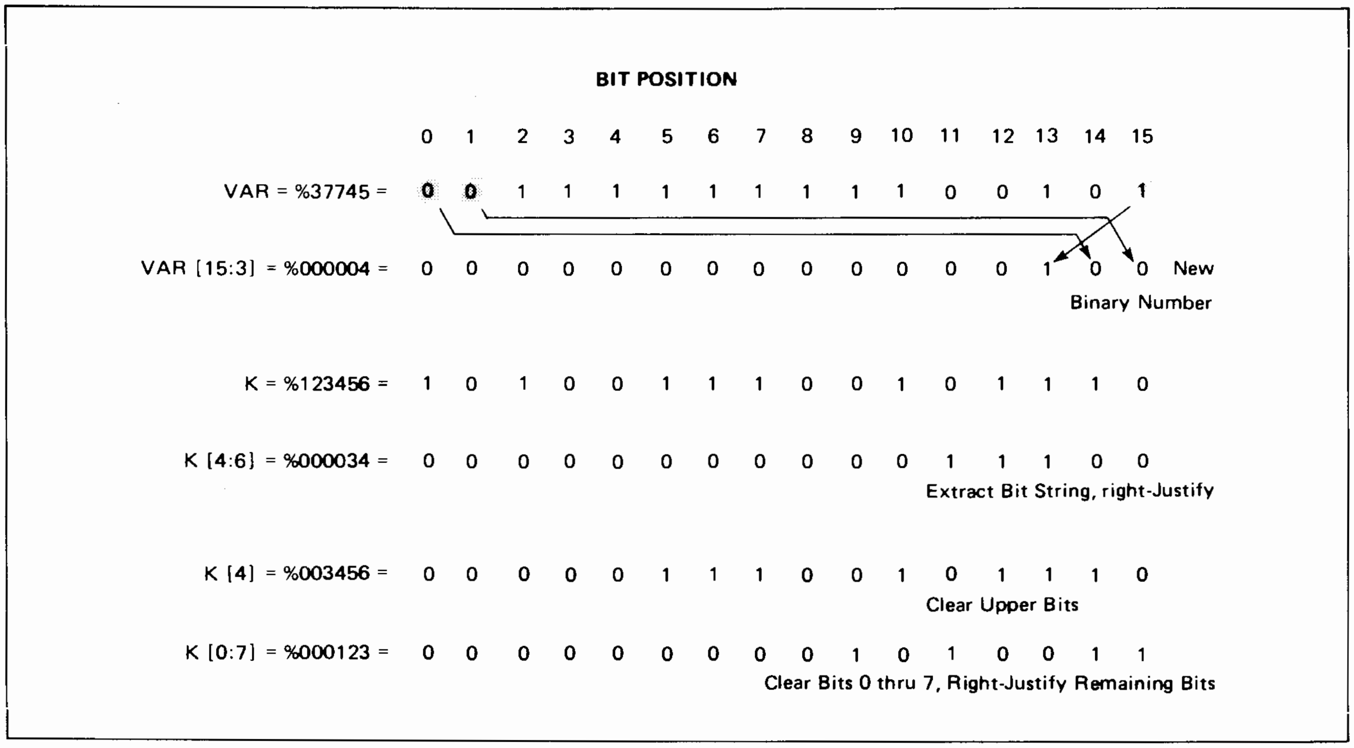
A = A + 1

name *expression*

where

name -	is a variable
expression -	is an expression of arithmetic, logical or character type.

Assignment statements are the basic computational tool of FORTRAN/3000 programs. When an assignment statement is executed, the expression is evaluated and the resulting value is assigned to the variable.



BIT POSITION

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
VAR = %37745 =	0	0	1	1	1	1	1	1	1	1	1	0	0	1	0	1	
VAR [15:3] = %000004 =	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	New Binary Number
K = %123456 =	1	0	1	0	0	1	1	1	0	0	1	0	1	1	1	0	
K [4:6] = %000034 =	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	Extract Bit String, right-Justify
K [4] = %003456 =	0	0	0	0	0	1	1	1	0	0	1	0	1	1	1	0	Clear Upper Bits
K [0:7] = %000123 =	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1	1	Clear Bits 0 thru 7, Right-Justify Remaining Bits

Figure 3-1. Partial-Word Designator Examples

Table 3-3. Substring Designators

```
        ┌first character
         ↘
      VAR[3:7]
         ↑ ↑
   name ┘  └ number of characters

           where
```

name -	a character variable or function reference. (Substring designators cannot be applied to character constants.)
first character -	a linear expression specifying the beginning of the substring. This expression is evaluated and converted to an integer, if necessary. Its value must range between one and the length of the character value.
number of characters -	a linear expression specifying the length of the extracted substring. This expression is evaluated and converted to an integer, if necessary. Its value must range between zero and (length of the character value-first character+1). If the number of characters is not specified, the default value (length of the character value-first character+1) is used.
EXAMPLES	
VAR[3:7]	When used with a value for VAR such as "THIS IS A STRING", the characters "IS IS A" would be extracted. (Starting with the third character and extending for six more characters (seven characters total).)
NEXTCHAR = 6	*Integer assignment statement.* Integer constant 6 is assigned to variable NEXTCHAR. (To be used in next example.)
VAR [NEXTCHAR+3]	*Expression used in substring designator.* When used with present value of VAR ("THIS IS A STRING"), would extract, "A STRING", It extracts the substring starting with the ninth character (NEXTCHAR + 3) and ending with the last character of the value. The number of characters was not specified, so the default value (length of value - first character + 1) was used.

Partial-word designators can be used with integer or logical variables in the *name* or *expression* side of an assignment statement, and have the effect of replacing the specified part of the variable. For example,

J[2:6]= K	This statement assigns six bits of K to J (starting at bit 2 of J), leaving all other bits unchanged. The six bits assigned to J are the six rightmost bits of K.
J = K[2:6]	This statement assigns six bits from K, starting at bit 2 in K, to J. The six bits are right justified in J.

Substring designators can be used with character variables to assign specific characters to a variable. For example,

VAR[3:7] = "THIS IS A STRING"	This statement extracts "THIS IS" from the value "THIS IS A STRING" and assigns it to VAR, starting at character 3 of VAR.
CHAR = AR[2:5]	If the value of AR is "LETbGO", then this statement extracts "ETbGO" from AR and replaces the leftmost characters of CHAR with these characters.

X[2:2] = X[1:2]

A	B
C	

Stage I

(Initial Stage)

A	A
C	

Stage II

(Second Character replaced)

A	A
A	

Stage III

(Third Character replaced)

In this example, both the source and the destination data are at the same location and are represented by variable X. The substring designators move the data character by character from the left. Thus, if X has the value ABC (stage I), this assignment statement would first change the data to AAC (stage II) and then to AAA (stage III).

If X and Y have the same initial value, ABC, the assignment statement X[2:2] = Y[1:2] would assign the value AAB to X.

Character expression values are truncated on the right if the defined character variable length is less than the expression value length. For example,

CHARACTER*10, X,Z*31

is a Type statement defining the variable X as type character with a length of 10 and Z as type character with a length of 31. (See Section V for a discussion of Type statements.) The assignment statement

```
Z = "THIS VARIABLE HAS 31 CHARACTERS"
```

assigns a value with 31 characters to Z. An assignment statement

```
X = Z
```

would then assign the value "THIS VARIA" to X because its length is less than that of Z.

Character expression values are left-justified and padded with blanks on the right if the defined character variable length is larger than the expression value length.

For example, assuming that X still retains the value "THIS VARIA", then

```
Z = X
```

would leave the value "THIS VARIA" left-justified in Z. The remainder of the length of Z would be padded with blanks. Some examples of assignment statements are shown in table 3-4.

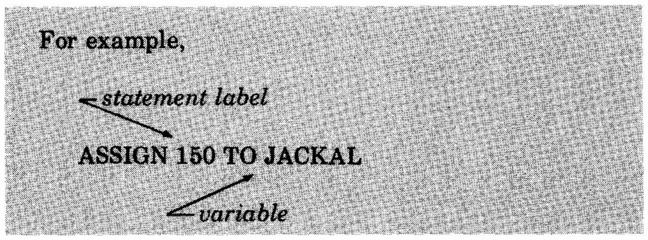
3-10. LABEL ASSIGNMENT STATEMENTS

Label assignment (ASSIGN) statements are used to assign *statement label* values to integer simple variables.

The form of an ASSIGN statement is

ASSIGN *statement label* TO *variable*

For example,



ASSIGN 150 TO JACKAL

where

statement label	is an integer constant used as a *label value* and not as the regular *integer value*.
variable	is an integer or double integer simple variable.

Some examples of ASSIGN statements are:

ASSIGN 150 TO JACKAL	The variable *JACKAL* is assigned the *statement label* 150.

Table 3-4. Assignment Statements

A = B * 6	*Arithmetic assignment statement.* Arithmetic expressions may be of type integer, real, double precision, or complex. The name need not be of the same arithmetic type as the expression. The value of the expression is converted to the name type before the value is assigned. (See table 3-5.)
LOGICAL A,B,C	*Type statement* declaring the variables A, B, and C as type logical (so that they may be used in a logical assignment statement).
A = B .AND. C	*Logical assignment statement.* A (previously defined as type logical) will assume a value of .TRUE. or .FALSE. depending on whether B and C (previously defined as type logical) are both true or if one or both of them is false. A then could be used as follows: IF (A) GOTO 100 If A is .TRUE., the statement GOTO 100 will be executed and control will pass to statement number 100. If A is .FALSE., GOTO 100 will not be executed and control will pass to the statement following the IF statement. (See Section IV for a description of IF statements.)

ASSIGN 70 TO ITEM	The variable *ITEM* is assigned the *statement label* 70.

ASSIGN statements dynamically assign statement label values to integer or double integer simple variables during program execution. The values thus assigned can be used only in assigned GOTO statements (see Section IV).

Please bear in mind that the *statement label* value assigned to an integer or double integer simple variable and its regular integer value are different. Through the ASSIGN statement, an integer or double integer simple variable can have *two* separate values: one of type integer and one of the pseudotype "label." These two values are independent of each other and can exist simultaneously. The "label" value is referred to in only two FORTRAN/3000 statements: the *label assignment* statement which assigns the "label" value to an integer or double integer simple variable, and the *assigned GOTO* statement (see Section IV) which transfers control to the variable which has been assigned this "label." All other references to the variable are to its integer value.

```
:FORTGO FTRAN2

 PAGE 0001   HP32102B.00.0


           PROGRAM ASSIGN
 C
 C ASSIGN STATEMENT EXAMPLE
 C
           REAL INCOME,INTEREST
           INTEGER AGE
      100  FORMAT(T10,"THE AMOUNT OF TAX IS: ",M10.2)
           ACCEPT AGE,INCOME,RETAX,GASTAX,INTEREST
           IF(AGE.GE.65)GOTO 10
 C
 C THE NEXT STATEMENT IS AN ASSIGN STATEMENT
 C
           ASSIGN 40 TO AGE
           GOTO 20
      10   ASSIGN 50 TO AGE
      20   ADJGROSS=INCOME-(RETAX+GASTAX+INTEREST)
      30   GOTO AGE
      40   AGE=750
           GOTO 60
      50   AGE=1500
      60   TAX=(ADJGROSS-AGE)*.22
           WRITE(6,100)TAX
           STOP
           END


 ****      GLOBAL STATISTICS      ****
 ****    NO ERRORS,   NO WARNINGS  ****
 TOTAL COMPILATION TIME  0:00:01
 TOTAL ELAPSED TIME      0:00:06

  END OF COMPILE

  END OF PREPARE


?68,22321.67,456.98,234,276.56

           THE AMOUNT OF TAX IS:  $4,367.91
  END OF PROGRAM
```

Figure 3-2. Example of ASSIGN Statement Usage

```
:FORTGO FTRAN2

 PAGE 0001   HP32102B.00.0


            PROGRAM ASSIGN
      C
      C ASSIGN STATEMENT EXAMPLE
      C
            REAL INCOME,INTEREST
            INTEGER AGE
       100  FORMAT(T10,"THE AMOUNT OF TAX IS: ",M10.2)
            ACCEPT AGE,INCOME,RETAX,GASTAX,INTEREST
            IF(AGE.GE.65)GOTO 10
      C
      C THE NEXT STATEMENT IS AN ASSIGN STATEMENT
      C
            ASSIGN 40 TO AGE
            GOTO 20
       10   ASSIGN 50 TO AGE
       20   ADJGROSS=INCOME-(RETAX+GASTAX+INTEREST)
       30   GOTO AGE
       40   AGE=750
            GOTO 60
       50   AGE=1500
       60   TAX=(ADJGROSS-AGE)*.22
            WRITE(6,100)TAX
            STOP
            END


  ****      GLOBAL STATISTICS       ****
  ****    NO ERRORS,    NO WARNINGS   ****
  TOTAL COMPILATION TIME   0:00:01
  TOTAL ELAPSED TIME       0:00:06

  END OF COMPILE

  END OF PREPARE


?68,22321.67,456.98,234,276.56

          THE AMOUNT OF TAX IS:  $4,367.91
  END OF PROGRAM
```

Figure 3-2. Example of ASSIGN Statement Usage

WRITING CONTROL STATEMENTS IN FORTRAN/3000

SECTION IV

In FORTRAN/3000, program execution normally proceeds sequentially from statement to statement. Control statements alter this sequence by transferring control to a specified statement or by repeating a predetermined group of statements.

Statements in a program unit to which control is to be passed are labeled by unsigned integers in the range of 1 through 99999. Embedded blanks and *leading* zeros in the label are ignored (1, 01, 0b1, 0001, and 00bb1 are identical to the compiler). Note that 0100 is interpreted by the compiler as 100, *trailing* zeros are not ignored by the compiler.

4-1. GO TO STATEMENTS

A GO TO statement transfers control to a labeled statement in the same program unit. There are three kinds of GO TO statements: *unconditional, computed,* and *assigned.*

4-2. UNCONDITIONAL GO TO STATEMENT

The unconditional GO TO statement provides for the absolute transfer of control to a given labeled statement.

The form of an unconditional GO TO statement is

GO TO *k*

For example,

GO TO 30

or

GOTO 30 (The words GO TO can be written without the space.)

where

k (30)
is an integer statement label number. Control is passed to the statement labeled *k* every time the unconditional GOTO statement is executed.

Unconditional GOTO statements can be used to bypass certain statements in a program unit or to cause repetition of certain statements.

An example of unconditional GOTO statement usage is shown in figure 4-1. Statement number 20 transfers control to statement number 60, thus skipping statements 30, 40, and 50. Statement number 50 provides repetition by transferring control to statement number 10.

4-3. COMPUTED GO TO STATEMENT

In a computed GO TO statement, an index expression is evaluated and control is passed to one of several labeled statements depending on the result of the evaluation.

The form of a computed GO TO statement is:

GO TO *(label, label, . . . label), index expression*

For example,

$$\text{GO TO }\underbrace{(30, 40, 50)}_{\textit{labels}}, \underbrace{\text{I}}_{\textit{index expression}}$$

where

label
is the statement number to which control is transferred when *index expression* is evaluated

index expression
is a test argument and can be an arithmetic expression of any type except complex. (For example, *index expression* can be a simple variable, a subscripted array variable or an arithmetic expression such as A + B.)

The computed GOTO statement is used when it is desirable to pass control to one of several labeled statements, depending on the result of an evaluation. The *index expression* is evaluated and truncated to an integer value (the *index*). This integer value, or *index,* then is used to select the Ith statement label in the label list. For example, if the index is 1, the first label in the list is used and control passes to the statement whose label is this number. If the index value is 2, the second label in the list is used, and so on. If the index expression evaluates to less than 1, the first label in the label list is used. If the index expression evaluates to a value greater than the number of labels in the label list, the last label in the list is used.

An example of the computed GOTO statement is shown in figure 4-2. I is the index expression. The first time the GOTO statement is executed, I = 1 and control is transferred to statement 10. The second time the GOTO statement is executed, I = 2 and control is passed to statement 20. The third time, control is passed to statement 30.

```
:FORTGO FTRAN3

PAGE 0001    HP32102B.00.0

               PROGRAM GOTO
         C
         C UNCONDITIONAL GOTO STATEMENT EXAMPLE
         C
          100  FORMAT('0',T10,F12.4)
               ACCEPT A,B,C
               D=A*2+B*2+C*2
          10   E=100
               IF(D.LE.E)GOTO 30
         C
         C THE NEXT STATEMENT IS AN UNCONDITIONAL GOTO STATEMENT
         C
          20   GOTO 60
          30   WRITE(6,100)D
          40   D=D**2
          50   GOTO 10
          60   WRITE(6,100)E
               STOP
               END


****       GLOBAL STATISTICS        ****
****    NO ERRORS,    NO WARNINGS   ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:00:06

 END OF COMPILE

 END OF PREPARE


?12.4,2,6


             40.8000

            100.0000
 END OF PROGRAM
```

Figure 4-1. Unconditional GOTO Statement Example

```
:FORTGO FTRAN4

PAGE ØØØ1   HP321Ø2B.ØØ.Ø


      PROGRAM COMPUTED GOTO
C
C COMPUTED GOTO STATEMENT EXAMPLE
C
 1ØØ  FORMAT('Ø',T1Ø,"THE SQUARE ROOT OF ",F12.3,
     #" IS ",F12.5)
      ACCEPT X,Y,Z
      DO 4Ø I=1,3
C
C THE NEXT STATEMENT IS A COMPUTED GOTO STATEMENT
C
      GOTO(1Ø,2Ø,3Ø),I
 1Ø   B=X
      A=SQRT(X)
      GOTO 4Ø
 2Ø   B=Y
      A=SQRT(Y)
      GOTO 4Ø
 3Ø   B=Z
      A=SQRT(Z)
 4Ø   WRITE(6,1ØØ)B,A
      STOP
      END


****      GLOBAL STATISTICS        ****
****   NO ERRORS,   NO WARNINGS    ****
TOTAL COMPILATION TIME   Ø:ØØ:Ø1
TOTAL ELAPSED TIME       Ø:ØØ:Ø3

 END OF COMPILE

 END OF PREPARE


?2,12,122


         THE SQUARE ROOT OF         2.ØØØ IS       1.41421

         THE SQUARE ROOT OF        12.ØØØ IS       3.4641Ø

         THE SQUARE ROOT OF       122.ØØØ IS      11.Ø4536
 END OF PROGRAM
```

Figure 4-2. Computed GOTO Statement Example

4-4. ASSIGNED GOTO STATEMENT

The assigned GOTO statement passes control to a statement label which has been assigned to a given variable.

The form of an assigned GOTO statement is

GOTO *variable*

For example,

GOTO JACKAL

or

GOTO *variable (label, label, . . . label)*

For example,

GOTO JACKAL (30, 40, 50, 60)

Note:

> The *assigned* GOTO statement passes control to whichever label is *assigned* to *variable*. This is different than the *computed* GOTO statement which passes control to one of the labels based on the result of a computation.

In the assigned GOTO statement,

variable
is an integer or double integer simple variable.

label
is an unsigned integer from 1 to 99999.

In either of the two statement forms (GOTO JACKAL or GOTO JACKAL(20, 30, 40)), *variable* must be given a *label value* through an ASSIGN statement prior to execution of the GOTO statement. (See Section III for a discussion of the ASSIGN statement.) When the assigned GOTO statement is executed, control is transferred to the statement whose label matches the *label value* of *variable*. (*Variable*, then, has two values, its *integer* or *double integer value* and the *label value* which has been assigned through an ASSIGN statement.)

The second form of the assigned GOTO statement includes a list of possible label values that *variable* might take and is included in the coding by the programmer merely to remind him of the places to which control might be transferred when this particular *variable* is mentioned in an assigned GOTO statement. (The computer does not verify that one of the label values in the list has actually been assigned to the variable.)

In the statement, GOTO JACKAL(30, 40, 50, 60), the variable JACKAL will have been assigned the label 30, 40, 50, or 60. Thus, when the statement is executed, control will transfer to statement number 30, 40, 50, *or* 60, depending on which *label value* has been assigned to JACKAL.

The assigned GOTO statement, whereby a *variable* is referenced instead of a *statement label,* is used whenever there is a chance that decision-making data may have been used and discarded earlier in the program. For an example of this type of usage, refer to the flowchart in figure 4-3 and the program in figure 4-4.

In figure 4-3, the computer reads a card to determine the sex of a retired person. If the person is female (SEX = 1), the label 60 is assigned to the variable SEX; if male (SEX = 2), the label 50 is assigned to SEX. Next, the computer reads another data item to determine the person's age. If the person is over 70 years old and is female, then she is assigned to a certain medical clinic for medical care. If the person is over 70 and is male, he is assigned to a different clinic. All persons under 70 are assigned to still another clinic. The sex of each retiree, then, must be checked again after the age has been determined. The sex infomation has been saved through the use of the ASSIGN statement which has assigned the *label value* 60 to the *variable* SEX if female, and 50 to SEX if male. When the age is checked, the assigned GOTO statement (GOTO SEX) passes control to statement 50 *or* 60, depending on the sex of the subject, and, from there to an output statement which prints the clinic where the retiree can receive medical care in the future.

4-5. IF STATEMENTS

Two types of IF statements are provided by FORTRAN/3000 for decision making. An *arithmetic* IF statement transfers control to one of three labeled statements depending on whether the expression evaluates to a negative, zero, or positive value. A *logical* IF statement causes execution of a statement if an evaluated expression is true.

4-6. ARITHMETIC IF STATEMENT

The arithmetic IF statement provides a means of directing control to one of three possible statements.

The form of the arithmetic IF statement is

IF *(expression) label, label, label*

For example,

$$\text{IF } \underset{\textit{expression}}{(\text{A})}\underbrace{30,40,50}_{\textit{labels}}$$

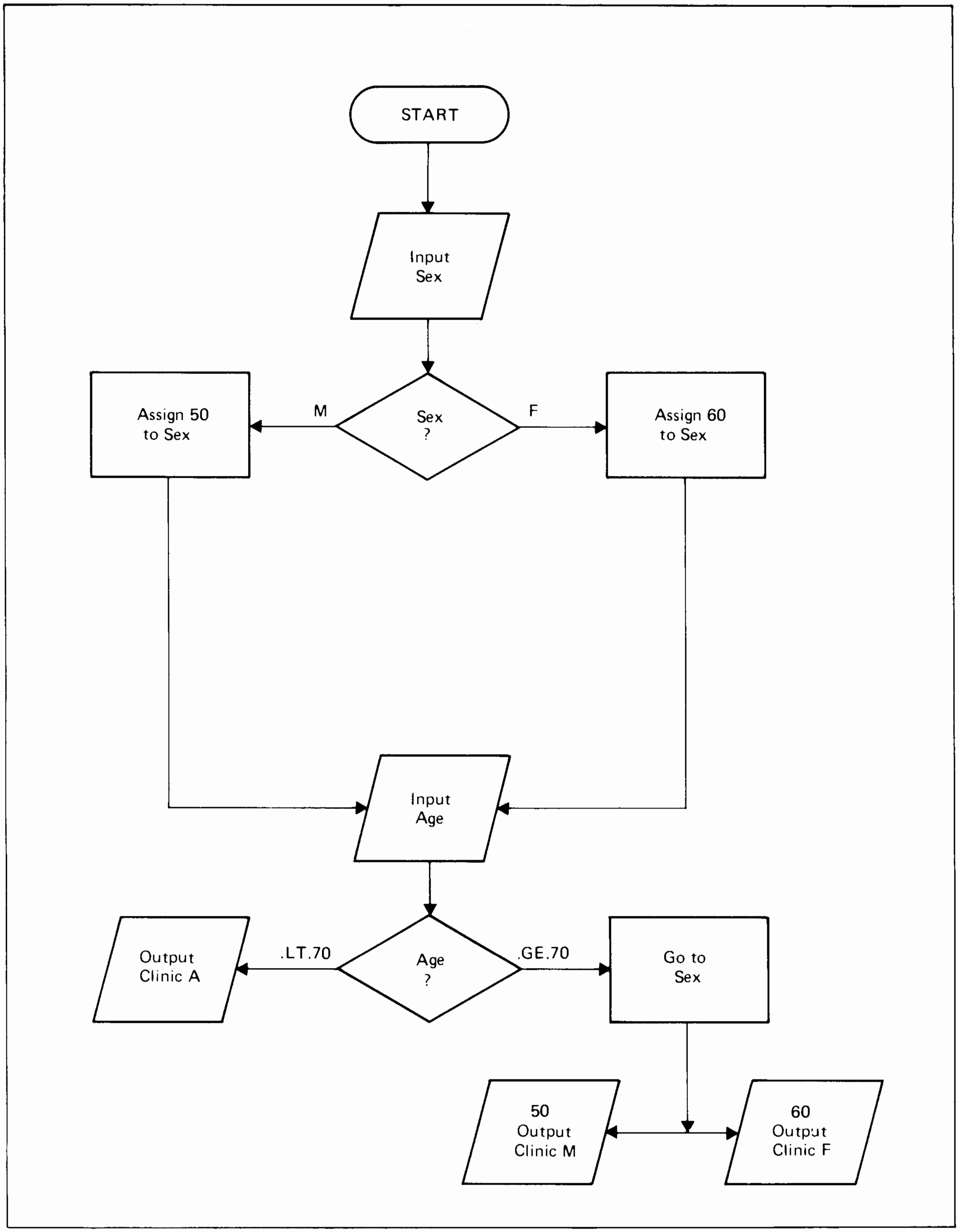


Figure 4-3. Assigned GOTO Statement Flowchart Example

```
:FORTGO FTRAN5

PAGE 0001   HP32102B.00.0

            PROGRAM ASSIGNED GOTO
      C
      C ASSIGNED GOTO STATEMENT EXAMPLE
      C
            INTEGER AGE,SEX
       100  FORMAT(I2)
       200  FORMAT('0',T10,"SUBJECT ASSIGNED TO CLINIC A")
       300  FORMAT('0',T10,"SUBJECT ASSIGNED TO CLINIC M")
       400  FORMAT('0',T10,"SUBJECT ASSIGNED TO CLINIC F")
       10   READ(5,100)SEX
            IF(SEX.EQ.1)GOTO 20
            ASSIGN 50 TO SEX
            GOTO 30
       20   ASSIGN 60 TO SEX
       30   READ(5,100)AGE
            IF(AGE.LT.70)GOTO 40
      C
      C THE NEXT STATEMENT IS AN ASSIGNED GOTO STATEMENT
      C
            GOTO SEX
       40   WRITE(6,200)
            GOTO 10
       50   WRITE(6,300)
            GOTO 10
       60   WRITE(6,400)
            GOTO 10
       500  STOP
            END


****      GLOBAL STATISTICS      ****
****   NO ERRORS,   NO WARNINGS  ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:00:03

 END OF COMPILE

 END OF PREPARE

1
78


        SUBJECT ASSIGNED TO CLINIC M
```

Figure 4-4. Assigned GOTO Statement Example

where

expression
is an arithmetic expression of any type except complex.

label
is an unsigned integer from 1 to 99999 denoting a statement label number.

The *expression* is evaluated. If the resulting value is negative, control is passed to the statement whose *label* is first in the list. If the evaluated value is zero, control is passed to the statement whose *label* is second in the list. If the evaluated value is positive, then the last *label* in the list is chosen and control passes to this statement number. Two of the labels in the label list may be the same in which case control will branch to one of two possible statements, or *all* of the labels in the list may be the same and control will branch to the statement bearing this label number regardless of the result of the evaluation.

The arithmetic IF statement is used to evaluate arguments and to branch to one of three possible statements depending on the outcome of the evaluation. For example, a program to compute income tax could branch to two different tax tables if the adjusted income is less than $25,000 or greater than or equal to $25,000 (the arithmetic IF statement does not have to branch to three different statements, two (or more) of the labels in the label list can be the same).

Figure 4-5 shows an example of arithmetic IF statement usage. In the sample program, if the expression (INCOME—25000) evaluates to a negative value or to zero, control is passed to statement 10. If the expression evaluates to a positive value, control is passed to statement number 20.

When system intrinsics (such as FOPEN, see Section VIII) are called by a FORTRAN/3000 statement, a *condition code* is returned which tells the program whether the intrinsic was accessed successfully or not. The condition returned signifies less than zero, equal to zero, or greater than zero.

The condition code is used in an arithmetic IF statement as follows:

```
     IF (.CC.) 30, 40, 50
```

The program will branch to statement 30 (if CC is less than zero), 40 (if CC equals zero) or 50 (if CC is greater than zero).

See Section VIII and Appendix A for a further discussion of condition codes.

4-7. LOGICAL IF STATEMENT

The logical IF statement evaluates a logical expression and executes a statement if the result of the evaluated expression is true.

The form of a logical IF statement is

IF *(logical expression) statement*

For example,

$$\text{IF } \underbrace{\text{(A .EQ. B)}}_{\textit{logical expression}}\ \underbrace{\text{GO TO 100}}_{\textit{statement}}$$

where

logical expression
is an expression as defined in Section III.

statement
is any executable statement other than a DO statement. The statement is executed if the *logical expression* evaluates *true,* otherwise the statement is skipped.

The logical IF statement is used as a two-way decision maker. If the *logical expression* contained in the IF statement is *true* when evaluated, then the *statement* contained in the IF statement is *executed.* If the *logical expression,* when evaluated, is *false,* then the statement contained in the IF statement is *not executed* and control passes to the next sequential statement in the program.

Figure 4-6 shows an example of logical IF statement usage.

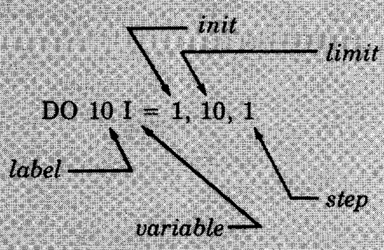
4-8. DO STATEMENT

A DO statement controls execution of a group of statements by causing the statements to be repeated a certain number of times.

The form of the DO statement is

DO *label variable = init, limit, step*

For example,



or

DO *label variable = init, limit*

For example,

DO 10 I = 1, 10 (*step* omitted)

```
:FORTGO FTRAN6

PAGE 0001    HP32102B.00.0

          PROGRAM ARITHMETIC IF
C
C ARITHMETIC IF STATEMENT EXAMPLE
C
          REAL NETINCOME
  100     FORMAT('0',"THE INCOME TAX IS ",M12.2)
          ACCEPT NETINCOME
C
C THE NEXT STATEMENT IS AN ARITHMETIC IF STATEMENT
C
          IF(NETINCOME-25000)10,10,20
  10      TAX=NETINCOME*.32
          GOTO 30
  20      TAX=NETINCOME*.36
  30      WRITE(6,100)TAX
          STOP
          END


****       GLOBAL STATISTICS       ****
****    NO ERRORS,    NO WARNINGS   ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME       0:00:05

 END OF COMPILE

 END OF PREPARE


?26457.98


THE INCOME TAX IS     $9,524.87
 END OF PROGRAM
```

Figure 4-5. Arithemtic IF Statement Example

```
:FORTGO FTRAN7

PAGE 0001   HP32102B.00.0

      PROGRAM LOGICAL IF
C
C LOGICAL IF STATEMENT EXAMPLE
C
 100  FORMAT(F10.4)
 200  FORMAT('0',T10,"A AND B ARE EQUAL")
 300  FORMAT('0',T10,"B IS LARGER")
 400  FORMAT('0',T10,"A IS LARGER")
      READ(5,100)A
      READ(5,100)B
C
C THE NEXT TWO STATEMENTS ARE LOGICAL IF STATEMENTS
C
      IF(A.EQ.B)GOTO 10
      IF(A.LT.B)GOTO 20
      WRITE(6,400)
      STOP
 10   WRITE(6,200)
      STOP
 20   WRITE(6,300)
      STOP
      END


****      GLOBAL STATISTICS       ****
****   NO ERRORS,    NO WARNINGS  ****
TOTAL COMPILATION TIME   0:00:01
TOTAL ELAPSED TIME       0:00:09

 END OF COMPILE

 END OF PREPARE

32.978
678.9


         B IS LARGER
 END OF PROGRAM
```

Figure 4-6. Logical IF Statement Example

where

label
is the statement label for the last statement of the group controlled by the DO statement.

variable
is an integer or double integer simple variable (the index) which is changed by the amount specified in *step* each time the group of statements within the DO loop is executed.

init
is the initial value given to *variable* at the start of execution of the DO statement.

limit
is the termination value for *variable*.

step
is the increment by which *variable* is changed after each execution of the group of statements defined by *label* (i.e., up through and including the statement defined by *label*). *Step* can be positive or negative, but not zero.

Init, limit, and *step* are indexing parameters. All three are arithmetic expressions of any type except complex, although their values are truncated to integer (or double integer) whenever they are used by the DO statement. If *step* is omitted, as for example, DO 10 I = 1, 10, it is assumed to be equal to 1. *Init* and *limit* can be positive, negative, or zero.

An example of DO statement usage is shown in figure 4-7. Note that the initial value of *init* is 2, and *step* is incremented by 2 each time the DO loop is executed. Thus, I (the index variable) assumes the values 2, 4, 6, 8, 10, 12, 14, 16, 18, and 20.

The termination statement of a DO loop may not be a GO TO statement, arithmetic IF statement, RETURN statement, STOP statement, DO statement, or a logical IF statement which contains any of the foregoing statements. A DO statement is used whenever it is necessary to cause a series of FORTRAN/3000 statements to be repeated. The DO statement defines this repetition, or loop. The *range* of the DO loop is defined as the first statement following the DO statement, up to and including the terminal statement referenced by *label*.

For example,

```
    A = 6

    DO 20 I = 1, 10, 1

    B = SQRT(A) ────┐
                    │
    WRITE (6, 200)B ├── Range of DO loop
                    │
 20 A = A + 1 ──────┘
```

When a DO statement is executed, the following steps occur:

1. The control variable (*variable* in the DO statement) is assigned the value of *init*.
2. Control is passed to the first executable statement after the DO statement and the *range* is executed.
3. The termination statement (defined by *label* in the DO statement) of the range is executed and *variable* is incremented by the value of *step*. If *step* is not mentioned in the DO statement, then *variable* is incremented by 1.
4. *Variable* is compared with *limit*.
   - If *step* is positive, the sequence is repeated if *variable* is less than or equal to *limit*. If *variable* exceeds *limit*, the DO loop is satisfied, and control transfers to the statement following the termination statement.
   - If *step* is negative, the sequence is repeated if variable is greater than or equal to *limit*. If *variable* is less than *limit*, the DO loop is satisfied and control transfers to the statement following the termination statement.

Step 4 indicates that two possible cases exist when comparing the control variable (*variable* in the DO statement) with the limit parameter (*limit* in the DO statement). When *step* is negative, *variable* must be less than *limit* before the DO loop passes control. When *step* is positive, *variable* must be greater than *limit* before the DO loop passes control. If either of the two cases exist when the loop is first entered, the statements in the range of the DO loop are executed once only.

Limit and *step* are computed only when the loop is entered. *Variable* can be redefined during execution of the DO loop.

For example,

```
    J = 1

    DO 10 I = 1, 20, 1

    K = 2 * * J

    J = J + 1

    IF (K .GT. 25000)I = 21

 10 CONTINUE
```

In the example, K is checked by the IF statement every time the DO loop is executed. When K becomes greater than 25000 the statement I = 21 is executed. The next time that *variable* (I) is compared with *limit* (20), *variable* equals 21 and exceeds *limit*. The DO loop is not repeated and control passes to the statement following statement number 10. Thus, variable was redefined during execution of the DO loop (K would become greater than 25000 after 16 iterations of the DO loop, at which time I would equal 16).

```
:FORTGO FTRAN8

PAGE 0001   HP32102B.00.0


         PROGRAM DO
C
C DO STATEMENT EXAMPLE
C
 100     FORMAT('0',T6,"NUMBER",T15,"SQUARE ROOT"//)
 200     FORMAT(T5,F7.4,T17,F7.5)
         WRITE(6,100)
C
C THE NEXT STATEMENT IS A DO STATEMENT
C
         DO 10 I=2,20,2
         A=I
         B=SQRT(A)
 10      WRITE(6,200)A,B
         STOP
         END



****       GLOBAL STATISTICS       ****
****    NO ERRORS,    NO WARNINGS  ****
TOTAL COMPILATION TIME   0:00:01
TOTAL ELAPSED TIME       0:00:03

 END OF COMPILE

 END OF PREPARE



    NUMBER    SQUARE ROOT


    2.0000      1.41421
    4.0000      2.00000
    6.0000      2.44949
    8.0000      2.82843
   10.0000      3.16228
   12.0000      3.46410
   14.0000      3.74166
   16.0000      4.00000
   18.0000      4.24264
   20.0000      4.47214
 END OF PROGRAM
```

Figure 4-7. DO Statement Example

Examples of DO loops are as follows:

```
100  FORMAT (I5)
     J = 0
     DO 10 I = 1, 10, 1
5    J = J + 1
10   K = J**J
     WRITE (6,100)K
     STOP
     END
```

Statements 5 and 10 will be executed 10 times. Each time the DO loop is executed, I is incremented by 1.

```
100  FORMAT (I5)
     J = 0
     DO 10 I = 1, 10, 2
5    J = J + 1

10   K = J**2
     WRITE (6,100)K
     STOP
     END
```

I will be incremented by 2 each time the preceding DO loop is executed.

```
100  FORMAT (I5)
     J = 0
     DO 10 I = 1, 10
5    J = J + 1
10   K = J**2
     WRITE (6, 100) K
     STOP
     END
```

Step was omitted from the last example. It is assumed to be 1 and I will be incremented by 1 each time the DO loop is executed.

4-9. NESTING DO LOOPS

If a DO loop contains another DO loop, the second (inner) loop is nested within a first (outer) loop. The last statement of the inner (nested) loop must either be the same as the last statement of the outer loop, or must occur before the last statement of the outer loop.

In the following example, the last, or terminating, statement of the innermost loop ocurs before the last statement of the outer loops.The two outer loops have the same terminating statement.

```
     DO 100 I = 1, 10, 1 ──────────────┐
        DO 100 J = a, 10, 2 ────────┐  │
        DO 90 K = 1, 10, 1 ────┐    │  │
        L = I**J               │    │  │
        WRITE (6, 100)L        │    │  │
 90     CONTINUE ──────────────┘    │  │
100     CONTINUE ───────────────────┴──┘
```

Control passes statement 100 only after all three loops are satisfied.

DO loops may be nested to as many levels as desired, as long as the ranges do not overlap. An example of overlapping ranges is as follows:

```
     DO 100 I = 1, 10 ───────┐
     DO 500 J = 1, 10 ────┐  │   ILLEGAL: THE RANGES
100  X = Y ───────────────┼──┘   OF THE TWO LOOPS
200  C = A*B ─────────────┘      OVERLAP
```

ILLEGAL

4-10. ENTERING AND EXITING DO LOOPS

A DO loop may be exited at any time, e.g., by a GO TO statement or a subprogram call, as long as the statement causing the passing of control out of the DO loop is not the termination statement of the loop. For example,

```
     DO 50 I = 1, 10, 2
     A = SQRT (C * D)
     WRITE (6, 100)A
     GO TO 500
50   CONTINUE
     C = C + 1
     D = C**2
500  D1 = C * D
```

In this example, control passes out of the DO loop during the first execution by means of the statement, GO TO 500.

It is possible to pass control into the range of a DO loop, but *only* if a transfer out of that same DO loop had occurred previously. The following example shows a legal transfer out of the range of a DO loop and back into the same range.

```
     DO 50 I = 1, 10, 1
     GO TO 70
20   X = Y * V + R
50   CONTINUE
70   VAL = BAN + 6
     GO TO 20
```

Instructions executed after a transfer out of a DO loop can include other DO statements. The range of any new DO statements, however, must not contain any means for exiting and reentering the range of the original DO loop before the limit of the new DO loop is satisfied. The following is an example of an illegal transfer:

```
     DO 100 I = 1, 10, 1
     Y = Y** 2
     GO TO 150
     GO TO 170
30   A = X
100  CONTINUE
150  DO 250 J = 1, 10, 2
     B = A * Y
170  IF (B .LE. 6.0) GO TO 30
250  CONTINUE
```

ILLEGAL

The loop defined by statements 150 and 250 contains a possible transfer out of the range of this loop (through statement 170) and into the range of the original before the limit of the second loop is satisfied.

When a transfer is made out of the first loop (through the statement, GO TO 150) and the second DO statement is executed, the first DO loop mechanism is altered. An attempt to reenter the first DO loop range before the second DO loop limits are satisfied might cause unpredictable results. Thus, transferring from the range of one DO loop and executing another DO statement is advisable only if the second DO loop range does not contain possible exit points other than those that result in normal satisfaction of the DO loop.

4-11. CONTINUE STATEMENT

A CONTINUE statement creates a reference or junction point in a FORTRAN/3000 program.

The form of a CONTINUE statement is

CONTINUE

or

***label* CONTINUE**

where

label
is a statement label number (an integer value from 1 to 99999). Because it is most always used as a reference or junction point in a program (such as the termination statement in a DO loop), a CONTINUE statement usually is labeled.

The CONTINUE statement usually is used as the last statement in a DO loop that otherwise would end in a prohibited statement such as a GO TO instruction. If a CONTINUE statement is used elsewhere in a program, or if it is not labeled, it has no function and control passes to the next statement.

EXAMPLE:

```
     DO 100 I = 1, 10
20   X = 1
     Y = SQRT(X)
     WRITE (6, 200) Y
     IF (X .LT. 25.0) GO TO 100
     GO TO 20
100  CONTINUE
```

The last executable statement of the DO loop in the example is a GOTO statement, which is not allowed to be the last statement in a DO loop. The CONTINUE statement is used therefore to terminate the loop.

4-12. STOP STATEMENT

A STOP statement causes termination of program execution.

The form of a STOP statement is

```
        STOP
or
        STOP integer
or
        STOP "character string"
or
        STOP 'character string'
```

where

integer
is an unsigned integer used to identify a specific STOP and is useful in determining at which point a program has terminated execution.

"character string"
is a message used to identify a particular stop. One use might be to state the reason for unexpectedly terminating execution.

Example 1,

```
 100   FORMAT (2F7.3)

 200   FORMAT (T12, "A IS LARGER")

 300   FORMAT (T12, "B IS LARGER")

       READ (5, 100) A, B

       IF (A .LT. B) GO TO 50

       WRITE (6, 200)

       STOP 10

 50    WRITE (6, 300)

       STOP 20

       END
```

In the example, if B is less than A, the program will terminate at the first STOP. If B equals or is greater than A, statement 50 will be executed and the program will terminate at the second STOP.

The STOP statements are followed by unsigned integers. When execution is terminated, the computer will output a message signifying the STOP at which program execution terminated.

Example 2,

```
 100   FORMAT (2F7.3)

 200   FORMAT (T12,"A IS LARGER")

 300   FORMAT (T12, "B IS LARGER")

       READ (5, 100) A, B

       IF (A.LT.B) GO TO 50

       WRITE (6, 200)

       STOP 'A GREATER THAN OR EQUAL TO B'

  50   WRITE (6, 300)

       STOP " A LESS THAN B "

       END
```

In the example, if B is less than A, the program will terminate at the first STOP. If B equals or is greater than A, statement 50 will be executed and the program will terminate at the second STOP.

The STOP statements are followed by the character strings " A LESS THAN B " and "A GREATER THAN OR EQUAL TO B". When execution is terminated, the computer will output a message signifying the STOP at which program execution terminated.

4-13. END STATEMENT

The END statement informs the compiler that the end of the code for a program unit has been reached. If there are other program units following, the compiler will progress to these units.

4-14. PAUSE STATEMENT

The PAUSE statement causes a program break if the program is operating in interactive mode and merely prints PAUSE (but does not cause a program break) if the program is operating in batch mode.

The form of a PAUSE statement is

```
        PAUSE
or
        PAUSE integer
or
        PAUSE "character string"
or
        PAUSE 'character string'
```

where

integer
is an unsigned integer used to identify a specific PAUSE. A PAUSE statement is used in interactive sessions whenever it is desirable to cause the program to break.

"character string"
is a message to be displayed on the job or session list device. It might be used to inform the user as to the reason for the pause.

Figure 4-8 is an example of PAUSE statements identified by integers (PAUSE 10 and PAUSE 20). In the program, if A is not less than or equal to B, program execution will pause and the pause number will be printed out on the terminal along with a prompt character (:). To resume program execution, type RESUME. For example,

For example,

PAUSE 10	*Typed out by the computer (along with the colon in the next line).*
:RESUME	*When RESUME is typed, the program will continue to execute.*

Note: The above performs the same function as using the BREAK key, and all MPE/3000 commands allowed in BREAK can be used.

Figure 4-9 contains STOP and PAUSE statements which are followed by character strings. If the first STOP is executed, the message STOP FOPEN FAILED. REASON UNKNOWN is displayed. If the PAUSE statement executes, the user is asked to create a file.

4-15. CALL STATEMENT

A CALL statement references and transfers control to an external procedure.

The form of a CALL statement is

CALL *name*

For example,

$\text{CALL } \overset{\textit{name}}{\text{FORMS}}$

or

CALL *name (param, param, . . . , param)*

as

$\text{CALL FORMS } (\underbrace{\text{A, B, C}}_{\textit{parameters}})$

or

CALL *name (param, param, . . . , param, $label, $label, . . . ,$label)*

as

$\text{CALL FORMS (A, B, C, } \underbrace{\text{\$30, \$40, \$50}}_{\textit{labels}})$

where

name
identifies the symbolic name of the procedure being called and must be identical to the name of the procedure.

param
is an actual argument defined by the program unit containing the CALL statement. Actual arguments must agree in number, order and type with the dummy arguments defined in the procedure being called. The actual arguments may be constants, simple variables, array names, expressions or function subprogram names.

label
is a statement label (prefixed with a $). *Label* identifies a statement in the calling program to which control may be returned when the procedure ends. The $ prefix is necessary to distinguish statement labels from integer constants. Labels must follow all other parameters (if any) in the parameter list.

CALL statements are *not* used to transfer control to function subprograms. Function subprograms are called *implicitly* by being referenced in an expression. For example, A = SQRT(B) calls the function subprogram for computing square root. The subprogram is called *implicitly* (i.e., merely by being referenced in the expression SQRT(B)). Thus, CALL statements are used only for transferring control to subroutine subprograms. When the subroutine is executed, the actual arguments (param) in the CALL statement are associated with their equivalent dummy arguments in the SUBROUTINE statement. The subroutine is then executed using the actual arguments. When a RETURN statement is executed (in the subroutine), control is returned to the statement following the CALL statement in the calling program unit. Control also can be returned to other statements in the calling program if a RETURN *n* statement is executed (see paragraph 4-16).

A call statement example is shown in figure 4-10.

In the example shown in figure 4-10, the main program first loads array JACK with integers from 1 to 10 by setting JACK(I) = I each time I is incremented. Integer variable J is assigned the value 6. CALL MULT(J) transfers control from the main program to the subroutine MULT. In the statement CALL MULT(J), J is the *actual* argument that is passed to the subroutine. In the state-

```
:FORTGO FTRAN9

PAGE ØØØ1    HP321Ø2B.ØØ.Ø


ØØØØ1ØØØ          PROGRAM PAUSE
ØØØØ2ØØØ    C
ØØØØ3ØØØ    C     PAUSE STATEMENT EXAMPLE
ØØØØ4ØØØ    C
ØØØØ5ØØØ          ACCEPT A,B
ØØØØ6ØØØ          IF(A.LE.B)GOTO 5Ø
ØØØØ7ØØØ          PAUSE 1Ø
ØØØØ8ØØØ          ACCEPT C
ØØØØ9ØØØ          IF(A.LE.C)GOTO 5Ø
ØØØ1ØØØØ          PAUSE 2Ø
ØØØ11ØØØ          ACCEPT D
ØØØ12ØØØ          X=A*D
ØØØ13ØØØ          DISPLAY "X = ",X
ØØØ14ØØØ    5Ø    STOP "SUCCESSFUL COMPLETION"
ØØØ15ØØØ          END




****      GLOBAL STATISTICS      ****
****    NO ERRORS,   NO WARNINGS  ****
TOTAL COMPILATION TIME   Ø:ØØ:Ø1
TOTAL ELAPSED TIME       Ø:ØØ:21

 END OF COMPILE

 END OF PREPARE

?
297865.89,45.32
PAUSE 1Ø
:RESUME
?
567.43
PAUSE 2Ø
:RESUME
?
5319.67

X =   .158455E+1Ø STOP  SUCCESSFUL COMPLETION

 END OF PROGRAM
```

Figure 4-8. PAUSE Statement Example

```
:FORTGO PAUSEXP

PAGE 0001   HP32102B.00.0

             PROGRAM PAUSE
       C
       C THIS PROGRAM DEMONSTRATES THE PAUSE STATEMENT WITH
       C A MESSAGE. THE SYSTEM INTRINSIC STATEMENT IS
       C DESCRIBED LATER IN THE MANUAL.
       C
             SYSTEM INTRINSIC FOPEN,FCHECK
             CHARACTER*9 FILENAME
             DATA FILENAME/"USERFILE "/
       100   IFNUM=FOPEN(FILENAME,%3L)
             IF(.CC.)200,400,200
       200   CALL FCHECK(IFNUM,IERRNO)
             IF(IERRNO.EQ.52)GO TO 300
             STOP "FOPEN FAILED.  REASON UNKNOWN"
       300   PAUSE "FOPEN FAILED, NON-EXISTENT FILE. PLEASE CREATE"
             GO TO 100
       400   STOP "SUCCESSFUL OPEN"
             END




****       GLOBAL STATISTICS        ****
****    NO ERRORS,    NO WARNINGS   ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:00:38

 END OF COMPILE

 END OF PREPARE

PAUSE FOPEN FAILED, NON-EXISTENT FILE. PLEASE CREATE
:FILE USERFILE=LP;DEV=FASTLP
:RESUME
STOP  SUCCESSFUL OPEN

 END OF PROGRAM
```

Figure 4-9. PAUSE Statement with Message Example

```
:FORTGO FTRAN1Ø

PAGE ØØØ1   HP321Ø2B.ØØ.Ø


                PROGRAM CALL
          C
          C CALL STATEMENT EXAMPLE
          C
          C THIS PROGRAM UNIT IS THE CALLING PROGRAM
          C
           1ØØ  FORMAT(I4)
                COMMON INT(1Ø),JACK(1Ø)
                DO 1Ø I=1,1Ø
           1Ø   JACK(I)=I
                J=6
          C
          C THE NEXT STATEMENT CALLS THE SUBROUTINE 'MULT'
          C
                CALL MULT(J)
                DO 2Ø I=1,1Ø
           2Ø   WRITE(6,1ØØ)INT(I)
                STOP
                END




          C
          C THE NEXT PROGRAM UNIT IS THE SUBROUTINE SUBPROGRAM
          C
                SUBROUTINE MULT(K)
                COMMON INT(1Ø),JACK(1Ø)
                DO 1Ø I=1,1Ø
           1Ø   INT(I)=JACK(I)*K
                RETURN
                END



****      GLOBAL STATISTICS         ****
****   NO ERRORS,    NO WARNINGS    ****
TOTAL COMPILATION TIME  Ø:ØØ:Ø1
TOTAL ELAPSED TIME      Ø:ØØ:Ø5

 END OF COMPILE

 END OF PREPARE


   6
  12
  18
  24
  3Ø
  36
  42
  48
  54
  6Ø
 END OF PROGRAM
```

Figure 4-10. CALL Statement Example

ment SUBROUTINE MULT(K), K is the *dummy* argument. The address in memory where the variable J is located is passed to the subroutine.The subroutine uses this address to indirectly reference the actual variable J.

The actual result of calling the subroutine is to multiply the number stored in each element of array JACK by 6 (J = 6) and store the result in the corresponding element of array INT. J is substituted for K whenever K appears in the subroutine. When the RETURN statement in the subroutine is executed, control is passed back to the statement following CALL MULT(J) in the main program. Since array INT is accessible to both the subroutine and the main program through COMMON statements, the main program can use the results of the subroutine computations.

4-16. RETURN STATEMENT

A RETURN statement transfers control from a subprogram back to the calling program unit.

The form of a RETURN statement is:

RETURN
or
RETURN *n*

where

n
is a positive integer constant or integer simple variable with positive value used as an index to point to a statement label in the label list in a CALL statement.

In a subroutine subprogram, executing a RETURN statement of the first form (without the *n*) returns program control to the statement following the CALL statement in the calling program. For example, in the subroutine MULT (previous example), the RETURN statement returns control to statement DO 20 M = 1, 10 (the statement following CALL MULT(J), which is the calling statement) in the main program.

The second form, RETURN *n*, is used to return control to a statement in the calling program unit other than the one after the CALL statement. An example of RETURN *n* statement usage is shown in figure 4-11. In the SUBROUTINE statement, a list of asterisks follows the two dummy parameters (K, L) to show that alternate return points exist. In the CALL statement in the main program, the asterisks are replaced by $10, $20, and $30. (The $ prefix is necessary to distinguish statement labels from integer constants.) Control is passed to the subroutine by the CALL statement.

The addresses in memory of I and J (the *actual* arguments) are used by the subroutine to determine the values of K and L (the *dummy* arguments). If K and L are equal, I is set equal to 1 and the RETURN I statement selects the first label ($10) in the label list. Control returns to statement 10 in the main program. If K is less than L, control is returned to statement 20; and if K is greater than L, control is returned to statement 30. If the index (I) is less than 1 or greater than the number of statement labels listed in the label list, a compiler diagnostic message results when index is a constant, and a run error results when the index is a simple variable.

In a function subprogram, only the first form of the RETURN statement is allowed. When a RETURN statement is executed in a function subprogram, control is returned to the statement containing the expression in the calling program which referenced the function. The value obtained by the function subprogram's computations is assigned to the function name and is used to continue evaluation of the referencing expression.

EXAMPLE:

```
100  FORMAT (2X, I5)
     I = 8
     J= 4
30   K = 6 + IDIV(I, J)
     WRITE (6, 100) K
     STOP
     END
     FUNCTION IDIV(L, M)
     IDIV = (L**2) + (M**3)
     RETURN
     END
```

The function reference (IDIV) in statement 30 of the main program passes control to function subprogram IDIV which computes a value for IDIV. The values of the actual arguments I and J are used by the dummy arguments L and M to compute the value of IDIV. The resulting value is assigned to function name IDIV.

When the RETURN statement is executed, control is passed back to the calling program and the evaluation of the expression in statement 30 continues.

If a RETURN statement is encountered in the main program, it is treated as a STOP statement. However, it is not recommended to use the RETURN statement in this fashion since the execution of the program is terminated at that stage, rather than transferred to some other point.

```
:FORTGO FTRAN11

PAGE 0001    HP32102B.00.0


00001000          PROGRAM RETURN I
00002000    C
00003000    C RETURN STATEMENT EXAMPLE
00004000    C
00005000     100  FORMAT('0',T10,"I AND J ARE EQUAL")
00006000     200  FORMAT('0',T10,"I IS LESS THAN J")
00007000     300  FORMAT('0',T10,"I IS LARGER THAN J")
00008000          ACCEPT I,J
00009000          CALL LARGE(I,J,$10,$20,$30)
00010000     10   WRITE(6,100)
00011000          STOP 10
00012000     20   WRITE(6,200)
00013000          STOP 20
00014000     30   WRITE(6,300)
00015000          STOP 30
00016000          END




00017000          SUBROUTINE LARGE(K,L,*,*,*)
00018000          IF(K.EQ.L)I=1
00019000          IF(K.LT.L)I=2
00020000          IF(K.GT.L)I=3
00021000          RETURN I
00022000          END


****      GLOBAL STATISTICS        ****
****   NO ERRORS,    NO WARNINGS   ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:00:04

 END OF COMPILE

 END OF PREPARE


?32,78


       I IS LESS THAN J
STOP 20
 END OF PROGRAM
```

Figure 4-11. RETURN n Statement Example

4-17. TRAP HANDLING

The trap handling mechanism allows you to write a subroutine to which control is transferred if a trap condition is encountered. Upon exit from the subroutine, execution continues with the instruction following the one which was interrupted.

Whenever a major error occurs during the execution of a hardware instruction, a procedure from the System Library, or an intrinsic called by the user, normally the user's program is aborted and an error message is output. You can, however avoid an immediate abort by enabling any of the following software traps:

1. Arithmetic Trap
2. System Trap
3. Basic External Function Trap
4. Internal Function Trap
5. Format Trap
5. Plot Trap
7. CONTROLY Trap (not actually an error)

When an error occurs, the corresponding trap, if enabled, suppresses output of the normal error message, transfers control to a *trap procedure* defined by you, and passes one or more parameters describing the error to this procedure. This procedure may attempt to analyze or recover from the error, or may execute some other programming path specified by you.

Upon exiting from the trap procedure, control returns to the instruction following the one that activated the trap. In the case of the library traps, however, you can specify that the process be aborted when control exits from the trap procedure. Trap intrinsics can be invoked from within trap procedures.

The flow of control must pass through a TRAP statement before the action specified can occur, since there are some internal parameters which must be set. In particular, ON statements may appear only where any executable statement may appear. Once enabled, the condition can only be disabled by another ON statement which specifies the same TRAP condition.

The form of a Trap statement is

ON *error condition* CALL *subroutine*

or

ON *error condition* ABORT

or

ON CONTROLY CALL *subroutine*

For example,

ON INTEGER*2 DIV 0 CALL DIVZERO

A FORTRAN subroutine containing a TRAP statement may not be added to the System Library since the Trap handling mechanism requires a special COMMON to store the external label of the called subroutine (*plabel*).

Error conditions described in the following paragraphs are recognized.

4-18. ARITHMETIC ERRORS

The following syntax is recognized for arithmetic error conditions.

1. REAL DIV 0
2. REAL OVERFLOW
3. REAL UNDERFLOW
4. DOUBLE PRECISION DIV 0
5. DOUBLE PRECISION OVERFLOW
6. DOUBLE PRECISION UNDERFLOW
7. INTEGER*2 DIV 0
8. INTEGER*2 OVERFLOW
9. INTEGER*4 OVERFLOW
10. INTEGER*4 DIV 0
11. INTEGER DIV 0
    This maps into INTEGER*4 DIV 0 when the $INTEGER*4 control option has been invoked. Otherwise this maps into INTEGER*2 DIV 0.
12. INTEGER OVERFLOW
    This maps into INTEGER*4 OVERFLOW when the $INTEGER*4 control option has been invoked. Otherwise this maps into INTEGER*2 OVERFLOW.

In each of the above cases, the corresponding subroutine requires one reference parameter which is of the same type as that associated with the error condition. When the subroutine is called, the parameter will be the result of the operation which caused the trap to be invoked.

Parameter type checking will be performed by the segmenter.

Note:

> No provision is made for handling commercial instruction traps due to the diversity of the possible number of elements in the stack and the nature of each of those elements.

Example,

```
$CONTROL USLINIT
      PROGRAM TRAPTEST
C     TRAP TESTING EXAMPLE
      ACCEPT I,J,K,
      ON INTEGER*2 DIV 0 CALL DIVZERO
      L=I/J+K
      WRITE(6,100)L
```

```
100  FORMAT (T5, "THE VALUE OF L IS",F10.3)
     STOP
     END
     SUBROUTINE DIVZERO(IRESULT)
     INTEGER*2 IRESULT
     DISPLAY "DIVIDE BY ZERO ENCOUNTERED"
     DISPLAY "RESULT DEFAULTED TO ZERO"
     IRESULT= 0
     RETURN
     END
```

4-19. SYSTEM ERRORS

The syntax for system errors is

SYSTEM ERROR

The subroutine which is called as a result of this kind of error must have one parameter which is an integer array. Parameter type checking will be performed by the segmenter. The contents of the parameter when the subroutine is called will be the array of eight parameters which are placed there by the system and defined in the *MPE Intrinsics Reference Manual* under System Traps. This parameter group immediately follows the parameters to the intrinsic in which the error occurred.

4-20. BASIC EXTERNAL FUNCTION ERRORS

The syntax for basic external function errors is

EXTERNAL ERROR

The subroutine which is called must have four formal parameters in the following order.

1. Single integer containing the error number (1 to 50) which is determined by the external function in which the error occurred. If the error number is set to zero by the user upon exit from the user-provided subroutine, then a normal termination sequence will occur with the standard error message being printed.

2. The result.

3. The first operand.

4. The second operand. (contains garbage for errors 5 to 13; only one argument).

An external function error is shown in figure 4-12.

Note: The correspondence between error numbers and external routines is listed in the *Compiler Library Reference Manual.*

4-21. INTERNAL FUNCTION ERRORS

Internal functions are those which are called implicitly by a user program, such as exponential routines.

The syntax for internal function errors is

INTERNAL ERROR

The subroutine called requires two formal parameters in the following order.

1. Single integer containing the error number (51 to 99) which is determined by the internal function in which the error occurred. If the error number is set to zero by the user upon exit from the user-provided subroutine, then a normal termination sequence will occur with the standard error message being printed.

2. The result.

Note: The correspondence between the error number and internal routine is listed in the *Compiler Library Reference Manual*.

An internal function error is shown in figure 4-12.

4-22. FORMAT ERRORS

Errors occurring during formatting of input or output result in a call to a subroutine specified by the user if the format error condition is specified in a trap statement.

The syntax for format errors is

FORMAT ERROR

The subroutine called must have seven formal parameters in the following order.

1. Single integer containing the error number (101 to 149) which corresponds to a particular format error (see the *Compiler Library Reference Manual* for correspondence). If the error number is set to zero by the user upon exit from the user-provided subroutine then a normal termination sequence will occur with the standard error message being printed.

2. The format. (character array)

```
:FORTGO LIBTRAPX

PAGE 0001    HP32102B.00.0

           PROGRAM STEST
    C
    C      FORCE LIBRARY ERRORS TO DEMONSTRATE TRAP HANDLING
    C
           REAL R
           COMMON /Z/II
    C
           ACCEPT II
           ON INTERNAL ERROR CALL INTERROR
           ON EXTERNAL ERROR CALL EXTERROR
           IF (II.GE.0) GO TO 5
           II = -II
           ON INTERNAL ERROR ABORT
           ON EXTERNAL ERROR ABORT
         5 GO TO (10,160),II
        10 CONTINUE
           R = ATAN2(0E0,0E0)
       160 CONTINUE
           R = 0.0**(-.5)
           STOP 1
           END


    $CONTROL CHECK=0
           SUBROUTINE INTERROR(ERRNO,RES)
           INTEGER*2 ERRNO
           DOUBLE PRECISION RES
           REAL RRES
           INTEGER*4 DIRES
           INTEGER*2 IRES
           EQUIVALENCE (RRES,DIRES,IRES)
           COMMON /Z/II
           WRITE(6,1) ERRNO
    1      FORMAT("0","*** INTERNAL ERROR NUMBER ",I4)
           IF (ERRNO.LT.57.OR.ERRNO.GT.59) GO TO 100
           DISPLAY "DOUBLE PRECISION RESULT = ",RES
           GO TO 9000
    100    RRES = RES
           IF (ERRNO.NE.56.AND.ERRNO.NE.55) GO TO 110
           DISPLAY "REAL RESULT = ",RRES
           GO TO 9000
    110    IF (ERRNO.LT.67) GO TO 120
           DISPLAY "DOUBLE INTEGER RESULT = ",DIRES
           GO TO 9000
    120    DISPLAY "INTEGER RESULT = ",IRES
    9000   IF (II.NE.0) ERRNO = 0
           RETURN
           END
```

Figure 4-12. Internal, External Function Error Example (Sheet 1 of 3)

```
$CONTROL CHECK=Ø
      SUBROUTINE EXTERROR(ERRNO,RES,OP1,OP2)
      INTEGER*2 ERRNO
      DOUBLE PRECISION RES,OP1,OP2
      REAL RRES,ROP1,ROP2
      LOGICAL ONEARG
      COMMON /Z/II
      ONEARG = .FALSE.
      WRITE(6,1) ERRNO
1     FORMAT("Ø","*** EXTERNAL ERROR NUMBER ",I4)
      IF (ERRNO.GE.5) ONEARG = .TRUE.
      IF (ERRNO.EQ.3.OR.ERRNO.EQ.4.OR.ERRNO.EQ.6.OR.ERRNO.EQ.8
     1 .OR.ERRNO.EQ.11.OR.ERRNO.EQ.13) GO TO 1ØØ
      ROP1 = OP1
      DISPLAY "FIRST OPERAND = ",ROP1
      IF (ONEARG) GO TO 11Ø
      ROP2 = OP2
      DISPLAY "SECOND OPERAND = ",ROP2
11Ø   RRES = RES
      DISPLAY "RESULT = ",RRES
      IF (II.NE.Ø) ERRNO = Ø
      RETURN
1ØØ   DISPLAY "FIRST OPERAND = ",OP1
       IF (.NOT.ONEARG) DISPLAY "SECOND OPERAND = ",OP2
      DISPLAY "RESULT = ",RES
      IF (II.NE.Ø) ERRNO = Ø
      RETURN
      END




****      GLOBAL STATISTICS        ****
****   NO ERRORS,   NO WARNINGS    ****
TOTAL COMPILATION TIME  Ø:ØØ:Ø2
TOTAL ELAPSED TIME      Ø:Ø2:Ø8

 END OF COMPILE

 END OF PREPARE

?
Ø
```

Figure 4-12. Internal, External Function Error Example (Sheet 2 of 3)

```
*** EXTERNAL ERROR NUMBER
FIRST OPERAND =    .ØØØØØØE+ØØ
SECOND OPERAND =    .ØØØØØØE+ØØ
RESULT =    .ØØØØØØE+ØØ

*** INTERNAL ERROR NUMBER   56
REAL RESULT =    .ØØØØØØE+ØØ
STOP 1

 END OF PROGRAM
:RUN $OLDPASS

?
1


*** EXTERNAL ERROR NUMBER     1
FIRST OPERAND =    .ØØØØØØE+ØØ
SECOND OPERAND =    .ØØØØØØE+ØØ
RESULT =    .ØØØØØØE+ØØ ATAN2:  ARGUMENTS ZERO
X=              .ØØØØØØE+ØØ
Y=              .ØØØØØØE+ØØ

       ***       STACK  DISPLAY       ***


                          S=ØØØ144     DL=177602     Z=ØØ1467
          Q=ØØØ15Ø P=ØØ2Ø6Ø  LCST= S153  STAT=U,1,Ø,L,Ø,Ø,CCE    X=ØØØØØ2

          Q=ØØØ137 P=ØØ4352  LCST= S153  STAT=U,1,Ø,L,Ø,Ø,CCE    X=ØØØØØ2
          Q=ØØØØ7Ø P=ØØØ735  LCST= S154  STAT=U,1,Ø,L,Ø,1,CCG    X=ØØØ715
          Q=ØØØØ51 P=ØØØ714  LCST=  ØØØ  STAT=U,1,1,L,Ø,1,CCG    X=ØØØØØ1

  ABORT :$OLDPASS...%Ø.%714:SYSL.%153.%4354: PROCESS QUIT

ERR 2
:RUN $OLDPASS
?
-2
RTOR: ILLEGAL ARGUMENTS
X=              .ØØØØØØE+ØØ
Y=             -.5ØØØØØE+ØØ

       ***       STACK  DISPLAY       ***


                          S=ØØØ13Ø     DL=177602     Z=ØØ1467
          Q=ØØØ134 P=ØØ2Ø6Ø  LCST= S153  STAT=U,1,1,L,Ø,Ø,CCE    X=ØØØØ2Ø

          Q=ØØØ123 P=ØØ4352  LCST= S153  STAT=U,1,1,L,Ø,Ø,CCE    X=ØØØØ2Ø
          Q=ØØØØ54 P=ØØØ432  LCST= S154  STAT=U,1,1,L,Ø,Ø,CCG    X=ØØØØØØ
          Q=ØØØØ45 P=ØØØ722  LCST=  ØØØ  STAT=U,1,1,L,Ø,Ø,CCL    X=ØØØØØØ

  ABORT :$OLDPASS...%Ø.%722:SYSL.%153.%4354: PROCESS QUIT

ERR 2
:
```

Figure 4-12. Internal, External Function Error Example (Sheet 3 of 3)

3. Offset to the location in the format where the error was detected from the start of the format.

4. The I/O buffer. (character array)

5. Offset to the location in the I/O buffer where the error was detected from the start of the buffer.

6. Single integer containing the FORTRAN unit number.

7. Single integer containing the MPE file number.

Figure 4-13 is an example showing a format error.

4-23. PLOT ERROR

Errors occurring during the generation of plots result in a call to a subroutine specified by the user, if the plotting error condition is specified in a trap statement.

The syntax for plotting errors is

PLOT ERROR

The subroutine called must have the following formal parameter.

> Single integer containing the error number (150 to 157) which corresponds to a particular plot error (see the *Compiler Library Reference Manual* for correspondence). If the error number is set to zero by the user upon exit from the user-provided subroutine then a normal termination sequence will occur with the standard error message being printed.

4-24. CONTROL Y

If CONTROLY is specified in a trap statement, and you enter a CONTROLY from the terminal during execution of the program, then the specified user subroutine is called.

There are no parameters allowed in the subroutine.

4-25. ABORT

If ABORT is specified instead of "CALL subroutine" for an error condition, then when the trap condition is encountered the contents of the registers S, DL, Z, Q, P, LCST, and X are listed (DB relative where applicable). A trace back of stack markers also is made. The program terminates at this point. If you desire more information than this, a subroutine may be written to call STACKDUMP to list whatever is required, using the normal mechanism.

An example showing the use of ABORT in an error condition is shown in figure 4-13.

4-26. ENABLING OF TRAPS BY USER. You can enable traps by appropriate calls to system intrinsics.

You may not, however, enable a trap explicitly with the appropriate system intrinsic and have a trap statement in effect at the same time within a category (see below). Enabling a trap with an intrinsic disables all trap statements in the same category, and a trap statement disables any explicitly enabled trap. The reason for this is that FORTRAN must use the same linkages and intrinsics as you would use if you set up your trap handlers using intrinsics described in the *MPE Intrinsics Reference Manual.*

Correspondence between MPE intrinsics which enable traps and trap statements is shown below:

Trap Enabled	**Trap Statements**
XARITRAP	Arithmetic Traps
XLIBTRAP	External, Internal, Format, Plot
XSYSTRAP	System Traps
XCONTRAP	CONTROLY

Note: For more information about traps, see the *MPE Intrinsics Reference Manual.*

```
:FORTGO FMTTRAPX

PAGE 0001    HP32102B.00.0


          COMMON /Z/ J
          DISPLAY "TEST FORMATTER ERROR RECOVERY"
          DISPLAY " "
          ACCEPT J
          ON FORMAT ERROR CALL FMTERROR
          IF (J.GE.0) GO TO 220
          J=-J
          ON FORMAT ERROR ABORT
     220  I=11
          DISPLAY "NUMBER OUT OF RANGE ERROR"
          READ(5,215) A
     215  FORMAT(E13.6)
          STOP
          END


$CONTROL CHECK=0
          SUBROUTINE FMTERROR(NUM,FMT,FMTLOC,BUF,BUFLOC,UNIT,FNUM)
          CHARACTER*100 FMT,BUF
          INTEGER*2 FMTLOC,BUFLOC,NUM,UNIT,FNUM
          COMMON /Z/ K
          WRITE(6,1) NUM
1          FORMAT(" ","FORMAT ERROR NUMBER ",I4)
          WRITE(6,2) UNIT
2         FORMAT(" ","UNIT NUMBER ",I4)
          WRITE(6,3) FNUM
3         FORMAT(" ","MPE FILE NUMBER",I4)
          IF (NUM.EQ.102.OR.NUM.EQ.113) GO TO 9000
          IF (FMTLOC.EQ.0) GO TO 100
          DISPLAY "GOOD FORMAT = ",FMT[1:FMTLOC]
100       IF (BUFLOC.EQ.0) GO TO 9000
          DISPLAY "GOOD DATA = ",BUF[1:BUFLOC]
9000      IF (K.NE.0) NUM=0
          RETURN
          END




****       GLOBAL STATISTICS       ****
****    NO ERRORS,    NO WARNINGS  ****
TOTAL COMPILATION TIME   0:00:02
TOTAL ELAPSED TIME       0:00:54

 END OF COMPILE

 END OF PREPARE


TEST FORMATTER ERROR RECOVERY

?
0

NUMBER OUT OF RANGE ERROR 1.2345678E96
```

Figure 4-13. Format Error Example (Sheet 1 of 2)

```
FORMAT ERROR NUMBER  106
UNIT NUMBER    5
MPE FILE NUMBER   4
GOOD FORMAT =  (E13.
GOOD DATA =  1.2345678E96

 END OF PROGRAM
:RUN $OLDPASS


TEST FORMATTER ERROR RECOVERY

?
1

NUMBER OUT OF RANGE ERROR 1.2345678E96

FORMAT ERROR NUMBER  106
UNIT NUMBER    5
MPE FILE NUMBER   4
GOOD FORMAT =  (E13.
GOOD DATA =  1.2345678E96
1.2345678E96
            ^
NUMBER OUT OF RANGE

       ***       STACK  DISPLAY       ***


                        S=000332     DL=177602     Z=001467
          Q=000336 P=002060  LCST= S153  STAT=U,1,1,L,0,0,CCE   X=000072

          Q=000325 P=003174  LCST= S153  STAT=U,1,1,L,0,0,CCE   X=000072
          Q=000244 P=000512  LCST= S114  STAT=U,1,1,L,0,1,CCG   X=177777
          Q=000054 P=000427  LCST=  000  STAT=U,1,1,L,0,0,CCG   X=000000

  ABORT :$OLDPASS...%0.%427:SYSL.%153.%3176: PROCESS QUIT

ERR 2
:RUN $OLDPASS


TEST FORMATTER ERROR RECOVERY

?
-121!!!
-1

NUMBER OUT OF RANGE ERROR 1.2345678E96
1.2345678E96

NUMBER OUT OF RANGE

       ***       STACK  DISPLAY       ***


                        S=000332     DL=177602     Z=001467
          Q=000336 P=002060  LCST= S153  STAT=U,1,1,L,0,0,CCE   X=000072

          Q=000325 P=003174  LCST= S153  STAT=U,1,1,L,0,0,CCE   X=000072
          Q=000244 P=000512  LCST= S114  STAT=U,1,1,L,0,1,CCG   X=177777
          Q=000054 P=000427  LCST=  000  STAT=U,1,1,L,0,0,CCG   X=000000

  ABORT :$OLDPASS...%0.%427:SYSL.%153.%3176: PROCESS QUIT

ERR 2
```

Figure 4-13. Format Error Example (Sheet 2 of 2)

WRITING DECLARATION STATEMENTS IN FORTRAN/3000

SECTION V

Declaration statements define the characteristics of data used in FORTRAN/3000 source programs, and, as such, are non-executable. When compiled, declaration statements do not provide instructions in the object program. Declaration statements must appear before the first executable statement in each program unit (whether main program or subprogram), and in the following order:

$$\left.\begin{array}{l}\text{IMPLICIT}\end{array}\right\}\text{level 1}$$

$$\left.\begin{array}{l}\text{DIMENSION}\\\text{COMMON}\\\text{EQUIVALENCE}\\\text{Type statements}\\\text{EXTERNAL}\end{array}\right\}\text{level 2}$$

$$\left.\begin{array}{l}\text{DATA}\\\text{Statement functions}\end{array}\right\}\text{level 3}$$

Order within the same level can change.

5-1. TYPE STATEMENT

Type statements assign an explicit type to symbolic names representing variables, arrays and function subprograms which would otherwise have their type implicitly determined by the first letter of their symbolic names. (See Section II, paragraph 2-19.)

The Type statement form is

type list

For example,

$$\underset{\uparrow\ \textit{type}}{\text{INTEGER}}\ \underbrace{\text{A, B, C, BETA}}_{\textit{list}}$$

where

type
may be INTEGER, INTEGER*2, INTEGER*4, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, CHARACTER or CHARACTER**x* (**x* is the length attribute of the CHARACTER heading and specifies the length of character variables. If the length attribute is omitted, the length is assumed to be 1.) If the length attribute is a variable it must be enclosed in parentheses.

INTEGER and INTEGER*2 define single word integers. INTEGER*4 defines a double integer.

list

is one or more variable names, array names, function names or array declarators (array names with dimensions). When type is CHARACTER, the variable or array name may have **x* as a suffix to designate the length.

In addition to being used for assigning explicit types to symbolic names, the Type statement, by using a complete array declarator (see paragraph 5-2) in *list*, may be used to reserve memory space in the same manner as a DIMENSION statement. (See paragraph 5-4 for a discussion of DIMENSION statements.)

For example,

```
    INTEGER RN
```

The above statement specifies that RN is type integer.

```
    REAL ITEM
```

ITEM is type real.

```
    INTEGER A(5,5,5)
```

The above statement specifies that array A is type integer and that it has three dimensions of five elements each.

```
    INTEGER*4 C(4,4)
```

The above statement specifies that array C is type double integer and that it has two dimensions of four elements each.

If a complete array declarator is used in *list* (see the last example), the declarator for that array must not be used in any other declaration statement (such as DIMENSION or COMMON). If a variable name only is used, then an array declarator must appear within a DIMENSION or COMMON statement (but not both) somewhere within the same program unit (so that the compiler will recognize the variable as an array name).

Type character is assigned to symbolic names as follows:

```
    CHARACTER A(3),B(4),C(5)
```

The above statement specifies that array A is type character with 3 elements, array B is type character with 4 elements and array C is type character with 5 elements. Since the length attribute (**x*) was omitted, the number of characters in each element of each array is assumed to be one.

```
    CHARACTER*10 A(3), B(4), C(5)
```

The above statement specifies that each element of arrays A, B, and C has a length of 10 characters (contains 10 characters in each element).

```
    CHARACTER*10 A(3), B*6(3)
```

This statement specifies that array A is type character

with 3 elements and that each element has a length of 10 characters; array B is type character with 3 elements and each element has a length of 6 characters. Thus, the length of type character variables can be defined in three ways:

1. By using the CHARACTER heading only, in which case the length of each element is implied to be one.

2. Through the length attribute (*x) following the CHARACTER heading. If the length attribute is a variable it must be enclosed in parentheses.

3. Through individual length attributes following character symbolic names. For example, CHARACTER*40 W,X,Y,Z defines the variable names W, X, Y and Z as type character, each with a length of 40. Thus, the length attribute following the CHARACTER heading specifies the length of all variables not having individual length attributes specified. Individual variables, however, can have their length specified. For example, CHARACTER*30 X,Y*20,Z specifies character variables (X and Z) of length 30, and character variable (Y) of length 20.

Character variables and character array names are the only data elements which may have length attributes. Elements in a Type statement *list* with any other heading cannot have length attributes.

A character variable occupies memory space according to the following rule:

(Character string length) x (number of elements) x (one-half word)

For example,

CHARACTER*3 CH(3)

The Type statement specifies that CH is a character array of 3 elements and that each element has a length of 3 characters. Array CH, therefore, would occupy: character string length (3) x elements (3) x 1/2 = 4-1/2 words of memory, or 9 *bytes* (byte = one-half word (8 bits)).

5-2. ARRAY DECLARATORS

Array declarators are used in conjunction with DIMENSION, COMMON and Type statements to define the number of elements, the type of data to be stored in the elements, and the arrangement of the elements in an array. This information is supplied to the computer through the array declarator.

The form of an array declarator is

name (b_1 , . . . , b_n)

For example,

ARR(3,3)

where

name
is a variable denoting the name of the array. The array type is specified through a Type statement (see paragraph 5-1), IMPLICIT statement (see paragraph 5-13) or through the implicit typing convention of using the first letter of the variable name to specify the variable type. (The letters I, J, K, L, M and N specify type integer; variable names starting with any other letter are type real.)

b_1, . . . ,b_n
are integers and specify the array *bounds*. The array bounds indicate the number of dimensions of the array (the maximum is 255), and the maximum number of elements in each dimension. The product, *elements x dimensions* determines the maximum number of elements allowed. The *absolute* maximum is approximately 30,000; the *actual* maximum, however, varies with each installation and is dependent upon the maximum stack size established at configuration time.

The array declarator I(3,4,5) indicates a three-dimensional array (signified by the fact that there are three subscripts (3,4,5) of type integer. The maximum subscript (bound) prescribed for each of the three dimensions is 3, 4, and 5, respectively.

The total number of elements in an array is calculated by multiplying the array bounds. For example, I(3,4,5) indicates that array I contains 3 x 4 x 5 = 60 elements. The number of words of memory needed to store an array is determined by the number of elements in the array and the type of data which the elements contain. Integer and logical arrays store each element of an array in a single 16-bit computer word; double integer arrays store each element of an array in two words; real arrays store each element in two words; double precision real arrays and complex arrays store each element in four words.

If array I(3,4,5) is type integer, then, it takes 3 x 4 x 5 x 1 = 60 words of memory. Real array A(3,4,5) takes 3 x 4 x 5 x 2 = 120 words of memory. Double integer array A(3,4,5) takes 3 x 4 x 5 x 2 = 120 words of memory.

As mentioned previously, character arrays take one-half word of memory storage for each character per element. Character array CH(3,3,3), then, with elements consisting of 3 characters each (CHARACTER*3) would take 3 x 3 x 3 x 3 x 1/2 = 40-1/2 words of memory storage.

Arrays are stored as one-dimensional arrays in memory according to the Array Successor Function, described in paragraph 5-8, *Equivalence Between Arrays of Different Dimensions*.

A complete array declarator can be declared only once in a program unit, while the array name may appear in several declaration statements. For example, if the array declarator is used in a DIMENSION statement, the array *name* only (not the *complete* array declarator) can be used in a COMMON or Type statement.

If the complete array declarator is used in a COMMON or Type statement, the array need not be mentioned in a DIMENSION statement. The array declarator used in the COMMON or Type statement creates the necessary storage space in memory, just as if the array were mentioned in a DIMENSION statement.

For example,

```
        INTEGER ARR(4,4)
```

has the same effect as

```
        INTEGER ARR
        DIMENSION ARR(4,4)
```

or

```
        INTEGER*4 ARR(4,4)
```

has the same effect as

```
        INTEGER*4 ARR
        DIMENSION ARR(4,4)
```

Normally, array bounds ($b_1 \ldots, b_n$) and character length are specified by positive integer constants and the bounds (and length) are fixed by the values of these constants. It is possible, however, to use adjustable array declarators in subprograms. In this case, the array bounds (or character length) are specified by integer simple variables instead of integer constants. (See Paragraph 11-6 for more information.)

Figure 5-1 is an example of adjustable array declarators and character length. The example declares an array IARR, and a character variable A, in the main program. Array IARR has two dimensions of 10 elements each and A has a length of 10 characters. A subroutine (SB) then is called to fill array IARR with values and to set A equal to "THE START." The variables I and J are set equal to the array bounds and K is set equal to the character length and these variables are used as the actual arguments to be passed to the subroutine. The subroutine dummy arguments L, M, and N assume the values passed to them through I, J, and K. These variables (L, M, and N) then are used in a DIMENSION and Type statement to establish the bounds for array IVAR and the length for variable Z in the subroutine. When character length is expressed as a variable as in CHARACTER Z*(N), the variable must be enclosed in parentheses. Note that the array bounds were passed to the subroutine.

Adjustable array declarators cannot be used in COMMON statements.

5-3. PARAMETER STATEMENT

A PARAMETER statement allows a constant to be given a symbolic name. All types are allowed, including character. The name then may appear anywhere a constant may appear, except in another PARAMETER statement, FORMAT statement, composite number, or Type statement. The name may not appear where a label value is required.

The form of a PARAMETER statement is

PARAMETER ***name=constant*** **[*,name=constant*] . . .**

For example,

```
                 ┌─name
    PARAMETER I=%3615D
    PARAMETER N=3.2E+5,S=1
              └──────┴─ names
```

where

name
is the symbolic name given to the constant.

The type is determined solely by the constant and not by the initial letter of the name.

In the first example the constant is of type double precision and in the second example the constants are real and integer, respectively. Thus I is of type double precision, N is type real and S is type integer.

The name may not be used as part of another constant except as the real or imaginary part of a complex constant.

A PARAMETER statement must appear before DATA statements, statement function statements, and executable statements.

Examples:

```
        PARAMETER I=%3615D       double precision
        PARAMETER N=3.2E+5,S=1      real, integer
```

5-4. DIMENSION STATEMENT

A DIMENSION statement defines the dimensions and bounds of arrays.

The form of a DIMENSION statement is

DIMENSION ***name(bounds),name(bounds), . . . , name(bounds)***

For example,

```
                  ┌── name ──┐
    DIMENSION ARR(3,3),ABLE(10)
                 └────────┴── bounds
```

where

name(bounds)
is an array declarator:
name(b_1, ...b_n).

DIMENSION statements are used to allocate storage space for arrays specified by array declarators. The DIMENSION statement need not be used to define all arrays in a program unit. An array declarator for each array used in a program unit must appear only once in the program unit (either in a DIMENSION, COMMON or Type statement). Using an array declarator in a COMMON or Type statement is equivalent (in terms of memory space allocated for the array) to using the array declarator in a DIMENSION statement.

```
:FORTGO FTRAN12

 PAGE 0001   HP32102B.00.0

         PROGRAM ADJUSTABLE
   C
   C ADJUSTABLE ARRAY DECLARATORS AND
   C CHARACTER LENGTH EXAMPLE
   C
    100  FORMAT('0',T8,S//)
    200  FORMAT(T5,10I4)
         DIMENSION IARR(10,10)
         CHARACTER A*10
         I=10
         J=10
         K=10
         CALL SB(IARR,A,I,J,K)
         WRITE(6,100)A
         WRITE(6,200)IARR
         STOP
         END




         SUBROUTINE SB(IVAR,Z,L,M,N)
         DIMENSION IVAR(L,M)
         CHARACTER Z*(N)
         DO 10 NR=1,L
         DO 10 NC=1,M
    10   IVAR(NR,NC)=NR*NC
         Z="THE START"
         RETURN
         END


 ****      GLOBAL STATISTICS       ****
 ****   NO ERRORS,   NO WARNINGS   ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:00:05

  END OF COMPILE

  END OF PREPARE


       THE START


       1   2   3   4   5   6   7   8   9  10
       2   4   6   8  10  12  14  16  18  20
       3   6   9  12  15  18  21  24  27  30
       4   8  12  16  20  24  28  32  36  40
       5  10  15  20  25  30  35  40  45  50
       6  12  18  24  30  36  42  48  54  60
       7  14  21  28  35  42  49  56  63  70
       8  16  24  32  40  48  56  64  72  80
       9  18  27  36  45  54  63  72  81  90
      10  20  30  40  50  60  70  80  90 100
  END OF PROGRAM
```

Figure 5-1. Adjustable Array Declarators and Character Length Example

For example,

INTEGER ARR

Type statement specifying ARR as type integer. The *name* of the array, not the *complete* array declarator, appears.

DIMENSION ARR(4,4)

DIMENSION statement using the array declarator (ARR(4,4)) which causes 16 words of memory to be allocated for the array element values. The two previous statements can be replaced by one statement if, instead of the array *name*, the complete array declarator is used in the Type statement.

For example,

> INTEGER ARR(4,4) specifies array ARR as type integer, and because of the complete array declarator, also allocates 16 words in memory for the array. This one statement has the same effect as the two previous statements.

or

INTEGER*4 ARR

Type statement specifying ARR as type double integer. The name of the array, not the complete array declarator, appears.

DIMENSION ARR(5,5)

DIMENSION statement using the array declarator (ARR(5,5)) which causes 50 words of memory to be allocated for the array element values. The two previous statements can be replaced by one statement if, instead of the array name, the complete array declarator is used in the Type statement.

For example,

> INTEGER*4 ARR(5,5) specifies array ARR as type double integer, and because of the complete array declarator, also allocates 50 words in memory for the array. This one statement has the same effect as the two previous statements.

5-5. EQUIVALENCE STATEMENT

The EQUIVALENCE statement associates simple variables and array elements so that they may share all or part of allocated storage space in the same program unit.

The form of an EQUIVALENCE statement is

EQUIVALENCE *(list),(list), . . . , (list)*

For example,

EQUIVALENCE (A,B,C),(ABLE,BETA)

list

where

list
is two or more simple variables, or subscripted or unsubscripted array names, all separated by commas. All items in a *list* share the same storage space. (In the example, A, B, and C share the same storage space and ABLE and BETA share the same storage space.) An array name (without the subscript) starts the equivalencing at the first element of the array (element 1 for a one-dimensional array, 1,1 for a two-dimensional array, and so on).

Function names, subroutine names, dummy variables and dummy array elements cannot appear in an EQUIVALENCE statement. A data element occurring in a DATA statement cannot be put into a common block through an EQUIVALENCE statement (with the exception of block data subprograms, see paragraph 5-17). In addition, none of the following items can occur in an EQUIVALENCE statement:

- An array with an adjustable declarator.
- A character array of adjustable length.
- A character variable of adjustable length.

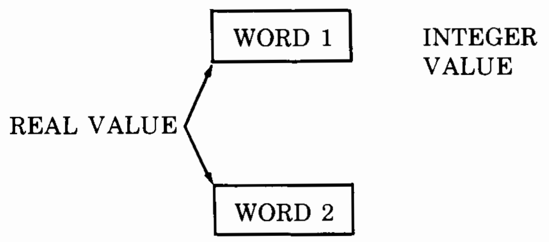
The EQUIVALENCE statement can be used to conserve memory space. For example, an integer array can be dimensioned at the beginning of a program and various other arrays, being manipulated at different times in the same program, can be equivalenced to this array so that the same memory space is used. The types of arrays being equivalenced need not be the same; an array of type integer can be equivalenced to an array of type real (see paragraph 5-6).

Another application for the EQUIVALENCE statement (although not considered to be good programming practice) is in long programs, perhaps written by several programmers. The different programmers may have inadvertently changed variable names such that, for example, A, B, C, and D all mean the same thing. To overcome this problem, it is sometimes easier to write an EQUIVALENCE statement (EQUIVALENCE(A,B,C,D)) than to search the program and change all the variable names.

5-6. EQUIVALENCE OF DIFFERENT TYPES

Equivalence between data elements of different types is allowable in FORTRAN/3000, but care should be taken when attempting to match data types which store data in different size storage space. For example, if an integer and a real value are equivalenced, the integer value will share the same space as the most significant word of the two-word real value.

For example,

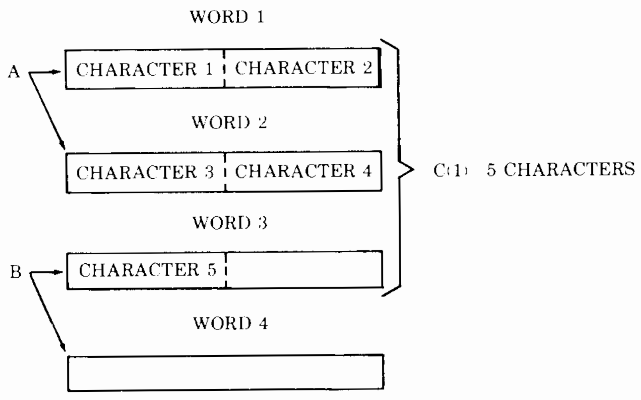


Equivalencing character variables with other variable types requires special care. All data values other than character are stored in multiples of whole 16-bit computer words. Character values are stored in multiples of 8 bits (two 8-bit characters (bytes) per word). Character values may be equivalenced with other data types only if the resulting group can be allocated so that all noncharacter data elements begin on a whole-word boundary.

For example,

ILLEGAL

CHARACTER *5 C(3)
EQUIVALENCE (A,C(1)),(B,C(2))
requires A and B to be allocated 5 bytes apart, which is not allowed.



B CANNOT START ON HALF-WORD BOUNDARY

5-7. EQUIVALENCE OF ARRAY ELEMENTS

Array elements can be equivalenced to elements of a different array or to simple variables.

For example,

```
DIMENSION A(3),C(5)
EQUIVALENCE (A(2),C(4))
```

In the preceding example array element A(2) shares the same storage space as array element C(4). This implies that:

- A(1) shares storage space with C(3), and A(3) shares storage space with C(5).
- No equivalence occurs outside the bounds of any of the arrays. For example, C(1) and C(2) do not share storage space with any elements of A. They are outside the bounds (3) of A.

The following two statements indicate that arrays A and C are type integer and that each array has four elements and one dimension. C(1) and C(2) have unique storage areas and A(3) and A(4) also have unique storage areas. A(1) shares space with C(3), and A(2) shares storage space with C(4):

```
INTEGER A(4),C(4)
EQUIVALENCE (A(2),C(4))
```

ARRAY A	STORAGE SPACE WORD NUMBER	ARRAY C
	1	C(1)
	2	C(2)
A(1)	3	C(3)
A(2) --------	4 ---------	C(4)
A(3)	5	
A(4)	6	

In case of double integer, the following two statements indicate that arrays B and D are type INTEGER*4, and that each array has 5 elements and one dimension. D(1) and D(2) have unique storage areas and B(4) B(5) also have unique storage areas. B(1) shares space with D(3), B(2) shares space with D(4), and B(3) shares space with D(5).

```
INTEGER*4 B(5),D(5)
EQUIVALENCE (B(3),D(5))
```

ARRAY B	STORAGE SPACE WORD NUMBER	ARRAY D
	1,2	D(1)
	3,4	D(2)
B(1)	5,6	D(3)
B(2)	7,8	D(4)
B(3)	9,10	D(5)
B(4)	11,12	
B(5)	13,14	

Array elements are equivalenced on the basis of storage elements. If the arrays are not of the same type, they may not line up element by element.

Example 1,

```
INTEGER A(4)
REAL IBAR(2)
EQUIVALENCE (A(1),IBAR(1))
```

A(1) and A(2) share the two computer words with the real array element IBAR(1). A(3) and A(4) share the two computer words used to store the value of IBAR(2), as follows:

ARRAY A	STORAGE SPACE WORD NUMBER	ARRAY IBAR
A(1)	1	IBAR(1)
A(2)	2	
A(3)	3	IBAR(2)
A(4)	4	

Example 2,

```
INTEGER*4 B(2)
REAL JSLASH (2)
EQUIVALENCE ( B(2),JSLASH(2) )
```

B(1) shares two computer words with JSLASH(1), and B(2) shares two computer words with JSLASH(2).

ARRAY B	STORAGE SPACE WORD NUMBER	ARRAY JSLASH
B(1)	1 2	JSLASH(1)
B(2)	3 4	JSLASH(2)

5-8. EQUIVALENCE BETWEEN ARRAYS OF DIFFERENT DIMENSIONS

To determine equivalence between arrays with different dimensions, FORTRAN/3000 provides an array successor function which views all elements of an array in linear sequence. This means that all arrays, regardless of their dimension, are stored in memory as one-dimensional arrays.

For example,

Array elements are stored in memory in ascending sequential order. Each (non-character) element occupies 1, 2, or 4 words, depending on the array type.

```
Integer array I(5)
  I(1) I(2) I(3) I(4) I(5)

Integer array J(2,3)
  J(1,1) J(2,1) J(1,2) J(2,2)
    J(1,3) J(2,3)

Integer array K(2,2,3)
  K(1,1,1) K(2,1,1) K(1,2,1)
    K(2,2,1) K(1,1,2) K(2,1,2)
  K(1,2,2) K(2,2,2) K(1,1,3)
    K(2,1,3) K(1,2,3) K(2,2,3)
```

General Rule: The first index counts fastest,then the second index, then the third, etc.

Integer arrays I, J and K show the storage space used for one-dimensional, two-dimensional and three-dimensional integer arrays. The principle is the same for double integer, real, and double precision arrays except that double integer and real occupy two words per element, and double precision arrays occupy four words per element.

A character array also is stored in the same manner; however, this type of array requires only one byte (8 bits) for each *character* per *element.* For example, CHARACTER*3 A(5) would require 15 bytes; there are 5 elements of 3 characters each.

EQUIVALENCE statements must avoid contradictory declarations.

For example,

ILLEGAL

```
INTEGER A(5,5),B(90),C(2,10)
EQUIVALENCE (A,B),(A(5,5),C),
   (B(25),C(1,10))
```

is not allowed because A and B, being equivalenced, cannot share the *same* space with C.

Similarly, while unnamed COMMON may be extended by an EQUIVALENCE statement, it may not be re-ordered.

For example,

ILLEGAL

```
COMMON A,B,C
EQUIVALENCE (A,C)
```

is not allowed.

The following statements equivalence array elements A(2,2,2) and I(3). A is a three-dimensional array and I is one-dimensional.

For example,

```
INTEGER A(3,3,3),I(10)
EQUIVALENCE (A(2,2,2),I(3))
```

ARRAY A	STORAGE WORD RELATIVE NUMBER	ARRAY I
A(1,1,1)	1	
A(2,1,1)	2	
A(3,1,1)	3	
A(1,2,1)	4	
A(2,2,1)	5	
A(3,2,1)	6	
A(1,3,1)	7	
A(2,3,1)	8	
A(3,3,1)	9	
A(1,1,2)	10	
A(2,1,2)	11	
A(3,1,2)	12	I(1)
A(1,2,2)	13	I(2)
A(2,2,2)	14	I(3)
A(3,2,2)	15	I(4)
A(1,3,2)	16	I(5)
A(2,3,2)	17	I(6)
A(3,3,2)	18	I(7)
A(1,1,3)	19	I(8)
A(2,1,3)	20	I(9)
A(3,1,3)	21	I(10)
A(1,2,3)	22	
A(2,2,3)	23	
A(3,2,3)	24	
A(1,3,3)	25	
A(2,3,3)	26	
A(3,3,3)	27	

5-9. COMMON STATEMENT

The COMMON statement reserves a block of storage space that can be referenced (and used) by several different program units, thus allowing variables to be used by different program units without being passed as explicit parameters. For example, two variables in different subprogram units, or in a main program unit and a subprogram unit, can use the same storage space and the value assigned to one variable automatically becomes the value of the other.

The form of a COMMON statement is

COMMON *list*

For example,

COMMON $\underbrace{\text{A,B,C}}_{\textit{list}}$

where

list
is one or more simple variables, array names, or array declarators. Using an array name only (instead of the complete array declarator) in the *list* implies that the array declarator appears in a Type or DIMENSION statement elsewhere in the same program unit.

Figure 5-2 is an example of COMMON statement usage.

In the example, the variable A in the main program shares storage space with X in the subroutine subprogram. When a value for A is determined by the ACCEPT statement, X automatically shares this value. Similarly, B and Y also share storage space, as does the variable SIDE both in the main program and the subprogram.

The subroutine uses the values read in for A and B (which are shared by X and Y) to compute the length of the hypotenuse of a right-angled triangle. As can be seen from the example, the variable names in the two COMMON statements need not be the same.

Thus, the COMMON statement allows one program unit to store data in a memory area which can be read, manipulated and stored by the program unit specifying the COMMON statement and by other program units.

The maximum number of pointers allowed for referencing the variables (simple variables and/or arrays) in COMMON is 254. The segmenter uses one or two words in the DB area. This limits the number of COMMON variables allowed to 254, even though the Primary DB ranges from DB + 0 to DB + 255. However, if simple variables precede the array variable in COMMON, it is possible to have more than 254 COMMON variables (see Appendix F for details). Alternatively, use the $CONTROL MORECOM compiler command (see paragraph 9-20A).

5-10. LABELED COMMON BLOCKS

Common blocks can be labeled by changing the form of the COMMON statement as follows:

COMMON /blockname/list/blockname/list

For example,

COMMON /$\overset{\textit{blockname}}{\text{SAM}}$/$\underbrace{\text{A,B,C}}_{\textit{list}}$/$\overset{\textit{blockname}}{\text{ABLE}}$/$\underbrace{\text{X,Y,Z}}_{\textit{list}}$

```
:FORTGO FTRAN13

PAGE ØØØ1    HP32102B.ØØ.Ø

ØØØØ1ØØØ          PROGRAM COMMON
ØØØØ2ØØØ    C
ØØØØ3ØØØ    C COMMON STATEMENT EXAMPLE
ØØØØ4ØØØ    C
ØØØØ5ØØØ          COMMON A,B,SIDE
ØØØØ6ØØØ          ACCEPT A,B
ØØØØ7ØØØ          CALL TRIANGLE
ØØØØ8ØØØ          DISPLAY "THE THIRD SIDE IS ",SIDE
ØØØØ9ØØØ          STOP
ØØØ1ØØØØ          END



ØØØ11ØØØ          SUBROUTINE TRIANGLE
ØØØ12ØØØ          COMMON X,Y,SIDE
ØØØ13ØØØ          SIDE=SQRT((X**2)+(Y**2))
ØØØ14ØØØ          RETURN
ØØØ15ØØØ          END


****       GLOBAL STATISTICS        ****
****    NO ERRORS,   NO WARNINGS    ****
TOTAL COMPILATION TIME  Ø:ØØ:Ø1
TOTAL ELAPSED TIME      Ø:ØØ:Ø4

 END OF COMPILE

 END OF PREPARE


?3Ø,4Ø

THE THIRD SIDE IS   5Ø.ØØØØ
 END OF PROGRAM
```

Figure 5-2. COMMON Statement Example

where

blockname
is used to identify different common blocks and is an alphanumeric name of from 1 to 15 characters (the first character must be a letter). In the example, A, B and C share block SAM and X, Y and Z share block ABLE. Different program units reference the same common block by using the same *blockname* in their COMMON statements.

A null block can be signified by two slashes with no intervening non-blank characters (//). If a null block is specified, the items in *list* that follow are assigned to the unlabeled or general common area. There may be only one unlabeled (null) common.

list
is one or more simple variables, array names or array declarators. Using an array name in a COMMON statement implies that the array declarator appears in a Type or DIMENSION statement elsewhere in the same program unit.

The double slash, indicating a null or unlabeled common block, may be omitted if the user desires that the first common block go unnamed.

For example,

COMMON A,B,C/ABLE/X,Y/BETA/S,T,U

The first common block containing variables A, B and C is unlabeled. If any block in a COMMON statement other than the first is to go unlabeled, however, the double slash must be inserted.

For example,

COMMON /ABLE/A,B,C//X,Y/BETA/S,T,U

The common block containing variables X and Y is unlabeled. The length of a common block is determined by the number and type of the items in *list* associated with that block. The *list* items are stored contiguously within their block according to their listed order within the COMMON statement.

For example,

INTEGER*4 A(3)
Type statement indicating a double integer array, A, of 3 elements. Area reserved for A= 6 words. (Double integer data uses two words per value.)

DIMENSION ARR(3)
DIMENSION statement indicating a real array, ARR, consisting of three elements. Area reserved for ARR= 6 words. (Real data requires two words per value.)

COMMON /BLOCKA/B, ARR,A
COMMON statement. Total common area reserved = 16 words.

If the arrays have their types implied (by the first letter of the name), a single COMMON statement is sufficient.

For example,

COMMON /BLOCKA/I(4),ARR(3)
Array I is implied type integer because the name begins with the letter I. ARR is implied type real. Total space reserved = 10 words.

Common block storage is allocated at the time the program is loaded into core for execution and is not local to any one program unit. No dummy variable name, function, subroutine name, or array with an adjustable array declarator or adjustable length character variable may be used in a COMMON statement, nor may any of these elements be put in a common block with an EQUIVALENCE statement. (See EQUIVALENCE, paragraph 5-5.) No name used in a DATA statement (see paragraph 5-15) may be used in a COMMON statement nor may it be put in a common block through equivalence, with the exception of block data subprograms (see paragraph 5-17). Data space within the common area for the arrays I and ARR shown in the preceding example is allocated as follows:

WORD	**COMMON BLOCK**
1	I(1)
2	I(2)
3	I(3)
4	I(4)
5 6	AAR(1)
7 8	AAR(2)
9 10	ARR(3)

Each program unit that uses the common block must include a COMMON statement which contains the *blockname* (if a name was defined). The *list* assigned to the common block by the program unit need not correspond by name, type, or number of elements with those of any other program unit. The size of a particular common block referenced in a program unit need not be the same as the size of the same common block declared in any other program unit. The largest declared size determines the size of the common block. For example, if one program unit declares COMMON A(5) and another, COMMON A(5000), the unlabeled common block is 10000 words long when A is real. If the common block is labeled, a warning message appears during program preparation when the declared size of the block is different in different program units.

Example 1,

In program unit 1:

$$\text{COMMON /BLOCKA/}\overbrace{\text{I(4),J(6)}}^{\text{Integer}},\overbrace{\text{ALPHA,SAM}}^{\text{Real}}$$

In program unit 2:

$$\text{COMMON /BLOCKA/}\overset{\text{Real}}{\text{GEO}},\overbrace{\text{L(10),INDIA,JACK}}^{\text{Integer}}$$

BLOCKA is the same size (14 words) in both program units. Thus, referencing I(4) in program unit 1 is equivalent to referencing L(2) in the program unit 2 since both variables pertain to the same word of the labeled common block.

PROGRAM 1 REFERENCE	**COMMON BLOCK WORD NUMBER**	**PROGRAM 2 REFERENCE**
I(1)	1	GEO
I(2)	2	
I(3)	3	L(1)
I(4)	4	L(2)
J(1)	5	L(3)
J(2)	6	L(4)
J(3)	7	L(5)
J(4)	8	L(6)
J(5)	9	L(7)
J(6)	10	L(8)
ALPHA	11	L(9)
	12	L(10)
SAM	13	INDIA
	14	JACK

Example 2,

In program unit 1:

REAL MULT
INTEGER*4 I,J

$$\text{COMMON/BUFFER/}\overbrace{\text{I(2) , J(3)}}^{\text{Double Integer}}\text{ , }\overbrace{\text{SUM, MULT}}^{\text{Real}}$$

In program unit 2:

INTEGER*4 J,K

$$\text{COMMON/BUFFER/}\overset{\text{Real}}{\text{DIV}}\text{, }\overbrace{\text{J(5) , K}}^{\text{Double Integer}}$$

BUFFER is the same size (14 words) in both program units. Thus, referencing I(2) in program unit 1 is equivalent to referencing J(1) in program unit 2 since both variables pertain to the same word of the labeled common block.

PROGRAM 1 REFERENCE	**COMMON BLOCK WORD NUMBER**	**PROGRAM 2 REFERENCE**
I(1)	1 2	DIV
I(2)	3 4	J(1)
J(1)	5 6	J(2)
J(2)	7 8	J(3)
J(3)	9 10	J(4)
SUM	11 12	J(5)
MULT	13 14	K

In the following example, the unlabeled common block is a different length in program unit 2. The last five words of the common block are not used in program unit 2.

PROGRAM 1 REFERENCE	**COMMON BLOCK WORD NUMBER**	**PROGRAM 2 REFERENCE**
I(1)	1	JAR(1)
I(2)	2	JAR(2)
I(3)	3	JAR(3)
I(4)	4	JAR(4)
I(5)	5	JAR(5)
I(6)	6	JAR(6)
I(7)	7	JAR(7)
I(8)	8	Unused
I(9)	9	Unused
I(10)	10	Unused
I(11)	11	Unused
I(12)	12	Unused

In the following example, the labeled block sizes referenced by program units 1 and 2 are different. The last four words referenced by program unit 1 are unused.

In program unit 1:

```
                  Integer
COMMON /BLOCKA/I(5),J(3)
```

In program unit 2:

```
                Real   Integer
COMMON /BLOCKA/ARR(4),K(4)
```

PROGRAM 1 REFERENCE	COMMON BLOCK WORD NUMBER	PROGRAM 2 REFERENCE
I(1)	1	ARR(1)
I(2)	2	ARR(1)
I(3)	3	ARR(2)
I(4)	4	ARR(2)
I(5)	5	ARR(3)
J(1)	6	ARR(3)
J(2)	7	ARR(4)
J(3)	8	ARR(4)
Unused	9	K(1)
Unused	10	K(2)
Unused	11	K(3)
Unused	12	K(4)

Common block elements are each individually pointed to by a pointer established by the segmenter. These pointers normally are allocated one per distinct element name, with a limit of 254 pointers. If a program has so much common that this limit is exceeded, some optimization can be achieved by placing simple variables before arrays of the same type. In this case, the simple variable is treated as the zero'th element of the array.

For example,

```
REAL A,ARR(10)
COMMON ARR,A
```

will result in 2 pointers, while

```
REAL A,ARR(10)
COMMON A,ARR
```

will result in only 1 pointer being allocated.

5-11. CHARACTER VARIABLES AND ARRAYS IN COMMON BLOCKS

Each type of data element in common starts at the next word boundary following the preceding data element, except for character values, which start on the next byte.

For example,

```
INTEGER A(3),BA(3)
CHARACTER*3 CH(3)
COMMON A,CH,BA
```

PROGRAM REFERENCE	COMMON BLOCK WORD NUMBER	
A(1)	1	
A(2)	2	
A(3)	3	
CH(1)	4	1 word
CH(1)	4	
CH(1)	5	1 word
CH(2)	5	
CH(2)	6	1 word
CH(2)	6	
CH(3)	7	1 word
CH(3)	7	
CH(3)	8	1 word
Unused	8	
BA(1)	9	
BA(2)	10	
BA(3)	11	

Thus, the least significant half of word 8 is unused since the integer array AB starts on the first word boundary after the character array CH.

In the following example, the double integer array BA occupies six words in the common block. The character array CH takes four and one-half words in the common block, and the double integer array DA takes four words in the common block.

For example,

```
INTEGER*4 BA(3) ,DA(2)
CHARACTER*3 CH(3)
COMMON BA,CH,DA
```

PROGRAM REFERENCE	COMMON BLOCK WORD NUMBER
BA(1)	1
	2
BA(2)	3
	4
BA(3)	5
	6
CH(1)	7
CH(1)	7
CH(1)	8
CH(2)	8
CH(2)	9
CH(2)	9
CH(3)	10
CH(3)	10
CH(3)	11
Unused	11
DA(1)	12
	13
DA(2)	14
	15

In the following example, the character array CH takes four and one-half words in the common block. Character array BN starts in the least significant half of the fifth word (starts on a byte boundary).

For example,

```
CHARACTER*3 CH(3),BN*1(3)
COMMON CH,BN
```

PROGRAM REFERENCE	COMMON BLOCK WORD NUMBER	
CH(1)	1	1 word
CH(1)	1	
CH(1)	2	1 word
CH(2)	2	
CH(2)	3	1 word
CH(2)	3	
CH(3)	4	1 word
CH(3)	4	
CH(3)	5	1 word
BN(1)	5	
BN(2)	6	1 word
BN(3)	6	

5-12. EQUIVALENCE IN COMMON BLOCKS

Data elements may be put into a common block by specifying them as equivalent to data elements mentioned in a COMMON statement. If one element of an array is equivalenced to a data element within a common block, the whole array is placed in the common block with equivalence maintained for storage units preceding and following the data element in common. The common block is always extended, if it is necessary to fit an equivalenced array into the common block, but no array can be equivalenced into a common block if storage elements would have to be prefixed to the common block to contain the entire array. Equivalences cannot insert storage into the middle of the common block or rearrange storage within the block. Since the elements in a common block are stored contiguously according to the order in which they are mentioned in the COMMON statement, two elements in common cannot be equivalenced. In the following example, array I is in a common block. Array element J(2) is equivalent to I(3).

```
DIMENSION I(6), J(6)
COMMON I
EQUIVALENCE (I(3),J(2))
```

The common block is extended to accommodate array J as follows:

ARRAY I	COMMON BLOCK WORD NUMBER	ARRAY J
I(1)	1	Not defined
I(2)	2	J(1)
I(3)	3	J(2)
I(4)	4	J(3)
I(5)	5	J(4)
I(6)	6	J(5)
Not defined	7	J(6)

The equivalence set up by the following example is not allowed. In order to set array I into the common block, an extra word must be inserted in *front* of the common block.

For example,

ILLEGAL

```
DIMENSION I(6),J(6)
COMMON I
EQUIVALENCE (I(1),J(2))
```

ARRAY I	COMMON BLOCK WORD NUMBER	ARRAY J
		J(1)
I(1)	1	J(2)
I(2)	2	J(3)
I(3)	3	J(4)
I(4)	4	J(5)
I(5)	5	J(6)
I(6)	6	

Element J(1) would be stored in front of the common block; thus, EQUIVALENCE (I(1),J(2)) is not allowed.

5-13. IMPLICIT STATEMENT

The IMPLICIT statement overrides or confirms the type associated with the first letter of a symbolic name (variable). If a symbolic name is not mentioned in a Type statement, the type of the data element is determined by the first letter of the symbolic name. As mentioned earlier, names starting with the letters I, J, K, L, M or N are type integer; names starting with any other letter are type real. The IMPLICIT statement can be used to override this convention (or, although it is not necessary, the IMPLICIT statement can be used to confirm the first letter convention).

The form of an IMPLICIT statement is

IMPLICIT *type(letter, . . . , letter), . . . , type(letter, . . . , letter)*

For example,

```
IMPLICIT INTEGER  (A,B,C,F),REAL(I,J,M)
             \_______type_______/
```

where

type
can be INTEGER, INTEGER*2, INTEGER*4, REAL, DOUBLE PRECISION, LOGICAL, COMPLEX or CHARACTER (optionally followed by **x*, the length attribute of a character value).

letter
is a letter of the alphabet which assumes the type specified by the heading preceding it in the IMPLICIT statement; *letter* can be a single letter or a range of letters — for example, A-C means A, B, C.

The IMPLICIT statement itself can be overridden for specific symbolic names when these names are used in a Type statement. For example, IMPLICIT INTEGER (A) specifies that symbolic names starting with the letter A are type integer. A Type statement such as REAL ABLE, however, indicates that the variable ABLE is type real, overriding the IMPLICIT statement in this case.

5-14. EXTERNAL STATEMENT

EXTERNAL statements identify function subprogram and subroutine subprogram names which are used as arguments in a CALL or function reference statement in one program unit but are defined in another (external) program unit.

The form of an EXTERNAL statement is

EXTERNAL *name,name, . . . , name*

For example,

EXTERNAL ALPHA, BETA, SOLVE

where

name
is the symbolic name of a function or subroutine subprogram. The EXTERNAL statement must be used within a calling program unit to identify the names of function or subroutine subprograms which are to be used as arguments.

For example, a CALL statement of the form,

CALL SIDE(X,Y)

passes X and Y as actual arguments to the subroutine SIDE to be used in its computations.

In an *implicit* call (such as to an intrinsic function), a CALL statement does not need to be used. For example, in

A = SQRT(B)

the intrinsic function SQRT is called *implicitly* merely by being referenced in the expression SQRT(B). The actual argument B is passed to SQRT to be used in its computations.

The EXTERNAL statement provides a means of using the names of function and subroutine subprograms as actual arguments in a CALL statement. The EXTERNAL statement is necessary to inform the compiler that these names are function or subroutine subprograms and not variable or array names.

Figure 5-3 is an example of EXTERNAL statement usage.

In the example, the main program declares HYPOT in an EXTERNAL statement, which defines it as a function name.

When the CALL statement is executed, the function name hypot is passed as an actual argument (along with ADDSIDES, A and B) to the function subprogram CIRCUM to be used in its computations.

First, CIRCUM dummy arguments U and V assume the values of A and B (which were passed to CIRCUM as actual arguments by the main program).

Next, U and V are passed as actual arguments to the function subprogram HYPOT, where their values are assumed by dummy arguments X and Y to compute a value for HYPOT.

```
:FORTGO FTRAN14

PAGE 0001    HP32102B.00.0

               PROGRAM EXTERNAL
         C
         C EXTERNAL STATEMENT EXAMPLE
         C
               EXTERNAL HYPOT
               ACCEPT A,B
          30   CALL CIRCUM(ADDSIDES,HYPOT,A,B)
               DISPLAY "THE CIRCUMFERENCE IS ",ADDSIDES
               STOP
               END



               FUNCTION HYPOT(X,Y)
               HYPOT=SQRT((X**2)+(Y**2))
               RETURN
               END



               SUBROUTINE CIRCUM(S,T,U,V)
               S=T(U,V)+U+V
               RETURN
               END


 ****      GLOBAL STATISTICS       ****
 ****    NO ERRORS,    NO WARNINGS  ****
 TOTAL COMPILATION TIME  0:00:01
 TOTAL ELAPSED TIME      0:00:11

 END OF COMPILE

 END OF PREPARE


?30,40

THE CIRCUMFERENCE IS   120.000
 END OF PROGRAM
```

Figure 5-3. EXTERNAL Statement Example

Finally, statement 10 (S = T(U,V) + U + V) in CIRCUM computes a value for S using the actual values for HYPOT, A and B in place of the dummy values T, U, and V. The value of S is passed back to the main program as ADDSIDES.

A function or subroutine subprogram name can be used as an actual argument without being declared in an EXTERNAL statement by inserting empty parentheses after the name. For example, statement 30 in the previous sample program could be written as follows:

```
30 CALL CIRCUM(ADDSIDES,HYPOT(),A,B)
```

The blank parentheses following HYPOT identify it as an external subprogram.

Function subprograms are always called *implicitly* by being referenced in a statement (the same as intrinsic functions).

For example,

```
100 FORMAT(2F10.4)
200 FORMAT(F10.4)
    READ (5,100)A,B
 10 S = HYPOT(A,B)
    WRITE (6,200)S
    STOP
    END
    FUNCTION HYPOT(X,Y)
    HYPOT = SQRT ((X**2) + (Y**2))
    RETURN
    END
```

Statement 10 (S = HYPOT(A,B)) calls the function subprogram HYPOT implicitly by referencing it in the expression HYPOT(A,B). A and B are actual arguments passed to HYPOT to be used in its computations.

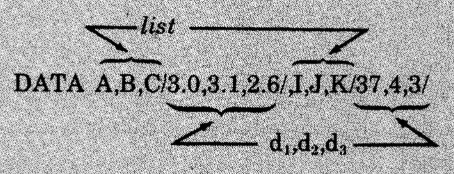
5-15. DATA STATEMENT

A DATA statement is used to assign initial values to data elements. All such data values are set at load time and are never re-initialized. (Variables declared with DATA statements in subroutines have their values preserved from one call to another.)

The form of a DATA statement is

DATA *list*/$d_1,d_2, \ldots , d_n$/
,*list*/$d_1,d_2, \ldots , d_n$/

For example,



where

list
is one or more simple variables, array names or array elements.

$d_1,d_2, \ldots , d_n$
are the actual constants (including Hollerith constants), which are to be assigned to the corresponding items in *list*.

The values declared in a DATA statement are compiled into the object program and become the values assumed by the variables when program execution begins.

For example,

```
DATA A,B,C,D,E/3.0,3.1,2.6,1.09,.2643/
```

assigns the values 3.0, 3.1, 2.6, 1.09 and .2643 to variables A, B, C, D and E, respectively.

If the same value is to be assigned to several variables, the data statement can take the following form:

```
DATA A,B,C,E,X/5*3.6/
```

The previous statement assigns the value 3.6 to all the variables in the list.

Mentioning an array name is the same as mentioning all the elements of the array.

For example,

```
DIMENSION I(3)
DATA I/3*5/
```

assigns I(1) = 5, I(2) = 5 and I(3) = 5. Thus, the statement,

```
DATA I/3*5/
```

is the same as

```
DATA I(1)/5/, I(2)/5/,I(3)/5/
```

or

```
DATA I(1),I(2),I(3)/5,5,5/
```

A DATA statement initializes the value of a particular storage location only once. Incorrect initial values result if the same storage location is reinitialized.

With two exceptions, the constants in a DATA statement must be the same type as the variables to which they are assigned. One exception is that a real variable can be initialized to a whole number by an integer constant (e.g., "20" can be used instead of "20."). The variable is still treated as real. Otherwise, integer variables must be initialized by integer constants, real variables by real constants, and so on.

The second exception is that a string constant can be used to initialize a variable of any type. For a character variable, the initial value represented by the string character is the character itself. For any other type variable, the 8-bit ASCII patterns of the constant are stored left-justified in the storage space reserved for the variable. If the constant does not fill the entire storage space, the remaining part of the storage word is padded with the 8-bit ASCII code for blanks. Because one string character requires only 8 bits of storage, two characters can be stored in one 16-bit computer word.

For example,

```
DATA I,J,K, L/2HSA, "SA", 'SA', 2HSA/
```

2HSA tells the compiler that two Hollerith characters (SA) follow. I, J, K and L are integer variables. The ASCII 8-bit patterns for S and A are loaded into each 16-bit word for I, J, K and L.

It is not necessary to set initial values for all of the variables listed in a DATA statement.

For example,

```
REAL IVAR(20)
DATA IVAR/10*8.0/
```

The statement REAL IVAR(20) sets up 20 elements of storage for array IVAR. The DATA statement, however, only initializes the first 10 elements to 8.0. The other 10 elements of IVAR are not given an initial value.

5-16. EQUIVALENCE IN DATA STATEMENTS

Variables and array elements can share storage space in memory with other variables and array elements through the use of EQUIVALENCE statements. If an array element is equivalenced to another element in a DATA statement, the *entire* array is allocated to the storage space. The storage space is extended either at the beginning or the end to accommodate data elements set into the space through the EQUIVALENCE statement.

Example 1,

```
INTEGER A(5),B(7)
DATA A/5*0/
EQUIVALENCE (A(1),B(2))
```

Since the element B(2) is mentioned in the EQUIVALENCE statement, the entire array will be allocated space in memory with array A.

Note, however, that the statements INTEGER A(5),B(7) and DATA A allocated only five words of storage in the storage area for array A. To accommodate B(1), the storage area is prefixed with one storage word, and extended one word to accommodate B(7).

ARRAY A	STORAGE SPACE WORD NUMBER	ARRAY B
Unused	1	B(1)
A(1)	2	B(2)
A(2)	3	B(3)
A(3)	4	B(4)
A(4)	5	B(5)
A(5)	6	B(6)
Unused	7	B(7)

Example 2,

```
INTEGER*4 C(4),D(6)
DATA C/4*1/
EQUIVALENCE ( C(1),D(3) )
```

Since the element D(3) is mentioned in the EQUIVALENCE statement, the entire array will be allocated space in memory with array C.

Note, however, that the statements INTEGER C(4),D(6) and DATA C allocated only four words of storage in the storage area for array A. To accommodate D(1), D(2), the storage area is prefixed with two storage words.

ARRAY C	STORAGE SPACE WORD NUMBER	ARRAY D
Unused	1 2	D(1)
Unused	3 4	D(2)
C(1)	5 6	D(3)
C(2)	7 8	D(4)
C(3)	9 10	D(5)
C(4)	11 12	D(6)

Equivalence can rearrange the order of storage allocation in a storage area as long as all arrays remain contiguous within themselves.

For example,

```
DIMENSION I(5),J(5),K(10)
DATA I,J/10*5/
EQUIVALENCE (K(1),J(1)),(K(6),I(1))
```

The previous statements produce the result:

ARRAY K	STORAGE SPACE WORD NUMBER	ARRAYS I AND J
K(1)	1	J(1)
K(2)	2	J(2)
K(3)	3	J(3)
K(4)	4	J(4)
K(5)	5	J(5)
K(6)	6	I(1)
K(7)	7	I(2)
K(8)	8	I(3)
K(9)	9	I(4)
K(10)	10	I(5)

Normally, a data element occurring in a DATA statement cannot be put into a common block through the use of an EQUIVALENCE statement (see Block Data Subprograms, paragraph 5-18 for exceptions). No dummy arguments, or arrays with adjustable declarators, can belong to a data block. EQUIVALENCE statements cannot be used if the purpose is to store two elements of the same array into the same space in memory, or if they destroy the contiguity of the array elements. Noncharacter values will be stored starting on a full-word boundary in memory.

5-17. BLOCK DATA SUBPROGRAMS

Block data subprograms are used for the sole purpose of supplying initial values to elements contained in common blocks. DATA statements are used in block data subprograms to supply these initial values. Storage space is allocated by COMMON statements,and the initial values are supplied by the DATA statements. (For a further discussion of block data subprograms, see Section XI.)

5-18. STATEMENT FUNCTIONS

In some programs, relatively simple computations are used repeatedly. These computational functions may be used only in one program so there would be no need to set up a new external function. Instead, a function can be defined in the program and then used whenever necessary in that program.

The definition of a statement function must occur before the first executable statement in the program unit and after all other declaration statements except DATA statements.

The form of a statement function is:

name (param,param, . . . ,param) = expression

For example,

$$\overset{\textit{name}}{\text{DISP}}\underbrace{\text{(C,R,H)}}_{\textit{param}} = \underbrace{\text{C * (3.14159 * (R**2) * H)}}_{\textit{expression}}$$

where

name
is a symbolic name starting with a letter.

param
is a simple variable used as a dummy argument. No other symbolic names except simple variable names may be used.

expression
is an arithmetic or logical expression of constants, simple variables, array variables, function subprogram references, intrinsic references, and the appropriate operators for the type expression.

The statement function name may not be used in an EXTERNAL statement. The definition is a single statement similar to an arithmetic or logical assignment statement. (See Section III).

The expression defines the actual computational procedure which derives one value. When referenced, this value is assigned to the function name. The expression must be either a logical expression or an arithmetic expession; no character expressions or character-valued function statements are allowed. Any statement function referenced in the definition of another statement function must be defined before it is used in the definition. Statement function definitions are not recursive, that is, a statement function cannot reference itself.

The value of any dummy arguments in the expression are supplied at the time the statement function is referenced. All other expression elements are local to the program unit containing the reference and derive their values from statements in the containing program unit.

The type of statement function is determined by using the statement function name in a Type statement or by the first letter of the statement function name (names beginning with I, J, K, L, M or N are type integer, while names beginning with any other letter are type real). This convention may be altered by using an IMPLICIT statement.

The type of expression in a statement function definition must be compatible with the defined type of the statement function's symbolic name. For example, logical expressions must be used in logical statement functions and arithmetic expressions in arithmetic statement functions. The arithmetic expression used in an arithmetic statement function need not be the same arithmetic type as the statement function symbolic name. (For example, the expression can be type integer, the statement function name can be defined as type real.) The expression value is converted to the statement function type at the time it is assigned to the statement function's symbolic name. (See Section III for a discussion of type conversion.) Figure 5-4 shows a program using the statement function DISP to compute the displacement of internal combustion engines.

The statement function DISP is defined with the dummy parameters C, R, and H. The engine size is computed by referencing this function and providing the actual parameters A, B, and C (which have just been read). The import tax then is computed using the result (SIZE) of the statement function's computations.

The example shown is, of course, too short to warrant using a statement function inasmuch as it is referenced once only. Please bear in mind, however, that in actual use a statement function may be referenced many times in a much longer program.

```
:FORTGO FTRAN15

PAGE 0001   HP32102B.00.0

          PROGRAM STATEMENT FUNC
C
C STATEMENT FUNCTION EXAMPLE
C
 100      FORMAT('0',T10,"THE SIZE OF THE ENGINE IS: ",F12.5)
 200      FORMAT('0',T10,"THE IMPORT TAX IS: ",M12.2)
          DISP(C,R,H)=C*(3.14159*(R**2)*H)
          DISPLAY "NUMBER OF CYLINDERS?"
          ACCEPT A
          DISPLAY "BORE SIZE?"
          ACCEPT B
          DISPLAY "STROKE?"
          ACCEPT C
          B=B/2
          SIZE=DISP(A,B,C)
          TAX=1.5*SIZE
          WRITE(6,100)SIZE
          WRITE(6,200)TAX
          STOP
          END


****      GLOBAL STATISTICS      ****
****   NO ERRORS,   NO WARNINGS  ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:00:03

 END OF COMPILE

 END OF PREPARE


NUMBER OF CYLINDERS?
?16

BORE SIZE?
?3.216

STROKE?
?2.978


        THE SIZE OF THE ENGINE IS:     387.04932

        THE IMPORT TAX IS:       $580.57
 END OF PROGRAM
```

Figure 5-4. Statement Function Example

WRITING INPUT/OUTPUT STATEMENTS IN FORTRAN/3000

SECTION VI

Input/output (I/O) statements transfer information between data elements in memory and external devices or between data elements in memory and other data elements at other locations in memory.

The FORTRAN/3000 statements which initiate the transfer of data are READ, WRITE, ACCEPT, and DISPLAY.

A READ statement causes data to be transferred from an external file or a buffer in memory to specified data elements in a list.

A WRITE statement transfers information from data elements in a list to an external file or to a buffer in memory.

An ACCEPT statement reads free-field format data from the standard input file ($STDIN, see Section XII).

A DISPLAY statement is used to output free-field data to the standard list file ($STDLIST, see Section XII).

An input/output statement can contain a list of names of simple variables, arrays, or array elements and, for output only, function subprograms. When an input statement is executed, the input values from the external file (or other data buffer in memory) are assigned to the data elements specified in the list. When an output statement is executed, the values assigned to the listed variables are transferred to the external file or to a buffer in memory. An input or output step normally requires two statements: an executable (READ or WRITE) statement and a non-executable declaration (FORMAT) statement.The FORMAT statement provides information on the external characteristics of the items to be transferred. See Section VII for a complete discussion or FORMAT statements.

External files are identified in I/O statements by a FORTRAN unit number. The FORTRAN/3000 file facility then transfers data to or from the file associated with this unit number. See Section VIII for a discussion of the FORTRAN/3000 file facility.

6-1. READ STATEMENT

A READ statement transfers information from an external file to specified data elements in a list, or from a character variable in memory to the element in the list.

The form of a READ statement is

READ (*source, format, END*=*sn*$_1$, *ERR*=*sn*$_2$) *list*

For example,

```
 source ↘  ↙format reference       list ↘
   READ  (5, 100, END = 500, ERR = 600) A, B, C
or
   READ(5,  100)          (END, ERR, and list omitted)
```

where

source

identifies the data source. It may be an integer constant or integer simple variable. If the variable contains a negative number, that number is assumed to be the negative of the MPE file number desired. (MPE file numbers are returned by the FOPEN intrinsic. See Section VIII.)

1. Sequential READ.
   Positive integer constant or integer simple variable indicates the FORTRAN unit number (1 through 99). For example, if the value is 3, then file FTN03 is read.
   Example:

```
     READ (3, 200) A
FORTRAN unit no.    Format reference
```

2. Direct READ.
   Same as above, with @ *record* appended to the unit number. *Record* is a constant or any linear expression whose integer value is taken as the record number to be read. This option is restricted to files on direct access devices such as disc, with fixed-length records.
   Example:

```
     READ (3 @ 18,200) A
FORTRAN unit no. @ record
```

3. Core-to-core READ.
   If the source where the data is to be read is a buffer in memory, then *source* consists of:
   Name, where *name* is a character simple variable or character array element specifying the name of the buffer in memory where the data is located. (See figure 6-7.)
   Example:

```
     READ (CARD, 150) A, B, C, D, E, F
name of buffer
```

format
specifies the conversion format to be used.

1. Integer constant specifying a FORMAT statement label.
2. Name of a character array, or a simple or subscripted character variable containing the format specification.
3. Asterisk (*) selects free-field format.
4. If *format* is omitted, a binary transfer takes place. (See Section VII, paragraph 7-50 for a discussion of binary tansfer.)

END = sn_1
enables the program to remain in control when an end-of-file (EOF) condition is detected on a READ. sn_1 is a statement label number. If an EOF condition is detected, control is transferred to the statement labeled with the number sn_1. If omitted, the program aborts with the appropriate message.

ERR = sn_2
enables the program to remain in control if an irrecoverable file error occurs. Control transfers to the statement label sn_2. If omitted, the program aborts with a message when the READ is attempted. (See paragraph 6-3 and figures 6-8 and 6-9.)

list
is one or more simple variable names, array names, array elements, function subprogram names, or DO-implied lists. (See paragrpah 6-8 for a discussion of DO-implied lists.) If *list* is omitted, the file is moved to the next record without any data transfer. Exception: a formatted READ without a *list* will read characters into Hollerith fields of the FORMAT statement (see figure 6-3).

6-2. READ STATEMENT EXECUTION

Reading always starts at the beginning of a record from an external file or a character variable in memory. Reading stops when the list is satisfied, provided that the format specifications and the record length are in agreement with the list. The list may always be shorter than the record (i.e., it is possible to read part of a record). After the READ, the external file will be positioned at the beginning of the next record. If the list is longer than the format specifications, or longer than the record, the following conditions exist:

1. Unformatted (binary) READ and list longer than record (list exceeds data). For a sequential READ, successive records are read until the list is satisfied. For a direct READ, the program aborts with message: DIRECT BUFFER OVERFLOW.
2. Formatted READ and format specification shorter than list. When the end of the format specification is reached, the external file skips to the next record and reads it using the format specification again. This proceeds until the list is satisfied. If read is from a character variable in memory, the program aborts with message: BUFFER OVERFLOW.
3. Formatted READ and record shorter than list (list exceeds data). The program aborts with message: FORMAT BEYOND RECORD.

When a READ statement is executed, data values are transferred from *source* to the elements specified in *list*. Elements are assigned values left-to-right according to their positions in *list*.

Each record in a file can contain several values. For example, one card can be thought of as one *input record* and a line printed on a line printer can be thought of as one *output record.*

Figure 6-1 shows a card punched with six integer values. Each value on the card is assigned to a field of a specific width (the field width for each value in figure 6-1 is six columns) but does not have to occupy all the columns in the field. A FORMAT statement is used to specify the type(s) of data contained in this one record. The FORMAT statement for the card shown would be:

100 FORMAT (6I6)

The letter I specifies that the data is type integer, the suffix 6 specifies that the field width is six characters, and the prefix 6 refers to the number of integer values contained in the record. The prefix 6 is used as a shorthand method of writing the specifications; the statement could be written as follows:

100 FORMAT (I6, I6, I6, I6, I6, I6)

For a complete discussion of FORMAT statements, see Section VII.

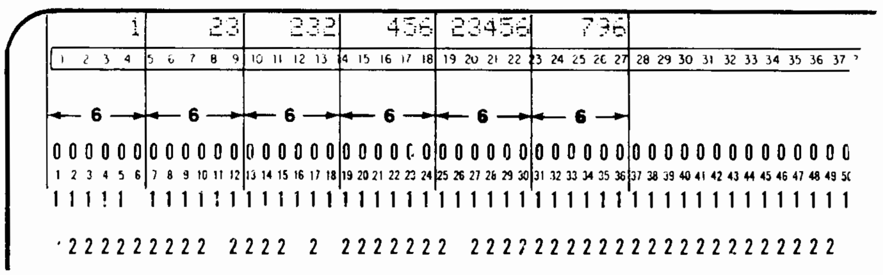


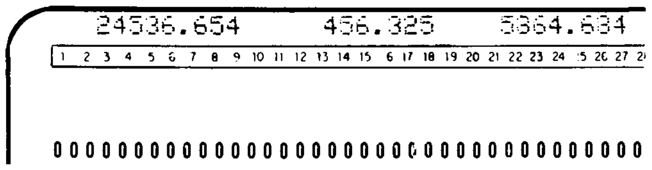
Figure 6-1. Input Record Example

A READ statement such as READ (5, 100) I, J, K, L, M, N would read the six values and assign 1 to I, 23 to J, 232 to K, 456 to L, 23456 to M and 796 to N. A READ statement with less than six elements in *list,* such as READ (5, 100) I, J would read only the first two values from the card. The remaining values in the record would be ignored because any subsequent READ statements would read values from the *next* record.

Thus, each READ statement begins reading values from a fresh record of the file, ignoring any values left unread in records accessed by previous READ statements. If a READ statement *list* does not contain any elements, no transfer of data occurs *except* as follows: The operation of the FORTRAN/3000 formatter is such that a format string is processed up to the point of requiring a list value, then control is returned to the calling program to provide the needed list value. Thus, even if the *list* is empty, format edit descriptors are executed up to the first field descriptor. (See Section VII.) In any case, the next record pointer is advanced to the next record.

Array names appearing in *list* represent all the elements in the array (unless the array name is subscripted to represent a specific element). Values are transferred to the array elements in accordance with the array successor function (see Section V).

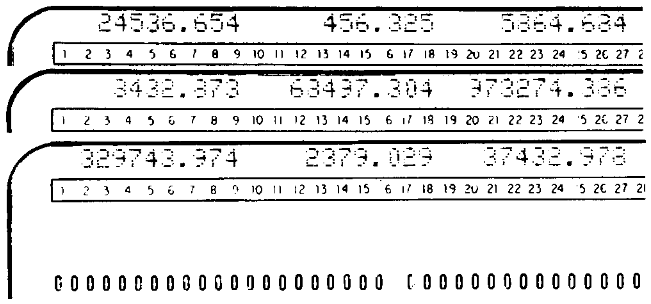
READ statement examples are shown in figures 6-2 through 6-7. The program in figure 6-2 reads three values from a single record (the card shown below).



Each value occupies 12 positions in F format. The program assigns the values as type real to variables A, B, and C. If the current record of FTN05 is shorter than 36 characters, the program aborts with the message: FORMAT BEYOND RECORD. If file FTN05 is empty, or is positioned beyond the last record, an end-of-file condition occurs and control transfers to statement 300, which calls the subroutine ENDOF. If a read error occurs, control is passed to statement 400, which calls the subroutine ERROR.

The first WRITE statement in the program shown in figure 6-3 will print HEADLINE. The READ statement reads a card (bNEWbTITLE) and replaces the characters in the H field of the FORMAT statement with those read from the card. The second WRITE statement then prints NEW TITLE.

In figure 6-4, the FORMAT statement of figure 6-2 (100 FORMAT(3F12.3)) is replaced by 100 FORMAT(F12.3). Then, instead of reading three values from one record (the first record shown below), the program reads one value from each of the three records shown below, (i.e., A is read from record 1, B is read from record 2, and C is read from record 3). This is a convenient way of reading several records with one READ statement. The FORMAT statement also could have been written as 100 FORMAT (F12.3, F12.3, F12.3)

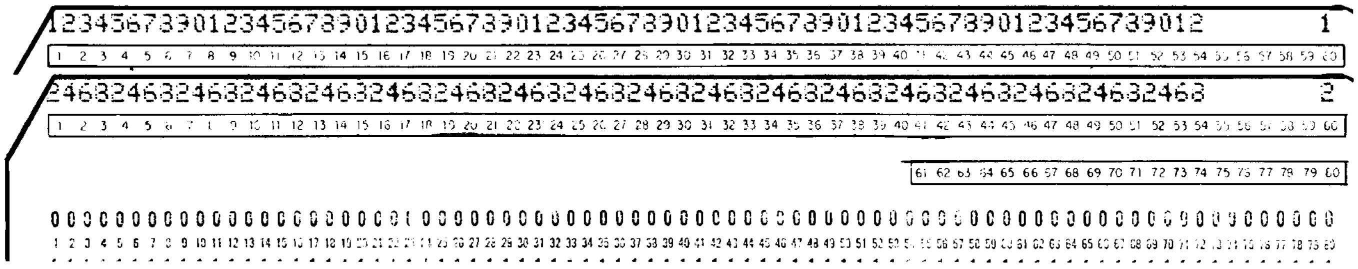


Figures 6-5 and 6-6 demonstrate a direct read. Figure 6-5 is an example of a mailing list, sorted on first names within last names, and stored on disc under the file name MAIL1. The program in figure 6-6 reads record 9 from the mailing list.

Figure 6-7 is an example of a core-to-core tansfer. The data to be read into memory is shown below and consists of 72 numerical characters packed on cards.

Position 80 of the card contains a code number which tells the program if the rest of the data on the card is to be stored as real or integer values.

All of the data on the card is read into memory and assigned to the character variable CARD. Next, the statement, READ (CARD, 100) ICOUNT, reads the code number and assigns it to variable ICOUNT. The computed GOTO statement then passes control to one of two core-to-core READ statements. If ICOUNT = 1, control passes to statement 400 and the data on the card are converted to real values and assigned to variables A, B, C, D, E, and F. If ICOUNT = 2, control passes to statement 500 and the data on the card are converted to integer values and assigned to variables I1, I2, I3, I4, I5, I6, I7, I8, I9, I10, I11, I12, I13, I14, I15, I16, I17, and I18.



```
:FORTGO


 PAGE 0001   HP32102B.00.0


                 PROGRAM READ EXAMPLE 1
           C
           C READ STATEMENT EXAMPLE
           C
            100  FORMAT(3F12.3)
            200  FORMAT(F12.3)
                 READ(5,100,END=300,ERR=400)A,B,C
                 D=SQRT(A*B*C)
                 GOTO 500
            300  CALL ENDOF
                 GOTO 500
            400  CALL ERROR
            500  WRITE(6,200)D
                 STOP
                 END








                 SUBROUTINE ENDOF
            100  FORMAT(T10,"END OF FILE")
                 WRITE(6,100)
                 RETURN
                 END








                 SUBROUTINE ERROR
            100  FORMAT(T108"READ ERROR")
                 WRITE(6,100)
                 RETURN
                 END


 ****      GLOBAL STATISTICS       ****
 ****    NO ERRORS,   NO WARNINGS  ****
 TOTAL COMPILATION TIME  0:00:01
 TOTAL ELAPSED TIME      0:00:04



 END OF COMPILE

 END OF PREPARE



    256251.906



 END OF PROGRAM
:EOD
IGNORED
:EOJ
```

Figure 6-2. READ Statement Example 1

```
:FORTGO


PAGE 0001   HP32102B.00.0

                 PROGRAM READ EXAMPLE 2
          C
          C READ STATEMENT EXAMPLE
          C
           100   FORMAT(10H HEADLINE )
                 WRITE(6,100)
                 READ(5,100)
                 WRITE(6,100)
                 STOP
                 END


 ****      GLOBAL STATISTICS       ****
 ****   NO ERRORS,   NO WARNINGS   ****
 TOTAL COMPILATION TIME  0:00:01
 TOTAL ELAPSED TIME      0:00:04



  END OF COMPILE

  END OF PREPARE



  HEADLINE
  NEW TITLE



  END OF PROGRAM
:EOD
IGNORED
:EOJ
```

Figure 6-3. READ Statement Example 2

```
:FORTGO

 PAGE 0001   HP32102B.00.0

               PROGRAM READ EXAMPLE 3
          C
          C READ STATEMENT EXAMPLE
          C
           100 FORMAT(F12.3)
               READ(5,100,END=300,ERR=400)A,B,C
               D=SQRT(A*B*C)
               GOTO 500
           300 CALL ENDOF
               GOTO 500
           400 CALL ERROR
           500 WRITE(6,100)D
               STOP
               END









               SUBROUTINE ENDOF
           100 FORMAT(T10,"END OF FILE")
               WRITE(6,100)
               RETURN
               END









               SUBROUTINE ERROR
           100 FORMAT(T108"READ ERROR")
               WRITE(6,100)
               RETURN
               END


  ****       GLOBAL STATISTICS       ****
  ****    NO ERRORS,    NO WARNINGS  ****
  TOTAL COMPILATION TIME  0:00:01
  TOTAL ELAPSED TIME      0:00:05


  END OF COMPILE

  END OF PREPARE

     8260073.000


  END OF PROGRAM
 :EOD
 IGNORED
 :EOJ
```

Figure 6-4. READ Statement Example 3

```
PAGE 1      HEWLETT-PACKARD 32201A.4.01 EDIT/3000 WED, FEB 19, 1975, 11:25 AM

     1      LOIS      ANYONE    6190 COURT ST       METROPOLIS  NY 20115 619-732-4997
     2      KING      ARTHUR    329 EXCALIBUR ST    CAMELOT     CA 61322 812-200-0100
     3      ALI       BABA      40 THIEVES WAY      SESAME      CO 69142 NONE
     4      JOHN      BIGTOWN   965 APPIAN WAY      METROPOLIS  NY 20013 619-407-2314
     5      KNEE      BUCKLER   974 FISTICUFF DR    PUGILIST    ND 04321 976-299-2990
     6      SWASH     BUCKLER   497 PLAYACTING CT   MOVIETOWN   CA 61497 NONE
     7      JAMES     DOE       4193 ANY ST         ANYTOWN     MD 00133 237-408-7100
     8      JANE      DOE       3959 TREEWOOD LN    BIGTOWN     MA 21843 714-399-4563
     9      JOHN      DOUGHE    239 MAIN ST         HOMETOWN    MA 26999 714-411-1123
    10      JENNA     GRANDTR   493 TWENTIETH ST    PROGRESSIVE CA 61335 799-191-9191
    11      KARISSA   GRANDTR   7917 BROADMOOR WAY  BIGTOWN     MA 21799 713-244-3717
    12      SPACE     MANN      9999 GALAXY WAY     UNIVERSE    CA 61239 231-999-9999
```

Figure 6-5. Mailing List Example

```
:FILE FTN20=MAILLIST,OLD
:FORTGO FTRAN18

PAGE 0001    HP32102B.00.0


$CONTROL FILE=20
      PROGRAM DIRECT READ
C
C READ @ RECORD EXAMPLE
C
 100  FORMAT(S)
      CHARACTER NAME*72
      READ(20@9,100)NAME
      DISPLAY NAME
      STOP
      END


****       GLOBAL STATISTICS        ****
****    NO ERRORS,    NO WARNINGS   ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:00:05

 END OF COMPILE

 END OF PREPARE


JOHN      DOUGHE   239 MAIN ST          HOMETOWN    MA 26999 714-411-1123

 END OF PROGRAM
```

Figure 6-6. Direct READ Example

```
:FORTGO

PAGE 0001   HP32102B.00.0


                  PROGRAM CORETOCORE
           C
           C CORE-TO-CORE READ EXAMPLE
           C
            100   FORMAT(78X,I2)
            150   FORMAT(6F12.3)
            200   FORMAT(18I4)
                  CHARACTER CARD*80
            300   READ(5,*,END=600)CARD
                  READ(CARD,100)ICOUNT
                  GOTO(400,500),ICOUNT
            400   READ(CARD,150)A,B,C,D,E,F
                  DISPLAY A,B,C,D,E,F
                  GOTO 300
            500   READ(CARD,200)I1,I2,I3,I4,I5,I6,I7,I8,
                 *I9,I10,I11,I12,I13,I14,I15,I16,I17,I18
                  DISPLAY I1,I2,I3,I4,I5,I6,I7,I8,I9,I10,I11,
                 *I12,I13,I14,I15,I16,I17,I18
                  GOTO 300
            600   STOP
                  END


****      GLOBAL STATISTICS        ****
****   NO ERRORS,   NO WARNINGS    ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME        0:00:06


            END OF COMPILE

            END OF PREPARE


.123457E+09  .345679E+09  .567890E+09  .789012E+09  .901235E+09  .123457E+09
 2468   2468   2468   2468   2468   2468   2468   2468   2468   2468   2468   2468   2468   2468   2468   2468   2468   2468

            END OF PROGRAM
           :EOJ
```

Figure 6-7. Core-to-Core READ Example

6-3. READ STATEMENT ERR PARAMETER

Figures 6-8 an 6-9 demonstrate the effect of file read errors.

In figure 6-8, a non-existent file, MAIL3, is equated to FTN20 with a :FILE command and a program which tries to access this file is executed. The program aborts with a file information display, naming error number 52 (the file referenced does not exist in the system file domain) as the cause. The remainder of the program does not execute.

Figure 6-9 demonstrates the use of the ERR parameter to return control to the program even if a file read error occurs.

The same non-existent file (MAIL3) is named in the :FILE command. This time, when the error occurs, control is passed to statement number 25. Statement number 25 calls the file system intrinsic FCHECK to obtain the error number and this information, along with the message "FILE READ ERROR" is output. The remainder of the program continues to execute. See the *MPE Intrinsics Reference Manual* and Section VIII of this manual for a discussion of file system intrinsics and see Appendix A for a description of parameters passed by value (use of back slash).

Note: The error (error number 52) referenced in figures 6-8 and 6-9 is a *file error* and is not covered in Appendix D (Error and Warning Messages) in this manual. Refer to Section VI of the *MPE Intrinsics Reference Manual* for a discussion of file errors.

6-4. WRITE STATEMENT

A WRITE statement transfers information from specified data elements in memory to an external file or to a character variable in memory.

The form of a WRITE statement is

WRITE *(destination, format, END = sn$_1$, ERR = sn$_2$) list*

For example,

destination **→** ***format* reference** ***list***

WRITE (6, 100, END = 500, ERR = 600) A, B, C

or

WRITE (6, 100) ***(END, ERR,* and *list*** **omitted)**

where

destination
identifies the destination of the data to be transferred. It may be an integer constant or integer simple variable. If the variable contains a negative number, that number is assumed to be the negative of the MPE file number desired. (MPE file numbers are returned by the FOPEN intrinsic. (See Section VIII.)

1. Sequential WRITE
Positive integer constant or integer simple variable indicates the FORTRAN unit number of the file which is to receive the data. For example, if the value is 6, then the information is transferred to FTN06.
Example:
WRITE (6, 100) A

FORTRAN unit no. — *format* reference

2. Direct WRITE
Same as above, with @ *record* appended to the unit number. *Record* is a constant or any linear expression whose integer value is taken as the record number to be written. This option is restricted to files on direct access devices such as disc, with fixed length records. FORTRAN references the first record in the file as record number 1.
Example:
WRITE (6 @ 18, 200) A

FORTRAN unit no. @ record

3. Core-to-core WRITE.
If the destination where the data is to be written is a buffer in memory, then *destination* consists of: *Name*, where *name* is a character simple variable or character array element specifying the name of the buffer in memory. (See figure 6-12.)
Example:

WRITE (ZIP, 150)"11256"

name of buffer

*format, END,*and *ERR*
are identical to the READ statement parameters *format, END* and *ERR*. See paragraph 6-1.

list
is one or more simple variable names, array names, array elements, function subprogram names, expressions, character constants, or DO-implied lists. (See paragraph 6-8 for a discussion of DO-implied lists.)

6-5. WRITE STATEMENT EXECUTION

When a WRITE statement is executed, values are transferred from the data elements in *list* to the output file indicated by *unit* in the WRITE statement *destination* part. Values are transferred left-to-right as the elements appear in *list*.

Each WRITE statement begins writing values into a fresh record of the destination file, ignoring any space left unused in records accessed by previous write statements. The first byte of the output record created under format control

```
:FILE FTN2Ø=MAIL3,OLD
:FORTGO FTRAN45

PAGE ØØØ1    HP321Ø2B.ØØ.Ø


ØØØØ1ØØØ  $CONTROL FILE=2Ø
ØØØØ2ØØØ        PROGRAM IRRECOVERABLE
ØØØØ3ØØØ  C
ØØØØ4ØØØ  C IRRECOVERABLE FILE ERROR EXAMPLE
ØØØØ5ØØØ  C
ØØØØ6ØØØ   1ØØ  FORMAT(2X,S)
ØØØØ7ØØØ        CHARACTER*72 REC
ØØØØ8ØØØ   1Ø   READ(2Ø,*,END=5Ø)REC
ØØØØ9ØØØ        WRITE(6,1ØØ)REC
ØØØ1ØØØØ        GOTO 1Ø
ØØØ11ØØØ   5Ø   DO 75 I=1,1Ø
ØØØ12ØØØ   75   DISPLAY I
ØØØ13ØØØ        STOP
ØØØ14ØØØ        END



****      GLOBAL STATISTICS      ****
****    NO ERRORS,   NO WARNINGS  ****
TOTAL COMPILATION TIME  Ø:ØØ:Ø1
TOTAL ELAPSED TIME      Ø:ØØ:Ø4

 END OF COMPILE

 END OF PREPARE

FILE SYSTEM ERROR ON UNIT # 2Ø

+-F-I-L-E---I-N-F-O-R-M-A-T-I-O-N---D-I-S-P-L-A-Y+
!  FILE NUMBER -2Ø    IS UNDEFINED.              !
!  ERROR NUMBER: 52    RESIDUE: Ø                 !
!  BLOCK NUMBER: Ø             NUMREC: Ø          !
+-------------------------------------------------+
SEGMENT   LOCATION
  217       ØØØØ16

  ABORT :$OLDPASS...%Ø.%16:SYSL.%24.%3327: PROCESS QUIT
```

Figure 6-8. Irrecoverable File Error Example

```
:FILE FTN2Ø=MAIL3,OLD
:FORTGO FTRAN46

PAGE ØØØ1   HP321Ø2B.ØØ.Ø


$CONTROL FILE=2Ø
      PROGRAM ERR EXAMPLE
C
C ERR PARAMETER EXAMPLE
C CONTROL RETURNS TO THE PROGRAM
C
 1ØØ  FORMAT(2X,S)
      CHARACTER*72 REC
 1Ø   READ(2Ø,*,END=5Ø,ERR=25)REC
      WRITE(6,1ØØ)REC
      GOTO 1Ø
 25   CALL FCHECK(\Ø\,IERR,Ø,Ø.Ø,Ø,\%3Ø\)
      DISPLAY "FILE READ ERROR"
      DISPLAY "ERROR NUMBER = ",IERR
 5Ø   DO 75 I=1,1Ø
 75   DISPLAY I
      STOP
      END



****       GLOBAL STATISTICS        ****
****    NO ERRORS,    NO WARNINGS   ****
TOTAL COMPILATION TIME  Ø:ØØ:Ø1
TOTAL ELAPSED TIME      Ø:ØØ:Ø4

 END OF COMPILE

 END OF PREPARE


FILE READ ERROR
ERROR NUMBER =       52
     1
     2
     3
     4
     5
     6
     7
     8
     9
    1Ø
 END OF PROGRAM
```

Figure 6-9. READ Statement ERR Parameter Example

is always considered to be a carriage control byte for files which recognize carriage control (CCTL). As with reads, the function of the FORTRAN/3000 formatter is to execute the operations implied by the FORMAT statement until either a value is required from the list or the end of the format is encountered. Thus, edit descriptors (such as the " . . . " descriptor, see Section VII) will be executed if they are encountered before one of the above conditions is true. In any case, after the "write" the next record pointer is advanced.

If a WRITE statement contains *no* format reference, a binary write is initiated. For sequential access (*destination* reference = *unit*), records are written sequentially until the value of the last element in *list* has been transferred. For a direct-access file (*destination* reference = *unit @ record*), only one record is written. The destination file record size must be large enough to accept all the values indicated in the WRITE statement *list*. Otherwise, a run error occurs. An unformatted WRITE statement must have an element *list*. If a WRITE statement *does* contain a format reference, a formatted write is initiated. Records are written sequentially until all of the *list* element values are transferred, regardless of whether the file is direct or sequential access (unless an end-of-file or format beyond record error condition occurs).

Array names appearing in *list* represent all the elements of the array (unless the array name is subscripted to represent a specific element). Array element values are transferred in accordance with the array successor function (see Section V).

Figures 6-10 through 6-12 are examples of WRITE statement execution. Figure 6-10 demonstrates a sequential write. The first 19 characters of the file MAIL1 (see figure 6-5) are read and assigned to the character variable NAME.The WRITE statement then transfers NAME to FORTRAN unit number 6.

Figure 6-11 illustrates a direct write. A new value is written into record 9 of the file MAIL1. The REWIND statement (see paragraph 6-9) then positions the "record pointer" to record 1 of the file and the second WRITE statement transfers each record, in sequence, to the output file. The program in figure 6-12 performs a core-to-core write. The READ statement reads the zip code from each record of file MAIL1 and assigns the value to the variable ZIP and, if it equals "20013" or "20115", transfers control to statement 400. The second WRITE statement writes a new value into ZIP. Note that the value of the zip code in the file is not changed inasmuch as the write is to the character variable ZIP and not to FORTRAN unit number 20 (the unit number of the file).

6-6. ACCEPT STATEMENT

An ACCEPT statement is a read statement intended for free-field format programs which are to be run in interactive mode from a terminal device (such as a teleprinter). ACCEPT statements are not resticted to interactive mode, however, and can be used to input free-field data from any input device.

Data transfers by ACCEPT statements conform to free-field transfers with one exception: FORTRAN/3000 determines if the standard input device (FTN05) to be used is a terminal; if the device is a terminal, a prompt character, ?, is printed before the data is accepted.

The form of an ACCEPT statement is

ACCEPT *list*

For example,

ACCEPT A, B, C

where

list
is one or more simple variable names, array names, array elements, function subprogram names, or DO-implied lists. (See paragraph 6-8 for a discussion of DO-implied lists.)
Example:

```
INTEGER A, B, C
ACCEPT A, B, C
STOP
END
```

When the program executes, it types the prompt character, ? (if $STDLIST is a terminal). The user answers, for example, 35, 455, 733. The commas are used to separate, or *delimit,* the three data values. (Other delimiters which can be used are blank spaces or any ASCII characters which are not part of the data item.) The program then assigns 35 to A, 455 to B, and 733 to C.

If the program is not executing from a terminal, the prompt character would not be printed and the program would read three values. The ACCEPT statement is equivalent to

```
READ (5,*)
```

(except for the prompt) where 5 represents the unit number of the standard input file and * specifies free-field format.

6-7. DISPLAY STATEMENT

A DISPLAY statement is a write statement intended for (but not restricted to) programs outputting free-field data in interative mode.

The form of a DISPLAY statement is

DISPLAY *list*

For example,

DISPLAY A, B, C

```
:FILE FTN2Ø=MAIL1,OLD
:FORTGO FTRAN2Ø

PAGE ØØØ1    HP321Ø2B.ØØ.Ø


ØØØØ1ØØØ   $CONTROL FILE=2Ø
ØØØØ2ØØØ         PROGRAM SEQUENTIAL
ØØØØ3ØØØ   C
ØØØØ4ØØØ   C SEQUENTIAL WRITE EXAMPLE
ØØØØ5ØØØ   C
ØØØØ6ØØØ    1ØØ  FORMAT(S)
ØØØØ7ØØØ    2ØØ  FORMAT(T1Ø,S)
ØØØØ8ØØØ         CHARACTER*19 NAME
ØØØØ9ØØØ    3ØØ  READ(2Ø,1ØØ,END=4ØØ)NAME
ØØØ1ØØØØ         WRITE(6,2ØØ)NAME
ØØØ11ØØØ         GOTO 3ØØ
ØØØ12ØØØ    4ØØ  STOP
ØØØ13ØØØ         END



****       GLOBAL STATISTICS       ****
****    NO ERRORS,   NO WARNINGS   ****
TOTAL COMPILATION TIME  Ø:ØØ:Ø1
TOTAL ELAPSED TIME      Ø:ØØ:Ø9

 END OF COMPILE

 END OF PREPARE


        LOIS      ANYONE
        KING      ARTHUR
        ALI       BABA
        JOHN      BIGTOWN
        KNEE      BUCKLER
        SWASH     BUCKLER
        JAMES     DOE
        JANE      DOE
        JOHN      DOUGHE
        JENNA     GRANDTR
        KARISSA   GRANDTR
        SPACE     MANN
 END OF PROGRAM
```

Figure 6-10. Sequential WRITE Example

```
:FILE FTN2Ø=MAIL1,OLD
:FORTGO FTRAN21

 PAGE ØØØ1   HP321Ø2B.ØØ.Ø


$CONTROL FILE=2Ø
              PROGRAM DIRECT
C
C DIRECT WRITE EXAMPLE
C
 1ØØ         FORMAT(S)
 2ØØ         FORMAT(T1Ø,S)
              CHARACTER*19 NAME
              NAME="SPEEDY    RABBIT"
              WRITE(2Ø@9,1ØØ)NAME
              REWIND 2Ø
 3ØØ         READ(2Ø,1ØØ,END=4ØØ)NAME
              WRITE(6,2ØØ)NAME
              GOTO 3ØØ
 4ØØ         STOP
              END


****       GLOBAL STATISTICS       ****
****    NO ERRORS,    NO WARNINGS   ****
TOTAL COMPILATION TIME   Ø:ØØ:Ø1
TOTAL ELAPSED TIME       Ø:ØØ:Ø6

 END OF COMPILE

 END OF PREPARE


         LOIS       ANYONE
         KING       ARTHUR
         ALI        BABA
         JOHN       BIGTOWN
         KNEE       BUCKLER
         SWASH      BUCKLER
         JAMES      DOE
         JANE       DOE
         SPEEDY     RABBIT
         JENNA      GRANDTR
         KARISSA    GRANDTR
         SPACE      MANN
 END OF PROGRAM
```

Figure 6-11. Direct WRITE Example

```
:FILE FTN20=MAILLIST,OLD
:FORTGO FTRAN22

PAGE 0001    HP32102B.00.0


$CONTROL FILE=20
            PROGRAM COREWRITE
C
C CORE-TO-CORE WRITE
C
 100        FORMAT(54X,S)
 150        FORMAT(S)
 200        FORMAT(//54X,S//)
            CHARACTER*5 ZIP
 300        READ(20,100,END=500)ZIP
            WRITE(6,100)ZIP
            IF(ZIP.EQ."20013".OR.ZIP.EQ."20115")GOTO 400
            GOTO 300
 400        WRITE(ZIP,150)"11256"
            WRITE(6,200)ZIP
            GOTO 300
 500        STOP
            END



****       GLOBAL STATISTICS        ****
****    NO ERRORS,    NO WARNINGS   ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:00:04

 END OF COMPILE

 END OF PREPARE


                                                      20115


                                                      11256


                                                      61322
                                                      69142
                                                      20013


                                                      11256


                                                      04321
                                                      61497
                                                      00133
                                                      21843
                                                      26999
                                                      61335
                                                      21799
                                                      61239
 END OF PROGRAM
```

Figure 6-12. Core-to-Core WRITE Example

where

list
is one or more simple variable names, array names, array elements, function subprogram names, or DO-implied lists. (See paragraph 6-8 for a discussion of DO-implied lists.) An expression of any type also can be used in *list*. The expression is evaluated and its value is transferred to the standard list device (unit number 6 (FTN06)).

When a DISPLAY statement is executed, the values of the data elements in *list* are output in free-field format to the unit number 6 (such as a teleprinter in interactive mode or a line printer in batch mode). The DISPLAY statement creates as many records as needed to output all of the element values in *list*.
EXAMPLE:

```
        INTEGER A, B, C
        A = 7
        B = 45
        C = 7666
        DISPLAY A, B, C
        STOP
        END
```

When the program is executed, the DISPLAY statement causes the following output:

```
        7       45       7666
```

A DISPLAY statement is equivalent to

```
        WRITE (6, *)
```

where 6 represents the file number of the standard list file and * specifies free-field format.

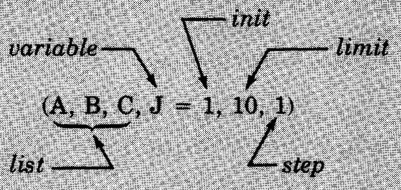
6-8. DO-IMPLIED LISTS

READ, WRITE, ACCEPT and DISPLAY statements can contain DO-implied lists. A DO-implied list contains a list of data elements to be input (read) or output (written), and a set of indexing parameters.

The form of a DO-implied list is

(list, variable = init, limit, step)
or
(list, variable = init, limit)

For example,



or

(A, B, C, J = 1, 10)

where

list
is one or more simple variable names, array names, array elements, function subprogram names, expressions, or other DO-implied lists.

variable
is an integer or double integer simple variable used as an index variable which controls the number of times the elements in *list* are read or written.

init
is the initial value given to *variable* at the start of the DO-implied list.

limit
is the termination value for *variable*.

step
is the increment by which *variable* is changed after each execution of the DO-implied list. *Step* can be positive or negative, but not zero. If *step* is omitted, it is assumed to be 1.

Init, limit and *step* can be arithmetic expressions of any type except complex. They are converted to integer when used.

The DO-implied list acts as a DO loop (see Section IV). The range of the implied DO loop is the list of elements to be input/output. The implied DO loop can transfer a list of simple variables or array elements, etc., or any combination of allowable data elements. The control variable is assigned the value of *init* at the start of the loop.

The list of elements is transferred. The control variable is then incremented by the value of *step*, or by 1 if *step* is omitted. The control variable is compared with *limit*. If *step* is positive and the control variable is greater than *limit*, the implied DO loop terminates; otherwise, the list is transmitted again.

If *step* is negative and the control variable is less than *limit*, the implied DO loop terminates; otherwise, the list is transferred again.

For example,

```
        WRITE (6, 100) (A,I = 1,3)
```

(A,I = 1,3) is a DO-implied list. A is a simple variable of type real. The effect of the DO-implied list is to write the value of A three times in succession. If A = 35.6, the output would consist of one record as follows:

```
        35.6        35.6        35.6
```

The preceding example is similar to, but more efficient than:

```
       DO 10 I = 1,3
       WRITE (6, 100) A
  10   CONTINUE
```

The last example would output three records, as follows:

```
35.6
35.6
35.6
```

If the *list* of an implied DO contains several simple variables, each of the variables in the list is input/output for each pass through the loop.

For example,

```
READ (5, 100) (A, B, C, J = 1,2)
```

is the same as

```
READ (5,100) A, B, C, A, B, C
```

A DO-implied list also can transmit arrays and array elements:

```
WRITE (6,100) (A (I), I = 1,10)
```

results in the array A being written in the following order:

```
A(1) A(2) A(3) A(4) A(5) A(6) A(7) A(8) A(9) A(10)
```

If an unsubscripted array name is used in *list,* the entire array is transmitted:

```
DIMENSION A(5)
WRITE (6, 100) A
STOP
END
```

The WRITE statement causes output as follows:

```
A(1) A(2) A(3) A(4) A(5)
```

The result of the following statements:

```
DIMENSION A(3)
WRITE (6, 100) (A,I = 1,2)
STOP
END
```

is to write the elements of array A twice as follows:

```
A(1) A(2) A(3) A(1) A(2) A(3)
```

DO-implied lists also can be nested to transmit arrays of more than one dimension. The form of a nested DO-implied list is:

((list, variable$_1$ = init, limit, step), variable$_2$ = init, limit, step)

For example,

```
READ (5, 100) ((ARRAY(M, N), M = 1, 10, 1), N = 1,
10, 1)
```

The nesting of DO-implied lists follows the same rules as nested DO loops.

For example,

```
WRITE (6, 100) ((A(I, J), I = 1,2), J = 1,2)
```

produces the following output:

```
A(1, 1) A(2, 1) A(1, 2) A(2, 2)
```

The first, or nested DO loop is satisfied before the outer DO loop begins its executions. For a complete discussion of nested DO loops, see Section IV.

6-9. AUXILIARY INPUT/OUTPUT STATEMENTS

Auxiliary input/output statements consist of the REWIND, BACKSPACE, and ENDFILE commands. These statements are used for control of magnetic tape and disc devices. If a referenced device does not have the capability to perform a request, an error is generated when the program is executed.

The form of these statements is:

REWIND *unit*

BACKSPACE *unit*

ENDFILE *unit*

For example,

```
REWIND 12
BACKSPACE 12
ENDFILE 12
```

where

unit
: is a file reference consisting of a positive integer constant or integer simple variable within the value range of from 1 through 99.

The REWIND command positions the "record pointer" to the first record of a referenced file. This command may invoke a physical rewind of the referenced device, if necessary, to point to the first record.

The BACKSPACE command positions the "record pointer" to the preceding record. The BACKSPACE statement may not reference files of variable length records.

ENDFILE writes an end-of-file record. The file can be reopened and accessed by other I/O statements at a later time.

For further descriptions of REWIND, BACKSPACE, and ENDFILE, see UNITCONTROL in Section VIII.

WRITING FORMAT STATEMENTS IN FORTRAN/3000

SECTION VII

FORMAT statements are nonexecutable statements which describe to the computer how input and output information is to be arranged. The *Formatter* is a subroutine called by FORTRAN/3000 compiler-generated code. The FORTRAN/3000 compiler interprets READ, WRITE, ACCEPT, and DISPLAY statements of a program to generate the calls to the Formatter.

The Formatter can perform the following functions:

1. Convert between external ASCII records and an internally represented list of variables. Formatting proceeds according to parameters derived from the FORTRAN/3000 program's FORMAT statements.

2. Convert free-field external ASCII records to an internally represented list of variables according to predefined formats and/or edit control characters embedded in the input records.

3. Convert an internally represented list of variables to external ASCII records which are free-field input-compatible.

4. Convert between an internally represented list of variables and a user-defined ASCII buffer storage area (core-to-core).

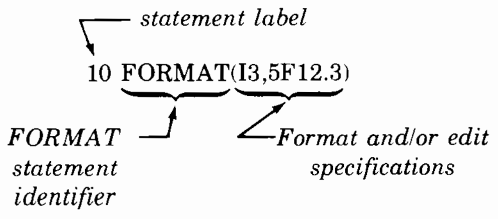
5. Transfer (unformatted and without conversion) between an internally represented list of variables and an external file.

The Formatter derives edit and format parameters from FORMAT statements or the data itself.

7-1. FORMAT STATEMENTS

FORTRAN/3000 FORMAT statements consist of a statement label and a series of format and/or edit specifications enclosed in parentheses. The specifications must be separated by commas or record terminators (see paragraph 7-32 for a discussion of record terminators). On rare occasions, the FORTRAN/3000 compiler may fail to detect the absence of these separators, and an error may result during program execution.

The form of a FORMAT statement is:


statement label
10 FORMAT(I3,5F12.3)
FORMAT statement identifier
Format and/or edit specifications

In the preceding FORMAT statement, the specification I3 specifies an integer number with a field width of 3 (see paragraph 7-16) and five real numbers, each with a field width of 12 with three significant digits to the right of the decimal point (see paragraph 7-11). A READ statement referencing this FORMAT statement could be of the form:

```
READ(5,10)ITEM,A,B,C,D,E
```

The input data as it might appear on a card is shown in figure 7-1.

Format and edit specifications can include another set of format and/or edit specifications enclosed in parentheses; this is called nesting. For example, the information shown on the card in figure 7-2 could be represented in a FORMAT statement as follows:

```
10 FORMAT(I3,F7.4,3(F7.2,I3),F12.4)
```

A READ statement corresponding with the FORMAT statement could be:

```
READ(5,10)I,A,B,J.D,K,E,L,F
```

The READ statement would read values for I and A, then repeat the parenthetical statement (F7.2,I3) three times to read values for B and J, D and K, and E and K and, finally, read a value for F.

FORTRAN/3000 allows nesting to a depth of four levels. For a further discussion of nesting, see paragraph 7-36.

See paragraph 7-51 for information about input/output of complex numbers.

7-2. FORMAT SPECIFICATIONS

Format specifications are written as:

- A field descriptor.
- A scale factor followed by a field descriptor.
- A repeat specification followed by a field descriptor.
- A scale factor followed by a repeat specification.

7-3. FIELD DESCRIPTORS

For output of data, a field descriptor specifies the data field into which the value of a list element will be written. For input, a field descriptor defines the field from which data will be read for assignment to a list element.

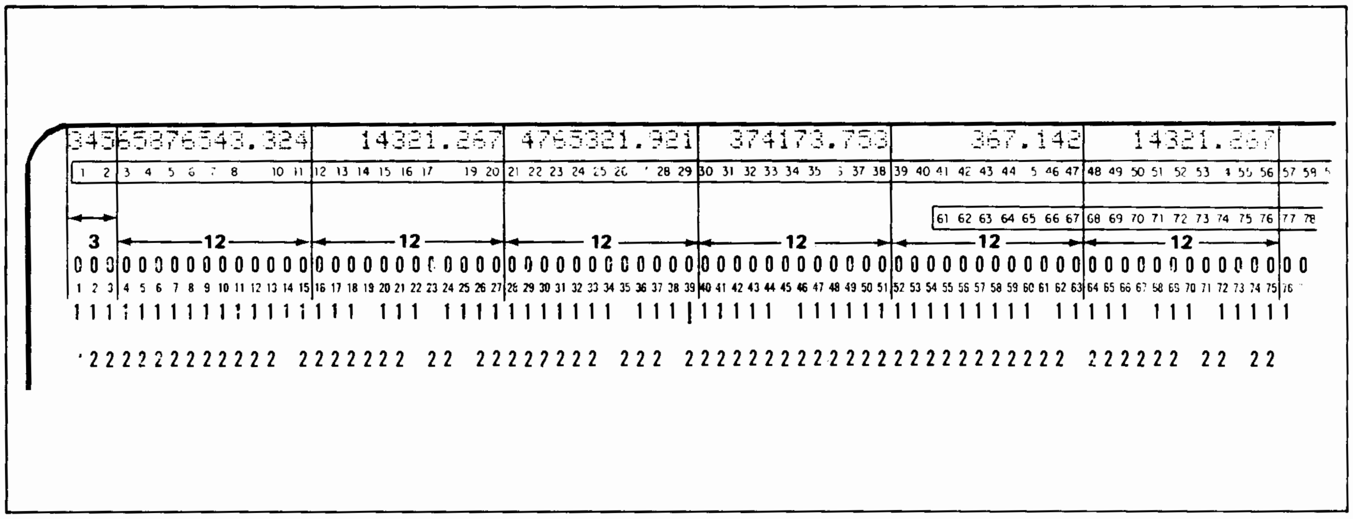


Figure 7-1. Input Data (Example 1)

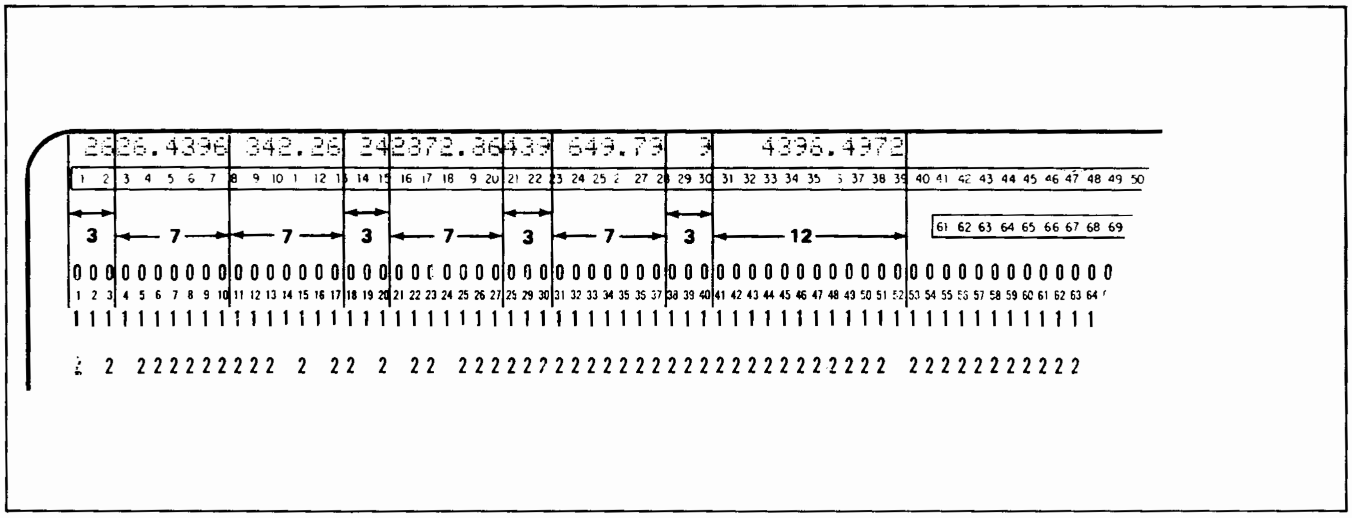


Figure 7-2. Input Data (Example 2)

7-4. DECIMAL NUMERIC CONVERSIONS. Seven field descriptor forms are provided for decimal numeric conversions, as follows:

1.	D*w.d*	External representation in double precision, floating point (with an exponent field) form. See paragraph 7-15.
2.	E*w.d*	External representation in real, floating point (with an exponent field) form. See paragraph 7-10.
3.	F*w.d*	External representation in real, fixed point (with *no* exponent field) form. See paragraph 7-11.
4.	G*w.d*	External representation in either the F*w.d* format or the E*w.d* format, depending on the relative size of the number to be converted. See paragraph 7-12.
5.	M*w.d*	External representation in monetary (business) form (real, fixed point, including $ and commas). For example, $4,379.89. See paragraph 7-13.
6.	N*w.d*	External representation in numeration form (same as the M*w.d* format, but without the $). For example, 3,267.54. See paragraph 7-14.
7.	I*w*	External representation in integer form. See paragraph 7-16.

In the preceding field descriptor forms,

w = the length of the external data field in characters; *w* must be greater than zero.

d = the number of fraction field digits in a floating- or fixed-point output. On input, if the external data does *not* include a decimal point, the integer is multiplied by 10^{-d}. If the external data *does* include a decimal point, this specification has no effect. In all the listed field descriptor forms, *d* must be stated even if zero.

7-5. RULES FOR INPUT. All of the field descriptors listed in paragraph 7-4 will accept ASCII numeric input in the following formats:

Note: I*w*, on input, is interpreted as F*w*.0.

1. A series of integer number digits with or without a sign.

   2314 or +26783 or −96

2. Any of the above with an exponent field with or without a sign.

   2314+2 or +26783E−4 or −96D+4

3. A series of real number digits with or without a sign.

   2.314 or +267.83 or +.96

4. Any of the items in 3, with an exponent field with or without a sign.

   2.314+2 or +267.83E−4 or −.96D+4

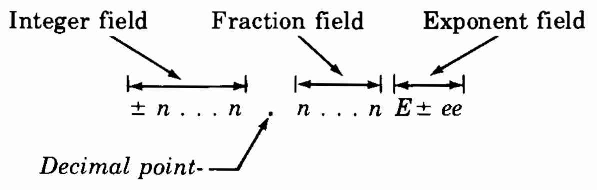
5. Any of the items in 1 or 3, in monetary form.

   \$2,314 or \$2,678.30 or −\$.96

6. Any of the items in 1 or 3, in numeration form.

   2.314 or +2,678.30 or −961,534,873

In summary, the input field can include integer, fraction, and exponent subfields:



Rules:

1. The number of characters in the input field, including \$ and commas, must not exceed *w* in the field descriptor used.

2. The exponent field input can be any of several forms:

+*e*	+*ee*	E*e*	E*ee*	D*e*	D*ee*
−*e*	−*ee*	E+*e*	E+*ee*	D+*e*	D+*ee*
		E−*e*	E−*ee*	D−*e*	D−*ee*

   where *e* is an exponent value digit.

3. Embedded or trailing blanks (to the right of any character read as a value) are treated as zeros; leading blanks are ignored. A field of all blanks is treated as zero.

EXAMPLES:

1b23 = 1023	.2b56bE+b4 = .20560E+04
12.b34 = 12.034	2b2,b45.bb3 = 202045.003
−\$1,b34.bb5 = −1034.005	2.b02−b13 = 2.022−013

4. The type of the internal storage is independent of either the ASCII numeric input or the field descriptor used to read the input. The data is stored according to the type of the list element (variable) currently using the field descriptor. The conversion rules are as follows:

   - Type INTEGER or DOUBLE INTEGER truncates a fractional input.
   - Type REAL rounds a fractional input.
   - Type DOUBLE PRECISION rounds a fractional input.

7-6. OCTAL NUMERIC CONVERSION. One field descriptor form is provided for octal numeric conversion on input or output, as follows:

O*w* for octal numbers from 0 through 1777777777777777777777_8

where

w is the length (in characters) of the external data field (must be greater than zero).

This field descriptor accepts ASCII numeric input up to twenty-two octal digits long. Non-numeric or non-octal characters cause a conversion error. For a complete discussion of this field descriptor, see paragraph 7-17.

7-7. HEXADECIMAL NUMERIC CONVERSION. One field descriptor form is provided for hexadecimal numeric conversion on input or output, as follows:

Z*w* for hexadecimal numbers from 0 through $\text{FFFFFFFFFFFFFFFF}_{16}$

where

w is the length (in characters) of the external data field (must be greater than zero).

This field descriptor accepts ASCII inputs up to sixteen hexadecimal digits long. Non-hexadecimal characters cause a conversion error. For a complete discussion of this field descriptor, see paragraph 7-18.

7-8. LOGICAL CONVERSION. One field descriptor form is provided for logical conversion for input or output:

L*w*	for logical values (T or F followed by any other characters).

This field descriptor accepts any ASCII characters input that begins with either T or F. See paragraph 7-19 for a further discussion of this field descriptor.

7-9. ALPHANUMERIC CONVERSIONS. Three field descriptor forms are provided for alphanumeric conversions:

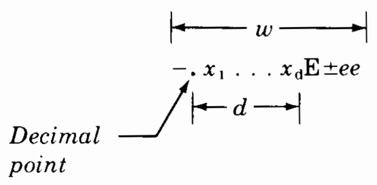
A*w*	for alphanumeric characters to and from the leftmost bytes of a list element. See paragraph 7-20.
R*w*	for alphanumeric characters to and from the rightmost bytes of a list element. See paragraph 7-21.
S	for alphanumeric characters to and from a character string (user-defined character list elements). See paragraph 7-22.

Each of these field descriptors accepts (but provides differing storage of) any ASCII character.

7-10. FLOATING-POINT REAL NUMBERS (E*w.d*). The field descriptor for a floating-point real number with an exponent is of the form E*w.d*.

On output, the E*w.d* field descriptor causes normalized output of a variable (internal representation value: integer, double integer, real or double precision) in ASCII character floating-point form, right-justified. The least significant digit of the output is rounded.

The external field width is *w* positions:



where

$x_1 \ldots x_d$	=	the most significant digits of the value
ee	=	the digits of the exponent value
w	=	the width of the external field
d	=	the number of significant digits allowed in *w* (for output, *d* must be greater than zero)
−(minus)	=	is present if the value is negative.

The field width *w* must follow the general rule:

$$w \geqslant d + 6$$

to provide positions for the sign of the value, the decimal point, *d* digits, the letter E, the sign of the exponent, and the exponent's two digits. If *w* is greater than the number of positions required for the output value, the output is right-justified in the field with blank spaces to the left. If *w* is less than the number of positions required for the value (with the sign, decimal point, and exponent field), the entire field is filled with #'s.

See table 7-1 and figure 7-3 for examples of E*w.d* output. Note that the program in figure 7-3 is written with multiple FORMAT and WRITE statements merely to illustrate how the values are output.

On input, the E*w.d* field descriptor causes interpretation of the next *w* positions in an ASCII input record. The number is converted to an internal representation value for the variable (list element) currently using the field descriptor.

See table 7-2 for E*w.d* input.

Placing the decimal point in an input value is optional. For A and B in table 7-2, the decimal point in the input value agrees with the value of *d* (3 in A and 6 in B). C has no decimal point in the input value so the field descriptor (E8.2) causes the values +3462E3 to be stored as 34.62 x 10^3 (it places the decimal point to agree with the field descriptor). In D, E and F, the decimal point placement in the input value does not agree with the value of *d* in the field descriptor. In these cases, the placement of the decimal point in the actual input value overrides the field descriptor and the values are stored as they appeared on input.

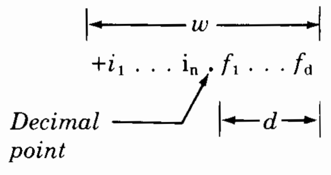
If *w* is less than the number of positions required for an input value (as in E in table 7-2), the least significant digits of the input value are not stored.

Trailing blanks are treated as zeros and can cause errors. For example, the blank after the value read in for F causes this value to be stored as 3.462 x 10^{30}, which is incorrect if the value was intended to be 3.462 x 10^3.

7-11. FIXED-POINT REAL NUMBERS (F*w.d*). The F*w.d* field descriptor defines a field for a real number without an exponent (fixed-point).

On output, the F*w.d* field descriptor causes output of a variable (internal representation value: integer, double integer, real, or double precision) in ASCII character fixed-point form, right-justified. The least significant digit of the output is rounded.

The external field is *w* positions:



where

$i_1 \ldots i_n$	=	the integer digits
$f_1 \ldots f_d$	=	the fraction digits
w	=	the width of the external field
d	=	the number of fractional digits allowed in w
n	=	the number of integer digits
−(minus)	=	is present if the value is negative.

The field width w must follow the general rule:

$w \geq d + n + 3$

to provide positions for the sign, n digits, the decimal point, d digits, and a rollover digit if needed. If w is greater than the number of positions required for the output value, the output is right-justified in the field with blank spaces to the left. If w is less than the number of positions required for the value (with the sign and the decimal point), the entire field is filled with #'s.

Note: Rollover digit is the digit that rolls to the left of a floating point number after rounding the least significant digit of the floating point number.

For example, if −999.996 is the internal value of a number and F8.2 is the field descriptor, then by rounding the least significant digit 6, the digit 1 rolls to the left of the internal value and gives as output the number −1000.00.

See table 7-3 for F*w.d* output.

On input, the F*w.d* field descriptor causes interpretation of the next w positions in an ASCII input record. The number is converted to an internal representation value for the variable (list element) currently using the field descriptor.

Only w positions in an input record will be used to obtain the value.

See table 7-4 for F*w.d* input.

Table 7-1. E*w.d* Output

```
10 FORMAT(E10.3,E10.3,E12.4,E12.4,E7.3,E5.1)
   WRITE(6,10)A,B,C,D,E,F
```

VARIABLE	FIELD DESCRIPTOR	INTERNAL VALUE	OUTPUT
A	E10.3	+12.342	bb.123E+02
B	E10.3	−12.341	b−.123E+02
C	E12.4	+12.340	bbb.1234E+02
D	E12.4	−12.345	bb−.1234E+02
E	E7.3	+12.34	#######
F	E5.1	+12.34	#####

If rounding of the least significant digit occurs and rollover results (for example, 99.99 becomes 100.00), the rollover value is normalized and the exponent is adjusted.

For example,

FIELD DESCRIPTOR	INTERNAL VALUE	OUTPUT
E12.5	−999.998	b−.10000E+04
E11.5	999.996	b.10000E+04
E10.5	−99.9997	##########

```
:FORTGO FTRAN23

 PAGE 0001    HP32102B.00,0


                  PROGRAM FORMAT
             C
             C E FORMAT STATEMENT EXAMPLES
             C
              10    FORMAT(T15,E10.3)
              20    FORMAT(T15,E10.3)
              30    FORMAT(T15,E12.4)
              40    FORMAT(T15,E12.4)
              50    FORMAT(T15,E7.3)
              60    FORMAT(T15,E5.1)
                    A=12.342
                    B=-12.341
                    C=12.340
                    D=-12.345
                    E=12.34
                    F=12.34
                    WRITE(6,10)A
                    WRITE(6,20)B
                    WRITE(6,30)C
                    WRITE(6,40)D
                    WRITE(6,50)E
                    WRITE(6,60)F
              70    FORMAT(T15,E12.5)
              80    FORMAT(T15,E11.5)
              90    FORMAT(T15,E10.5)
                    G=-999.998
                    H=999.996
                    X=-99.9997
                    WRITE(6,70)G
                    WRITE(6,80)H
                    WRITE(6,90)X
             100    FORMAT(T15,"↑")
             110    FORMAT(T5,"POSITION 15")
                    WRITE(6,100)
                    WRITE(6,110)
                    STOP
                    END


 ****       GLOBAL STATISTICS       ****
 ****    NO ERRORS,    NO WARNINGS  ****
 TOTAL COMPILATION TIME   0:00:01
 TOTAL ELAPSED TIME       0:00:04

  END OF COMPILE

  END OF PREPARE


               .123E+02
              -.123E+02
                .1234E+02
               -.1234E+02
             #######
             #####
              -.10000E+04
              .10000E+04
             ##########
             ↑
    POSITION 15
  END OF PROGRAM
```

Figure 7-3. E*w.d* Format Output Example

Table 7-2. E*w.d* Input

```
10 FORMAT(E9.3,E13.6,E8.2,E9.0,E3.2,E8.2)
   READ(5,10)A,B,C,D,E,F
```

INPUT CHARACTERS	FIELD DESCRIPTOR	VARIABLE	INTERPRETED AS
+3.462E03	E9.3	A	3462.00
−7.243242E+02	E13.6	B	−724.324
b+3462E3	E8.2	C	34620
−34.62E−3	E9.0	D	$-.346200 \times 10^{-1}$
3462.E−5	E3.2	E	3.46000
3.462E3b	E8.2	F	3.46200×10^{30}

Table 7-3. F*w.d* Output

```
10 FORMAT(F10.3,F10.3,F12.3,F12.3,F4.3,F43.)
   WRITE(6,10)A,B,C,D,E,F
```

VARIABLE	FIELD DESCRIPTOR	INTERNAL VALUE	OUTPUT
A	F10.3	+12.3402	bbbb12.340
B	F10.3	−12.3413	bbb - 12.341
C	F12.3	+12.3434	bbbbbb12.343
D	F12.3	−12.3456	bbbbb−12.346
E	F4.3	+12.34	####
F	F4.3	+12345.12	####

If rounding of the least significant digit occurs and rollover results, the stated formula for *w* provides enough positions for the value.

For example,

FIELD DESCRIPTOR	INTERNAL VALUE	OUTPUT
F8.2	+999.997	b1000.00
F8.2	−999.996	−1000.00
F7.2	−999.995	#######

Table 7-4. F*w.d* Input

```
10 FORMAT(F8.0,F10.2,F5.3,F10.4,F6.4,F6.1)
   READ(5,10)A,B,C,D,E,F
```

INPUT CHARACTERS	FIELD DESCRIPTOR	VARIABLE	INTERPRETED AS
bbb+b362	F8.0	A	+362
bbbbb−3624	F10.2	B	−36.24
b−.36	F5.3	C	−.36
b−362.4567	F10.4	D	−362.4567
b36240	F6.4	E	3.624
b3.624	F6.1	F	3.624

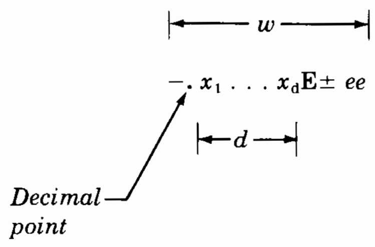
Placing the decimal point in an input value is optional. For example, in table 7-4, the value (−3624) to be read for variable B has no decimal point. The field descriptor (F10.2) places the decimal point in this case and the internal representation becomes −36.24. In case the placement of the decimal point in the input value and the field descriptor do not agree (as in F in table 7-4), the input value overrides the field descriptor and the value stored internally represents the actual value that is read.

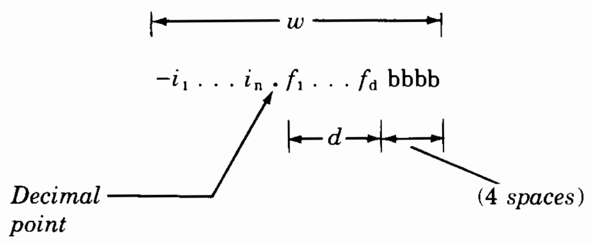
7-12. FIXED-POINT OR FLOATING-POINT REAL NUMBERS (G*w.d*).

The G*w.d* field descriptor defines a field for a fixed-point (without an exponent) *or* a field for a floating-point (with an exponent) number, as needed. The list element determines whether a fixed-point or floating-point transfer will occur.

On output, the G*w.d* field descriptor causes output of a variable (internal representation value: integer, double integer, real or double precision) in ASCII character floating-point form, right-justified.

The external field is *w* positions:





where

$i_1 \ldots i_n$	=	the integer digits (F*w.d* descriptor)
$f_1 \ldots f_d$	=	the fraction digits
$x_1 \ldots x_d$	=	the most significant digits of the value (E*w.d* descriptor)
ee	=	the digits of the exponent value (E*w.d* descriptor)
w	=	the width of the external field
d	=	the number of fractional digits allowed in *w*
n	=	the number of integer digits (F*w.d* descriptor)
− (minus)	=	is present if the value is negative.

The G*w.d* field descriptor is interpreted as an F*w.d* descriptor for fixed-point form or as an E*w.d* descriptor for floating-point form, according to the internal representation absolute value (N) after rounding. If the number of integer digits in N is > *d*, or if N < .1, the E*w.d* field descriptor is used; otherwise the F*w.d* field descriptor is used.

See table 7-5 for G*w.d* output.

Table 7-5. G*w.d* Output

IF		N<0.1	THEN E*w.d*
IF	0.1	≤N<1	THEN F(*w* − 4) . *d* plus 4X (spaces)
IF	1	$\leq N<10^1$	THEN F(*w* − 4) . (*d* − 1) plus 4X
IF	10^1	$\leq N<10^2$	THEN F(*w* − 4) . (*d* − 2) plus 4X
IF	10^2	$\leq N<10^3$	THEN F(*w* − 4) . (*d* − 3) plus 4X
IF	10^3	$\leq N<10^4$	THEN F(*w* − 4) . (*d* − 4) plus 4X
.	.	.	. .
.	.	.	. .
IF	$10^{(d-1)}$	$\leq N<10^d$	THEN F(*w* − 4) . 0 plus 4X
IF	10^d	≤N	THEN E*w.d*

EXAMPLES:

G12.6,N = 1234.5: F(*w* − 4) . (*d* − 4) = F8.2,4X: b1234.50bbb

G13.7,N = 123456.7: F(*w* − 4) . (*d* − 6) = F9.1,4X: b123456.7bbbb

G9.2,N = 123.4: E*w.d* = E9.2: bb.12E+03

The field width *w* must follow the general rule for the E*w.d* descriptor:

$w \geq d + 6$

to provide positions for the sign of the value, *d* digits, the decimal point (preceding x_1), and, if needed, the letter E, the sign of the exponent and the exponent's two digits. If *w* is greater than the number of positions required for the output value, the output is right-justified in the field with blank spaces to the left. If *w* is less than the number of positions required for the value (with the sign, the decimal point, and the exponent field − or 4 spaces), the entire field is filled with #'s.

```
10 FORMAT(G10.3,G10.3,G12.4,G12.4,G12.4,G7.1,G5.1)
   WRITE(6,10)A,B,C,D,E,F,G
```

VARIABLE	FIELD DESCRIPTOR	INTERPRETED AS	INTERNAL VALUE	OUTPUT
A	G10.3	E10.3	+1234	bb.123E+04
B	G10.3	E10.3	−1234	b−.123E+04
C	G12.4	E12.4	+12345	bbb.1235E+05
D	G12.4	F8.0,4X	+9999	bbb9999.bbbb
E	G12.4	F8.1,4X	−999	bb−999.0bbbb
F	G7.1	E7.1	+.09	b.9E−01
G	G5.1	E5.1	−.09	#####

When the E*w.d* descriptor is used, if rounding of the least significant digit occurs and rollover results, the rollover value is normalized and the exponent is adjusted.

For example,

FIELD DESCRIPTOR	INTERNAL VALUE	OUTPUT
G12.1 (E12.2)	+9999	bbbbb.10E+05
G8.2 (E8.2)	+999	b.10E+04
G7.2 (E7.2)	−999	#######

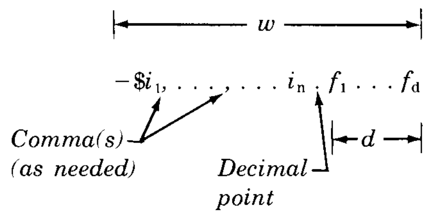
On input, the G*w.d* field descriptor causes interpretation of the next *w* positions in an ASCII input record. The number is converted to an internal representation value for the variable (list element) currently using the field descriptor.

See table 7-6 for examples of G*w.d* input.

7-13. MONETARY FORM (M*w.d*). The M*w.d* field descriptor defines a field for a real number without an exponent (fixed-point) written in monetary form.

On output, the M*w.d* field descriptor causes output of a variable (internal representation value: integer, double integer, real or double precision) in ASCII character fixed-point form right-justified, with a dollar sign ($) and commas. The least significant digit of the output is rounded.

The external field is *w* positions:



where

$i_1 \ldots . i_n$	= the integer digits (without commas)
$f_1 \ldots . f_d$	= the fraction digits
commas = *c*	= the number of output commas needed: one to the left of every third digit left of the decimal point; see general rule for *w* below.
d	= the number of fractional digits allowed in *w*
n	= the number of integer digits
w	= the width of the external field

– (minus) is present if the value is negative.

The field width *w* must follow the general rule:

$w \geq d + n + c + 4$

to provide positions for the sign, $, *n* digits, *c* commas, the decimal point, *d* digits, and a rollover digit if needed. If *w* is greater than the number of positions required for the output value, the output is right-justified in the field with blank spaces to the left. If *w* is less than the number of positions required for the output value (with the plus or minus sign, $ sign, comma(s), and the decimal point), the entire field is filled with #'s.

See table 7-7 for M*w.d* output.

Table 7-6. G*w.d* Input

```
10 FORMAT(G10.3,G10.3,G12.4,G12.4,G12.4,G7.1,G5.1)
   READ(5,10)A,B,C,D,E,F
```

INPUT CHARACTERS	FIELD DESCRIPTOR	INTERPRETED AS	VARIABLE	INTERPRETED AS
bb.123E+04	G10.3	E10.3	A	$.123 \times 10^4$
b–.123E+04	G10.3	E10.3	B	$-.123 \times 10^4$
bbb.1235E+05	G12.4	E12.4	C	$.1235 \times 10^5$
bbbbbb+9999.	G12.4	F8.0	D	9999.
bbbbbbbb–999	G12.4	F8.1	E	99.9
bbb+.09	G7.1	E7.1	F	$.900000 \times 10^{-1}$
b–.09	G5.1	E5.1	G	$-.900000 \times 10^{-1}$

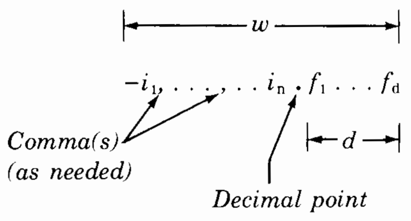
On input, the M*w*.*d* field descriptor causes interpretation of the next *w* positions in an ASCII input record. The field width is expected (but not required) to have a $ and comma(s) embedded in the data as described for M*w*.*d* outputs (the $ and commas are ignored). The number is converted to an internal representation value for the variable (list element) currently using the field descriptor.

See table 7-8 for M*w*.*d* input.

7-14. NUMERATION FORM (N*w*.*d*). The N*w*.*d* field descriptor defines a field for a real number without exponent (fixed-point) written in numeration form (same as M*w*.*d* but without $ on output).

On output, the N*w*.*d* field descriptor causes output of a variable (internal representation value: integer, double integer, real or double precision) in ASCII character fixed-point form, right-justified, with commas. The least significant digit is rounded.

The external field is *w* positions:



where

$i_1 \ldots i_n$	= the integer digits (without commas)
$f_1 \ldots f_d$	= the fraction digits
commas = *c*	= the number of output commas needed: one to the left of every third digit left of the decimal point; see general rule for *w* below.
d	= the number of fractional digits allowed in *w*
n	= the number of integer digits

Table 7-7. M*w*.*d* Output

```
10 FORMAT(M10.3,M10.3,M13.3,M12.2,M12.2)
   WRITE(6,10)A,B,C,D,E
```

VARIABLE	FIELD DESCRIPTOR	INTERNAL VALUE	OUTPUT
A	M10.3	+12.3402	bbb$12.340
B	M10.3	−12.3404	bb−$12.340
C	M13.3	+80175.3965	bb$80,175.397
D	M12.2	−80175.396	b−$80,175.40
E	M12.2	+28705352.563	############

If rounding of the least significant occurs and rollover results, the stated formula for *w* provides enough positions.

For example,

FIELD DESCRIPTOR	INTERNAL VALUE	OUTPUT
M12.2	+99999.996	b$100,000.00
M12.2	−99999.998	−$100,000.00
M11.2	−99999.995	###########

w	= the width of the external field

– (minus) is present if the value is negative.

The field width w must follow the general rule:

$w \geq d + n + c + 3$

to provide positions for the sign, n digits, c commas, the decimal point, d digits, and a rollover digit if needed. If w is greater than the number of positions required for the output value, the output is right-justified in the field with blank spaces to the left. If w is less than the number of positions required for the output value (with the sign, comma(s), and the decimal point), the entire field is filled with #'s.

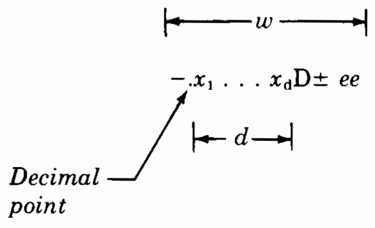
See table 7-9 for N*w*.*d* output.

On input, the N*w*.*d* field descriptor causes interpretation of the next w positions in an ASCII input record as a real number without exponent (fixed-point). The field width is expected (but not required) to have comma(s) embedded in the data as described for N*w*.*d* outputs (the commas are ignored). The number is converted to an internal representation value for the variable (list element) currently using the field descriptor.

See table 7-10 for N*w*.*d* input.

7-15. DOUBLE PRECISION REAL NUMBERS (D*w*.*d*). The D*w*.*d* field descriptor defines a field for a double precision number with an exponent (floating-point). On output, the D*w*.*d* field descriptor causes normalized output of a variable (internal representation value: integer, real or double precision) in ASCII character floating-point form, right-justified. The least significant digit of the output is rounded.

The external field is w positions:



where

$x_1 \ldots x_d$	= the most significant digits of the value
ee	= the digits of the exponent value
w	= the width of the external value
d	= the number of significant digits allowed in w

– (minus) is present if the value is negative.

The field width w must follow the general rule:

$w \geq d + 6$

to provide positions for the sign of the value, the decimal point, d digits, the letter D, the sign of the exponent, and the exponent's two digits. If w is greater than the number of positions required for the output value, the output is right-justified in the field with blank spaces to the left. If w is less than the number of positions required for the value (with the sign, decimal point, and exponent field), the entire field is filled with #'s.

See table 7-11 for D*w*.*d* output.

Table 7-8. M*w*.*d* Input

```
10 FORMAT(M10.3,M10.3,M13.3,M12.2,M12.2)
   READ(5,10)A,B,C,D,E
```

INPUT CHARACTERS	FIELD DESCRIPTOR	VARIABLE	INTERPRETED AS
bbb$12.340	M10.3	A	12.340
bb$12.3402	M10.3	B	12.3402
bbbb80175.397	M13.3	C	80175.397
–$80,175.397	M12.2	D	80175.397
bbb99999.996	M12.2	E	99999.996

Table 7-9. N*w*.*d* Output

```
10 FORMAT(N9.3,N9.3,N12.3,N11.2,N11.2)
   WRITE(6,10)A,B,C,D,E
```

VARIABLE	FIELD DESCRIPTOR	INTERNAL VALUE	OUTPUT
A	N9.3	+12.3402	bbb12.340
B	N9.3	−12.3404	bb−12.340
C	N12.3	+80175.3965	bb80,175.397
D	N11.2	−80175.396	b−80,175.40
E	N11.2	+28705352.563	###########

If rounding of the least significant occurs and rollover results, the stated formula for *w* provides enough positions.

For example,

FIELD DESCRIPTOR	INTERNAL VALUE	OUTPUT
N11.2	+99999.995	b100,000.00
N11.2	−99999.997	−100,000.00
N10.2	−99999.999	##########

Table 7-10. N*w*.*d* Input

```
10 FORMAT(N9.3,N9.3,N12.3,N11.2,N11.2)
   READ(5,10)A,B,C,D,E
```

INPUT CHARACTERS	FIELD DESCRIPTOR	VARIABLE	INTERPRETED AS
bb12.3402	N9.3	A	12.3402
b−12.3404	N9.3	B	−12.3404
b+80,175.396	N12.3	C	80175.396
bb−80175.39	N11.2	D	−80175.39
bb99999.996	N11.2	E	99999.996

Table 7-11. D*w.d* Output

```
10 FORMAT(D10.3,D10.3,D12.4,D12.4,D7.3,D5.1)
   WRITE(6,10)A.B.C.D.E.F
```

VARIABLE	FIELD DESCRIPTOR	INTERNAL VALUE	OUTPUT
A	D10.3	+12.342	bb.123D+02
B	D10.3	−12.341	b−.123D+02
C	D12.4	+12.340	bbb.1234D+02
D	D12.4	−12.345	bb−.1234D+02
E	D7.3	+12.343	#######
F	D5.1	+12.344	#####

If rounding of the least significant digit occurs and rollover results, the rollover value is normalized and the exponent is adjusted.

For example,

FIELD DESCRIPTOR	INTERNAL VALUE	OUTPUT
D11.5	−999.997	−.10000D+04
D11.5	+999.996	b.10000D+04
D10.5	−99.9995	##########

On input, the D*w.d* field descriptor causes interpretation of the next *w* positions in an ASCII input record. The number is converted to an internal representation value for the variable (list element) currently using the field descriptor.

See table 7-12 for D*w.d* input.

7-16. INTEGER NUMBERS (I*w*). The I*w* field de-criptor defines a field for an integer or double integer number.

On output, the I*w* field descriptor causes output of a variable (internal representation value: integer, double integer, real or double precision) in ASCII character integer form, right-justified. If the internal representation is real or double precision, the least significant digit of the output is rounded.

The external field is *w* positions:

$$|\longleftarrow w \longrightarrow|$$

$$-i_1 \ldots\ldots i_n$$

where

$i_1 \ldots\ldots i_n$	=	the integer digits
n	=	the number of significant digits
w	=	the width of the external field

− (minus) is present if the value is negative.

The field width *w* must follow the general rule:

$$w \geq n + 2$$

to provide positions for the sign, *n* digits, and a rollover digit if needed. If *w* is greater than the number of positions required for the output value, the output is right-justified in the field with blank spaces to the left. If *w* is less than the number of positions required for the output (all digits of the integer, and, when needed, the sign), the entire field is filled with #'s.

See table 7-13 for I*w* output.

Table 7-12. D*w.d* Input

```
10 FORMAT(D10.3,D10.3,D12.4,D12.4,D5.1)
   READ(5,10)A,B,C,D,E
```

INPUT CHARACTERS	FIELD DESCRIPTOR	VARIABLE	INTERPRETED AS
bb.123D+03	D10.3	A	123
b−.123D+02	D10.3	B	−12.3
bb.12345D+02	D12.4	C	12.345
bb−.1235D+02	D12.4	D	−12.35
+.123D+02	D5.1	E	12

Table 7-13. I*w* Output

```
10 FORMAT(I5,I5,I5,I5,I4,I4,I6)
   WRITE(6,10)I,J,K,L,M,N,ITEM
```

VARIABLE	FIELD DESCRIPTOR	INTERNAL VALUE	OUTPUT
I	I5	−123	b−123
J	I5	+123	bb123
K	I5	+12345	12345
L	I5	−12345	#####
M	I4	+12.4	bb12
N	I4	−12.7	b−13
ITEM	I6	−.3765E+03	bb−377

If rounding of the least significant digit occurs and rollover results, the stated formula for *w* provides enough positions.

For example,

FIELD DESCRIPTOR	INTERNAL VALUE	OUTPUT
I5	−999.8	−1000
I5	+999.6	b1000
I4	−999.5	####

On input, the I*w* field descriptor functions as an F*w.d* descriptor with $d = 0$; it causes interpretation of the next *w* positions in the ASCII input record. The number is converted to an internal representation value for the variable (list element) currently using the field descriptor.

See table 7-14 for I*w* input.

7-17. OCTAL INTEGER NUMBERS (O*w*). The O*w* field descriptor defines a field for an octal integer number.

On output, the O*w* field descriptor causes output of a variable (internal representation value: integer, double integer, real, or double precision) in ASCII character octal integer form, right-justified.

The external field is *w* positions:

$$\left|\longleftarrow w \longrightarrow\right|$$

$$i_1 \ldots\ldots i_n$$

where

$i_1 \ldots i_n$ = the octal integer digits

n = the number of significant octal digits (maximums: 6 for an integer variable, 11 for a real variable or double integer, or 22 for a double precision variable)

w = the width of the external field

The field width *w* can be any desired value but should be equal to or greater than 6 for an integer variable, equal to or greater than 11 for a real variable or double integer, and equal to or greater than 22 for a double precision variable for complete accuracy. If *w* is greater than the number of positions required for the output value, the output is right-justified in the field with blank spaces to the left. If *w* is less than the number of positions required for the entire octal integer, only the *w* least significant digits are output.

See table 7-15 for O*w* output.

Table 7-14. I*w* Input

```
10 FORMAT (I5,I5,I5,I5,I4,I4,I6)
 READ(5,10)I,J,K,L,M,N,ITEM
```

INPUT CHARACTERS	FIELD DESCRIPTOR	VARIABLE	INTERPRETED AS
b−123	I5	I	−123
bb123	I5	J	123
12345	I5	K	12345
−12345	I5	L	−1234
12.4	I4	M	12
−12.7	I4	N	−13
.3765E+03	I6	ITEM	377

Table 7-15. O*w* Output

```
10 FORMAT(O8,O4,O16,O11)   WRITE(6,10)I,J,A,B
```

VARIABLE	FIELD DESCRIPTOR	INTERNAL VALUE	OUTPUT
I	O8	%102077	bb102077
J	O4	%30554677321	7321
A	O16	%56774532673	bbbbb56774532673
B	O11	%0000003435645327422113	45327422113

On input, the O field descriptor causes interpretation of the next *w* positions in the ASCII input record as an octal integer number. The number is converted to an internal representation value for the variable (list element) currently using the field descriptor.

The input field can consist of octal digits only. No more than six digits (no larger than 177777_8) are interpreted for an integer variable; no more than 11 digits (no larger than 37777777777_8) are interpreted for a double integer variable; no more than 11 digits (no larger than 37777777777_8) are interpreted for a real variable; and no more than 22 digits (no larger than 1777777777777777777777_8) are interpreted for a double precision variable. Any non-octal or non-numeric character (including a blank) anywhere in the field will produce a conversion error. If *w* is less than the maximum number allowed by the variable using the descriptor, *w* digits are right-justified in that variable's internal representation (one, two, or four words of memory).

See table 7-16 for O*w* input.

7-18. HEXADECIMAL INTEGER NUMBERS (Z*w*).

The Z*w* field descriptor defines a field for a hexadecimal integer number.

On output, the Z*w* field descriptor causes output of a variable (internal representation value: integer, double integer, real, or double precision) in ASCII character hexadecimal integer form, right-justified.

The external field is *w* positions:

$$|\!\leftarrow w \rightarrow\!|$$

$$i_1 \ldots i_n$$

where

$i_1 \ldots i_n$	= the hexadecimal integer digits
n	= the number of significant hexadecimal digits (maximums: 4 for an integer variable, 8 for a double integer variable, 8 for a real variable, or 16 for a double precision variable)
w	= the width of the external field

The field width *w* can be any desired value but should be equal to or greater than 4 for an integer variable, equal to or greater than 8 for a double integer, equal to or greater than 8 for a real variable, and equal to or greater than 16 for a double precision variable for complete accuracy. If *w* is greater than the number of positions required for the output value, the output is right-justified in the field with blank spaces to the left. If *w* is less than the number of positions required for the entire hexadecimal integer, only the *w* least significant digits are output.

See table 7-17 for Z*w* output.

Table 7-16. O*w* Input

```
10 FORMAT(3O6,3O9,3O13)
   READ(I,A,B,J,C,D,K,E,F)
```

INPUT CHARACTERS	FIELD DESCRIPTOR	VARIABLE TYPE	VARIABLE	INTERPRETED AS
134577	O6	Integer or logical	I	%134577
134577	O6	Real	A	%00000134577
134577	O6	Double Precision	B	%0000000000000000134577
545563274	O9	Integer or logical	J	%563274
545563274	O9	Real	C	%00545563274
545563274	O9	Double Precision	D	%0000000000000545563274
4367436521051	O13	Integer or logical	K	%521051
4367436521051	O13	Real	E	%67436521051
4367436521051	O13	Double Precision	F	%0000000004367436521051

On input, the Z*w* field descriptor causes interpretation of the next *w* positions in the ASCII input record as a hexadecimal integer number. The number is converted to an internal representation value for the variable (list element) currently using the field descriptor.

The input field can consist of hexadecimal digits only. No more than four digits (no larger than $FFFF_{16}$) are interpreted for an integer variable; no more than eight digits (no larger than $FFFFFFFF_{10}$) are interpreted for a double integer variable; no more than eight digits (no larger than $FFFFFFFF_{16}$) are interpreted for a real variable; and no more than 16 digits (no larger than $FFFFFFFFFFFFFFFF_{16}$) are interpreted for a double precision variable. Any non-hexadecimal character (including a blank except leading blanks, which are ignored) anywhere in the field will produce a conversion error. If *w* is less than the maximum number allowed by the variable using the descriptor, *w* digits are right-justified in that variable's internal representation (one, two, or four words of memory).
See table 7-18 for Z*w* input.

7-19. LOGICAL (BOOLEAN) VALUES (L*w*). The L*w* field descriptor defines a field for a logical value.

On output, the L*w* field descriptor causes output of a variable (internal representation value: integer or logical (boolean)) in ASCII character logical value form (T or F).

The external field is *w* positions:

$$|\leftarrow w \rightarrow|$$

$$X_1 \ldots X_n c$$

where

$X_1 \ldots . X_n$	= $w - 1$ blanks
c	= either of two logical characters: T(true) or F(false)
n	= the number of blank spaces to the left of c
w	= the width of the external field.

The field width *w* can be any value $\geqslant 1$.

Table 7-17. Z*w* Output

```
10 FORMAT(Z6,Z4,Z16,Z8)
   WRITE(I,J,A,B)
```

VARIABLE	FIELD DESCRIPTOR	INTERNAL VALUE	OUTPUT
I	Z6	5AFC	bb5AFC
J	Z4	FCD473BE	73BE
A	Z16	32AB698A	bbbbbbbb32AB698A
B	Z8	9BE84893E6FF	4893E6FF

Table 7-18. Z*w* Input

```
10 FORMAT(3Z4,3Z6,3Z10)
   READ(I,A,B,J,C,D,K,E,F)
```

INPUT CHARACTERS	FIELD DESCRIPTOR	VARIABLE TYPE	VARIABLE	INTERPRETED AS
1AD6	Z4	Integer or logical	I	1AD6
1AD6	Z4	Real	A	000000001AD6
1AD6	Z4	Double precision	B	000000001AD6
AB12F6	Z6	Integer or logical	J	12F6
AB12F6	Z6	Real	C	00AB12F6
AB12F6	Z6	Double Precision	D	0000000000AB12F6
5489BB3A6C	Z10	Integer or logical	K	3A6C
5489BB3A6C	Z10	Real	E	89BB3A6C
5489BB3A6C	Z10	Double Precision	F	0000005489BB3A6C

On input, the Lw field descriptor causes a scan of the next w positions in an ASCII input record to find a logical character (T or F). All positions to the left of the logical character must be blank; any other character(s) can follow the logical character. The character T is converted to -1 (177777_8), F is converted to 0 (000000_8).

See table 7-20 for Lw input.

The logical character c is T if the least significant bit of the internal representation is 1; c is F if that bit is 0.

See table 7-19 for Lw output.

7-20. LEFTMOST ASCII CHARACTERS FIELD DESCRIPTOR (Aw). The Aw field descriptor defines a field for ASCII alphanumeric characters.

On output, the Aw field descriptor causes output of w bytes of a variable in ASCII character alphanumeric form. The maximum number (n) of bytes (thus, the maximum number of characters available to a single Aw descriptor) depends on the type of the variable: for logical or integer, $n = 2$; for double integer, $n = 4$; for real, $n = 4$; for double precision, $n = 8$; for character, $n =$ the length attribute (as defined in a Type statement such as: CHARACTER*8 CHAR) of the character variable. The length attribute can be any integer in the range of from 1 to 255.

The external field is w positions, left-justified:

$$|\longleftarrow w \longrightarrow|$$
$$s_1 \ldots . s_r\, c_1 \ldots . c_n$$

where

$c_1 \ldots . c_n$	= the alphanumeric characters
n	= the number of characters
w	= the width of the external field
r	= any remaining positions not used by n ($r = w - n$)
$s_1 \ldots . s_r$	= blank spaces

Table 7-19. Lw Output

```
      LOGICAL A,B,C
   10 FORMAT(L1,L13,L5)
      WRITE(6,10)A,B,C
```

VARIABLE	FIELD DESCRIPTOR	INTERNAL VALUE	OUTPUT
A	L1	102033_8	T
B	L13	32767(77777_8)	bbbbbbbbbbbbT
C	L5	+124(174_8)	bbbbF

Table 7-20. Lw Input

```
      LOGICAL A,B,C,D,E
   10 FORMAT(L8,L1,L6,L2,L1)
      READ(5,10)A,B,C,D,E
```

VALUE	FIELD DESCRIPTOR	VARIABLE	INTERPRETED AS
bbbbTRUE	L8	A	177777_8
F	L1	B	000000_8
bFALSE	L6	C	000000_8
T5	L2	D	177777_8
5	L1	E	INVALID

The field width *w* can be any value ⩾ 1. If *w* is greater than *n*, the output is right-justified in the field with *w* − *n* blanks to the left. If *w* is less than *n*, the leftmost *w* bytes of the variable are output. The *n* − *w* remaining bytes are ignored.

See table 7-21 for A*w* output.

On input, the A*w* field descriptor causes transmittal of *w* positions in an ASCII input record to *n* bytes of a variable (list element) currently using the field descriptor. If *w* is greater than *n*, the first *w* − *n* characters of input are skipped, and *n* characters are transferred. If *w* is less than *n*, *w* characters are transferred to the leftmost bytes of the variable, and all remaining *n* − *w* bytes are set to blank. See table 7-22 for A*w* input.

7-21. RIGHTMOST ASCII CHARACTERS FIELD DESCRIPTOR (R*w*). The R*w* field descriptor defines a field for ASCII alphanumeric characters.

On output, the R*w* field descriptor causes output of *w* bytes of a variable in ASCII character alphanumeric form. The maximum number (*n*) of bytes (thus, the maximum number of characters) available to a single R*w* field descriptor depends on the type of the variable: for logical or integer, *n* = 2; for double integer, *n* = 4; for real, *n* = 4; for double precision, *n* = 8; for character, *n* = the length attribute (as defined in a Type statement such as CHARACTER *8 CHAR) of the character variable. The length attribute can be any integer in the range of from 1 to 255.

Table 7-21. A*w* Output

```
10 FORMAT(A3,A3,A7,A10,A4,A12,A6)
WRITE(6,10)I,A,B,J,C,CHAR1,CHAR2
```

VARIABLE	FIELD DESCRIPTOR	INTERNAL CHARACTERS	VARIABLE TYPE (*n* =)	OUTPUT
I	A3	AB	Logical or Integer (2)	bAB
A	A3	ABCD	Real (4)	ABC
B	A7	ABCDEF	Double Precision (6)	bABCDEF
J	A10	AB	Logical or Integer (2)	bbbbbbbbAB
C	A4	ABCDEF	Double Precision (8)	ABCD
CHAR1	A12	LEFTMOST	Character (8)	bbbbLEFTMOST
CHAR2	A6	LEFTMOST	Character (8)	LEFTMO

Table 7-22. A*w* Input

```
10 FORMAT(A3,A2,A10,A4,A4,A7)
READ(5,10)I,J,K,A,B,CHAR
```

FIELD DESCRIPTOR	INPUT CHARACTERS	VARIABLE	VARIABLE TYPE (*n* =)	INTERPRETED AS
A3	ABC	I	Integer or Logical (2)	BC
A2	AB	J	Integer or Logical (2)	AB
A10	COMPLEMENT	K	Integer or Logical (2)	NT
A4	REAL	A	Double Precision (8)	REALbbbb
A4	REAL	B	Real (4)	REAL
A7	PROGRAM	CHAR	Character (8)	PROGRAMb

The external field is w positions:

$$\xleftrightarrow{\quad w \quad}$$

$$s_1 \ldots\ldots s_r\, c_1 \ldots\ldots c_n$$

where

$c_1 \ldots\ldots c_n$	= the alphanumeric characters
n	= the number of characters
w	= the width of the external field
r	= any remaining positions not used by n ($r = w - n$)
$s_1 \ldots\ldots s_r$	= blank spaces (when needed)

The field width w can be any value ≥ 1. If w is greater than n, the output is right-justified in the field with $w - n$ blanks to the left. If w is less than n, the rightmost bytes of the variable are output. The $n - w$ remaining bytes are ignored.

See table 7-23 for Rw output.

On input, the Rw field descriptor causes transmittal of w positions in an ASCII input record to n bytes of the variable currently using the field descriptor. If w is greater than n, the first (leftmost) $w - n$ characters of input are skipped, and n characters are transferred. If w is less than n, w characters are transferred to the rightmost bytes of the variable, and all bits of the remaining $n - w$ bytes are set to 0 (ASCII Null).

See table 7-24 for Rw input.

Table 7-23. Rw Output

```
10 FORMAT(R3,R3,R7,R10,R4,R12,R6)
WRITE(6,10)I,A,B,J,C,CHAR1,CHAR2
```

VARIABLE	FIELD DESCRIPTOR	INTERNAL CHARACTERS	VARIABLE TYPE (*n* =)	OUTPUT
I	R3	AB	Logical or Integer (2)	bAB
A	R3	ABCD	Real (4)	BCD
B	R7	ABCDEF	Double Precision (6)	bABCDEF
J	R10	AB	Logical or Integer (2)	bbbbbbbbAB
C	R4	ABCDEF	Double Precision (8)	CDEF
CHAR1	R12	RIGHTMOST	Character (9)	bbbRIGHTMOST
CHAR2	R6	RIGHTMOST	Character (9)	HTMOST

Table 7-24. Rw Input

```
10 FORMAT(R3,R2,R10,R4,R4,R7)
READ(5,10)I,J,K,A,B,CHAR
```

FIELD DESCRIPTOR	INPUT CHARACTERS	VARIABLE	VARIABLE TYPE (*n* =)	INTERPRETED AS
R3	CAB	I	Integer or Logical (2)	AB
R2	CA	J	Integer or Logical (2)	CA
R10	COMPLEMENT	K	Integer or Logical (2)	NT
R4	REAL	A	Double Precision (8)	*aaaa*REAL
R4	REAL	B	Real (4)	REAL
R7	PROGRAM	CHAR	Character (8)	*a*PROGRAM

NOTE: *a* = ASCII null

7-22. STRINGS OF ASCII CHARACTERS FIELD DESCRIPTOR (S). The S field descriptor defines a field for a string of ASCII alphanumeric characters.

On output, the S field descriptor causes output of a variable (internal representation value: character only) in ASCII character alphanumeric form. If the variable (list element) is not type character, the error message FMT: STRING MISMATCH occurs.

The external field is *l* positions:

$$|\longleftarrow l \longrightarrow|$$

$$c_1 \ldots\ldots\ldots c_n$$

where

$c_1 \ldots\ldots .c_n$	= the alphanumeric characters
n	= the number of characters
l	= the length attribute of the character variable; thus, the width of the external field.

See table 7-25 for S output.

On input, the S field descriptor causes transmittal of *l* positions in an ASCII input record to the character variable currently using the field descriptor.

See table 7-26 for S input.

Table 7-26. S Input

```
    CHARACTER*8 DAY
 10 FORMAT(S)
    READ(5,10)DAY
```

INPUT CHARACTERS	INTERPRETED AS
MONDAY	MONDAYbb
SATURDAY	SATURDAY

7-23. SCALE FACTOR

The scale factor is a format specification which modifies the normalized *output* of the D*w.d,* E*w.d* and the G*w.d*–selected E*w.d* field descriptors and the fixed-point *output* of the F*w.d,* M*w.d* and N*w.d* field descriptors. The scale factor also modifies the fixed-point and integer (no exponent field) *inputs* to the D*w.d,* E*w.d,* F*w.d,* G*w.d,* M*w.d* and N*w.d* field descriptors. This scale factor has no effect on output of the G*w.d*-selected F*w.d* field descriptor or floating-point (with exponent field) inputs.

A scale factor is written in one of two forms:

nPf

or

nPrf

where

n	= an integer constant or – (minus) followed by an integer constant which is the scale value
P	= the scale factor identifier
f	= the field descriptor
r	= a repeat specification for a field descriptor (see paragraph 7-24).

Table 7-25. S Output

```
    CHARACTER *3 NAME1, NAME2*6
 10 FORMAT(" MY NAME IS ",S," JONES")
    WRITE(6,10)NAME1
    WRITE(6,10)NAME2
```

VARIABLE	INTERNAL CHARACTERS	OUTPUT
NAME1	JIM	MY NAME IS JIM JONES
NAME2	GEORGE	MY NAME IS GEORGE JONES

When a FORMAT statement is interpreted, the scale factor is set to zero. Each time a scale factor specification is encountered in that FORMAT statement, a new value is set. This scale value remains in effect for all subsequent affected field descriptors or until use of that FORMAT statement ends.

EXAMPLES

FORMAT SPECIFICATIONS	**COMMENTS**
(E10.3,F12.4,I9)	No scale factor change, previous value remains in effect.
(E10.3,2PF12.4,I9)	Scale factor for E10.3 unchanged from previous value, changes to 2 for F12.4, has no effect on I9.

If the FORMAT statement includes one or more nested groups (see paragraph 7-36, *Nesting*), the last scale factor value encountered remains in effect.

See table 7-27 for nested groups.

Table 7-27. Scale Factor for Nested Groups

```
10 FORMAT(G9.2,2PF9.4,E7.1,2(D10.2,-1PG8.1))
```

FIELD DESCRIPTOR	SCALE VALUE
G9.2	Unchanged from previous value
F9.4	2
E7.1	2
D10.2	2
G8.1	−1
D10.2	−1
G8.1	−1

On output, the scale factor affects D*w.d,* E*w.d,,* F*w.d,* M*w.d,* N*w.d* and G*w.d*-selected E*w.d* field descriptors only.

For D*w.d* and E*w.d* field descriptors, the scale factor has the effect of shifting the output number left *n* places while reducing the exponent by *n* (the *value* of the printed number remains the same).

- If $n \leq 0$, the output fraction field has $-n$ leading zeros, followed by $d + n$ significant digits. The least significant digit is rounded.
- If $n > 0$, the output has *n* significant digits in the integer field, and $(d - n)$ digits in the fraction field. The least significant digit is rounded.
- The field width specification *w* normally required may have to be increased by 1.

See table 7-28 and figure 7-4 for E*w.d* scale factors.

Table 7-28. Scale Factor for E*w.d* Output

SCALE FACTOR AND FIELD DESCRIPTOR	INTERNAL VALUE	OUTPUT
E12.4	+12.345678	bbb.1235E+02
3PE12.4	+12.345678	bb123.46E−01
−3PE12.4	+12.345678	bbb.0001E+05

For F*w.d,* M*w.d* and N*w.d,* the internal value is multiplied by 10^n, then output in the normal manner.

See table 7-29 and figure 7-4 for F*w.d* and M*w.d* output.

For G*w.d*-selected E*w.d* field descriptors, the effect is exactly the same as described for E*w.d* scale factors. (See table 7-28.)

The scale factor has no effect for the G*w.d*-selected F*w.d* field descriptor.

Table 7-29. Scale Factors for F*w.d* and M*w.d* Output

SCALE FACTOR AND FIELD DESCRIPTOR	INTERNAL VALUE	OUTPUT
F11.3	1234.500	bbb1234.500
−2PF11.3	1234.500678	bbbbb12.345
2PF11.3	1234.500678	b123450.068
1PM11.3	1234.500678	$12,345.007

```
:FORTGO FTRAN24

PAGE 0001   HP32102B.00.0


                  PROGRAM SCALE FACTOR
            C
            C SCALE FACTOR EXAMPLE
            C
             100  FORMAT(T5,E12.4)
             200  FORMAT(T5,3PE12.4)
             300  FORMAT(T5,-3PE12.4)
             400  FORMAT(T5,F11.3)
             500  FORMAT(T5,-2PF11.3)
             600  FORMAT(T5,2PF11.3)
             700  FORMAT(T5,1PM11.3)
                  A=12.345678
                  B=12.345678
                  C=12.345678
                  D=1234.500
                  E=1234.500678
                  F=1234.500678
                  G=1234.500678
                  WRITE(6,100)A
                  WRITE(6,200)B
                  WRITE(6,300)C
                  WRITE(6,400)D
                  WRITE(6,500)E
                  WRITE(6,600)F
                  WRITE(6,700)G
                  STOP
                  END



****       GLOBAL STATISTICS       ****
****    NO ERRORS,    NO WARNINGS   ****
TOTAL COMPILATION TIME   0:00:01
TOTAL ELAPSED TIME       0:00:04

  END OF COMPILE

  END OF PREPARE


         .1235E+02
       123.46E-01
         .0001E+05
        1234.500
           12.345
      123450.073
     $12,345.007
  END OF PROGRAM
```

Figure 7-4. Scale Factor Examples

On input, the scale factor effect is the same for integer or fixed-field (no exponent field) inputs to the D*w.d,* E*w.d,* F*w.d,* G*w.d,* M*w.d* and N*w.d* field descriptors. The external value is multiplied by 10^{-n}, then converted in the usual manner.

If the input includes an exponent field, the scale factor has no effect. See table 7-30 for the scale factor effect on input.

Table 7-30. Scale Factor for Input

SCALE FACTOR AND FIELD DESCRIPTOR	INPUT VALUE	INTERPRETED AS
E10.4	bb123.9678	.1239678E+03
2PD10.4	bb123.9678	.1239678E+01
−2PG11.5	bb123.96785	.12396785E05
−2PE13.5	1239.6785E+02	.12396785E+06

7-24. REPEAT SPECIFICATION FOR FIELD DESCRIPTORS

The repeat specification is a positive integer written to the left of the field descriptor it controls. If a scale factor is needed also, it is written to the left of the repeat specification.

The repeat specification allows one field descriptor to be used for several list elements. It can also be used for nested groups of format specifications.

EXAMPLES:

```
(4E12.4) = (E12.4,E12.4,E12.4,E12.4)

(−2P3D8.2,2I6) = (−2PD8.2,D8.2,D8.2,I6,I6)

(E8.2,3F7.1,3(I6,D12.3) = (E8.2,F7.1,F7.1,I6,D12.3,
                           I6,D12.3,I6,D12.3)

(2(M8.2)) = (M8.2,M8.2)
```

7-25. EDIT SPECIFICATIONS

Edit specifications are written as an edit descriptor or a repeat specification followed by an edit descriptor.

Note: The repeat specification cannot be used directly on the *n*H or *n*X edit descriptors. (See paragraph 7-34.)

7-26. EDIT DESCRIPTORS

There are six edit descriptors, as follows:

DESCRIPTOR	FUNCTION
". . ."	Fix the next *n* characters of an edit specification.
'. . .'	Fix the next *n* characters of an edit specification.
*n*H	Initialize the next *n* characters of an edit specification.
*n*X	Skip *n* positions of the external record on input, fill with *n* blanks on output.
T*n*	Select the position in an external record where data input/output is to begin or resume.
/	Signal the end of a current record and the beginning of a new record.
%*n*C	Put a one-byte character in the output buffer.

7-27. FIXED ASCII STRING EDIT DESCRIPTOR (". . ."). The ". . ." edit descriptor fixes *n* characters in the edit specification, where *n* is the number of ASCII characters enclosed in the quotation marks.

The form of the ". . ." edit descriptor is:

"THIS IS A FIXED STRING"

If a quotation mark is to be used as one of the characters, this must be signaled by an adjacent quotation mark.

For example,

```
10 FORMAT(2X, "OUTPUT ""LOAD""")
   WRITE(6,10)
```

would cause OUTPUT "LOAD" to be printed.

Any other ASCII characters, including apostrophe ('), can be used without restriction.

On output, the ". . ." edit descriptor causes *n* characters to be transmitted to the external record. Any adjacent pair of quotation marks is transmitted as one quotation mark. A blank is counted as one character.

See table 7-31 for ". . ." output.

Table 7-31. “. . .” Edit Descriptor Output

```
10 FORMAT(2X, “OUTPUT ““LOAD”””)
20 FORMAT(2X, “USER’S PROGRAM”)
   WRITE(6,10)
   WRITE(6,20)
```

EDIT DESCRIPTOR	OUTPUT
“OUTPUT ““LOAD”””	OUTPUT “LOAD”
“USER’S PROGRAM”	USER’S PROGRAM

On input, the “. . .” edit descriptor causes *n* positions of the input record to be skipped. Each pair of adjacent quotation marks counts as one position and a blank counts as one position.

See table 7-32 for “. . .” input.

7-28. ALTERNATE FIXED ASCII STRING EDIT DESCRIPTOR (‘. . .’). An alternate method of fixing *n* number of ASCII characters is through the use of the ‘. . .’ edit descriptor. The ‘. . .’ edit descriptor functions the same as the “. . .” edit descriptor (see paragraph 7-27) except that *n* is the number of ASCII characters enclosed in *apostrophes* (’) instead of *quotation marks* (”). Any one or more of the characters can be an apostrophe if signaled by an adjacent apostrophe. Any other ASCII characters, including quotation marks, can be used without restriction.

On output, the ‘. . .’ edit descriptor causes *n* characters to be transmitted to the external record. Any adjacent pair of apostrophes is transmitted as one apostrophe, and counts as one character. A blank is counted as one character.

See table 7-33 for ‘. . .’ edit descriptor output.

Table 7-33. ‘. . .’ Edit Descriptor Output

```
10 FORMAT(2X, ‘PRINT “DATA”’)
20 FORMAT(2X, ‘SAM’’S “SCORE”’)
   WRITE(6,10)                  
   WRITE(6,20)
```

two apostrophes

EDIT DESCRIPTOR	OUTPUT
‘PRINT “DATA”’	PRINT ‘DATA’
‘SAM’’S “SCORE”’	SAM’S “SCORE”

On input, the ‘. . .’ edit descriptor causes *n* positions of the input record to be skipped. Each pair of adjacent apostrophes counts as one position and blanks count as one position.

See table 7-34 for ‘. . .’ edit descriptor input.

Table 7-32. “. . .” Edit Descriptor Input

```
   CHARACTER* 17 SKIP1,SKIP2*18
10 FORMAT(2X, “HEADING HERE”, S)
20 FORMAT(2X, “ENDINGS HERE”, S)
   READ(5,10)SKIP1
   READ(5,10)SKIP2
```

INPUT CHARACTERS	EDIT DESCRIPTOR	INTERPRETED AS
THIS IS THE START	“HEADING HERE”	START (12 positions of the input are skipped)
THIS IS THE END OF	“ENDINGS HERE”	END OF (12 positions of the input are skipped)

7-29. ASCII STRING (MODIFIABLE) EDIT DESCRIPTOR (*n*H). The *n*H edit descriptor initializes the next *n* characters of the edit specification. Any ASCII character is permitted. If included, *n* must be a positive integer constant greater than zero (if omitted, the default value of *n* is 1).

On output, the *n*H edit descriptor causes the current *n* characters in the edit specification to be transferred to the external record.

For example,

```
10 FORMAT(2X,6HOUTPUT)
   WRITE(6,10)
```

would cause OUTPUT to be printed.

If the *n*H edit descriptor is referenced by a READ statement, then the value read in replaces the characters in the edit descriptor.

For example,

```
10 FORMAT(2X,6HOUTPUT)
   READ(5,10)
   WRITE(6,10)
```

If the input characters read in by the READ statement are: INPUT, then, when the WRITE statement is executed, these characters will have replaced the original characters in the *n*H edit descriptor and the ouput would be: INPUTb. Thus, on input, the *n*H edit descriptor causes the next *n* characters of an external record to replace the next *n* characters in the edit specification.

Table 7-35 shows some examples.

Table 7-34. '. . .' Edit Descriptor Input

```
   CHARACTER*16 IN1, IN2*14
10 FORMAT(2X, 'COLUMN HEAD', S)
20 FORMAT(2X, 'ROW LABEL', S)
   READ(5,10)IN1
   READ(5,20)IN2
```

EDIT DESCRIPTOR	INPUT CHARACTERS	VARIABLE	INTERPRETED AS
'COLUMN HEAD'	BEGIN DATA INPUT	IN1	INPUT (11 positions of the input are skipped)
'ROW LABEL'	END DATA INPUT	IN2	INPUT (9 positions of the input are skipped)

Table 7-35. *n*H Edit Descriptor Input and Output

EDIT DESCRIPTOR	INPUT LAST READ	OUTPUT
4HMULT	(None)	MULT
7HFORTRAN	ALGOLbb	ALGOLbb
12HPROGRAM DATA	BINARY LOADER	BINARY LOADE
10HCALCULATED	PASSEDbb	PASSEDbbbb

7-30. ASCII BLANKS EDIT DESCRIPTOR (*n*X). The *n*X edit descriptor causes *n* positions of a record to be skipped. If included, *n* must be a positive integer constant greater than zero; if omitted, the default value of *n* is 1.

On output, the *n*X edit descriptor causes *n* positions of the external record to be filled with blanks.

See table 7-36 for *n*X output.

Table 7-36. *n*X Edit Descriptor Output

```
10 FORMAT(E7.1,4X,"END")
20 FORMAT(F8.2,2X,I3)
   WRITE(6,10)A
   WRITE(6,20)A,J
```

FORMAT/EDIT SPECIFICATIONS	INTERNAL VALUE	OUTPUT
E7.1,4X,"END"	34.1	b3E+02bbbbEND
F8.2,2X,I3	5.87,436	bbbb5.87bb436

The *n*X edit descriptor, when used with the T*n* edit descriptor (see paragraph 7-31), may cause previous characters to be overlaid.

For example,

("ABCDEFG",T1,"M",2X,"N")

would produce

MbbNEFG

On input, the *n*X edit descriptor causes the next *n* positions of the input record to be skipped.

See table 7-37 for *n*X edit descriptor input.

7-31. TABULATE EDIT DESCRIPTOR (T*n*). The T*n* edit descriptor provides a means of tabulating an external record in order to select the position where data input/output is to begin or resume. The T*n* edit descriptor positions the record pointer to the *n*th position in the record.

See table 7-38 for examples.

Table 7-37. *n*X Edit Descriptor Input

```
10 FORMAT(D8.2,3X,M9.2)
20 FORMAT(5X,E9.2,I5)
   READ(5,10)A,B
   READ(5,20)A,J
```

FORMAT/EDIT SPECIFICATIONS	INPUT CHARACTERS	INTERPRETED AS
(D8.2,3X,M9.2)	b.25E+02END$1,563.79	.25E+02, 1563.79
(5X,E9.2,I5)	54321−98.7563814581	−.9876538E+02, 14581

As can be seen from table 7-38, the position numbers *n* need not be given in ascending order.

Note: The T*n* edit descriptor may cause previous characters to be overlaid.

7-32. RECORD TERMINATOR EDIT DESCRIPTOR (/). The / edit descriptor terminates the current external record and begins a new record (such as a new line on a line printer or a keyboard terminal or a new card on a card device). The / edit descriptor has the same result for both input and output: it terminates the current record and begins a new record. (On output, a new line is printed; on input, a new line or a new card is read.)

If a series of two or more / edit descriptors are written into a FORMAT statement, the effect is to skip $n - 1$ records, where *n* is the number of /'s in the series. A series of /'s can be written by using the repeat specification. (Note that a single slash (/) causes one record to be skipped.)

Note: If one or more / edit descriptors are the first item(s) in a series of format specifications, *n* (not $n - 1$) records are skipped for that series of /'s.

The / edit descriptor also can be used without a comma to separate it from other format and/or edit specifications; it has the same effect as a comma.

Table 7-38. T*n* Edit Descriptor Input and Output

```
           10 FORMAT(T11,"DESCRIPTION",T26,"QUANTITY,T2,"PART NO.")
              WRITE(6,10)

Output:    PART NO.bDESCRIPTIONbbbbQUANTITY
           ↑        ↑              ↑
           └position 2 └──position 11 └position 26


           10 FORMAT(T25,I3,T1,"HR124A",T10,"LOCK-WASHERS")
              WRITE(6,10)J

(Internal Value for J:  125)

Output:    HR124AbbbLOCK-WASHERSbbb125
           ↑        ↑              ↑
           └position 1 └──position 10 └position 25


           10 FORMAT(T13,E8.2,T1,I4,T24,M12.3)
              READ(5,10)A,J,B

Input:     1325COUNTEDbbb525.78LBSbb$4,365.78COST
           ↑           ↑          ↑
           └position 1 └position 13 └position 24

Results:
```

VARIABLE	INTERPRETED AS
A	.52578E+03
J	1325
B	.436578E+04

7-33. CARRIAGE CONTROL. The first character in a FORMAT statement is interpreted as a carriage control character. Carriage control characters and their meanings are as follows:

Blank	= single space
0	= double space
1	= page eject (form feed)
+	= no space (suppress space)

Additionally, a FORMAT statement may contain an octal number in the range 0 to 377, which will be treated as being equivalent to a byte character. The primary function served is that of a carriage-control character, especially where a particular number does not represent an available ASCII character. The octal number must be distinguished by a preceding percent sign and a trailing "C" (for uniformity with the compiler).

For example:

```
FORMAT(%306C,5HHELLO,L5)
```

The %306C gives a space of 1/4 of a page.

See the *MPE Intrinsics Reference Manual* for additional carriage control characters which may be used with FORTRAN/3000.

When the first character is interpreted, the spacing action requested by the character is performed and the remainder of the record is then printed. This is true unless the file is explicitly opened by the FOPEN intrinsic (see Section 8-8) and the spacing action then depends on the parameters passed to FOPEN.

7-34. REPEAT SPECIFICATION FOR EDIT DESCRIPTORS

The repeat specification is a positive integer written to the left of the edit descriptor it controls.

The repeat specification is written as: *r*". . .", *r*'. . .', *r*(*n*X), *r*(*n*H) or *r*/, where *r* is the repetition value.

Note: The forms *r*(*n*H) and *r*(*n*X) may include other field and/or edit descriptors within the parentheses.

EXAMPLES:

```
(E9.2/3F7.1,2(4HDATA)) =
    (E9.2/F7.1,F7.1,F7.1,4HDATA,4HDATA)

(2(5HABORT2/)) = (5HABORT,//,5HABORT// )

(G10.3,3("READ",E12.4)) =
    (G10.3,"READ",E12.4,"READ",E12.4,"READ",E12.4)
```

7-35. SPECIFICATION INTERRELATIONSHIPS

Two or more specifications (E9.3,I6) in a FORMAT statement cause the data to be concatenated.

For example,

Data 12.3 and −30303 produces:

b.123E+02−30303

Table 7-39. / Record Terminator Edit Descriptor Examples

EDIT/FORMAT SPECIFICATIONS	INTERNAL VALUES	OUTPUT	RECORD NO.
(E12.5,I3/"END")	.32456E+04, 95	bb.32456E+04b95	1
		END	2
(E12.5,I3///"END")	.32456E+04, 96	bb.32456E+04b96	1
			2
			3
		END	4
(I5,3HEND,4/"NEW DATA")	43592	43592END	1
			2
			3
			4
		NEW DATA	5
(2/"END")			1
			2
		END	3

The *n*X edit descriptor (E9.3,4X,I6) can insert blank spaces between fields. For example, the same data as before produces:

```
b.123E+02bbbb−30303
```

The / edit descriptor (E9.3/I6) places each field on a different line. For example, the same data produces:

```
b.123E+02
−30303
```

7-36. NESTING

The group of format and edit specifications in a FORMAT statement can include one or more other groups enclosed in parentheses (called "group(s) at nested level *x*"). Each group at nested level 1 can include one or more other groups at nested level 2; those at level 2 can include groups at nested level 3; those at level 3 can include groups at level 4.

For example,

(E9.3,I6,(2X,I4))

One group at nested level 1.

(T12,"PERFORMANCES"3/(E10.3,2(A2,L4)))

One group at nested level 1, one at nested level 2.

(T5,5HCOSTS,2(M10.3,(I6,E10.3,(A2,F8.2))))

One group at nested level 1, one at level 2, one at level 3.

A FORTRAN/3000 READ or WRITE statement references each element of a series of list elements and the corresponding FORMAT statement is scanned to find a field descriptor for each list element. As long as a list element and field descriptor pair occurs, normal execution continues.

7-37. UNLIMITED GROUPS

If a program does not provide a one-to-one match between list elements and field descriptors, execution continues only until all list elements have been transmitted. If there are fewer field descriptors than list elements, format specification groups at nested level 1 and deeper are used as "unlimited groups". After the effective rightmost field descriptor in a FORMAT statement has been referenced, three steps are performed:

1. The current record is terminated: on output, the current field is completed, then the record is terminated; on input, the rest of the record is ignored.
2. A new record is started.
3. Format control (field descriptor interpretation) is returned to the repeat specification for the rightmost specification group at nested level 1. Or, if there is no group at level 1, control returns to the first field descriptor in the FORMAT statement.

Note: In any case, the current scale factor is not changed until another scale factor is encountered. (See paragraph 7-23.)

Table 7-40 shows examples of format control.

Table 7-40. Format Control

(I5,2(3X,F8.2,8(I2)))	Control returns to (3X,F8.2,8(I2))
(I5,3X,4F8.2,3X)	Control returns to (I5,3X,4F8.2,3X)

If A equals .32E+04 these statements:

```
    WRITE(6, 100) A,A,A,A,A
100 FORMAT (" HEADER"/3(E10.2))
```

Yield this result:

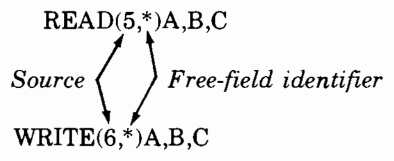
```
HEADER
bbb.32E+04bbb.32E+04bbb.32E+04
bbb.32E+04bbb.32E+04
```

7-38. FREE-FIELD INPUT/OUTPUT

Free-field input/output is format converted according to the data itself. That is, data are converted from or to external ASCII character form without using FORMAT statements.

For free-field input/output, FORTRAN/3000 READ and WRITE statements are written with an asterisk instead of a FORMAT statement identifier.

For example,



7-39. FREE-FIELD CONTROL CHARACTERS

Special ASCII characters embedded in the external data fields are used to control free-field input. These characters are shown in table 7-41. Predefined field and edit descriptions are used and a carriage control character gets embedded as the first character of every output record to control the format of free-field output. For example, in the case of character string data, the embedded carriage control character designates single spacing.

7-40. FREE-FIELD INPUT

Six data types can be input to free-field conversion: octal, integer, double integer, floating-point real, double-precision floating-point and character string. Numeric data types can be mixed freely with numeric list elements. For example, an integer data item can be input to a floating-point list element; the integer is converted to floating-point form and the double-word result is stored.

All rules for input to numeric and alphanumeric conversions apply. (See paragraph 7-5.) A character string item, however, must be input only to a character variable; if not, SOFTERROR' message FMT: STRING MISMATCH occurs and execution is terminated.

7-41. DATA ITEM DELIMITERS

A data item is any numeric or character string field occurring between data item delimiters. A data item delimiter is a comma, a blank space, or any ASCII character that is not part of the data item. The initial data item need not be preceded by a delimiter; the function of a delimiter is to signal the end of one data item and the beginning of another. Two commas with no data item in between indicate that no data item is supplied for the corresponding variable, and the previous contents of that variable are to remain unchanged. Any other delimiter appearing two or more consecutive times is equivalent to one delimiter.

7-42. DECIMAL DATA. Decimal data items are written in any of the field descriptor forms described for formatted data except the monetary or the numeration forms. Embedded commas and dollar signs are data item delimiters in free-field data transfers and, therefore, cannot be part of the data.

Notes: 1. Leading, embedded, or trailing blanks or commas, $, or any ASCII character that is not a part of the data item are data item delimiters.

2. All integer inputs have an implicit decimal point to the right of the last (least significant) digit.

3. The exponent field input can be any of several forms, as follows:

$+e$	$+ee$	Ee	Eee	De	Dee
$-e$	$-ee$	E$+e$	E$+ee$	D$+e$	D$+ee$
		E$-e$	E$-ee$	D$-e$	D$-ee$

where e is an exponent value digit.

7-43. OCTAL DATA. Octal data items are written:

$$\%i_1 \ldots i_n$$

where

$i_1 \ldots i_n$ = the octal integer digits

n = the number of significant digits (maximums: 6 for an integer variable, 11 for a double integer, 11 for a real variable, or 22 for a double precision variable)

% = the octal data identifier

On input, the data item is interpreted as an octal integer number. The number is converted to an internal representation value for the variable (list element) currently being referenced.

Table 7-41. Free-Field Control Characters

CHARACTER(S)	FUNCTION
Blank space or comma or any ASCII character not part of the data item.	Data item delimiter (terminator).
/ (slash)	Record terminator (when not part of a character string data item).
+ (plus) or − (minus)	Sign of data item.
. (period)	Defines the beginning of the fraction subfield of the data item.
E or + or − or D	Define the beginning of the exponent subfield of the data item.
% (percent)	Defines the data item as octal (not decimal).
". . ."	A character string enclosed by quotation marks to be input to a FORTRAN/3000 character type variable.
⟨⟨ · · · ⟩⟩	A character string enclosed by ⟨⟨and⟩⟩ signifies that the characters are a comment only for the external record; the string and symbols are ignored on input.

The input field can consist of octal digits only. No more than six digits (no larger than 177777$_8$) are interpreted for an integer variable; no more than 11 digits (no larger than 37777777777$_8$) are interpreted for a real or double integer variable; and no more than 22 digits (no larger than 1777777777777777777777$_8$) are interpreted for a double precision variable. Any non-octal or non-numeric character (including a blank) anywhere in the field will produce a conversion error. If the number of digits in the data item is less than the maximum number allowed by that type of variable, the digits are right-justified in that variable's internal representation (one, two, or four words of memory).

7-44. CHARACTER STRING DATA. A character string data item is any series of ASCII characters, including blank spaces, usually enclosed in quotation marks. Any one or more of these characters can be a quotation mark if signaled by an adjacent quotation mark.

The corresponding variable must be of type CHARACTER of a specified string length. If the number of characters in the data item is greater than the length attribute *n* of the variable, *n* characters are transferred and the remaining characters are ignored. If there are fewer characters than *n,* all characters of the data item are transferred, left-justified, in the variable, followed by trailing blanks.

If an end-of-record condition occurs before the terminating edit descriptor of a character string data item, FORTRAN/3000 assumes that the data item is continued in the next record and resumes transfer with the first character of the next record.

7-45. RECORD TERMINATOR. The character / (slash), if not part of a character data item, terminates the current record and delimits the current data item. If this occurs before all list elements have been satisfied, the remainder of the current record is skipped and transfer resumes with the first character of the next record.

7-46. LIST TERMINATION. If an end-of-record occurs without the record terminator /, the effect is to end the list of variables (list elements). Any list elements not satisfied are left unchanged.

7-47. FREE-FIELD OUTPUT

Six data types can be output under free-field conversion: integer, floating-point real, double-precision floating-point, logical, complex and character string.

- Integer data items are output under the I6 field descriptor.
- Double integer data items are output under the I11 field descriptor.
- Floating-point data items are output under the G12.6 field descriptor.
- Double-precision floating-point data items are output under the G23.17 field descriptor.
- Character string data items are output under the ". . ." edit descriptor. The adjacent quotation rule is used (if a quotation mark is to be included in the output string, double quotation marks must be used).

7-48. DATA ITEM DELIMITER. Each field in the ouput record is delimited by one blank space.

7-49. RECORD TERMINATORS. If the width of a current numeric data item is too great for the remainder of a current record, a record terminator character (/) is output, and a new record is started with the next character of the data item.

If a character string data item overlaps record boundaries, subsequent records are output (without record terminator slashes) until the entire character string has been transferred.

7-50. UNFORMATTED (BINARY) TRANSFER

Data can be transferred to and from files in internal representation (binary) form without any conversion. Such transfers are faster than formatted data transfers.

Note: Every value transmitted, including character strings of odd length, will begin on a word boundary; transmission of a character string of odd length may thus leave a byte unused and unusable.

Two types of access to files on disc devices are available through the MPE/3000 file system: sequential or direct. When binary transfer is used, the READ or WRITE statements of a FORTRAN/3000 program are written without a FORMAT statement identifier.

For example, statements could be as follows for sequential access:

```
      READ(9)A,B,C
         ↗
Device file
number
         ↘
      WRITE(12)A,B,C
```

When direct access is used, the READ or WRITE statements of a FORTRAN/3000 program are written with an integer simple variable for the record identifier and without a FORMAT statement identifier.

For example,

```
      READ(8@IV)A,B,C
         ↗   ↑
Device file   Record identifier variable
number
         ↘   ↘
      WRITE(12@KR)A,B,C
```

In sequential access, as many records as needed are used in sequence until the entire list of elements has been transferred.

Note: If the storage required exceeds the size of the record, transfer continues into the next record; this usually leaves part of that next record unused.

In direct access, record access is terminated by the last element in the list. Any unused portion of the record just terminated is ignored since only one record may be accessed for any given WRITE statement. If the storage required by all the list elements exceeds the record size, error message FMT: DIRECT ACCESS OVERFLOW occurs.

Section VIII contains a discussion of the way files are created initially.

7-51. INPUT/OUTPUT OF COMPLEX NUMBERS

The format specification for a complex number consists of two field descriptors. The first one is used to interpret the real part of the complex number and the second to interpret the imaginary part. Formatted *or* free field input/output can be performed. The Formatter applies the same formatting rules to the real and imaginary parts of a complex number as it does to other types of numbers.

The following example illustrates formatted input of two complex numbers:

```
    COMPLEX A,B
    READ(5,500) A,B
500 FORMAT (2F10.2,G12.4,G10.2)
```

Both the real and imaginary parts of the complex variable A will be read according to the F10.2 edit descriptor. The real part of B will be read with the G12.4 edit descriptor and the imaginary part with the G10.2 descriptor.

An example of formatted output of complex numbers is:

```
    COMPLEX A
       :
    WRITE(6,600) A
600 FORMAT ("A = ",F10.2, F8.2)
```

The real part of A is printed according to the F10.2 descriptor and the imaginary part according to the F8.2 descriptor.

When a complex number is entered in free field format, parentheses may be placed around the pair of values representing the real and imaginary parts but they are not required and are ignored by the Formatter.

```
COMPLEX A,B
READ (5,*) A,B
```

accepts the following input:

```
3.7, 4.2, (6.1, 8.2)
```

The complex number A will equal (3.7,4.2) and the complex number B will equal (6.1,8.2).

When free field output is used for complex numbers, the Formatter displays the real part and then the imaginary part under the 2G12.6 format specification.

FORTRAN/3000 FILE FACILITY

SECTION VIII

Every peripheral input/output or storage device is linked to a *file* through the file facility of the MPE/3000 Operating System. Each file takes on the attributes of the hardware device associated with it. See the *MPE Intrinsics Reference Manual* for a discussion of the file facility.

FORTRAN/3000 input or output statements reference specific hardware devices (such as teleprinters or card readers) by referencing a *FORTRAN Logical Unit Number* associated with the device. The FORTRAN/3000 compiler then translates the input/output statements into requests for manipulations of the file referenced by this unit number. For example, an input statement of the form: READ(5,100)A,B tells the FORTRAN/3000 compiler that two values are to be read from the file whose FORTRAN unit number is 5, formatted in accordance with specifications contained in statement number 100, and assigned to variables A and B.

8-1. REFERENCING FILES

Files are referenced in FORTRAN/3000 input/output statements by using integer constants or integer simple variables with values in the range of 1 to 99, inclusive. The same number can be used in more than one input/output statement; however, the same file is referenced in each case. The FORTRAN/3000 compiler assumes that any file specifically referenced by an integer constant in an input/output statement has been defined. For example, if the statement READ(10,100)A,B were used in a program, the compiler assumes that a file has been defined for FORTRAN unit number 10. If the unit reference of an input/output statement appears as an integer simple variable, the FORTRAN unit number represented by the variable must appear in a $CONTROL FILE = *n* compiler command in the program and the integer simple variable must be set equal to this unit number. (See Section IX for a discussion of compiler commands.)

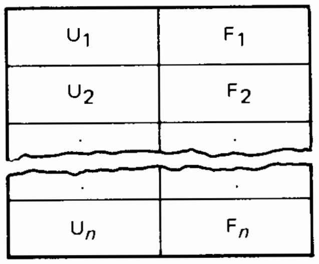
8-2. FORTRAN/3000 LOGICAL UNIT TABLE (FLUT)

The MPE/3000 Operating System Segmenter prepares a FORTRAN Logical Unit Table (FLUT) for FORTRAN/3000 programs which use the Formatter.

Note:

The discussion of the FLUT is provided for reference only. The FLUT is not explicitly manipulated by the FORTRAN/3000 programmer.

The structure of the FLUT is shown in figure 8-1.



Where

$U_1 \ldots U_n$ = the UNIT numbers (integers in the range [1,99]) in the left byte of each entry, to be specified in Formatter initialization calls

$F_1 \ldots F_n$ = 0 in the right byte, when the FLUT is prepared

Figure 8-1. FORTRAN Logical Unit Table (FLUT)

Reference to a specific FORTRAN logical unit either in a $CONTROL FILE = statement (see Section IX) or any other FORTRAN/3000 statement will result in that unit number being included in the FLUT as one of the UNIT numbers (U_1 through U_n). Failure to so define every unit required by a program will result in no entry for the unit and will cause an error when the unit is referenced.

F_1 through F_n (*n* is from 1 to 99 and is the number of entries in the FLUT) in the right byte of each word in the FLUT is the MPE/3000 file number assigned to the logical unit when the file was last opened by the MPE/3000 file intrinsic FOPEN. Thus the FLUT provides the necessary correspondence between FORTRAN unit numbers and MPE/3000 file numbers required by the MPE/3000 file system. (See the *MPE Intrinsics Reference Manual* for a discussion of the FOPEN intrinsic.)

When the Formatter is called by a FORTRAN/3000 statement, it must determine if the file to be used has been opened, and if it has, what the file parameters (such as the file options, access options, etc.) are. See paragraph 8-7 for a discussion of these options.

The Formatter first checks the FLUT for a U entry corresponding to the *unit* specified in the FORTRAN/3000 statement which initiated the call to the Formatter. If such an entry does not exist, the error message: FILE NOT IN TABLE FOR UNIT # *xx* occurs and the program aborts. If a U entry exists, the F entry is checked.

If the F entry is zero, the file has not been opened and the Formatter makes a call to the MPE/3000 file intrinsic FOPEN. The nominal FORTRAN/3000 parameters (see paragraph 8-3) are used in the FOPEN call. These include the file name created by appending the unit number to the ASCII characters FTN. For example, the file name for unit 3 is FTN03. The FOPEN intrinsic returns an integer which is stored in the FLUT as the F entry for the unit referenced.

If the file entry is not zero, the file has been opened and the Formatter calls the MPE/3000 file intrinsic FGETINFO which extracts the file parameters for that file. The Formatter also allocates space for an I/O buffer to be used in the input/output operation.

8-3. NOMINAL FORTRAN/3000 FILE PARAMETERS

Table 8-1 shows the nominal FORTRAN/3000 file parameters used in an FOPEN call. These parameters can be superseded with an MPE/3000 :FILE command (see paragraph 8-7.)

Table 8-1. Nominal FORTRAN/3000 File Parameters

FILEDESIGNATOR	FTN*dd*, where *dd* is the UNIT number in the FLUT (for example, FTN03).
FOPTIONS	Bits are set or cleared for the following file options: Domain (bits 15 and 14 clear): NEW
	BINARY file (bit 13 clear)[1]
	Default File Designator (bits 12, 11, 10): 000[2]
	Record Format (bits 9 and 8): If sequential, then VARIABLE (bits 9 and 8 = 01), else (direct) FIXED (bits 9 and 8 = 00).
	Carriage Control (bit 7 clear): none[3]
	Disallow File Equation (bit 5 clear): allow :FILE
	(Bits 4 through 0 are spares.)
AOPTIONS	Access Type: READ/WRITE
	No Multirecord
	No Dynamic Locking
	Exclusive Access: Default
	Buffered
RECSIZE	System default value: 128 words.
DEVICE	System default: DISC.
FORMMSG	None.
USERLABELS	System default: 0
BLOCKFACTOR	System default value: 128/physical record size
NUMBUFFERS	System default: OUTPRI = 8 COPIES = 1 BUFFERS = 2
FILESIZE	System default: 1023 records
NUMEXTENTS	System default: 8 extents
INITIALLOC	System default value: 1 extent
FILECODE	System default: 0.

[1]Except for FTN05 or FTN06: ASCII file (bit 13 set).
[2]Except for FTN05: 100 for $STDIN, and FTN06: 001 for $STDLIST.
[3]Except for FTN06: yes (bit 7 set).

8-4. STANDARD INPUT AND OUTPUT FILES

The integer values used for FORTRAN logical unit number references in a source program are converted to file names recognizable to the MPE/3000 operating system file facility. As discussed in paragraph 8-2, the name for any file is created by joining the characters FTN with the two-digit FORTRAN logical unit number. For example, file 8 is FTN08, file 10 is FTN10, etc.

The FORTRAN logical unit number 5 is assigned by default to the standard input file (usually a card reader file in batch mode or a teleprinter file in interactive mode), and 6 is assigned to the standard list (or output) file (usually a line printer file in batch mode or a teleprinter file in interactive mode).

The DISPLAY statement (intended for free-field output in interactive mode) implicitly declares and references FORTRAN unit number 6. The ACCEPT statement (intended for free-field input in interactive mode) implicitly references $STDLIST for prompting and unit 5 for the user's response.

8-5. CREATING AND ACCESSING FILES

Permanent files for the storage of data can be created and accessed through MPE/3000 system commands external to a FORTRAN/3000 program or through the use of system intrinsic calls within the program. MPE/3000 commands and intrinsics are described in detail in the *MPE Commands Reference Manual* and the *MPE Intrinsics Reference Manual* respectively.

A new file can be created, and its characteristics specified,by using the MPE/3000 system :BUILD command. The :BUILD command is written in the following format:

:BUILD *filereference*

For example,

```
:BUILD ITEM
```

where

filereference
is the name of the file to be opened.

The file may be defined further by using additional optional parameters, as follows:

```
:BUILD ITEM;DISC=100;REC=-72
```

where, for example,

DISC=100
specifies the device as disc with a total maximum file capacity of 100 records. The default value is 1023 records.

REC=-72
specifies the size of each record in the file. The negative number (-72) represents bytes. (A positive number represents words.) The default values generally specified by the system are:

Disc	=	128 (words)
Tape	=	128 (words)
Printer	=	-132 (bytes)
Card reader	=	-80 (bytes)
Card punch	=	-80 (bytes)
Terminal	=	-72 (bytes)

Of course, these values may vary from device to device and installation to installation. See your system management for your current default record sizes.

For a complete description of the :BUILD command, including additional optional parameters, see the *MPE Commands Reference Manual.*

A simple example using the :BUILD command is shown in figure 8-2.

The :BUILD command creates an empty file named TEST1. The :FILE command tells the operating system that FORTRAN unit number FTN02 is equated to the file named TEST1 and that TEST1 is an old file (it has been created by the :BUILD command and will not be created within the program). (See paragraph 8-7 for a discussion of the :FILE command.)

Within the program, the compiler command $CONTROL FILE = 2 tells the compiler that the file FTN02 may be accessed and the statement IN= 2 sets the variable IN (used in the WRITE statement) equal to 2. Thus, when the WRITE statement is executed, information will be written into TEST1, which has been equated to FORTRAN unit number 2 by the :FILE command.

The program outputs a prompt character (?) on the terminal (the program was run in interactive mode) and writes whatever is typed in into file TEST1.

Figure 8-3 shows a simple program to read the data contained in file TEST1. The integer simple variable, IOUT, used in the READ statement in the program, is set equal to 2.

8-6. COPYING FILES

The information contained in a file can be copied into another file in several ways using MPE/3000 system intrinsics and commands. A simple method of copying one file into another using the :BUILD and :FILE commands is shown in figure 8-4.

The file from which the information is to be copied (TEST1) is equated to FORTRAN unit number FTN02 with a :FILE command; the file into which the information is to be copied is created with a :BUILD command and equated to FORTRAN unit number FTN03 with a :FILE command.

In the program, the integer simple variable OLD (used in statement 10 to read the old file) is set equal to 2 and integer simple variable NEW is set equal to 3.

8-7. CHANGING STANDARD ATTRIBUTES OF FILES

The standard attributes of files used by a FORTRAN/3000 program can be modified through the use of the MPE/3000 :FILE command.

```
:BUILD TESTI
:FILE FTNØ2=TESTI,OLD
:FORTGO FTRAN3Ø

PAGE ØØØ1    HP321Ø2B.ØØ.Ø


       $CONTROL FILE=2
               PROGRAM CREATE FILE
       C
       C EXAMPLE PROGRAM TO CREATE FILE
       C
               CHARACTER*72 INFILE
               IN=2
        1Ø     ACCEPT INFILE
               IF(INFILE[1:3].EQ."END")GOTO 3Ø
               WRITE(IN,*,END=2Ø)INFILE
               GOTO 1Ø
        2Ø     DISPLAY "FILE FULL"
        3Ø     STOP
               END


****       GLOBAL STATISTICS       ****
****    NO ERRORS,    NO WARNINGS   ****
TOTAL COMPILATION TIME  Ø:ØØ:Ø1
TOTAL ELAPSED TIME      Ø:ØØ:Ø4

  END OF COMPILE

  END OF PREPARE


?THIS IS RECORD 1 OF FILE "TESTI"

?THIS IS RECORD 2

?RECORD 3

?4

?ETC

?END

  END OF PROGRAM
```

Figure 8-2. Example of a FORTRAN/3000 Program to Create and Access a File

```
:FILE FTN02=TEST1,OLD
:FORTGO FTRAN31

PAGE 0001   HP32102B.00.0


$CONTROL FILE=2
      PROGRAM READ FILE
C
C FILE READ EXAMPLE
C
      CHARACTER*72 OUTFILE
      IOUT=2
 10   READ(IOUT,*,END=20,ERR=30)OUTFILE
      DISPLAY OUTFILE
      GOTO 10
 20   DISPLAY "END OF FILE"
      STOP
 30   DISPLAY "FILE READ ERROR"
      STOP
      END


****      GLOBAL STATISTICS       ****
****   NO ERRORS,   NO WARNINGS   ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:00:03

 END OF COMPILE

 END OF PREPARE


THIS IS RECORD 1 OF FILE "TEST1"

THIS IS RECORD 2

RECORD 3

4

ETC

END OF FILE
 END OF PROGRAM
```

Figure 8-3. Example of a FORTRAN/3000 Program to Read a File

```
:BUILD TEST2
:FILE FTN02=TEST1,OLD
:FILE FTN03=TEST2,OLD
:FORTGO FTRAN32

PAGE 0001    HP32102B.00.0


$CONTROL FILE=2,FILE=3
      PROGRAM FILE COPY
C
C FILE COPY EXAMPLE
C
      CHARACTER*72 DATA
      INTEGER OLD
      OLD=2
      NEW=3
 10   READ(OLD,*,END=20,ERR=30)DATA
      WRITE(NEW,*,END=40,ERR=50)DATA
      GOTO 10
 20   DISPLAY "FILE COPIED"
      STOP
 30   DISPLAY "ERROR OCCURRED ON INPUT FILE"
      STOP
 40   DISPLAY "OUTPUT FILE TOO SMALL FOR COMPLETE COPY"
      STOP
 50   DISPLAY "ERROR OCCURRED ON OUTPUT FILE"
      STOP
      END


****       GLOBAL STATISTICS        ****
****    NO ERRORS,    NO WARNINGS   ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:00:03

 END OF COMPILE

 END OF PREPARE


FILE COPIED
 END OF PROGRAM
```

Figure 8-4. Example of a FORTRAN/3000 Program to Copy Information From One File to Another

Note: Read the discussion of files in the *MPE Commands Reference Manual* before attempting to change file attributes with the :FILE command.

The specifications in a :FILE command do not take effect until the compiled program is running and opens the file referenced. The :FILE command specifications hold throughout the entire program unless superseded (by another :FILE command) or revoked by a :RESET command. At job (batch) or session (interactive) termination, however, all :FILE commands are cancelled.

When two (or more) :FILE commands referencing the same file appear in a job or session, the last command takes precedence.

8-8. DIRECT INTRINSIC CALLS

Since a FORTRAN/3000 user can write and execute non-FORTRAN/3000 language programs, it is possible to access files directly. This is accomplished by writing direct calls to MPE/3000 operating system intrinsics which manipulate the indicated files.

The calls may require actual arguments passed by value (see Appendix A). Direct intrinsic calls may be used entirely by themselves or in conjunction with :FILE commands.

An example of the use of the FOPEN, FREAD, and FCLOSE intrinsics is shown in figure 8-5. (See appendix A for an explanation of the back slash and .CC. logical operator.)

Note: These examples could be considerably simplified if the SYSTEM INTRINSIC statement had been used. (See Appendix A.)

The program shown in figure 8-5 uses the intrinsic FOPEN to open the file "MAILLIST" and the intrinsic FREAD to read the file. The FOPEN intrinsic procedure contains thirteen parameters. The file name (MAILLIST) is passed to the FOPEN intrinsic in the character variable FILENAME. A 1 is passed for the *foptions* parameter, specifying that the file is an old permanent file and that the system file domain should be searched for the file. The default values are taken for the remainder of the *foptions*. Octal 105 is passed as the *aoptions* parameter, specifying that all file intrinsics, including FUPDATE, can be issued for the file and that, once the file is opened, other FOPEN requests against this file will be denied until the file is closed or the program terminates. Default values are taken for the remainder of the *aoptions*. Zeros are passed (as dummy parameters) to the FOPEN intrinsic for all remaining parameters.

The last parameter in the call to FOPEN (%16000) is an octal value used as a "bit map" to inform the intrinsic being called which parameters are being passed values and which are being passed dummy parameters.

When the file is opened, the FOPEN intrinsic returns an integer value to the variable FILENUMBER. A file read is performed on this file number by the FREAD intrinsic and the value that is obtained from the read is assigned to the logical variable LBUFFER. The WRITE statement prints the first 19 characters of this value (LBUFFER and BUFFER have been equivalenced).

Finally, the FCLOSE intrinsic is called to close the file MAILLIST.

Note: The MPE/3000 file number returned by FOPEN (as the value of FILENUMBER) can be made negative, assigned to a simple variable such as I, and then used in the *unit* part of a FORTRAN/3000 input/output statement (for example, I= -FOPEN (. . . .)). This number then is passed to the formatter as the file number on which I/O is to be performed, effectively bypassing the FLUT.

8-9. FSET PROCEDURE

The FSET procedure is an entry point to the HP 3000 Compiler Library procedure FTNAUX' (see the *Compiler Library Reference Manual*) and is used to change the MPE/3000 operating system file number assigned to a given FORTRAN logical unit number in the FLUT.

The FSET procedure can be called from a FORTRAN/3000 program as follows:

CALL FSET(UNIT,NEWFILE,OLDFILE)

where

UNIT
is a positive single integer in the range of 1 to 99 to specify the FLUT entry for which the change is to be made.

NEWFILE
is a positive single integer in the range of 1 to 254, or an integer variable, to specify the new MPE/3000 file number which is to be assigned to the UNIT specified above.

OLDFILE
is a variable to which the procedure returns the old value of the file number that was assigned to the UNIT specified above.

A program using the FSET procedure is shown in figure 8-6. See appendix A for an explanation of the back slash and .CC. logical operator.

When the call to FSET is executed, the MPE/3000 file number assigned to the variable FILENUMBER by the FOPEN intrinsic call is assigned to unit number 5 in the FLUT. Thus, when the READ statement (referencing FORTRAN unit number 5) is executed, a record from the file MAILLIST is read and assigned to character variable BUFFER.

```
:FORTGO FTRAN33

PAGE 0001   HP32102B.00.0


      PROGRAM INTRINSICS
C
C FOPEN, FREAD, AND FCLOSE EXAMPLE
C
  100 FORMAT(T2,S)
      CHARACTER*72 BUFFER,FILENAME*16
      LOGICAL LBUFFER(36)
      INTEGER FILENUMBER,FOPEN,FREAD
      EQUIVALENCE (LBUFFER,BUFFER)
      FILENAME="MAILLIST"
C
C THE NEXT STATEMENT CALLS THE FOPEN INTRINSIC
C
      FILENUMBER=FOPEN(FILENAME,\1\,\%105\,0,0,0,0,0,0,\0.0\,
     #0,0,0,\%16000\)
      IF(.CC.)40,10,40
   10 DISPLAY "FILENUMBER = ",FILENUMBER
C
C THE NEXT STATEMENT CALLS THE FREAD INTRINSIC
C
   20 N=FREAD(\FILENUMBER\,LBUFFER,\36\)
      IF(.CC.)50,30,60
   30 WRITE(6,100)BUFFER[1:19]
      GOTO 20
   40 DISPLAY "FOPEN FAILURE"
      STOP
   50 DISPLAY "FREAD ERROR"
      STOP
C
C THE NEXT STATEMENT CALLS THE FCLOSE INTRINSIC
C
   60 CALL FCLOSE(\FILENUMBER\,\1\,\0\)
      IF(.CC.)70,80,70
   70 DISPLAY "FCLOSE FAILURE"
      STOP
   80 DISPLAY "FILE CLOSED SUCCESSFULLY"
      STOP
      END



****      GLOBAL STATISTICS       ****
****   NO ERRORS,   NO WARNINGS   ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:00:06

 END OF COMPILE

 END OF PREPARE


FILENUMBER =          1
JOHN       BIGTOWN
LOIS       ANYONE
ALI        BABA
JAMES      DOE
JANE       DOE
JOHN       DOUGHE
SPACE      MANN
KING       ARTHUR
JENNA      GRANDTR
KARISSA    GRANDTR
SWASH      BUCKLER
KNEE       BUCKLER
FILE CLOSED SUCCESSFULLY
 END OF PROGRAM
```

Figure 8-5. Example of a FORTRAN/3000 Program to Call FOPEN, FREAD, and FCLOSE Intrinsics.

```
:FORTGO FTRAN34

PAGE 0001    HP32102B.00.0

      PROGRAM FSET
C
C FOPEN, FSET, AND FCLOSE EXAMPLE
C
 100  FORMAT(T2,S)
      CHARACTER*72 BUFFER,FILENAME*16
      INTEGER FILENUMBER,FOPEN,OLDNUM
      FILENAME="MAILLIST"
C
C THE NEXT STATEMENT CALLS THE FOPEN INTRINSIC
C
      FILENUMBER=FOPEN(FILENAME,\1\,\%105\,0,0,0,0,0,0,\0.0\,
     #0,0,0,\%16000\)
      IF(.CC.)30,10,30
C
C THE NEXT STATEMENT CALLS THE COMPILER LIBRARY
C PROCEDURE FSET TO ASSIGN FORTRAN UNIT NUMBER 5
C TO FILENUMBER
C
 10   CALL FSET(5,FILENUMBER,OLDNUM)
      DISPLAY "OLD FILE NUMBER = ",OLDNUM
      DISPLAY "FOPEN NUMBER = ",FILENUMBER
 20   READ(5,END=40)BUFFER
      WRITE(6,100)BUFFER[1:19]
      GOTO 20
 30   DISPLAY "FOPEN FAILURE"
      STOP
C
C THE NEXT STATEMENT CALLS THE FCLOSE INTRINSIC
C
 40   CALL FCLOSE(\FILENUMBER\,\1\,\0\)
      IF(.CC.)50,60,50
 50   DISPLAY "FCLOSE FAILURE"
      STOP
 60   DISPLAY "FILE CLOSED SUCCESSFULLY"
      STOP
      END


****      GLOBAL STATISTICS        ****
****   NO ERRORS,   NO WARNINGS    ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:00:04

 END OF COMPILE

 END OF PREPARE


OLD FILE NUMBER =        0
FOPEN NUMBER =        1
JOHN      BIGTOWN
LOIS      ANYONE
ALI       BABA
JAMES     DOE
JANE      DOE
JOHN      DOUGHE
SPACE     MANN
KING      ARTHUR
JENNA     GRANDTR
KARISSA   GRANDTR
SWASH     BUCKLER
KNEE      BUCKLER
FILE CLOSED SUCCESSFULLY
 END OF PROGRAM
```

Figure 8-6. FSET Example

8-10. FNUM AND UNITCONTROL PROCEDURES

The FNUM and UNITCONTROL procedures are HP 3000 Compiler Library procedures that can be called from FORTRAN/3000 programs.

The FNUM procedure enables a FORTRAN/3000 program to extract the MPE/3000 system file number assigned to a given FORTRAN logical unit number from the FLUT.

The FNUM procedure can be called from a FORTRAN/3000 program as an external function, as follows:

I = FNUM(UNIT)

where

UNIT
is a positive single integer in the range of 1 to 99 to specify the FLUT entry for which the MPE/3000 system file is to be extracted.

Note: FNUM must be declared type integer.

The UNITCONTROL procedure enables a FORTRAN/3000 program to request several actions (see below) for any FORTRAN logical unit.

The UNITCONTROL procedure is called as follows:

CALL UNITCONTROL(UNIT,OPT)

where

UNIT
is a positive single integer in the range of 1 to 99 to specify the FLUT entry of the file to be used.

OPT
is a single integer to specify one of the following options:

−1: REWIND (but don't close the file)
0: BACKSPACE
1: ENDFILE(write an EOF mark)
2: SKIP BACKWARD TO A TAPE MARK
3: SKIP FORWARD TO A TAPE MARK
4: UNLOAD TAPE AND CLOSE THE FILE
5: LEAVE TAPE AND CLOSE THE FILE
6: CONVERT FILE TO PRE-SPACING*
7: CONVERT FILE TO POST-SPACING*
8: CLOSE FILE

Note: If no file is open for the specified unit, FNUM and UNITCONTROL will open one (as does a READ or WRITE statement).

*See "Directing File Control Operations" in the *MPE Intrinsics Reference Manual.*

A program using FNUM and UNITCONTROL is shown in figure 8-7 and the output resulting from the program is shown in figure 8-8.

The BACKSPACE statement must not reference files of variable length records, otherwise the program terminates abnormally with the following message:

FILE SYSTEM ERROR ON UNIT #XX

In order to call the MERGE procedure (which merges two or more sorted input files into one sorted output file), the MPE/3000 system file numbers must be specified in an integer array. The FNUM external function is used to assign the file numbers for FTN20 and FTN21 to INFILES elements 1 and 2. In addition, the parameter FNUM(22) is passed to the MERGE procedure to specify the output file number.

Three calls to UNITCONTROL are made at the end of the program to close the three files.

Note: The UNITCONTROL procedure closes files under FCLOSE *disposition* 0 (no change). Thus, the file remains as it was before the file was opened. If the file is NEW (as is FTN22), it is deleted; otherwise it is assigned to the domain to which it belonged previously. See the *MPE Intrinsics Reference Manual* for a discussion of the FCLOSE *disposition* parameter.

Figure 8-8 shows the output resulting from the program in figure 8-7.

Note: The MAXDATA = 4000 parameter is appended to the :PREP command to provide the MERGE procedure with sufficient stack space for its operation and is of no other concern to the FORTRAN/3000 programmer.

```
:FORTRAN FTRAN35

PAGE 0001   HP32102B.00.0


$CONTROL FILE=20,FILE=21,FILE=22
      PROGRAM PROCEDURES
 100  FORMAT(T2,S)
      CHARACTER BUFFER*72
      INTEGER KEYS(6),FNUM,INFILES(2)
      LOGICAL FAILURE
C
C MERGE TWO SORTED FILES (MAIL1 (FTN20) AND MAIL2 (FTN21))
C INTO A THIRD FILE (MAIL3 (FTN22))
C
C ESTABLISH KEYS FOR SORT - MAJOR AT 11 FOR 9 BYTES
C (LAST NAME) AND MINOR AT 1 FOR 10 BYTES (FIRST NAME)
C
      KEYS(1)=11
      KEYS(2)=9
      KEYS(3)=0
      KEYS(4)=1
      KEYS(5)=10
      KEYS(6)=0
C
C ESTABLISH MPE/3000 FILENUMBERS FOR INPUT FILES
C (MAIL1 AND MAIL2) BY REFERENCING FNUM PROCEDURE
C
      INFILES(1)=FNUM(20)
      INFILES(2)=FNUM(21)
C
C CALL MERGE PROCEDURE
C
      CALL MERGE(\2\,INFILES,\FNUM(22)\,\0\,\2\,KEYS,
     #\0\,\0\,\0\,\0\,\0\,FAILURE,\%7301\)
      IF(FAILURE)STOP 10
C
C READ AND WRITE OUTPUT FILE (FORTRAN UNIT NO. 22)
C
      REWIND 22
 20   READ(22,END=30)BUFFER
      WRITE(6,100)BUFFER[1:54]
      GO TO 20
C
C THE FOLLOWING STATEMENTS CALL THE COMPILER LIBRARY
C PROCEDURE UNITCONTROL TO CLOSE FILES FTN20,
C FTN21, AND FTN22
C
 30   CALL UNITCONTROL(20,8)
      CALL UNITCONTROL(21,8)
      CALL UNITCONTROL(22,8)
      STOP
      END


 ****      GLOBAL STATISTICS        ****
 ****   NO ERRORS,   NO WARNINGS    ****
 TOTAL COMPILATION TIME  0:00:01
 TOTAL ELAPSED TIME      0:00:05


  END OF COMPILE
```

Figure 8-7. FNUM and UNITCONTROL Example

```
:BUILD PROGTEST;CODE=PROG
:FILE FTN20=MAIL1,OLD
:FILE FTN21=MAIL2,OLD
:FILE FTN22=MAIL3,NEW
:PREP $OLDPASS,PROGTEST;MAXDATA=4000

 END OF PREPARE
:RUN PROGTEST


PLAINS    ANTELOPE 201 OPENSPACE AVE   BIGCOUNTRY  WY
LOIS      ANYONE   6190 COURT ST       METROPOLIS  NY
KING      ARTHUR   329 EXCALIBUR ST    CAMELOT     CA
ALI       BABA     40 THIEVES WAY      SESAME      CO
BLACK     BEAR     47 ALLOVER DR       ANYWHERE    US
JOHN      BIGTOWN  965 APPIAN WAY      METROPOLIS  NY
KNEE      BUCKLER  974 FISTICUFF DR    PUGILIST    ND
SWASH     BUCKLER  497 PLAYACTING CT   MOVIETOWN   CA
ANIMAL    CRACKERS 1000 ANYWHERE PL    ALLOVER     US
MULE      DEER     963 FOREST PL       HIGHCOUNTRY CA
WHITETAIL DEER     34 WOODSY PL        BACKCOUNTRY ME
JAMES     DOE      4193 ANY ST         ANYTOWN     MD
JANE      DOE      3959 TREEWOOD LN    BIGTOWN     MA
PRAIRIE   DOG      493 ROLLINGHILLS DR OPENSPACE   ND
JOHN      DOUGHE   239 MAIN ST         HOMETOWN    MA
MALLARD   DUCK     79 MARSH PL         PUDDLEDUCK  CA
JENNA     GRANDTR  493 TWENTIETH ST    PROGRESSIVE CA
KARISSA   GRANDTR  7917 BROADMOOR WAY  BIGTOWN     MA
SNOWSHOE  HARE     742 FRIGID WAY      COLDSPOT    MN
MOUNTAIN  LION     796 KING DR         THICKET     NM
SPACE     MANN     9999 GALAXY WAY     UNIVERSE    CA
SWAMP     RABBIT   4444 DAMPLACE RD    BAYOU       LO
NASTY     RATTLER  243 DANGER AVE      DESERTVILLE CA
BIGHORN   SHEEP    999 MOUNTAIN DR     HIGHPLACE   CO
GREY      SQUIRREL 432 PLEASANT DR     FALLCOLORS  MA
 END OF PROGRAM
```

Figure 8-8. Output Resulting from FNUM and UNITCONTROL Example

WRITING COMPILER COMMANDS IN FORTRAN/3000

SECTION IX

9-1. COMPILER COMMANDS

Compiler commands are inserted in a source program to inform the compiler that an output listing is required, that statements in the program are in free-field format, or that various other compilation options are in effect.

The FORTRAN/3000 compiler is accessed with an MPE/3000 :FORTRAN, :FORTPREP, or :FORTGO command. See Section XII for a description of these commands and of the files (*textfile, uslfile, listfile, masterfile,* and *newfile*) associated with the commands.

Table 9-1 contains a summary of compiler commands and their parameters. The following paragraphs describe the commands (and parameters).

The basic form of a compiler command is

$*command parameter list*

For example,

$\overset{\textit{command}}{\text{\$CONTROL}}\ \overbrace{\text{CODE, NOLIST}}^{\textit{parameter list}}$

The $ sign must be the first character in the line and must be followed immediately by the command name (i.e., CONTROL in the example above). In free-field format the $ sign must be preceded by a blank. If more than one parameter is included in *parameter list*, the parameters must be separated from each other by commas. Blanks may be inserted freely between parameters in the list.

A command line can be continued for as many as 19 additional lines if the last non-blank character in each line to be continued is an ampersand (&) and the first character of each continuation line is $. Words (command names or parameters) must not be broken by the $.

9-2. $CONTROL COMMAND

The $CONTROL command specifies compilation options during source program compilation.

The form of a $CONTROL command is

$CONTROL *parameter list*

For example,

$\text{\$CONTROL}\ \underbrace{\text{MAP, LIST, CODE}}_{\textit{parameter list}}$

An example of a $CONTROL command with the parameters CODE, LABEL, and MAP is shown in figure 9-1.

9-3. BOUNDS PARAMETER. The BOUNDS parameter of the $CONTROL command has the form $CONTROL BOUNDS and can appear at the beginning of program units only.

The BOUNDS parameter requests the compiler to generate code which will dynamically validate array bounds. The compiler will check fixed-bound arrays, adjustable-bound arrays, and dummy arrays by verifying the information contained in DIMENSION declaration statements against any dynamic attributes associated with the arrays during execution.

It should be noted that this parameter generates a substantial amount of code which must be executed each time an array is referenced.

The BOUNDS parameter is cleared by default if not specified.

9-4. CODE/NOCODE PARAMETERS. The CODE/NOCODE parameters have the form $CONTROL CODE or $CONTROL NOCODE and can appear anywhere in a program unit. These parameters take effect at the end of the program unit.

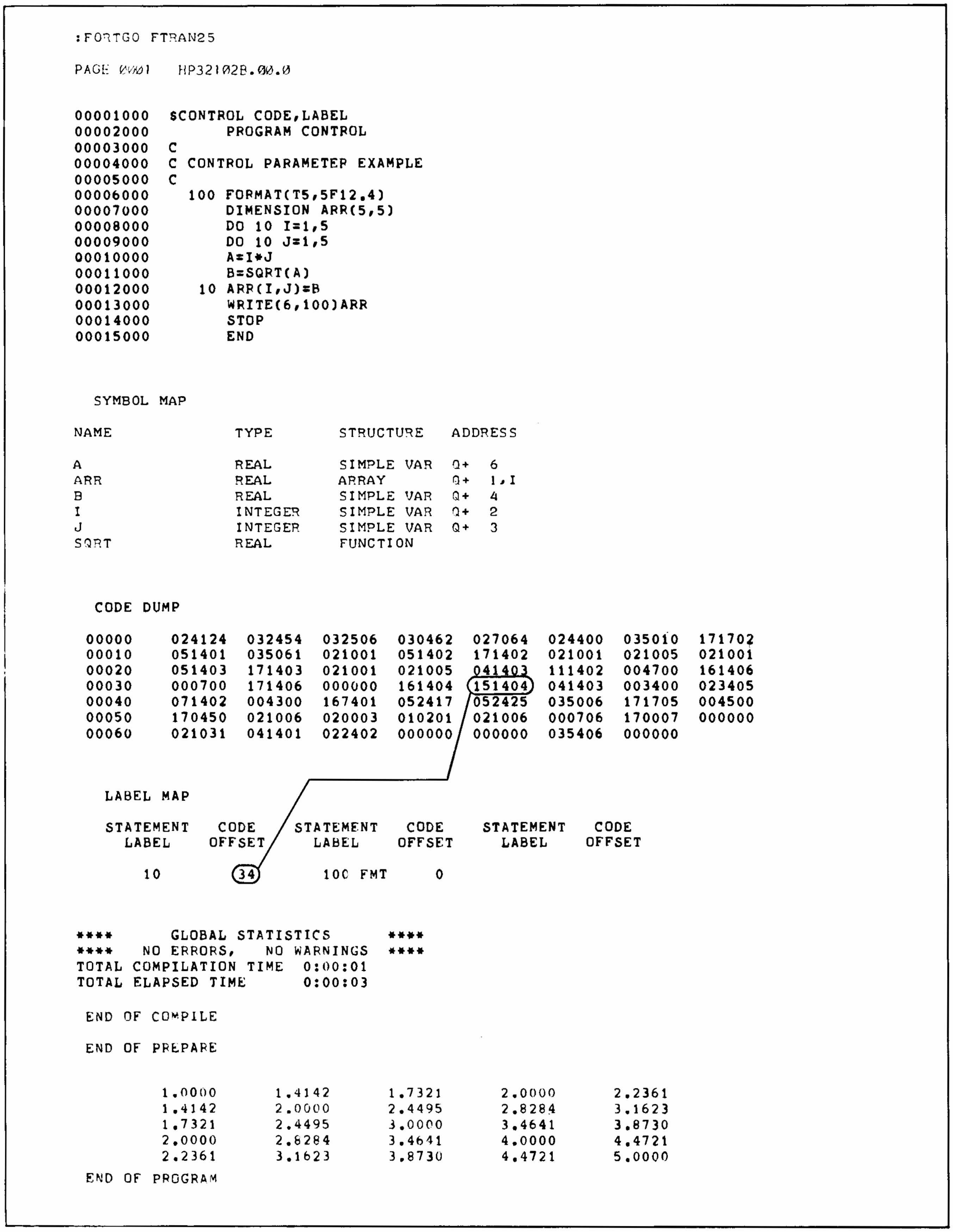
The CODE parameter causes an octal listing of the machine instruction code which has been generated to be output to the user's *listfile* (see figure 9-1 for the code dump).

The NOCODE parameter clears the CODE parameter. If the CODE parameter is not specified, the default is NOCODE.

9-5. CHECK PARAMETER. The CHECK parameter has the form $CONTROL CHECK=$n$ (where n=0,1,2,3), and can appear at the beginning of program units only.

Table 9-1. Summary of Compiler Commands

COMMAND	FORM	PARAMETERS	DEFAULT
$CONTROL	$CONTROL *parameter list*	BOUNDS	Clear
		CHECK= *number*	CHECK= 3
		CODE (NOCODE)	NOCODE
		CROSSREF	No crossref
		CROSSREF ALL	No crossref
		ERRORS= *number*	50 severe errors
		FILE= *number*	
		FILE= *number – number*	
		FIXED	FIXED
		FREE	
		INIT	Clear
		LABEL (NOLABEL)	NOLABEL
		LIST (NOLIST)	LIST
		LOCATION	NOLOCATION
		MAP (NOMAP)	NOMAP
		SEGMENT= *name*	SEG
		SOURCE (NOSOURCE)	SOURCE (Batch) NOSOURCE (Interactive)
		STAT (NOSTAT)	NOSTAT
		USLINIT	Clear
		WARN (NOWARN)	WARN
		MORECOM	CLEAR
$EDIT	$EDIT *parameter list*	FIXED	
		FREE	
		INC= *number*	
		NOSEQ	
		SEQNUM = *sequence number*	
		VOID = *sequence number*	
$IF	$IF X*n* = ON / OFF		
$INTEGER*4	$INTEGER*4		
$PAGE	$PAGE or $PAGE *character string list*		
$SET	$SET X0 = ON/OFF, SET X1 = ON/OFF, . . ., SET X*n* = ON/OFF or $SET X0 = ON/OFF, X1 = ON/OFF, ..., XN = ON/OFF		
$TITLE	$TITLE *character string list*		
$TRACE	$TRACE *program unit; identifier, identifier, . . . ,identifier*		



```
:FORTGO FTRAN25

PAGE 0001   HP32102B.00.0


00001000  $CONTROL CODE,LABEL
00002000        PROGRAM CONTROL
00003000  C
00004000  C CONTROL PARAMETER EXAMPLE
00005000  C
00006000    100 FORMAT(T5,5F12.4)
00007000        DIMENSION ARR(5,5)
00008000        DO 10 I=1,5
00009000        DO 10 J=1,5
00010000        A=I*J
00011000        B=SQRT(A)
00012000     10 ARR(I,J)=B
00013000        WRITE(6,100)ARR
00014000        STOP
00015000        END


  SYMBOL MAP

NAME              TYPE        STRUCTURE   ADDRESS

A                 REAL        SIMPLE VAR  Q+  6
ARR               REAL        ARRAY       Q+  1,I
B                 REAL        SIMPLE VAR  Q+  4
I                 INTEGER     SIMPLE VAR  Q+  2
J                 INTEGER     SIMPLE VAR  Q+  3
SQRT              REAL        FUNCTION


  CODE DUMP

 00000    024124  032454  032506  030462  027064  024400  035010  171702
 00010    051401  035061  021001  051402  171402  021001  021005  021001
 00020    051403  171403  021001  021005  041403  111402  004700  161406
 00030    000700  171406  000000  161404  151404  041403  003400  023405
 00040    071402  004300  167401  052417  052425  035006  171705  004500
 00050    170450  021006  020003  010201  021006  000706  170007  000000
 00060    021031  041401  022402  000000  000000  035406  000000


   LABEL MAP

   STATEMENT   CODE    STATEMENT   CODE    STATEMENT   CODE
     LABEL    OFFSET     LABEL    OFFSET     LABEL    OFFSET

       10       34       100 FMT     0


****       GLOBAL STATISTICS       ****
****    NO ERRORS,    NO WARNINGS  ****
TOTAL COMPILATION TIME   0:00:01
TOTAL ELAPSED TIME       0:00:03

 END OF COMPILE

 END OF PREPARE

         1.0000      1.4142      1.7321      2.0000      2.2361
         1.4142      2.0000      2.4495      2.8284      3.1623
         1.7321      2.4495      3.0000      3.4641      3.8730
         2.0000      2.8284      3.4641      4.0000      4.4721
         2.2361      3.1623      3.8730      4.4721      5.0000
 END OF PROGRAM
```

Figure 9-1. $CONTROL Compiler Command Example

The CHECK parameter allows you to set the level of checking of all calls *to* the subroutine it immediately precedes. If not specified, it is set to 3 by default. This parameter does not affect the checking level of calls to other subroutines *from* this one. The checking level for FORTRAN program units is always established by the called program unit.

The significance of each level is as follows:

- 0 No checking.
- 1 Checking of function type.
- 2 Checking of function type and number of parameters.
- 3 Checking of function type, number of parameters, parameter type and parameter structure. (See paragraph 11-3.)

9-6. CROSSREF PARAMETER. The CROSSREF parameter of the $CONTROL command has the form $CONTROL CROSSREF or $CONTROL CROSSREF ALL.

The CROSSREF parameter requests the compiler to provide a cross reference listing. At the end of each program unit, a listing of all symbolic names, in alphabetic order, and all statement labels will be made with all sequence numbers. This listing will appear immediately following each name or label. Except as noted below, this option is effective only at the beginning of a program unit and is only effective for the program unit it precedes. The default is no cross reference. This option is disabled if the LOCATION parameter is also used and an error exists in the program unit.

The use of "ALL" specifies that a CROSSREF is desired for all program units in the program. In addition, at the end of the entire compilation, a listing will be produced, cross referencing common block names, program units, and references to program units. This option is only effective at the beginning of a compilation and supersedes any CROSSREF requests at the beginning of each program unit.

9-7. ERRORS PARAMETER. The ERRORS parameter has the form $CONTROL ERRORS = *n* (where *n* is between 0 and 999) and can appear anywhere in a program unit. The ERRORS parameter takes effect immediately.

The ERRORS parameter sets a maximum of severe errors allowable before the compiler terminates compilation of the program unit. This parameter is useful for correcting long programs which may have many errors; the compiler will output all error messages in one pass. These errors then can be corrected and the program resubmitted for compilation.

If omitted, the compiler sets the maximum number of severe errors at 50 by default for each program unit.

9-8. FILE PARAMETER. The FILE parameter has the form $CONTROL FILE = n or $CONTROL FILE = $n_1 - n_n$, or $CONTROL FILE = n_1, FILE = n_2, . . . , FILE = n_n (where *n* is between 1 and 99) and can appear anywhere in a program unit. The form $CONTROL FILE = $n_1 - n_n$ is equivalent to the form $CONTROL FILE = n_1, FILE = n_2, ... FILE = n_n. That is, a range may be specified.

The compiler command $CONTROL FILE=*n* in a program defines the file for the FORTRAN Logical Unit Table (FLUT, see Section VIII). The FLUT is used by the FORTRAN/3000 Formatter subsystem for file system access during program execution. If a file is referenced as a constant FORTRAN unit number in an input/output statement, it need not be included as a FILE parameter in a $CONTROL command. However, if it is referenced only as a variable, it must be included.

Figure 9-2 contains an example of the FILE parameter. The FORTRAN unit number in the READ statement is referenced by the variable IPU, and the FORTRAN unit number in the WRITE statement by the constant 6. Only FORTRAN unit number 20, the value assigned to IPU, need be included as a FILE parameter $CONTROL command. The MPE :FILE command equates the formal file designator FTN20 to the actual designator MAILLIST. (See the *MPE Commands Reference Manual* for a discussion of the :FILE command.) When the READ statement is executed, file MAILLIST is read.

See Section VIII for a further discussion of the FORTRAN/3000 file facility.

9-9. FIXED PARAMETER. The FIXED parameter has the form $CONTROL FIXED and can appear anywhere in a program unit. The FIXED parameter takes immediate effect.

The FIXED parameter specifies that subsequent statements in the program unit are in fixed-field format (see Section I). The FIXED parameter is set by default unless changed by the FREE parameter.

9-10. FREE PARAMETER. The FREE parameter has the form $CONTROL FREE and can appear anywhere in a program unit. The FREE parameter takes immediate effect.

The FREE parameter informs the compiler that subsequent statements are in the free-field format (see Section I). The FREE parameter clears the FIXED parameter.

9-11. INIT PARAMETER. The INIT parameter has the form $CONTROL INIT and can appear only at the beginning of program units. It applies only to the program unit which immediately follows it.

The INIT parameter requests the compiler to generate code to initialize all local variables for the program unit.

Arithmetic variables are initialized to zero, logical variables to .FALSE., and character variables to all null characters.

The $CONTROL INIT command initializes each word of a character variable to %000000, which represents two null characters. This is different from putting blanks (or spaces) in the character variable, since a word containing 2 blanks is specified by %020040.

The INIT parameter is honored only at the beginning of program units and is cleared by default if not specified.

```
:FILE FTN2Ø=MAILLIST,OLD
:FORTGO FTRAN26

 PAGE ØØØ1    HP321Ø2B.ØØ.Ø


ØØØØ1ØØØ  $CONTROL FILE=2Ø
ØØØØ2ØØØ        PROGRAM FILE
ØØØØ3ØØØ  C
ØØØØ4ØØØ  C     CONTROL FILE PARAMETER EXAMPLE
ØØØØ5ØØØ  C
ØØØØ6ØØØ  1ØØ   FORMAT(T5,S)
ØØØØ7ØØØ        CHARACTER*19 NAME
ØØØØ8ØØØ        IPU=2Ø
ØØØØ9ØØØ  1Ø    READ(IPU,END=2ØØ)NAME
ØØØ1ØØØØ        WRITE(6,1ØØ)NAME[1:19]
ØØØ11ØØØ        GO TO 1Ø
ØØØ12ØØØ  2ØØ   STOP
ØØØ13ØØØ        END



****       GLOBAL STATISTICS       ****
****    NO ERRORS,    NO WARNINGS   ****
TOTAL COMPILATION TIME  Ø:ØØ:Ø1
TOTAL ELAPSED TIME      Ø:ØØ:19

 END OF COMPILE

 END OF PREPARE


   JOHN       BIGTOWN
   LOIS       ANYONE
   ALI        BABA
   JAMES      DOE
   JANE       DOE
   JOHN       DOUGHE
   SPACE      MANN
   KING       ARTHUR
   JENNA      GRANDTR
   KARISSA    GRANDTR
   SWASH      BUCKLER
   KNEE       BUCKLER
 END OF PROGRAM
```

Figure 9-2. $CONTROL FILE Parameter Example

```
:FILE FTN2Ø=MAILLIST,OLD
:FORTGO FTRAN25

PAGE ØØØ1    HP321Ø2B.ØØ.Ø


ØØØØ1ØØØ  $CONTROL FILE=2Ø,LOCATION
ØØØ1Ø   ØØØØ2ØØØ          PROGRAM FILE
ØØØ1Ø   ØØØØ3ØØØ   C
ØØØ1Ø   ØØØØ4ØØØ   C      CONTROL FILE PARAMETER EXAMPLE
ØØØ1Ø   ØØØØ5ØØØ   C
ØØØ1Ø   ØØØØ6ØØØ   1ØØ    FORMAT(T5,S)
ØØØ1Ø   ØØØØ7ØØØ          CHARACTER*19 NAME
ØØØ1Ø   ØØØØ8ØØØ          IPU=2Ø
ØØØ12   ØØØØ9ØØØ   1Ø     READ(IPU,END=2ØØ)NAME
ØØØ27   ØØØ1ØØØØ          WRITE(6,1ØØ)NAME[1:19]
ØØØ5Ø   ØØØ11ØØØ          GO TO 1Ø
ØØØ51   ØØØ12ØØØ   2ØØ    STOP
ØØØ52   ØØØ13ØØØ          END



****       GLOBAL STATISTICS       ****
****    NO ERRORS,    NO WARNINGS   ****
TOTAL COMPILATION TIME   Ø:ØØ:Ø1
TOTAL ELAPSED TIME       Ø:ØØ:25

 END OF COMPILE

 END OF PREPARE


   JOHN        BIGTOWN
   LOIS        ANYONE
   ALI         BABA
   JAMES       DOE
   JANE        DOE
   JOHN        DOUGHE
   SPACE       MANN
   KING        ARTHUR
   JENNA       GRANDTR
   KARISSA     GRANDTR
   SWASH       BUCKLER
   KNEE        BUCKLER
 END OF PROGRAM
```

Figure 9-3. $CONTROL FILE LOCATION Parameter Example

9-12. LABEL/NOLABEL PARAMETERS. The LABEL/NOLABEL parameters have the form $CONTROL LABEL or $CONTROL NOLABEL and can appear anywhere in a program unit. These parameters take effect at the end of the program unit. See figure 9-1 for a label map example. All statement labels are printed, along with a CODE OFFSET column. The code offset listing informs the user where the first machine instruction for that statement begins. For example, statement number 10 has a code offset of 35 (octal), which means that the first machine instruction for this statement is at location 35 (octal) in the label map (see the circled instruction in figure 9-1). The label map is helpful when correcting a program, in that it allows the user to observe the machine instructions for any labelled statement. The NOLABEL parameter clears the LABEL parameter (if it is on). The NOLABEL parameter is set by default.

9-13. LIST/NOLIST PARAMETERS. The LIST/NOLIST parameters have the form $CONTROL LIST or $CONTROL NOLIST and can appear anywhere in a program unit. These parameters take immediate effect.

The LIST parameter is set by default and all compilations will produce listings of all statements, symbol maps (if specified), label maps (if specified), etc., during compilation. If the NOLIST parameter is set, however, *no* listings will occur during compilation.

9-14. LOCATION/NOLOCATION PARAMETERS. The LOCATION parameter of the $CONTROL command has the form $CONTROL LOCATION, and can appear anywhere in the program unit.

The LOCATION parameter requests the compiler to list the P-relative address offset from the beginning of a program unit for each statement. This feature is useful with certain modes of debugging programs.

This option may appear only before a program unit and will be in effect until reset by NOLOCATION.

The P-relative address within the program unit will be listed on the left side of the listing. The default is NOLOCATION. This feature is disabled by errors because no code is generated. If the LOCATION parameter is used, any existing error(s) also disable the CROSSREF and LABEL options.

9-15. MAP/NOMAP PARAMETERS. The MAP/NOMAP parameters have the form $CONTROL MAP or $CONTROL NOMAP and can appear anywhere in a program unit. These parameters take effect at the end of the program unit.

If the MAP parameter is included in the compiler command, the output listing shows all the symbolic names and their structures, such as simple variable, array, function, subroutine and so forth. The listing also shows the various *types*, such as integer, character, or real. The addresses are listed in octal form. Character variables do not always start on word boundaries and therefore are referred to by their byte addresses. All other variables have word addresses. Figure 9-1 presents a listing of real and integer variables. Character variables appear in the symbol map shown in Figure 9-4. Variables which are grouped in a common block are referred to by addresses relative to the beginning of the common block.

The NOMAP parameter clears the MAP parameter (if it is on). The NOMAP parameter is set by default.

9-16. SEGMENT PARAMETER. The SEGMENT parameter has the form $CONTROL SEGMENT = *name*, where *name* can be any symbolic name (up to 15 characters). The SEGMENT parameter can appear only at the beginning of a program unit. The name is reset to SEG' at the end of each program unit.

By using the SEGMENT parameter, different segment names can be assigned to program units. When the complete source program is compiled, all program units having the same segment name are compiled into one segment. The various segments then are combined into one complete executable program. (All program units having no segment name are compiled into the segment containing the default name SEG'.)

See the *MPE Intrinsics Reference Manual* for a complete discussion of segmentation.

9-17. SOURCE/NOSOURCE PARAMETERS. The SOURCE/NOSOURCE parameters have the form $CONTROL SOURCE or $CONTROL NOSOURCE and can appear anywhere in a program unit. These parameters take immediate effect.

The SOURCE parameter is set by default in all the cases except in direct interactive mode when $STDIN and $STDLIST are same. Refer to figure 12-1 where $STDIN is the source file and is not listed during compilation.

The SOURCE parameter causes all source program statements to be listed during compilation.

If the NOSOURCE parameter is specified, the source program statements are not listed during compilation, but symbol maps, label maps, etc. (if specified) are listed. This parameter differs from the NOLIST parameter (see paragraph 9-13) which suppresses *all* listings.

9-18. STAT/NOSTAT PARAMETERS. The STAT/NOSTAT parameters have the form $CONTROL STAT or $CONTROL NOSTAT and can appear anywhere in a program unit. These parameters take immediate effect. Their function is to control the listing of compilation time statistics for each program unit.

If STAT and LIST are both defined, the following is listed on $STDLIST and the list file (if a list file is specified).

1. Number of errors and warnings.
2. Name and disposition of program unit.
3. Segment name if the program unit was successfully compiled.

```
:FORTRAN MAP1

PAGE ØØØ1    HP321Ø2B.ØØ


ØØØØ1ØØØ  $CONTROL MAP
ØØØØ2ØØØ        PROGRAM MAP
ØØØØ3ØØØ  C
ØØØØ4ØØØ  C CONTROL MAP PARAMETER EXAMPLE
ØØØØ5ØØØ  C
ØØØØ6ØØØ        CHARACTER*15 FIRSTNAME,LASTNAME
ØØØØ7ØØØ        CHARACTER*7 HEADING
ØØØØ8ØØØ        INTEGER ADULTS,CHILDREN,MEMBERSHIP
ØØØØ85ØØ        EXTERNAL COMPUTEPRICE
ØØØØ9ØØØ        COMMON FIRSTNAME,LASTNAME,MEMBERSHIP
ØØØ1ØØØØ        DISPLAY "THIS ROUTINE SHALL DETERMINE THE TOTAL"
ØØØ11ØØØ        DISPLAY "PRICE OF YOUR CONCERT OR DANCE TICKETS"
ØØØ12ØØØ        DISPLAY "PLEASE ENTER 'CONCERT' OR 'DANCE'"
ØØØ13ØØØ        ACCEPT HEADING
ØØØ14ØØØ        DISPLAY "LASTNAME?"
ØØØ15ØØØ        ACCEPT LASTNAME
ØØØ16ØØØ        DISPLAY "MEMBERSHIP?"
ØØØ161ØØ        ACCEPT MEMBERSHIP
ØØØ162ØØ        DISPLAY "NUMBER OF ADULTS?"
ØØØ163ØØ        ACCEPT ADULTS
ØØØ164ØØ        DISPLAY "NUMBER OF CHILDREN?"
ØØØ165ØØ        ACCEPT CHILDREN
ØØØ166ØØ        DISPLAY HEADING,"TICKETS"
ØØØ19ØØØ        CALL COMPUTEPRICE(ADULTS,CHILDREN)
ØØØ2ØØØØ        STOP
ØØØ21ØØØ        END


  SYMBOL MAP

NAME                TYPE        STRUCTURE    ADDRESS

ADULTS              INTEGER     SIMPLE VAR   Q+%1
CHILDREN            INTEGER     SIMPLE VAR   Q+%2
COMPUTEPRICE                    SUBROUTINE
FIRSTNAME           CHARACTER   SIMPLE VAR     %Ø      COMMON
HEADING             CHARACTER   SIMPLE VAR   Q+%3    ,I
LASTNAME            CHARACTER   SIMPLE VAR     %17     COMMON
MEMBERSHIP          INTEGER     SIMPLE VAR     %17     COMMON


  COMMON BLOCKS

NAME                LENGTH

COM'                   16


PROGRAM UNIT MAP COMPILED


****       GLOBAL STATISTICS       ****
****    NO ERRORS,    NO WARNINGS  ****
TOTAL COMPILATION TIME  Ø:ØØ:Ø1
TOTAL ELAPSED TIME      Ø:Ø1:11

END OF COMPILE
```

Figure 9-4. $CONTROL MAP Parameter Example.

4. Amount of code generated.
5. A rough estimate of the fixed stack required by the program unit.
6. CPU time.
7. Elapsed time.

If NOSTAT or NOLIST is defined, and a list file is specified, the following is listed on the list file:

1. Number of errors and warnings, if any.
2. Name and disposition of program unit.

If any errors or warnings were detected, then the above also is listed on $STDLIST.

If NOSTAT or NOLIST is defined, and a list file is not specified, then nothing is listed unless an error or warning is detected. In that case the following is listed:

1. Number of errors and warnings, if any.
2. Name and disposition of program unit.

NOSTAT is the default option

9-19. USLINIT PARAMETER. The USLINIT parameter initializes the *uslfile* (see Section XII) by deleting all existing contents. This ensures that the file receiving the output from the compiler is initialized to an empty condition before compilation starts. The USLINIT parameter must appear before the start of the first program unit (if it is used). Otherwise it is ignored.

9-20. WARN/NOWARN PARAMETERS. The WARN/NOWARN parameters have the form $CONTROL WARN or $CONTROL NOWARN and can appear anywhere in a program unit. These parameters take immediate effect.

The WARN parameter is set by default and causes the compiler to list all warning messages during compilation.

The NOWARN parameter clears the WARN parameter and suppresses all warning messages.

9-20A MORECOM PARAMETER

As noted in paragraph 5-9, the maximum number of pointers allowed for referencing the variables (simple variables and/or arrays) in COMMON is 254. This limit exists because only 254 variable cells are available for separate pointers in the primary DB area of memory. If your program exceeds this limit, the segmenter prints

```
TOO MANY DATA LABELS
```

In such a situation, you may override this limit by using the MORECOM parameter of the $CONTROL command to provide an alternative method for addressing COMMON variables. This method can increase COMMON storage up to 254 COMMON blocks, with one primary DB location assigned to each block. This removes the restriction on the number of variables, but also decreases the program's efficiency because it produces more code. When the $CONTROL MORECOM is used, the primary limiting factor on the number of variables allowed is the size of the stack. $CONTROL MORECOM can appear only at the beginning of the source program. However, if multiple compilations are combined in the same USL file, this parameter must appear in the beginning of the main program and each subprogram unit. Sometimes it is possible to have more than 254 COMMON variables without using the MORECOM parameter. See Appendix F for explanation.

9-21. $PAGE COMMAND

The $PAGE command is used to start a new page for the listing output to *listfile.*

The form of a $PAGE command is

$PAGE

or

$PAGE *character string list*

For example,

$PAGE "MIDDLE OF THE PROGRAM"

Normally, the *listfile* will start a new page based on the number of lines that have been printed on the current page. The $PAGE command, however, may appear anywhere in a source program and will cause the output listing to start on a new page.

Thus, the $PAGE command ejects the current page of *listfile* to the top of the next page and prints a heading (if one has been included in *character string list*) followed by two blank lines.

If the optional *character string list* is included in the command, a title will be printed for the new page and all subsequent pages until changed by another $PAGE command or a $TITLE command (see paragraph 9-22).

Character string list consists of one or more character strings separated by commas (each string is enclosed in quotation marks). When the list is printed, the quote marks, separating commas and any blanks between strings are deleted and the character strings are concatenated and placed in the heading. One character string in the list cannot be continued from one line to the next, but a new individual string in the list can be started on a continuation line. Any line to be continued must end with an & and the new line must begin with a $.

No page eject occurs if the NOLIST parameter of the $CONTROL command is specified since no listing will be printed. No page eject takes place if the $PAGE command is within the range of an unsatisfied $IF command (see paragraph 9-24).

9-22. $TITLE COMMAND

The $TITLE command is used to print a page heading whenever the top of the page is encountered in *listfile*.

The form of a $TITLE command is

$TITLE ***character string list***

For example,

$TITLE "TODAY IS 2/26/75"

The *character string list* is used for subsequent page headings until another $TITLE or $PAGE command is encountered. If no *character string list* is included with a $TITLE command, the command has no effect.

Character string list is the same as that defined for the $PAGE command in paragraph 9-21.

9-23. $INTEGER*4 COMMAND

The $INTEGER*4 command is used to cause the compiler to treat explicit and implicit INTEGER types as 32-bit integers instead of as 16-bit integers.

The form of a $INTEGER*4 command is

$INTEGER*4

When this compiler command is used, it must appear before the first source statement.

Double integers (INTEGER*4) may appear in any specification statement in which single integers may appear. All decimal integer constants will be treated as 32-bit constants. In this case, the use of INTEGER*2 is available for explicit typing, or in an IMPLICIT statement to force variables to be 16-bit integers. Composite, ASCII, and octal constants still require the trailing "J". The following intrinsics will accept and/or return 32-bit integers in place of their normal 16-bit integer values: BOOL,MOD,IDIM,IABS,IFIX,INT,FLOAT,MAX0,MAX1, MIN0,MIN1,AMAX0,AMIN0,ISIGN,IDINT.

Note: A double integer may appear anywhere an integer may appear. However, using double integer as loop parameters in a DO statement is considerably less efficient than using single integers.

It is important to note that $INTEGER*4 forces all decimal integers to double integers. Therefore, it may be necessary to force parameters of external procedures back to single integer for correct operation. Intrinsic IJINT will perform this function. Examples of external procedures for which this must be done are FNUM, UNITCONTROL, and FOPEN (unless these are defined in a system intrinsic statement).

9-24. $SET AND $IF COMMANDS

It is possible to set conditions in a FORTRAN/3000 source program which will cause certain portions of the program to be ignored (except for listing) during compilation. This function is helpful for unusually long programs or for programs which use several subprograms. (It may be desirable to compile and correct portions of the program. The $SET and $IF commands can be used for this purpose.)

The form of a $SET command is

$$\text{\$SET X1} = \begin{Bmatrix}\text{ON}\\\text{OFF}\end{Bmatrix}, \text{SET X2} = \begin{Bmatrix}\text{ON}\\\text{OFF}\end{Bmatrix}, \ldots, \text{SET X}_n = \begin{Bmatrix}\text{ON}\\\text{OFF}\end{Bmatrix}$$

or

$$\text{\$SET X1} = \begin{Bmatrix}\text{ON}\\\text{OFF}\end{Bmatrix}, \text{X2} = \begin{Bmatrix}\text{ON}\\\text{OFF}\end{Bmatrix}, \ldots, \text{X}_n = \begin{Bmatrix}\text{ON}\\\text{OFF}\end{Bmatrix}$$

where *n* varies from 0 to 9, inclusive.

Note: { } denotes that one of the parameters, *but not both*, must be included.

The $SET command sets or clears condition *n*.. If more than one parameter is specified, each parameter must be separated from the next by a comma. All conditions begin the compilation in the cleared (OFF) state.

The form of a $IF command is

$$\text{\$IF X}_n = \begin{Bmatrix}\text{ON}\\\text{OFF}\end{Bmatrix}$$

where *n* varies from 0 to 9, inclusive.

The $SET command *sets* the condition and the $IF command *checks* the condition. If the condition specified by the $IF command is true, the succeeding source program statements (until the next $IF command) are compiled.

If the specified condition is false, the succeeding source program statements are ignored by the compiler (except for listing) until the next $IF command is encountered. The only command not ignored is the $EDIT command (see paragraph 9-25).

An $IF command with no parameters merely terminates the preceding $IF command. Its form is $IF.

9-25. $EDIT COMMAND

The $EDIT command provides the following editing capabilities:

- Merging correction statements (*textfile*) with an old master source program (*masterfile*) to produce a new program source file (*newfile*) for compilation.
- Checking source record sequence numbers for ascending order.
- Bypassing sections of source programs.
- Renumbering source record sequence numbers.

The form of a $EDIT command is

$EDIT ***parameter list***

where any of the following *parameters* may be specified:

VOID = *sequence number*
SEQNUM = *sequence number*
NOSEQ
INC = *number*
FIXED
FREE

For example,

```
$EDIT VOID = 100, SEQNUM = 90
```

Use of the $EDIT command depends on the parameters specified (see paragraph 9-26) and the files specified in the MPE/3000 :FORTRAN command (see Section XII).

If the :FORTRAN command specifies both *masterfile* and *textfile*, source statements from both files are merged, and the statement sequence numbers are checked for ascending order. Each sequence number in columns 73 to 80 of the record unit must either be all blank or greater than the previous sequence number. In merging *masterfile* with *textfile,* one record is read from each file and their sequence numbers are compared. The record with the lower sequence number is compiled. If *newfile* was specified, this record is passed to *newfile*. If the sequence numbers are identical, the record from *textfile* is compiled and passed to *newfile* (if *newfile* was specified in the :FORTRAN command). Sequence numbers can be specified as numeric only.

By default, records sent to *newfile* are sent with unchanged sequence numbers. To renumber sequence numbers, use the SEQNUM parameter of the $EDIT command. Sequence numbers are checked by the compiler for proper order only if *masterfile* is specified in the :FORTRAN command. If *textfile* is specified and *masterfile* is not specified, sequence numbers are not checked by the compiler.

The first line of a $EDIT command can contain a sequence number to indicate placement in the *textfile*, but continuation lines must have blank sequence fields.

9-26. $EDIT COMMAND PARAMETERS. $EDIT command parameters and their meanings appear in the following paragraphs.

The VOID parameter form is

$EDIT VOID = ***sequence number***

When the VOID parameter appears in a $EDIT command, the compiler bypasses all *masterfile* records with a *sequence number* less than or equal to the *sequence number* in the VOID parameter. The *sequence number* can be specified either as a number or as a character string. If the number is less than eight digits, the compiler left-fills the number with zeros to achieve eight digits. If a character string is used and it is less than eight characters, the compiler strips the quote marks and left-fills the field with blank characters to achieve eight characters.

The form for the SEQNUM parameter is

$EDIT SEQNUM = ***sequence number***

The SEQNUM parameter renumbers succeeding source records sent to *newfile* starting with the *sequence number* specified in SEQNUM. If the INC parameter (see below) is specified, each subsequent source record *sequence number* is incremented by the value associated with INC. If INC is not specified, *sequence number* is incremented by the default value 1000 for each succeeding record.

The form for the NOSEQ parameter is

$EDIT NOSEQ

The NOSEQ parameter indicates that succeeding sequence numbers retain their current sequence numbers. If SEQNUM = *sequence number* is not specified, the NOSEQ condition occurs regardless of whether NOSEQ is specified.

The form for the INC parameter is

$EDIT INC = ***number***

where *number* indicates the value by which each source record *sequence number is* incremented when the SEQNUM parameter is specified. INC is ignored if *newfile* is not specified in the :FORTRAN command or if the last SEQNUM parameter was overridden by a NOSEQ parameter.

The form for the FIXED parameter is

$EDIT FIXED

The FIXED parameter following a $EDIT command informs the compiler that the source records in the *textfile* are in fixed-field format (sequence field is located in columns 73 through 80 of the source record).

The form for the FREE parameter is

$EDIT FREE

The FREE parameter informs the compiler that the source records in the *textfile* are in free-field format (sequence field is located in columns 1 through 8 of the source record).

When a record is read from the *textfile*, the computer must locate the sequence field to determine when the record is to be merged with the *masterfile*. At the beginning of compilation or after a $EDIT command specifying FIXED, the compiler takes characters 73 through 80 of the record as the sequence field. Following a $EDIT command specifying FREE, the compiler takes the sequence field to be the first character of the record up to (but not including) the first blank. The compiler uses only the last eight characters of this field and prefixes the sequence string with ASCII zero characters if the field is less than eight. A blank in column one indicates an all blank sequence field.

When a record from the *textfile* is merged with the *masterfile*, if the mode (FIXED or FREE) is the same as the mode of the record in the *masterfile*, then the *textfile* record is used as is. If the mode differs, the *textfile* record is converted to the *masterfile* mode. A line read as FIXED is converted to FREE (if the *masterfile* mode is FREE) by moving the sequence field (columns 73 through 80) to the beginning of the line, inserting a blank following the sequence field, and following that with the characters originally in columns 1 through 71 of the line. Column 72 of the line is lost since free-field records contain a maximum of 71 characters following the sequence field.

A line read as FREE is converted to FIXED (if the *masterfile* mode is FIXED) by moving the last eight characters of the sequence field (remember the blank is not part of the field) to columns 73 through 80.

Note that statements and comments in *textfile* which are to be converted should be written in the mode of the *masterfile* except for the sequence field. For example, comment lines merged into a fixed-field program must use the letter C (in the appropriate place in the line) instead of #, which is the sign for a comment line in free-field mode.

9-27. $TRACE COMMAND

The $TRACE command specifies variables, arrays, labels, and other program elements (*identifiers*) to be monitored by the HP 3000 Symbol Trace program (TRACE/3000) during program execution. For further information on the TRACE/3000 program, consult the *Trace/3000 Reference Manual.*

The form of the $TRACE command is

$TRACE ***program unit ;identifier,***
   identifier, . . . , identifier

where

program unit
is the name of the program unit to which the *identifiers* belong. If *program unit* is omitted from the command, the compiler uses the name MAIN'.

identifier
can be a variable within the referenced *program unit*, or an array, a statement label, or a subroutine or function subprogram within the referenced *program unit.*

For example,

```
$TRACE SUBX; ARRAY, AJAX,A,B
```

The compiler will generate code to invoke the TRACE subsystem within the *program unit* whenever it encounters a line of code with ARRAY, AJAX, A or B in it.

$TRACE commands must appear before the first FORTRAN/3000 statement of the *program unit* referenced in the $TRACE command. For a multiprogram unit compilation, $TRACE commands can appear before any program unit preceding the affected one,thus allowing all $TRACE commands to be grouped before the first program unit in the multiunit compilation.

FORTRAN/3000 INTRINSIC FUNCTIONS AND BASIC EXTERNAL FUNCTIONS

SECTION X

FORTRAN/3000 program units can call *FORTRAN/3000 intrinsic functions* and *basic external functions* by referencing these functions in FORTRAN/3000 statements.

A reference to a FORTRAN/3000 *intrinsic function* generates code to perform an indicated function (such as converting a real value to an integer value) or generates a call to a procedure (written in SPL/3000) to perform the function.

Note: FORTRAN/3000 intrinsic functions can be called from FORTRAN/3000 programs only and should not be confused with HP 3000 *system intrinsics* (such as FOPEN, ASCII, etc.) which can be called from FORTRAN/3000 and other programming languages. See the *MPE Intrinsics Reference Manual* for definitions of HP 3000 system intrinsics.

Basic external functions are written in SPL/3000 and can be called by FORTRAN/3000 program units by referencing the name of the function.

10-1. FUNCTION REFERENCES

A function reference is a symbolic name from one to 15 alphanumeric characters (the first of which must be alphabetic) followed by a list of arguments. The symbolic name references a computational process (defined elsewhere) which is designed to return a value to the function symbolic name.

The form of a function reference is

name (param,param,. . .,param)

where

name
is the symbolic name of the FORTRAN/3000 intrinsic function or basic external function.

param
is a variable name, constant, array name, array element, function subprogram name (see Section XI), subroutine subprogram name (see Section XI), Hollerith constant, or expression. (See Section XI for a discussion of actual and dummy arguments.)

An example of a function reference is as follows:

A = SQRT(6.5)

The above statement calls the basic external function SQRT to compute the square root of 6.5, which is the *param*, or actual argument, passed to the function.

A function reference returns a specific value of the type associated with the function and is equivalent in usage to a variable reference of the same type. In the previous example, a real number is returned as the value of SQRT(6.5). When a function reference is encountered during the evaluation of an expression (e.g., A = SQRT(6.5)), control is passed to the referenced function. The function is executed using the actual arguments listed in the function reference. The function name is assigned a value and passed back to the referencing expression, which continues its evaluation.

10-2. FORTRAN/3000 INTRINSIC FUNCTIONS

A FORTRAN/3000 intrinsic function is a computational process which performs an operation such as converting an integer value to a real value. To perform this operation, the intrinsic function may be generated code or it may generate a call to a procedure (already written). In either case, the process is invisible to the user and the intrinsic can be used merely by writing a function reference (along with the appropriate arguments). FORTRAN/3000 intrinsic functions are predefined to the FORTRAN/3000 compiler and the function reference will call the correct process.

For example,

J = IFIX(A)

converts the value of variable A from real to integer.

A list of all FORTRAN/3000 intrinsics and their uses and arguments is presented in table 10-1.

10-3. BASIC EXTERNAL FUNCTIONS

Some basic computational procedures (such as taking the square root of a number) are defined in FORTRAN/3000 as basic external functions. These functions are written in SPL/3000 but can be referenced and used by FORTRAN/3000. To use a basic external function, the function name, along with the appropriate arguments, must appear in a function reference.

For example,

```
A = SQRT(B)
```

where SQRT is the name of the basic external function for computing the square root of a value and B is the value (or argument) for which the square root will be computed.

Basic external functions available to FORTRAN/3000 are shown in table 10-2. Also included are the definitions of the functions, the number of arguments allowed for each function, the types of arguments that are allowed, and the function types.

To use a basic external function of type other than real or integer, the function must be declared in a Type statement or an IMPLICIT statement in the calling program.

For example, if it were necessary to use the basic external function DSQRT (which is type double precision) to compute the square root of A, then the function (DSQRT) and the argument to be passed to it (A) must both be declared in a Type or IMPLICIT statement, as follows:

```
DOUBLE PRECISION DSQRT,A
```

or

```
IMPLICIT DOUBLE PRECISION (A,D)
```

The last statement informs the compiler that all variables beginning with the letters A and D are double precision, thus DSQRT is typed as double precision.

When the basic external function is referenced, the actual arguments passed to it are checked for proper type. The basic external function type as defined is associated with the result.

A function subprogram (see Section XI) can be defined with the same name as a basic external function. For example, you may write your own procedure to compute square root, name it SQRT, and reference it in your programs. The new function takes the place of the system-defined basic external function in the program unit which defined the new function and in any other program unit which references it.

10-4. GENERIC FUNCTIONS

Generic names simplify the referencing of intrinsic and basic external functions because the same name may be used with more than one type argument.

The result of the function is determined by the type of argument(s), as shown in table 10-3. If a particular function, such as MOD, requires more than one argument, then all arguments in a particular call must have the same type.

For example, the following is illegal:

```
INTEGER I
REAL A
I=MOD(I,A)
```

If a generic name appears as a dummy argument, then that name does not identify an intrinsic function in that program unit.

If a *specific* name that is also a *generic* name appears in a Type statement confirming the type of the specific function, the function is available for referencing in the program unit only with the type of argument required with the specific name of the function. That is, the *generic* property of that function is disabled.

For example, the following is illegal:

```
INTEGER MOD
REAL A,B
A=MOD(A,B)
```

If a Type statement specifies some other type for a specific function name, that name is not available for referencing that pre-defined function. For example, redefinition of IABS as real will make the pre-defined function IABS unavailable, although ABS will still be generic for real and double precision numbers. Stated simply, redefinition in any way of a generic function name results in the loss of the generic property of that name. Changing the type of a specific function name results in the inability to reference that pre-defined function either directly or generically.

Note: If the main program references a function subprogram, a statement function, or a basic external function, the type information of the respective functions must be included either explicitly or implicitly; otherwise an error results during program execution. The type information is not necessary when generic functions or FORTRAN/3000 intrinsic functions are referenced. Refer to Section A-4 for information on HP 3000 system intrinsics.

Table 10-1. FORTRAN/3000 Intrinsic Functions

INTRINSIC FUNCTION	DEFINITION	NUMBER OF ARGUMENTS	FUNCTION REFERENCE	TYPE OF ARGUMENT	TYPE OF FUNCTION
Absolute Value	$\lvert a \rvert$	1	ABS*(a)*	Real	Real
			IABS*(a)*	Integer	Integer
			JABS*(a)*	Double Integer	Double Integer
			DABS*(a)*	Double Precision	Double Precision
Truncation	sign of a times largest integer $\leq \lvert a \rvert$	1	AINT*(a)*	Real	Real
			INT*(a)*	Real or Logical	Integer
			IJINT*(a)*	Double Integer	Integer
			JINT*(a)*	Real	Double Integer
			JIINT*(a)*	Integer	Double Integer
			IDINT*(a)*	Double Precision	Integer
			JDINT*(a)*	Double Precision	Double Integer
			DDINT*(a)*	Double Precision	Double Precision
Remaindering	a_1 *(mod* a_2*)*	2	AMOD(a_1, a_2)	Real	Real
			MOD(a_1, a_2)	Integer	Integer
			JMOD(a_1, a_2)	Double Integer	Double Integer
Choosing largest value	MAX $(a_1, a_2, \ldots)$	at least 2	AMAX0 $(a_1, a_2, \ldots, a_n)$	Integer	Real
			AMAX1 $(a_1, a_2, \ldots, a_n)$	Real	Real
			MAX0 $(a_1, a_2, \ldots, a_n)$	Integer	Integer
			MAX1 $(a_1, a_2, \ldots, a_n)$	Real	Integer
			DMAX1 $(a_1, a_2, \ldots, a_n)$	Double	Double
			AJMAX0	Double Integer	Real
			JMAX0 $(a_1, a_2, \ldots, a_n)$	Double Integer	Double Integer
			JMAX1 $(a_1, a_2, \ldots, a_n)$	Real	Double Integer

Table 10-1. FORTRAN/3000 Intrinsic Functions (Continued)

INTRINSIC FUNCTION	DEFINITION	NUMBER OF ARGUMENTS	FUNCTION REFERENCE	TYPE OF ARGUMENT	TYPE OF FUNCTION
Choosing smallest value	Min $(a_1, a_2, \ldots)$	at least 2	AMIN0 $(a_1, a_2, \ldots, a_n)$	Integer	Real
			AMIN1 $(a_1, a_2, \ldots, a_n)$	Real	Real
			MIN0 $(a_1, a_2, \ldots, a_n)$	Integer	Integer
			MIN1 $(a_1, a_2, \ldots, a_n)$	Real	Integer
			DMIN1 $(a_1, a_2, \ldots, a_n)$	Double Precision	Double Precision
			AJMIN0 $(a_1, a_2, \ldots, a_n)$	Double Integer	Real
			JMIN0 $(a_1, a_2, \ldots, a_n)$	Double Integer	Double Integer
			JMIN1 $(a_1, a_2, \ldots, a_n)$	Real	Double Integer
Float	Conversion from integer to real	1	FLOAT *(a)*	Integer	Real
			FLOATJ (a_1)	Double Integer	Real
Fix	Conversion from real to integer	1	IFIX*(a)*	Real	Integer
			JFIX*(a)*	Real	Double Integer
Transfer of Sign	Sign of a_2 times $\|a_1\|$	2	SIGN(a_1, a_2)	Real	Real
			ISIGN(a_1, a_2)	Integer	Integer
			DSIGN(a_1, a_2)	Double Precision	Double Precision
			JSIGN(a_1, a_2)	Double Integer	Double Integer
Positive Difference	a_1 − Min(a_1, a_2)	2	DIM(a_1, a_2)	Real	Real
			IDIM(a_1, a_2)	Integer	Integer
			JDIM(a_1, a_2)	Double Integer	Double Integer
Obtain most significant part of double precision argument		1	SNGL*(a)*	Double Precision	Real
Obtain real part of complex argument		1	REAL*(a)*	Complex	Real
Obtain imaginary part of complex argument		1	AIMAG*(a)*	Complex	Real

Table 10-1. FORTRAN/3000 Intrinsic Functions (Continued)

INTRINSIC FUNCTION	DEFINITION	NUMBER OF ARGUMENTS	FUNCTION REFERENCE	TYPE OF ARGUMENT	TYPE OF FUNCTION
Express single precision argument in double precision form		1	DBLE(a)	Real	Double Precision
Express two real arguments in complex form	$a_1 + a_2 \sqrt{-1}$	2	CMPLX(a_1, a_2)	Real	Complex
Obtain conjugate of a complex argument		1	CONJG(a)	Complex	Complex
Obtain the position of the character in the first argument which begins the substring which matches the second argument	INDEX	2	INDEX(a_1, a_2)	Character	Integer
Convert character expression to integer (INUM), double integer (JNUM), real (RNUM), or double precision (DNUM)	INUM	1	INUM(a)	Character	Integer
	JNUM	1	JNUM(a)	Character	Double Integer
	RNUM	1	RNUM(a)	Character	Real
	DNUM	1	DNUM(a)	Character	Double Precision
Convert an arithmetic expression (a_1) of any type except complex to a string of length a_2. a_2 must be integer constant.	STR If a_1 is: — Conversion format is: Integer — Ia_2 Double Integer — Ia_2 Real — Ga_2.6 Double Precision — Ga_2.17	2	STR(a_1, a_2)	Integer, Real, Double Precision, or Double Integer	Character
Convert integer expression to type logical	BOOL		BOOL(a)	Integer	Logical

Table 10-2. Basic External Functions

BASIC EXTERNAL FUNCTION	DEFINITION	NUMBER OF ARGUMENTS	FUNCTION REFERENCE	TYPE OF ARGUMENT	TYPE OF FUNCTION
Exponential	e^a	1	EXP(*a*)	Real	Real
		1	DEXP(*a*)	Double Precision	Double Precision
		1	CEXP(*a*)	Complex	Complex
Natural Logarithm	Log_e (*a*)	1	ALOG(*a*)	Real	Real
		1	DLOG(*a*)	Double Precision	Double Precision
		1	CLOG(*a*)	Complex	Complex
Common Logarithm	Log_{10} (*a*)	1	ALOG10(*a*)	Real	Real
		1	DLOG10(*a*)	Double Precision	Double Precision
Trigonometric Sine	Sin (*a*)	1	SIN(*a*)	Real	Real
		1	DSIN(*a*)	Double Precision	Double Precision
		1	CSIN(*a*)	Complex	Complex
Trigonometric Cosine	Cos (*a*)	1	COS(*a*)	Real	Real
		1	DCOS(*a*)	Double Precision	Double Precision
		1	CCOS(*a*)	Complex	Complex
Trigonometric Tangent	Tan (*a*)	1	TAN(*a*)	Real	Real
			DTAN(*a*)	Double Precision	Double Precision
			CTAN(*a*)	Complex	Complex
Square root	$(a)^{1/2}$	1	SQRT(*a*)	Real	Real
		1	DSQRT(*a*)	Double Precision	Double Precision
		1	CSQRT(*a*)	Complex	Complex
Arctangent	Arctan (*a*)	1	ATAN(*a*)	Real	Real
		1	DATAN(*a*)	Double Precision	Double Precision
	Arctan (a_1/a_2)	2	ATAN2(a_1, a_2)	Real	Real
		2	DATAN2(a_1, a_2)	Double Precision	Double Precision
Remaindering	a_1 (*mod* a_2)	2	DMOD(a_1, a_2)	Double Precision	Double Precision
Modulus		1	CABS(*a*)	Complex	Real
Hyperbolic Sine	Sinh (*a*)	1	SINH(*a*)	Real	Real
		1	DSINH(*a*)	Double Precision	Double Precision
		1	CSINH(*a*)	Complex	Complex
Hyperbolic Cosine	Cosh (*a*)	1	COSH(*a*)	Real	Real
		1	DCOSH(*a*)	Double Precision	Double Precision
		1	CCOSH(*a*)	Complex	Complex
Hyperbolic Tangent	**Tanh (*a*)**	1	TANH (*a*)	Real	Real
		1	DTANH(*a*)	Double Precision	Double Precision
		1	CTANH(*a*)	Complex	Complex

Table 10-3. List of Generic Functions

GENERIC FUNCTION	GENERIC NAME	SPECIFIC NAME	TYPE OF ARGUMENT	TYPE OF RESULT
Absolute value	ABS	ABS	Real	Real
		IABS	Integer	Integer
		DABS	Double Precision	Double Precision
		CABS	Complex	Complex
		JABS	Double Integer	Double Integer
Truncation to integer if $INTEGER*4 is not invoked	INT	INT	Real or Logical	Integer
		IFIX	Real	Integer
		IDINT	Double Precision	Integer
		IJINT	Double Integer	Integer
Truncation to integer if $INTEGER*4 is invoked		JINT	Real	Double Integer
		JDINT	Double Precision	Double Integer
		JFIX	Real	Double Integer
		JIINT	Integer	Double Integer
Conversion to double integer	JINT	JINT	Real	Double Integer
		JDINT	Double Precision	Double Integer
		JFIX	Real	Double Integer
		JIINT	Integer	Double Integer
Conversion of data item to real	REAL	FLOAT	Integer	Real
		SNGL	Double Precision	Real
		REAL	Complex	Real
		FLOATJ	Double Integer	Real
Remaindering	MOD	MOD	Integer	Integer
		AMOD	Real	Real
		DMOD	Double Precision	Double Precision
		JMOD	Double Integer	Double Integer
Sign Transfer	SIGN	SIGN	Real	Real
		ISIGN	Integer	Integer
		DSIGN	Double Precision	Double Precision
		JSIGN	Double Integer	Double Integer
Difference in magnitude	DIM	DIM	Real	Real
		IDIM	Integer	Integer
		JDIM	Double Integer	Double Integer
Select largest value from a list	MAX	MAX0	Integer	Integer
		AMAX1	Real	Real
		DMAX1	Double Precision	Double Precision
		JMAX0	Double Integer	Double Integer

Table 10-3. List of Generic Functions (Continued)

GENERIC FUNCTION	GENERIC NAME	SPECIFIC NAME	TYPE OF ARGUMENT	TYPE OF RESULT
Select smallest value from a list	MIN	MIN0	Integer	Integer
		AMIN1	Real	Real
		DMIN1	Double Precision	Double Precision
		JMIN0	Double Integer	Double Integer
Square Root	SQRT	SQRT	Real	Real
		DSQRT	Double Precision	Double Precision
		CSQRT	Complex	Complex
Exponential	EXP	EXP	Real	Real
		DEXP	Double Precision	Double Precision
		CEXP	Complex	Complex
Natural Logarithm	LOG	ALOG	Real	Real
		DLOG	Double Precision	Double Precision
		CLOG	Complex	Complex
Trignometric Sine	SIN	SIN	Real	Real
		DSIN	Double Precision	Double Precision
		CSIN	Complex	Complex
Trignometric Cosine	COS	COS	Real	Real
		DCOS	Double Precision	Double Precision
		CCOS	Complex	Complex
Trignometric Tangent	TAN	TAN	Real	Real
		DTAN	Double Precision	Double Precision
		CTAN	Complex	Complex
Arc Tangent	ATAN	ATAN	Real	Real
		DATAN	Double Precision	Double Precision
		ATAN2	Real	Real
		DATAN2	Double Precision	Double Precision
Hyperbolic Sine	SINH	SINH	Real	Real
		DSINH	Double Precision	Double Precision
		CSINH	Complex	Complex
Hyperbolic Cosine	COSH	COSH	Real	Real
		DCOSH	Double Precision	Double Precision
		CCOSH	Complex	Complex
Hyperbolic Tangent	TANH	TANH	Real	Real
		DTANH	Double Precision	Double Precision
		CTANH	Complex	Complex

WRITING MAIN PROGRAMS AND SUBPROGRAMS IN FORTRAN/3000

SECTION XI

An executable FORTRAN/3000 program consists of program *units* made up of a *main program* and any necessary *subprograms. Subprograms* are *subroutine* subprograms, *function* subprograms, or *block data* subprograms (written in FORTRAN/3000 and compiled by the FORTRAN/3000 compiler), or *procedure* subprograms (written in a language other than FORTRAN/3000, usually SPL/3000 (Systems Programming Language for the HP 3000) and compiled by the appropriate compiler) which can be called from a FORTRAN/3000 program. This section discusses the writing of main programs and subprograms and Section XII explains how to compile and execute these programs on the HP 3000 Computer System.

11-1. MAIN PROGRAMS

A main program consists of any necessary non-executable statements (such as FORMAT and declaration statements), one or more executable statements (assignment, control, or input/output), and an END statement. The main program can be assigned a symbolic name by using a PROGRAM statement as the first line of the program. The PROGRAM statement has the form

```
PROGRAM name
```

where

name
is an alphanumeric string from one to fifteen characters (the first character must be a letter).

Any main program not headed by a PROGRAM statement is assigned the special name MAIN' by the FORTRAN/3000 compiler.

An example of a short main program is shown in figure 11-1.

11-2. SUBPROGRAMS

As discussed earlier, there are four basic types of subprograms which can be called from FORTRAN/3000 program units:

- *Subroutine* subprograms, which may perform a computation and pass one or more values back to the calling program, or a subroutine may perform a computational process such as a sort.
- *Function* subprograms, which perform a computation and return a value through its *name*, and may pass one or more values back to the calling program through its arguments.
- *Block data* subprograms, which provide initial values for simple variables or array elements in labeled common blocks.
- *Procedure* subprograms, which are written in a language other than FORTRAN/3000 (usually SPL/3000) and can be called from a FORTRAN/3000 program.

11-3. DUMMY AND ACTUAL ARGUMENT CHARACTERISTICS

In the manipulation of subprograms, the calling program unit may pass arguments to the subprogram to be used in the computation.

The arguments passed by the calling program are called *actual* arguments. The subprogram, which is structured with *dummy* arguments, uses the actual arguments passed to it to replace the dummy arguments and perform the computation.

For example,

```
A = 6.5

B = 8.3

M = RFUNC(A,B) * 3.14159
```

When the call is made to the function subprogram, the addresses of the actual values A and B are passed to the subprogram. The subprogram could be as follows:

```
FUNCTION RFUNC(C,D)

RFUNC = (C ** 2) + (D ** 3)

RETURN

END
```

The subprogram uses the addresses of A and B(passed to it from the calling program) to indirectly reference these variables. Thus, in effect, the variables C and D, which are dummy arguments, assume the values of the actual arguments (A and B) used in the referencing expression (in this case, 6.5 for C and 8.3 for D).

```
:FORTGO FTRAN27

PAGE 0001   HP32102B.00.0


                 PROGRAM MAIN
      C
      C MAIN PROGRAM EXAMPLE
      C
       100   FORMAT('0',T10,"X",T15,"X SQUARED",T28,"X CUBED"//)
       200   FORMAT(T9,I2,T18,I3,T30,I4)
             WRITE(6,100)
             DO 10 I=1,10
             K=I**2
             L=I**3
       10    WRITE(6,200)I,K,L
             STOP
             END



****      GLOBAL STATISTICS       ****
****   NO ERRORS,   NO WARNINGS   ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:00:04

 END OF COMPILE

 END OF PREPARE               *



         X    X SQUARED    X CUBED


         1        1           1
         2        4           8
         3        9          27
         4       16          64
         5       25         125
         6       36         216
         7       49         343
         8       64         512
         9       81         729
        10      100        1000
 END OF PROGRAM
```

Figure 11-1. Main Program Example

Actual arguments in a subroutine call or function reference must agree in number, order, type (INTEGER, REAL, and so forth) and structure with the dummy arguments they replace. The structure must be either a simple variable, array, function or subroutine name. (A constant is treated as a simple variable.) One exception to the above rule is that an array element, for example, A(4), used as an actual argument may correspond to a dummy argument that is a simple variable or an array.

Note: A *function* is called *implicitly* by being referenced in a FORTRAN/3000 statement. A *subroutine* is called *explicitly* through the use of a CALL statement (CALL statements are described in Section IV). A further difference between a *function* and a *subroutine* is that a function performs a computation and can return values to the calling program through its arguments and it can assign a resulting *single* value to the function *name*, which then is passed back to the calling program unit. A *subroutine*, on the other hand, cannot assign a computed value to its name. Instead, it may return *one* or *more* values to the calling program unit through actual arguments supplied in the CALL statement, or it may perform some other computational process, such as sorting a list of characters, which define, or redefine, many values.

Within subprograms, dummy arguments may consist of simple variables, array names, subroutine names, or function names. All variable names are local to the program unit which defined them, and, similarly, dummy arguments are local to the subprogram unit or statement function containing them. Thus, they can be the same as names appearing elsewhere in another program unit. No element of a dummy argument list can occur in a COMMON, EQUIVALENCE, or DATA statement. When an array name is used as a dummy argument, the dummy array name must be dimensioned in a DIMENSION or Type statement within the body of the subprogram.

In subroutine subprograms, a list of asterisks (*) may follow the list of dummy arguments. When the subprogram is called using actual arguments, statement labels (prefixed by $'s) are substituted for the asterisks to indicate optional return points in the calling program. The method of choosing optional return points is described in paragraph 11-4. The types of actual arguments which may appear in subroutine calls or function references are shown below, with examples.

Constants: CALL SUBR(6.43,9.56)

Variable names: CALL SUBR(A,B)

Array names: CALL SUBR(ARR1, ARR2)

Array elements: CALL SUBR(ARR(1,2), ARR(9,9))

Expressions: CALL SUBR(X+Y,X/Y)

Subroutine names: CALL SUBR(SUB1,SUB2)

Function subprogram names: CALL SUBR (FUNC1,FUNC2)

Note: When subroutine or function subprogram names are used as actual arguments, they must be declared in an EXTERNAL statement (see Section V) or be followed by empty parentheses.

When a subroutine name is used as an actual argument, it does not pass a value as do other actual arguments. Instead, it passes the *name* itself to the referenced subroutine or function subprogram.

A statement function name (see paragraph 11-7) or intrinsic function name (see Section X) cannot be used as an actual argument, although an expression used as an actual argument can contain a reference to them as part of the expression.

For example,

CALL SUM(A,B,ABS) *not allowed*

CALL SUM(A,B,ABS(A)) *allowed*

11-4. SUBROUTINE SUBPROGRAMS

A subroutine is a computational procedure which may pass one or more values back to the calling program unit or perform some other type of computational process such as a sort.

The first statement of a subroutine subprogram must be a SUBROUTINE statement of the form

SUBROUTINE ***name***

or

SUBROUTINE ***name (param, param,. . .,param)***

or

SUBROUTINE name (param,param,. . ., param,*,*,. . .,*)

where

name
is an alphanumeric string from one to fifteen characters (the first character must be alphabetic). No type is associated with the name of a subroutine.

param

is a dummy argument of the subroutine. *Param* can be a simple variable, array name, subroutine name, or function subprogram name. The dummy argument must be the same type and structure as the actual argument which was passed to it. (See paragraph 11-3.)

the asterisks are used to show that alternate return points exist. In the CALL statement (see Section IV) in the main or other calling program, the asterisks are replaced by statement labels. The subroutine will return to one of these labeled statements depending upon the results of an evaluation in the subroutine.

Examples of SUBROUTINE statements are:

```
SUBROUTINE LARGE

SUBROUTINE LARGE (A,B)

SUBROUTINE LARGE (A,B,*,*,*)
```

A subroutine subprogram can contain any statement except another SUBROUTINE statement, or FUNCTION, PROGRAM, or BLOCK DATA statements. A subroutine is *re-entrant*, that is, it can contain a CALL statement which references, or calls, itself either directly or indirectly. (Care should be taken, of course, to ensure that this type of call does not result in an endless loop.)

The last line of a subroutine must be an END statement. One or more RETURN statements can be included to return control to the calling program unit. If no RETURN statement is included in the subroutine, the END statement will return control to the calling program unit. It is recommended, however, that the RETURN statement be used for this purpose.

An example of a main program and subroutine subprograms is shown in figure 11-2.

When statement 20 in the main program is executed, it calls SUM and passes the array names X and Y, the simple variable Z, and the name MIN to the SUM subroutine subprogram.

In the subroutine SUM, statement 10 substitutes the name MIN for MINMAX and calls subroutine MIN. Subroutine MIN compares two elements, one from each array, and returns the smaller value to SUM. When the DO-loop in SUM has been executed ten times, the ten smallest values of each of the ten pairs have been summed and this value is returned to the main program through W.

When statement 30 in the main program is executed, the name MAX is passed to SUM. This time SUM calls MAX and the ten largest values of each of the ten pairs of X and Y are summed.

11-5. FUNCTION SUBPROGRAMS

A function subprogram is a computational procedure which returns a single value through its name. As with subroutine subprograms, however, a function subprogram also may return one or more values through its arguments.

The first statement of a function subprogram must be a FUNCTION statement as follows:

FUNCTION *name (param,param,. . .,param)*

or

type* FUNCTION *name (param, param,. . .,param)

where

name
is an alphanumeric character string from one to fifteen characters (the first character must be alphabetic).

param
a dummy argument of the function. It can be a simple variable, array name, subroutine name, or a function subprogram name. The dummy argument must be the same type and structure as the actual argument that is passed to it.

type
either LOGICAL, INTEGER, INTEGER*2, INTEGER*4, REAL, DOUBLE PRECISION, COMPLEX or CHARACTER**n* (where *n* is a positive integer constant specifying the length of the character function value. If **n* is omitted, the length of the character function is assumed to be 1).

Examples of FUNCTION statements are:

```
FUNCTION IDIV(A,B,C)

REAL FUNCTION IDIV(A,B,C)

CHARACTER *10 FUNCTION NEXTWORD(B,C)
```

The type associated with the function name is determined in one of three ways:

1. If the type is mentioned as the first part of the FUNCTION statement, the function name is assigned that type.

2. If the type is not mentioned in the FUNCTION statement, the function name can be mentioned in a Type statement within the function subprogram.

```
:FORTGO FTRAN28

PAGE ØØØ1    HP321Ø2B.ØØ.Ø

      PROGRAM SUBROUTINE
C
C SUBROUTINE SUBPROGRAM EXAMPLE
C
 1ØØ  FORMAT(T5,"THE SUM OF THE 1Ø SMALLEST VALUES IS ",F14.5)
 2ØØ  FORMAT(T5,"THE SUM OF THE 1Ø LARGEST VALUES IS ",F14.5)
      DIMENSION X(1Ø),Y(1Ø)
C
C THE NEXT STATEMENT IS AN EXTERNAL STATEMENT
C AND DECLARES THAT THE ARGUMENTS MIN AND MAX
C ARE SUBROUTINE NAMES TO BE USED AS ACTUAL ARGUMENTS
C
      EXTERNAL MIN,MAX
      DO 1Ø I=1,1Ø
      X(I)=I*3.14159
 1Ø   Y(11-I)=X(I)
 2Ø   CALL SUM(X,Y,Z,MIN)
      WRITE(6,1ØØ)Z
 3Ø   CALL SUM(X,Y,Z,MAX)
      WRITE(6,2ØØ)Z
      STOP
      END



      SUBROUTINE SUM(A,B,S,MINMAX)
      DIMENSION A(1Ø),B(1Ø)
      S=Ø
      DO 2Ø I=1,1Ø
 1Ø   CALL MINMAX(A(I),B(I),W)
 2Ø   S=S+W
      RETURN
      END



      SUBROUTINE MIN(D,E,F)
      F=D
      IF(D.GT.E)F=E
      RETURN
      END



      SUBROUTINE MAX(P,Q,R)
      R=P
      IF(P.LT.Q)R=Q
      RETURN
      END


****       GLOBAL STATISTICS       ****
****    NO ERRORS,    NO WARNINGS  ****
TOTAL COMPILATION TIME  Ø:ØØ:Ø1
TOTAL ELAPSED TIME      Ø:ØØ:Ø4

  END OF COMPILE

  END OF PREPARE


    THE SUM OF THE 1Ø SMALLEST VALUES IS        94.24773
    THE SUM OF THE 1Ø LARGEST VALUES IS        251.32721
  END OF PROGRAM
```

Figure 11-2. Subroutine Subprogram Example

3. If the function name is not mentioned in a Type statement or the type is not included in the FUNCTION statement itself, the type is assigned implicitly according to the first letter of the name. Names starting with I, J, K, L, M or N are type integer, and names starting with any other letter are type real. (This convention may be modified by an IMPLICIT statement, see Section V.)

To associate a value with the function subprogram name, the name must be used within the function subprogram as a simple variable in one or more of the following ways:

1. The left side of an assignment statement.
2. An element of an input list in a READ statement.
3. An actual parameter of a function or subroutine subprogram. Examples of these three methods of determining a value for a function subprogram name are as follows:

```
      FUNCTION DIV(A,B)

      DIV = A/B

      RETURN

      END


      FUNCTION DIV(A,B)

  100 FORMAT(F12.5)

      READ(5,100)DIV(A,B)

      RETURN

      END


      FUNCTION DIV(A,B)

      DIMENSION A(10,10)

      DO 10 I = 1,10

      DO 10 J = 1,10

   10 A(I,J) = I*J

      CALL SUM(A,DIV)

      RETURN

      END
```

The value last assigned to the name of the function at the time a RETURN statement is executed within the subprogram is the value retained by the function name.

Note: The RETURN statement form RETURN *n*, where *n* is an integer constant or simple variable, is not allowed in function subprograms.

A function subprogram may contain a direct or indirect reference to itself (recursive call).

An example of a calling program unit and a function subprogram is shown in figure 11-3.

The function subprogram DISP is called implicitly (a CALL statement is not used) by the main program. The actual arguments C, D, E, A, B, and T are passed to DISP.

The function subprogram computes a value for the displacement of the vehicle's engine using the values passed to it by the actual arguments C, D, and E and assigns this value to the function name DISP.

Transportation charges are computed using actual arguments A (WEIGHT) and B (DISTANCE). This value is assigned to TRANSCHG and then passed back to the calling program through T.

Note that the function subprogram computes a value which is assigned to its name *and* computes and passes another value back through the argument T.

Whenever a function subprogram is defined, at least one parameter must be included in the declaration.

Fig. 11-2A references the Function Subprogram AFFIRM without any parameter. The function name fails to be typed Integer in the symbolic map. It is typed Real by default, even though it is defined as an integer. In Fig. 11-2B, the parameter IDUM is included and hence the function name is typed Integer.

11-6. MULTIPLE ENTRY POINTS

Multiple entry points allow you to begin execution of a subprogram at different locations. The name specified in a FUNCTION or SUBROUTINE statement is the *primary* entry point, and the names used in ENTRY statements are *secondary* entry points. In each case, calling a subprogram results in the transfer of control to the first executable statement following the entry name (primary or secondary) by which the subprogram was called.

The form of an ENTRY statement is

ENTRY *entryname*
or ENTRY *entryname (parameter name, parameter name, . . . ,parameter name)*

where

entryname
is an alphanumeric character string from one to fifteen characters (the first character must be alphabetic).

parameter name
is a dummy argument of the function. It can be a simple variable, array name, subroutine name, or a function subprogram name. If it is necessary to type these parameters, they should be included in the type statements which follow the primary entry point.

```
PAGE 0001   HP32102B.00.08  FORTRAN/3000  (C) HEWLETT-PACKARD CO. 1976  TUE, APR 11, 1978,  3:53 PM

          $CONTROL SEGMENT=IMN1,MAP
          C
          C
                LOGICAL FUNCTION AFFIRM
                INTEGER READLINE
                CHARACTER ANSWR(5),ANSR
                EQUIVALENCE (ANSWR,ANSR)
                ANSWR(1)="N"
                IDUM=READLINE (ANSWR,-5)
                AFFIRM=ANSR .EQ. "Y" .OR. ANSR .EQ. "Y"
                             -
*** ERROR  155 ***  NON-ARITHMETIC PRIMARY WHERE ARITHMETIC EXPECTED
                RETURN
                END


  SYMBOL MAP

NAME             TYPE       STRUCTURE   ADDRESS                NAME              TYPE       STRUCTURE   ADDRESS

AFFIRM           REAL       SIMPLE VAR                         ANSR              CHARACTER  SIMPLE VAR
ANSWR            CHARACTER  ARRAY                              FUNCTIONAFFIRM    LOGICAL    NULL
IDUM             INTEGER    SIMPLE VAR                         READLINE          INTEGER    FUNCTION


****   1 ERROR,    NO WARNINGS   ****
PROGRAM UNIT MAIN' FLUSHED
****       GLOBAL STATISTICS       ****
****     1 ERROR,    NO WARNINGS   ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:00:03
```

Figure 11-2a. Typed Real — Without Parameter

```
PAGE 0001   HP32102B.00.08  FORTRAN/3000  (C) HEWLETT-PACKARD CO. 1976  TUE, APR 11, 1978,  4:14 PM

          $CONTROL SEGMENT=IMN1,MAP
          C
          C
                LOGICAL FUNCTION AFFIRM (IDUM)
          C
                INTEGER READLINE
                CHARACTER ANSWR(5),ANSR
                EQUIVALENCE (ANSWR,ANSR)
          C
                ANSWR(1)="N"
                IDUM = READLINE (ANSWR,-5)
                AFFIRM =ANSR .EQ. "Y" .OR. ANSR .EQ. "Y"
                RETURN
                END


  SYMBOL MAP

NAME             TYPE       STRUCTURE   ADDRESS                NAME              TYPE       STRUCTURE   ADDRESS

AFFIRM           LOGICAL    SIMPLE VAR  Q-%5                   AFFIRM            LOGICAL    FUNCTION
ANSR             CHARACTER  SIMPLE VAR  Q+%2   ,I              ANSWR             CHARACTER  ARRAY       Q+%1   ,I
IDUM             INTEGER    SIMPLE VAR  Q-%4   ,I              READLINE          INTEGER    FUNCTION


PROGRAM UNIT AFFIRM COMPILED
****       GLOBAL STATISTICS       ****
****    NO ERRORS,   NO WARNINGS   ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:00:02
```

Figure 11-2b. Typed Logical — With Parameter

```
:FORTGO FTRAN29

PAGE 0001    HP32102B.00.0

      PROGRAM FUNCTION
C
C FUNCTION SUBPROGRAM EXAMPLE
C
 100  FORMAT(T5,"DISPLACEMENT OF THIS VEHICLE IS ",F14.5)
 200  FORMAT(T5,"TOTAL COST IS ",M12.2)
 300  FORMAT(T5,"REGISTRATION COST IS ",M12.2//)
      DISPLAY "WEIGHT?"
      ACCEPT A
      DISPLAY "DISTANCE?"
      ACCEPT B
      DISPLAY "COST?"
      ACCEPT COST
      DISPLAY "NO. CYLINDERS?"
      ACCEPT C
      DISPLAY "BORE?"
      ACCEPT D
      DISPLAY "STROKE?"
      ACCEPT E
      D=D/2
C
C THE NEXT STATEMENT CALLS THE FUNCTION DISP IMPLICITLY
C
      DISPL=DISP(C,D,E,A,B,T)
      WHOLECOST=COST+T
      TAX=1.5*DISPL
      REGISTRATION=SQRT(DISPL)*6.5
      WRITE(6,100)DISPL
      WRITE(6,200)WHOLECOST
      WRITE(6,300)REGISTRATION
      STOP
      END




      FUNCTION DISP(CYL,RAD,HGT,WEIGHT,DISTANCE,TRANSCHG)
      DISP=CYL*(3.14159*(RAD**2)*HGT)
      TRANSCHG=WEIGHT+DISTANCE*.0019
      RETURN
      END


****       GLOBAL STATISTICS        ****
****    NO ERRORS,    NO WARNINGS   ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:00:07

 END OF COMPILE

 END OF PREPARE


WEIGHT?
?3742

DISTANCE?
?6567

COST?
?8769.89

NO. CYLINDERS?
?6

BORE?
?3.798

STROKE?
?3.243

   DISPLACEMENT OF THIS VEHICLE IS       220.44366
   TOTAL COST IS   $12,524.37
   REGISTRATION COST IS       $96.51


 END OF PROGRAM
```

Figure 11-3. Function Subprogram Example

Examples of ENTRY statements are:

```
ENTRY ISUM
ENTRY ISUM(A,B,C)
```

An *entryname* in a subroutine identifies the name as a subroutine.

An *entryname* in a function identifies the name as a function.

One or more ENTRY statements are allowed, and all entry points are reentrant (the subroutine or function can call itself through this entry point).

An ENTRY statement may appear anywhere an executable statement may appear. It is not allowed in the range of a DO statement.

A subprogram may not contain both an ENTRY statement and an EXTERNAL statement referencing the same entryname.

If dynamic array bounds are used in a subprogram which contains secondary entry points, the dynamic bounds must be passed in each entry point even though the code being executed does not access the array. This is because the space must be allocated for the array even if it is not used. Since the compiler does not check to make sure that the bounds are passed at each entry point, failure to do so causes the program to abort with a bounds violation in the subprogram's initialization code when the calling program makes an entry at one of the points that does not include the necessary dynamic bounds variables.

A function entry name, like a function name, has a type associated with it. The type may be integer, double integer, real, double precision real, complex, logical or character and is determined as follows:

1. The entry name may appear in a type statement following the primary entry point of the function subprogram.
2. The type may be assigned implicitly according to the first letter of the name. Unless modified by an IMPLICIT statement, names starting with I, J, K, L, M, or N are type integer and names starting with any other letter are type real.

Note that the word ENTRY in an ENTRY statement may not be preceded by a type attribute.

Different entries in a function subprogram may be of different types. Since entries of the same type share the same memory location for returning the function's value to the calling program, the entry names can be used interchangeably in the function code. Figure 11-4 illustrates the use of multiple entries of the same type in a function. The main entry D1 and the secondary entries D2 and D3 share the same two words in memory in which a real number is stored. Therefore, D1 may be used regardless of where entry is made.

Use of an entry name of a different type than that by which entry was made will result in unpredictable consequences. This is also true if the entry name is used as anything other than a function call. Figure 11-4 illustrates the incorrect use of the entry name, A, when entry is made through an entry point, DI, of a different type.

Figure 11-6 is an example of a subroutine subprogram and an entry point.

When statement number six in the main program is executed, it calls entry point MULT and passes the variables A,B,C to the MULT entry point.

The entry point MULT calculates the product of A,B, and C and displays this product.

Figure 11-7 is an example of multiple entry points.

When statement number six in the main program is executed, it calls entry point MULT1 and passes the variables A and C to the MULT1 entry point.

The entry point MULT1 calculates the product of A and C and displays this product.

When statement number seven in the main program is reached, it calls entry point SUM2 and passes the variables A and C to the SUM2 entry point.

The entry point SUM2 calculates the sum of A and C and displays this sum.

When statement number eight in the main program is reached, it calls entry point MULT2 and passes the variables B and D to the MULT2 entry point.

The entry point MULT2 calculates the product of B and D and displays this product.

Only those parameters which are associated with the entry through which entry was made are defined. Any attempt to access formal parameters different from the parameters in the entry will result in unpredictable consequences.

Since all entry points in figure 11-7 are of different types, any reference to an entry name other than the one through which entry was made is illegal, unless it is a recursive call to the program unit. In the example of figure 11-7, the names B and D are defined only for the entries SUM1 and MULT2, thus they must not be referenced if entry was made through MULT1 or SUM2. In the same example, the names A and C are defined for the entries MULT1 and SUM2, thus they must not be referenced if entry was made through MULT2 or SUM1.

11-7. BLOCK DATA SUBPROGRAMS

As you recall from Section V, blocks of storage space can be reserved for use by several different program units through the use of a COMMON statement. These common blocks can be labeled so that the program units can reference the block by its *blockname*. Block data subprograms provide initial values for simple variables and array elements in these labeled common blocks. A block data subprogram consists of a BLOCK DATA statement and IMPLICIT, COMMON, DIMENSION, EQUIVALENCE, Type and DATA statements. EXTERNAL statements and executable statements are not allowed in block data subprograms. A block data subprogram must contain an END statement as the last line of the subprogram.

```
:FORTGO DISCNT

PAGE ØØØ1   HP321Ø2B.ØØ.Ø5  (C) HEWLETT-PACKARD CO. 1976

                PROGRAM DISCOUNT
                INTEGER QNTY
                DISPLAY "ENTER QUANTITY"
                ACCEPT QNTY
                DISPLAY "ENTER PRICE"
                ACCEPT PRICE
                IF (QNTY-5ØØ) 1Ø,1Ø,4Ø
             1Ø IF (QNTY-1ØØ) 2Ø,2Ø,3Ø
             2Ø TOTAL = D3 (QNTY,PRICE)
                GO TO 5Ø
             3Ø TOTAL = D2 (QNTY,PRICE)
                GO TO 5Ø
             4Ø TOTAL = D1 (QNTY,PRICE)
             5Ø WRITE(6,6Ø) QNTY, PRICE, TOTAL
             6Ø FORMAT (I1Ø,2X,M1Ø.2,2X,M1Ø.2)
                STOP
                END


          C
          C     FUNCTION SUBPROGRAM WITH MULTIPLE ENTRY POINTS
          C
                FUNCTION D1 (Q,P)
                INTEGER Q
                D1 = P*.9Ø
                GO TO 1ØØ
          C
          C     SECONDARY ENTRY D2
          C
                ENTRY D2 (Q,P)
                D1 = P*.95
                GO TO 1ØØ
          C
          C     SECONDARY ENTRY D3
          C
                ENTRY D3 (Q,P)
                D1 = P*.97
            1ØØ D1 = D1 *1.Ø6 * Q
                RETURN
                END



****      GLOBAL STATISTICS      ****
****   NO ERRORS,   NO WARNINGS  ****
TOTAL COMPILATION TIME  Ø:ØØ:Ø2
TOTAL ELAPSED TIME      Ø:Ø4:31

 END OF COMPILE

 END OF PREPARE


ENTER QUANTITY ?15Ø

ENTER PRICE ?5.ØØ

      15Ø       $5.ØØ     $755.25

 END OF PROGRAM
```

Figure 11-4. Entry Names of the Same Type Example

```
:FORTGO FTEST1

PAGE 0001   HP32102B.00.05  (C) HEWLETT-PACKARD CO. 1976


                PROGRAM FUNCTIONA
                INTEGER DI
                DISPLAY "ENTER VALUE"
                ACCEPT X
                Y = DI(X)
                DISPLAY Y
                STOP
                END


          C
          C     FUNCTION  SUBPROGRAM
          C
                FUNCTION DI(R)
                INTEGER DI
            100 A=R/2.345
                DI=A*100
                RETURN
          C
          C     REAL ENTRY
          C
                ENTRY A(R)
                GO TO 100
                END



****      GLOBAL STATISTICS      ****
****   NO ERRORS,   NO WARNINGS  ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:02:06

 END OF COMPILE

 END OF PREPARE


ENTER VALUE ?4.3

  ABORT :$OLDPASS...%0.%43
 PROGRAM ERROR #24: BOUNDS VIOLATION

ERR 2
```

Figure 11-5. Entry Names of Different Type Example

```
:FORTRAN TEST

PAGE 0001   HP32102B.00.08  (C) HEWLETT-PACKARD CO. 1976


          $CONTROL USLINIT
                PROGRAM ENTRYSUB
          C     ENTRY SUBROUTINE SUBPROGRAM EXAMPLE
                ACCEPT A,B,C
                IF (A.EQ.0 .OR. B.EQ.0) CALL SUM(A,B,C)
                CALL MULT (A,B,C)
                STOP
                END



PROGRAM UNIT ENTRYSUB COMPILED
                SUBROUTINE SUM(A,B,C)
          100   FORMAT(T5,"THE SUM OF A,B,C IS",F14.5)
          200   FORMAT(T5,"THE PRODUCT OF A,B,C IS",F15.5)
                S=A+B+C
                WRITE (6,100)S
                RETURN
                ENTRY MULT(A,B,C)
                P=A*B*C
                WRITE(6,200)P
                RETURN
                END



PROGRAM UNIT SUM COMPILED



****       GLOBAL STATISTICS       ****
****    NO ERRORS,    NO WARNINGS  ****
TOTAL COMPILATION TIME   0:00:01
TOTAL ELAPSED TIME       0:00:05

 END OF COMPILE
:SAVE $OLDPASS,P1
:PREP P1,P4

 END OF PREPARE
:RUN P4

?5,7,9

   THE PRODUCT OF A,B,C IS      315.00000

 END OF PROGRAM
:RUN P4

?0,12,15

   THE SUM OF A,B,C IS      27.00000
   THE PRODUCT OF A,B,C IS         .00000

 END OF PROGRAM
```

Figure 11-6. Subroutine Subprogram and an Entry Point Example

```
:FORTGO ENF4

PAGE ØØØ1    HP32102B.ØØ.Ø


$CONTROL USLINIT
            PROGRAM ENTRYFUNCTION
C
C           MULTIPLE ENTRY POINTS EXAMPLE
C
            ACCEPT A,B,C,D
            IF(A.EQ.Ø .OR. C.EQ.Ø) CALL SUM1(B,D)
            CALL MULT1(A,C)
            IF(B.EQ.Ø .OR. D.EQ.Ø) CALL SUM2(A,C)
            CALL MULT2(B,D)
            STOP
            END



C
C           THE FOLLOWING PROGRAM IS A  SUBROUTINE SUBPROGRAM
C           WITH ENTRY POINTS
C
            SUBROUTINE SUM1(B,D)
            S1=B+D
            WRITE(6,1ØØ)S1
            RETURN
            ENTRY MULT1(A,C)
            P1=A*C
            WRITE(6,2ØØ)P1
            RETURN
            ENTRY SUM2(A,C)
            S2=A+C
            WRITE(6,3ØØ)S2
            RETURN
            ENTRY MULT2(B,D)
            P2=B*D
            WRITE(6,4ØØ)P2
            RETURN
1ØØ         FORMAT(T5,"THE SUM OF B,D IS",F1Ø.3)
2ØØ         FORMAT(T5,"THE PRODUCT OF A,C IS",F12.4)
3ØØ         FORMAT(T5,"THE SUM OF A,C IS",F1Ø.3)
4ØØ         FORMAT(T5,"THE PRODUCT OF B,D IS",F12.4)
            END




****       GLOBAL STATISTICS       ****
****    NO ERRORS,    NO WARNINGS  ****
TOTAL COMPILATION TIME  Ø:ØØ:Ø2
TOTAL ELAPSED TIME      Ø:ØØ:53

 END OF COMPILE

 END OF PREPARE

?
Ø,Ø,5,8

   THE SUM OF B,D IS      8.ØØØ
   THE PRODUCT OF A,C IS        .ØØØØ
   THE SUM OF A,C IS      5.ØØØ
   THE PRODUCT OF B,D IS        .ØØØØ

 END OF PROGRAM
```

Figure 11-7. Multiple Entry Points Example

The first line of a block data subprogram must be a BLOCK DATA statement of the following form:

BLOCK DATA

or

BLOCK DATA *name*

where

name
is an alphanumeric string from one to fifteen characters (the first character must be alphabetic). The *name* may be included to identify the subprogram.

Block data subprograms use DATA statements to supply initial values to variables and array elements in labeled common blocks. The common blocks must be fully specified in a COMMON statement. EQUIVALENCE, DIMENSION and Type statements also can be used for defining the variables in the common blocks. The DATA statements indicate which variables mentioned in the COMMON statement will have initial values and what those values will be. No variable should be included in a DATA statement in a block data subprogram unless it is included in a COMMON statement. Not all variables mentioned in the COMMON statement need be mentioned in the DATA statement however, only those data elements which are to have initial values assigned need be mentioned in the DATA statement. DATA statements do not affect the storage allocation of any of the variables in block data subprograms. More than one common block can be initialized in a single block data subprogram. For example,

```
BLOCK DATA BL1

COMMON /COM1/I,J,K/COM2/L,M,N

REAL I,J,K,L,M,N

DIMENSION I(20)

DATA I,K/20 * 1.0,34.0/M,N/-4.3,67.9/

END
```

The preceding block data subprogram describes two common blocks, COM1 and COM2. COM1 contains a real array, I, of 20 elements and two simple real variables J and K. COM2 contains simple real variables L, M, and N. The DATA statement supplies initial values for all 20 elements of array I and variables K, M, and N. Note that initial values are not supplied for variables J or L, even though they are included in the COMMON statement.

11-8. STATEMENT FUNCTIONS

Another form of computational procedure that can be used in FORTRAN/3000 programs is a *statement function*. A statement function is a relatively simple computational procedure which is defined in a single statement and which may be referenced only in the program unit that defined it.

The form of a statement function is

name (param,param,. . .,param) = expression

For example,

$$\overset{\textit{name}}{\text{DISP}}\ \underbrace{\text{(C,R,H)}}_{\textit{param}} = \underbrace{\text{C * (3.14159 * (R**2) * H)}}_{\textit{expression}}$$

where

name
is a symbolic name starting with a letter.

param
is a simple variable used as a dummy argument. No other symbolic names except simple variable names may be used.

expression
is an arithmetic or logical expression of constants, simple variables, array variables, function subprogram reference, intrinsic functions, and the appropriate operators for the type expression.

See Section V, paragraph 5-18 for a complete discussion of statement functions.

11-9. NON-FORTRAN/3000 LANGUAGE SUBPROGRAMS

Procedure subprograms written in a language other than FORTRAN/3000 can be used as long as the calling sequence and the effect of execution are consistent with FORTRAN/3000. Consult Appendix A for details on the use of non-FORTRAN/3000 language subprograms.

COMPILING, PREPARING, AND EXECUTING SOURCE PROGRAMS IN FORTRAN/3000

SECTION XII

FORTRAN/3000 operates under control of the MPE/3000 Operating System.

12-1. MPE/3000 COMMANDS

Communication with MPE/3000 is initiated through *commands*. Commands are requests issued to MPE/3000 to perform various functions external to a FORTRAN/3000 source program. For example, commands are used to initiate and terminate batch jobs and interactive sessions, compile and execute source programs, call various MPE/3000 subsystems, and obtain job/session status information. Commands can be entered through any standard input file such as a card reader file or a terminal file. Commands that you will use most often are summarized in table 12-1. The commands listed in table 12-1 are those in effect with the MPE/3000C Operating System and are subject to enhancement or redefinition as later versions of the operating system are made available. Be sure to check the latest *MPE Commands Reference Manual* to verify the use of these commands.

A command consists of:

1. A *colon* (:) used as a command identifier. If in interactive mode, the colon is prompted by MPE/3000; in batch mode the colon must be provided in position 1 of the command record.

2. A *command name*. The command name requests MPE/3000 to perform some specific function.

3. A *parameter list*. The parameter list can contain zero, one, or more parameters that signify conditions for the command. The end of each parameter in a list is signified by a *delimiter*. (*Delimiter* is a name used for a character that separates, or *delimits*, one item from another.) Delimiters consist of commas, semicolons, equal signs, or other punctuation marks.

A command, then, could appear as follows:

```
:FORTRAN MYFILE,USLFILE,LISTF
```

parameter delimiters

In the preceding example, a blank space *delimits* the command name from the start of the parameter list.

Table 12-1. MPE/3000 Commands

COMMAND	FUNCTION
:JOB	Initiates a batch job
:HELLO	Initiates an interactive session
:FILE	Specifies characteristics of a file
:BUILD	Creates a new file
:PURGE	Deletes a file from the system
:CONTINUE	Disregards batch job error condition
:FORTRAN	Compiles a FORTRAN/3000 source program
:FORTPREP	Compiles and prepares a FORTRAN/3000 source program
:FORTGO	Compiles, prepares, and executes a FORTRAN/3000 source program
:PREP	Prepares a compiled program
:PREPRUN	Prepares and executes a compiled program
:RUN	Executes a prepared program
:EOD	Signifies the end of data
:EOJ	Terminates a job
:BYE	Terminates a session

The meanings of parameters in some commands are determined by their *positions* in the parameter list. For example, in a FORTRAN command:

:FORTRAN *textfile,uslfile,listfile,masterfile,newfile*

the parameters are *positional* and their positions in the list designate their meanings. The omission of an optional positional parameter from a parameter list is signified by adjacent delimiters, as follows:

:FORTRAN *textfile,,listfile*

uslfile is omitted

When parameters are omitted from the end of a list, no adjacent delimiters are required, as shown in the example by the omission of *masterfile* and *newfile.*

Commands are explained in greater detail in the *MPE Commands Reference Manual.*

12-2. SPECIFYING FILES FOR PROGRAMS

Both the FORTRAN/3000 compiler and the MPE/3000 Operating System read input from and write output to files handled through the file facility of MPE/3000. For example, the compiler reads source code from a *textfile,* writes object code to an *objectfile* (uslfile), produces listings to a *listfile,* and performs merging and editing operations using an old *masterfile* for input and a *newfile* for output. Each file has a *formal file designator* (a name by which the file is known to MPE/3000 and the FORTRAN/3000 compiler). For files used by the compiler or referenced by MPE/3000 commands, you equate these *formal file designators* to *actual file designators* (the actual names of the files to be used for input/output) through either of two methods:

1. By naming the files as positional parameters (*actual file designators*) in the compilation, preparation, or execution commands.

2. By omitting optional parameters from the compilation, preparation, or execution command, thus allowing FORTRAN/3000 to assign standard *default file designators* for input or output files.

Because of the importance of file references as command parameters, some of the rules for specifying files are introduced before the compilation, preparation, and execution commands themselves. The complete rules concerning files are discussed in the *MPE Commands Reference Manual* and in Section VIII of this manual.

You can equate formal file designators to actual file designators through the MPE/3000 :FILE command. You also can use this command to override any actual or default file designators specified in compilation, preparation, and execution commands, and to specify various file characteristics. See Section VIII for a discussion of the MPE/3000 :FILE command.

12-3. SPECIFYING FILES AS COMMAND PARAMETERS

You can name the following types of files as parameters in a compilation, preparation, or execution command:

- System-Defined Files
- User-Defined Files
- New Files
- Old Files

12-4. SYSTEM-DEFINED FILES. System-defined file designators indicate those files that MPE/3000 uniquely identifies as standard input/output files for a job/session. These files are shown in table 12-2.

Table 12-2. System-Defined Files

ACTUAL FILE DESIGNATOR	DEVICE/FILE REFERENCED
$STDIN	A filename indicating the standard job or session input file (from which the job or session is initiated). For a job, this is typically a card reader; for a session this typically indicates a terminal. Input data records in the $STDIN file should not contain a colon in position one, since this indicates the end of the source input. Use the :EOD command to indicate the physical end of a source program. (The same command is used to indicate the end of a data file.)
$STDINX	Equivalent to $STDIN, except that MPE/3000 command records (those with a colon in position one) encountered in a data file are read without indicating the end of data. (However, the commands :JOB, :DATA, :EOJ, and :EOD are exceptions that always indicate the end of data and are never read as data.)
$STDLIST	A filename indicating the standard job or session listing file corresponding to the particular job or session input device being used. (For each potential job/session input device, a user with MPE/3000 System Supervisor capability designates a corresponding job/session listing device during system configuration.) The job or session listing device is customarily a printer for a batch job and a terminal for a session.
$NULL	The name of a non-existent "ghost" file that is always treated as an empty file. When referenced as an input file by a program, that program receives only an end of data indication upon first access. When referenced as an output file, the associated write request is accepted by MPE/3000 but no physical output is actually performed. Thus, $NULL can be used to discard unneeded output from an executing program.

12-5. USER PRE-DEFINED FILES. A user pre-defined file is any file that was previously defined or redefined in a :FILE command as discussed in the *MPE Commands Reference Manual* and Section VIII of this manual. In other words, it is a back-reference to that :FILE command. In compilation, preparation, or execution commands, the *actual file designator* of this file is the *formal file designator* preceded by an asterisk (to indicate that it was previously defined).

12-6. NEW FILES. New files are files that have not yet been created, and are being created for the first time by the current batch job or interactive session. New files can have actual file designators as shown in table 12-3.

12-7. OLD FILES. Old files are existing files in the system. They may be named by the designators shown in table 12-4.

Table 12-3. New Files

ACTUAL FILE DESIGNATOR	FILE REFERENCED
$NEWPASS	A temporary disc file that can be passed automatically to any succeeding MPE/3000 command within the same job or session which references it by the filename $OLDPASS. (Passing is explained in the compilation, preparation, and execution command examples.) Only one such file can exist in the job or session at any one time. (When $NEWPASS is closed, its name is changed to $OLDPASS automatically, and any previous file named $OLDPASS is deleted.)
filereference	Any other new file to which you have access. Unless you specify otherwise, this is a temporary file, residing on disc, that is destroyed upon termination of the program. If no :FILE command specifies otherwise, any such FORTRAN/3000 files are closed as job/session temporary files, saved until the end of the job/session, and then are purged. If closed as permanent files (by specifying SAVE in a :FILE command), they are saved until you purge them. Typically, this format consists of a file name containing up to eight alphanumeric characters, beginning with a letter. In addition, other elements (such as a group name, account name, or lockword) can be specified. The complete rules governing the *filereference* format are contained in the *MPE Commands Reference Manual.*

Table 12-4. Old Files

ACTUAL FILE DESIGNATOR	FILE REFERENCED
$OLDPASS	The name of the temporary file last closed as $NEWPASS.
filereference	Any other old file to which you have access. It may be a job/session temporary file created in the current or a previous program in the current job/session, or a permanent file saved by any program in any job/session. The format is the same as *filereference,* noted in table 12-3.

12-8. INPUT/OUTPUT SETS. All of the preceding actual file designators can be classified as those used as input parameters (*input set*) and those used as output parameters (*output set*). These sets are defined as follows:

INPUT SET

$STDIN	The job/session input file.
$STDINX	The job/session input file with commands allowed.
$OLDPASS	The last file passed.
$NULL	A constantly-empty file that will produce an end-of-file condition whenever it is read.
**formaldesignator*	A back-reference to a previously-defined file.
filereference	A file name, and perhaps account and group names and a lockword.

OUTPUT SET

$STDLIST	The job/session listing file.
$OLDPASS	The last file passed.
$NEWPASS	A new temporary file to be passed.
$NULL	A constantly-empty file.
**formaldesignator*	A back-reference to a previously-defined file.
filereference	A file name, and perhaps account and group names and a lockword.

12-9. SPECIFYING FILES BY DEFAULT

When you omit an optional file parameter from a compilation, preparation, or execution command, MPE/3000 assigns one of the members of the *input* or *output* set by default. The file designator assigned depends on the specific command, parameter, and operating mode, as noted later in this section. The default file designators are shown in table 12-5.

12-10. COMPILING, PREPARING, AND EXECUTING FORTRAN/3000 SOURCE PROGRAMS

The commands used for compilation, preparation, and execution of FORTRAN/3000 source programs are:

:FORTRAN	*compiles* a source program.
:FORTPREP	*compiles* and *prepares* a source program.
:FORTGO	*compiles, prepares,* and *executes* a source program.
:PREP	*prepares* source programs that have been compiled into a USL file.
:RUN	*executes* programs that have been compiled and prepared (and,therefore, exist on program files).
:PREPRUN	*prepares* and *executes* programs compiled into USL files.

12-11. :FORTRAN COMMAND

The :FORTRAN command *compiles* a FORTRAN/3000 source program.

The form of a :FORTRAN command is

:FORTRAN *textfile, uslfile, listfile, masterfile, newfile*

where

textfile
is the name of an input file from which the source program is to be read. If omitted, the program will be read from the standard input file to which MPE/3000 will assign the default file name $STDIN. Do not use the designator FTNTEXT for this parameter.

Table 12-5. FORTRAN/3000 Compiler File Designators

FILE	PURPOSE	FORMAL FILE DESIGNATOR	DEFAULT FILE DESIGNATOR
Textfile	Contains source program, correction text to be merged, and/or compiler subsystem commands.	FTNTEXT	$STDIN
Listfile	Destination of listing output.	FTNLIST	$STDLIST
Uslfile	Destination of object program code.	FTNUSL	$NEWPASS
Masterfile	Old source program to be merged and edited with new text input from *textfile*.	FTNMAST	$NULL
Newfile	New source program resulting from (optional) merging of *textfile* and *masterfile*.	FTNNEW	$NULL
Progfile	Destination of executable object program.	None	$NEWPASS

uslfile
is the name of the USL file on which the object program is to be written. If this parameter is included in a :FORTRAN command, it must indicate a file previously created in one of two ways:

1. By saving a USL file with the :SAVE command from a previous compilation.
2. By creating a new file with the :BUILD command and designating it as a USL file with a *filecode* of 1024 or USL. For example,

```
:BUILD USLF;CODE=1024
```

or

```
:BUILD USLF;CODE=USL
```

If you specify a USL file, and that file does not exist, then a file will be opened with the specified name as the USL file and will be closed with a save disposition at the conclusion of the compilation.

listfile
the name of the file to which the program listing is to be generated. If omitted, the default file $STDLIST (typically the terminal in a session or the line printer in batch) is assigned. Do not use the designator FTNLIST for this parameter.

masterfile
the name of a file to be optionally merged with *textfile* and written onto a file named *newfile*. If *masterfile* is omitted, no merging takes place. Do not use the designator FTNMAST for this parameter.

newfile
the name of a file on which the (re-sequenced) records from *textfile* and *masterfile* are optionally merged. When *newfile* is omitted, no *newfile* is created. Do not use the designator FTNNEW for this parameter.

All parameters for the :FORTRAN command are optional.

Figure 12-1, shows a program entered from a terminal.

Note: Direct interactive input is not recommended. Typing error recovery is impossible once a carriage return is pressed. To create source files, use the HP 3000 Text Editor. (See the *EDIT/3000 Reference Manual.*)

All parameters are omitted from the :FORTRAN command. The compiler will use the default file $STDIN (meaning the program is to be read from the standard input file) for the parameter *textfile*. Omitting the *uslfile* parameter causes the compiler to write the object program on the default file $OLDPASS. When *listfile* is omitted, the program listing will occur on $STDLIST (which is the terminal in the example). No merging will occur and no *newfile* will be created when the *masterfile* and *newfile* parameters are omitted.

When the compiler encounters an END line in the program, it compiles the program unit containing the END line and outputs another prompt character (if in interactive mode). If this is the last program unit to be compiled, the :EOD command is used to exit from :FORTRAN.

The program is now compiled and a temporary USL file has been created during compilation. To prepare the program, a :PREP command could be entered as follows:

```
:PREP $OLDPASS,TEST
```

The temporary file created during compilation ($NEWPASS) is passed automatically to the preparation mechanism and renamed $OLDPASS. The non-existent file TEST is used for the *progfile* parameter (required parameter, see paragraph 12-12). A file of the correct size and type will be created by the Segmenter. After preparation, the program could be run by using a :RUN command as follows:

```
:RUN TEST
```

Note that TEST is a temporary file and will not be saved when the session is terminated unless the MPE/3000 :SAVE command is used.

12-12. :FORTPREP COMMAND

The :FORTPREP command *compiles* and *prepares* a FORTRAN/3000 source program.

The form of a :FORTPREP command is

:FORTPREP *textfile, progfile, listfile, masterfile, newfile*

where

textfile, listfile, masterfile, newfile
have the same meanings as described under the :FORTRAN command.

progfile
is the name of the file on which the prepared program is to be written. If this parameter is included, it must reference a file created in one of two ways:

1. By specifying a non-existent file in the parameter, in which case a temporary file of the correct size and type will be created. This method is recommended for optimal disc usage. Use the SAVE command to save the temporary file.
2. By using the :BUILD command with a filecode of 1029 or PROG. For example,

   :BUILD PROGF;CODE= 1029;DISC=*numrec*,1,1

   or

   :BUILD PROGF;CODE= PROG;DISC=*numrec*,1,1

   If *numrec* is omitted, the default is 1024 records.

```
:FORTRAN

PAGE ØØØ1    HP32102B.ØØ.Ø


>$CONTROL FREE
> 1Ø FORMAT(T6,"NUMBER",T15,"SQUARE ROOT"//)
> 2Ø FORMAT(T7,F4.1,T18,F7.5)
> WRITE(6,1Ø)
> A=1.Ø
> DO 5Ø I=1,1Ø
> B=SORT(A)
> WRITE(6,2Ø)A,B
> 5Ø A=A+1.Ø
> STOP
> END



>:EOD



****       GLOBAL STATISTICS       ****
****    NO ERRORS,    NO WARNINGS  ****
TOTAL COMPILATION TIME  Ø:ØØ:Ø1
TOTAL ELAPSED TIME      Ø:Ø4:32

 END OF COMPILE
```

Figure 12-1. :FORTRAN Command Example

If omitted, the default file $NEWPASS is assigned. (This file is renamed $OLDPASS upon completion.)

All :FORTPREP parameters are optional.

Figure 12-2 shows a program entered from cards for compilation and preparation using the :FORTPREP command.

12-13. :FORTGO COMMAND

The :FORTGO command *compiles, prepares,* and *executes* a FORTRAN/3000 source program.

The form of a :FORTGO command is

:FORTGO *textfile, listfile, masterfile, newfile*

where

textfile, listfile, masterfile, newfile
all have the same meanings as described under the :FORTRAN command.

All :FORTGO parameters are optional.

Figure 12-3 is an example of the :FORTGO command used to compile, prepare, and execute a source program which has been stored on disc under the file name "FTRAN36."

12-14. :PREP COMMAND

The :PREP command *prepares* source programs that have been compiled into a USL file.

The form of a :PREP command is

:PREP *uslfile, progfile*

where

uslfile
is the name of the USL file onto which the program has been compiled.

progfile
is the name of the program file onto which the prepared program is to be written. This file must be created in one of two ways:

1. By specifying a non-existent file in this parameter, in which case a temporary file of the correct size and type will be created. This method is recommended for optimal disc usage. Use the SAVE command to save the temporary file.

2. By creating a new file with the :BUILD command using a *filecode* of 1029 or PROG, as follows:

   :BUILD PROGF;CODE=1029;DISC=*numrec*,1,1

   or

   :BUILD PROGF;CODE=PROG;DISC=*numrec*,1,1

   If omitted, *numrec* is 1024 records by default.

Both parameters are required in a :PREP command.

Other (optional) parameters, called *keyword* parameters, that can be used with the :PREP command are summarized below. These (and all other *keyword* parameters) are discussed in the *MPE/3000 Operating System Reference Manual* and are summarized here for reference only.

;ZERODB

;PMAP

;MAXDATA = segsize

;STACK = stacksize

;DL = dlsize

;CAP = caplist

;RL = filename

where

ZERODB
is a request to set the initially defined DL-DB and DB-Q (initial) areas to zero. Default: DL-DB and DB-Q (initial) not set.

PMAP
is a request to list certain information about the prepared program. Default: no listing.

segsize
maximum stack area (Z - DL) size permitted, in words, normally estimated by Segmenter at preparation. This parameter allows you to override the Segmenter estimate. Default: MPE/3000 assumes stack will remain same size.

stacksize
When a process is created by the system, the user is allocated *MAXDATA* words of virtual memory but only *stacksize* words in main memory. The main memory space is expanded as required. This parameter allows you to override the Segmenter estimate. Default: estimated by the Segmenter for each individual program.

dlsize
the DL-DB area size to be assigned initially to the stack. Default: estimated by MPE/3000 for each program.

```
:JOB     HAL.GOODWIN, PUB
 PRI= DS;  INPRI= 13;  TIME= ?
 JOB NUMBER = #J16
 THU. MAR  6, 1975, 12:54 PM
 HP32000C.F0.51

:FORTPREP


PAGE 0001    HP32102B.00.0


                  PROGRAM FORTPREP
           C
           C THIS IS AN EXAMPLE OF A FORTRAN/3000
           C SOURCE PROGRAM COMPILED AND PREPARED
           C USING THE :FORTPREP COMMAND
           C
            100   FORMAT(T10,F12.4)
                  COMMON ARR(3,3),RESULT
                  DO 150 I=1,3
                  DO 150 J=1,3
                  ARR(I,J)=5.3
            150   CONTINUE
                  CALL SOLVE
                  WRITE(6,100)RESULT
                  STOP
                  END










                  SUBROUTINE SOLVE
                  COMMON ARR(3,3),RESULT
                  RESULT=ARR(1,1)*ARR(1,2)*ARR(1,3)*
                 #ARR(2,1)*ARR(2,2)*ARR(2,3)*
                 #ARR(3,1)*ARR(3,2)*ARR(3,3)
           C END OF SUBROUTINE
                  RETURN
                  END



****      GLOBAL STATISTICS       ****
****   NO ERRORS,   NO WARNINGS   ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:00:10



 END OF COMPILE

 END OF PREPARE
:EOJ

CPU (SEC) = 9
ELAPSED (MIN) = 2
THU, MAR  6, 1975, 12:55 PM
END OF JOB
```

Figure 12-2. :FORTPREP Command Example

```
:FORTGO FTRAN36

PAGE 0001   HP32102B.00.0

            PROGRAM FORTGO
      C
      C EXAMPLE OF :FORTGO COMMAND
      C
            CHARACTER*5 CHEX(8,8),BL,IN,CP,PP,FRAME*41
            BL="!    "
            IN="! ** "
            CP="! BL "
            PP="! WH "
            DO 100 I=1,3,2
            DO 90 J=2,8,2
       90   CHEX(I,J)=CP
      100   CONTINUE
            DO 110 J=1,7,2
      110   CHEX(2,J)=CP
            DO 130 I=6,8,2
            DO 120 J=1,7,2
      120   CHEX(I,J)=PP
      130   CONTINUE
            DO 140 J=2,8,2
      140   CHEX(7,J)=PP
            DO 160 I=1,7,2
            DO 150 J=1,7,2
      150   CHEX(I,J)=IN
      160   CONTINUE
            DO 180 I=2,8,2
            DO 170 J=2,8,2
      170   CHEX(I,J)=IN
      180   CONTINUE
            DO 190 J=1,7,2
      190   CHEX(4,J)=BL
            DO 200 J=2,8,2
      200   CHEX(5,J)=BL
      300   FORMAT(T12,"1",4X,"2",4X,"3",4X,"4",4X,"5"
           #,4X,"6",4X,"7",4X,"8")
      400   FORMAT(T10,S)
      500   FORMAT(T6,I2,T10,8S)
            FRAME="!----!----!----!----!----!----!----!----!"
            WRITE(6,300)
            DO 600 I=1,8
            WRITE(6,400)FRAME
      600   WRITE(6,500)I,(CHEX(I,J),J=1,8)
            WRITE(6,400)FRAME
            STOP
            END


****      GLOBAL STATISTICS       ****
****    NO ERRORS,   NO WARNINGS  ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:00:09

 END OF COMPILE

 END OF PREPARE

           1    2    3    4    5    6    7    8
        !----!----!----!----!----!----!----!----!
     1  ! ** ! BL ! ** ! BL ! ** ! BL ! ** ! BL
        !----!----!----!----!----!----!----!----!
     2  ! BL ! ** ! BL ! ** ! BL ! ** ! BL ! **
        !----!----!----!----!----!----!----!----!
     3  ! ** ! BL ! ** ! BL ! ** ! BL ! ** ! BL
        !----!----!----!----!----!----!----!----!
     4  !    ! ** !    ! ** !    ! ** !    ! **
        !----!----!----!----!----!----!----!----!
     5  ! ** !    ! ** !    ! ** !    ! ** !
        !----!----!----!----!----!----!----!----!
     6  ! WH ! ** ! WH ! ** ! WH ! ** ! WH ! **
        !----!----!----!----!----!----!----!----!
     7  ! ** ! WH ! ** ! WH ! ** ! WH ! ** ! WH
        !----!----!----!----!----!----!----!----!
     8  ! WH ! ** ! WH ! ** ! WH ! ** ! WH ! **
        !----!----!----!----!----!----!----!----!
 END OF PROGRAM
```

Figure 12-3. :FORTGO Command Example

caplist
the capability-class attributes associated with your program. Default: BA, IA (standard batch and interactive access).

filename
the name of a relocatable procedure library to be searched to satisfy external references during program preparation. Default: no library searched.

12-15. :PREPRUN COMMAND

The :PREPRUN command *prepares* and *executes* programs that have been compiled into USL files.

The form of the :PREPRUN command is

:PREPRUN *uslfile*

where

uslfile
is the name of the USL file on which the program has been compiled. (Required parameter.)

The (optional) keyword parameters that can be used with the :PREPRUN command are summarized below. These (and all other keyword) parameters are discussed in the *MPE Commands Reference Manual* and are summarized here for reference only.

;NOPRIV

;PMAP

;DEBUG

;LMAP

;ZERODB

;MAXDATA = segsize

;PARM = parameternum

;STACK = stacksize

;DL = dlsize

;RL = filename

;LIB = library

;NOCB

;CAP = caplist

where

NOPRIV
is a request to place a privileged program in non-privileged mode. Default: privileged program executes in privileged mode.

PMAP
a request to list certain information about the prepared program. Default: no listing.

DEBUG
a request to set a breakpoint on the first executable instruction of the program. Default: no breakpoint set.

LMAP
a request to list certain information about the loaded program. Default: no listing.

ZERODB
a request to set the initially defined DL-DB and DB-Q (initial) areas to zero. Default: DL-DB and DB-Q (initial) not set.

segsize
maximum stack area (Z-DL) size permitted, in words, normally estimated by Segmenter at preparation. This parameter allows you to override the Segmenter estimate. Default: MPE/3000 assumes stack will remain same size.

parameternum
a value that can be passed to your program as a general parameter for control or other purposes. Default: value is set to zero.

stacksize
When a process is created by the system, the user is allocated *MAXDATA* words of virtual memory but only *stacksize* words in main memory. The main memory space is expanded as required. This parameter allows you to override the Segmenter estimate. Default: estimated by the Segmenter for each individual program.

dlsize
the DL-DB area size to be assigned initially to the stack. Default: estimated by MPE/3000.

filename
the name of a relocatable procedure library to be searched to satisfy external references during program preparation. Default: no library searched.

library
the order in which segmented procedure libraries are to be searched to satisfy external references during segmentation. Default: S (system library searched).

NOCB
a request that file system not use stack segment (PCBX) for its control blocks, even if sufficient space is available. Use only if program absolutely requires largest stack possible.

caplist
the capability-class attributes associated with your program. Default: BA, IA (standard batch and interactive access only).

12-16. :RUN COMMAND

The :RUN command *executes* a program that has been compiled and prepared into a program file.

The form of the :RUN command is

:RUN *progfile*

For example,

:RUN MYPROG

where

progfile
is the name of the compiled and prepared program to be executed.

The *progfile* parameter is required in a :RUN command.

The following (optional) *keyword* parameters can be used with the :RUN command:

;NOPRIV

;LMAP

;DEBUG

;MAXDATA = segsize

;PARM = parameternum

;STACK = stacksize

;DL = dlsize

;LIB = library

;NOCB

The above parameters are the same as those summarized for the :PREPRUN command. (See paragraph 12-15.)

12-17. USING EXTERNAL PROCEDURE LIBRARIES

Compiled FORTRAN/3000 programs are stored in files called *User Subprogram Libraries* (*USL's*) that reside on disc. In any particular USL, each compiled program unit exists as a *Relocatable Binary Module* (RBM). To prepare a program (and any program units it references) for execution, the MPE/3000 Segmenter selects the appropriate RBM's from the USL and binds them into linked segments written on a program file. For more information on the Segmenter, USL's and RBM's refer to the *MPE Segmenter Reference Manual.*

When you prepare and run programs in FORTRAN/3000, it is possible to reference external procedures from *procedure libraries.* You can build,modify, and maintain two types of procedure libraries within your log-on group and account: *Relocatable Libraries* and *Segmented Libraries.*

12-18. RELOCATABLE LIBRARIES

A Relocatable Library (RL) is a specially-formatted file that is searched at *program preparation* time to satisfy references to external procedures called by your program. Within such libraries, these procedures exist in RBM form (as they would on a USL). When a program is prepared, these procedures are placed in a single segment and linked to your program in the resulting program file.

For example, to specify that an RL named RLPROC be searched during preparation of a program from the USL file USL1 to the program file PROG1, you would enter the following :PREP command:

```
PREP USL1,PROG1; RL=RLPROG
```

12-19. CREATING AND MAINTAINING RELOCATABLE LIBRARIES. To create and maintain relocatable libraries, you first must access the Segmenter by entering the MPE/3000 :SEGMENTER command.

The form of the :SEGMENTER command is

:SEGMENTER *listfile*

where

listfile
is an ASCII file from the output set (formal designator SEGLIST) to which is written any listable output generated by the Segmenter subsystem commands. (The designator SEGLIST should *not* be used as the *actual* file designator.) If *listfile* is omitted, the standard job/session list device ($STDLIST) is assigned by default. (Optional parameter.)

If in interactive session, the Segmenter will prompt with a dash (−). Once the Segmenter is accessed, the following commands are used to create and maintain an RL:

−BUILDRL
creates a permanent, formatted RL file.

−USL
references the USL file from which the procedure is to be obtained.

−RL
identifies an existing RL.

−ADDRL
adds a procedure to the currently identified RL.

−PURGERL
deletes a procedure from an RL.

−LISTRL
lists information concerning the currently identified RL.

The form of a −BUILDRL command is

−BUILDRL *filereference, records, extents*

where

filereference
is the filename of the new RL (optionally including group and account identifiers). (Required parameter.)

records
is the total maximum capacity of the file, specified in terms of 128-word, binary logical records. (Required parameter.)

extents
the total number of disc extents that can be dynamically allocated to the file as logical records are written to it. The size of each extent is determined by the *records* parameter value divided by the *extents* parameter value. The *extents* value must be an entry from 1 to 16. (Required parameter.)

The form of a −USL command is

−USL *filereference*

where

filereference
is the name (and optional group and account names) of the USL file to be manipulated. (Required parameter.)

The form of an −RL command is

−RL *filereference*

where

filereference
is the name (and optional group and account names) of the RL to be modified. (Required parameter.)

The form of an −ADDRL command is

−ADDRL *name (index)*

where

name
is the name of the procedure to be added to the RL. This name is called the *primary entry point* of the RBM containing the procedure. (Required parameter.)

index
is an integer further identifying the RBM. The index may be used when the currently-managed USL contains more than one active RBM of the same name. If *index* is omitted, a value of 0 is assigned by default. (Optional parameter.)

The form of a −PURGERL command is

−PURGERL *rlspec, name*

where

rlspec
is *UNIT*, to delete the procedure identified by *name*; or *ENTRY*, to delete the entry point identified by *name*. The default parameter is ENTRY. (Optional parameter.)

name
if *rlspec* is UNIT, *name* is the name of the *procedure* to be deleted. If *rlspec* is ENTRY, *name* is the name of the *entry point* to be deleted. (Required parameter.)

The form of a −LISTRL command is

−LISTRL

See the *MPE Segmenter Reference Manual* (Section VII) for further discussions of these Segmenter commands.

Figures 12-4 through 12-6 demonstrate how to compile a procedure into a USL, build an RL file and add the procedure to this RL, and, finally, how to run a program referencing this external procedure.

In figure 12-4, USL file USL1 is created using the MPE/3000 :BUILD command. The text file FTRAN37 then is compiled into this USL using the :FORTRAN command.

In figure 12-5, the Segmenter is accessed with the MPE/3000 :SEGMENTER command. Once accessed, the Segmenter command −BUILDRL is used to create the RL file RLPROC (consisting of 300 records maximum and one disc extent); the −USL command identifies the USL file (USL1) which contains the procedure; and the −ADDRL command adds the procedure START to RLPROX.

```
:BUILD USL1;CODE=USL
:FORTRAN FTRAN37,USL1

PAGE 0001    HP32102B.00.0


00001000           SUBROUTINE START
00002000           CHARACTER*5 CHEX(8,8),BL,IN,CP,PP,FRAME*41
00003000           BL="!    "
00004000           IN="! ** "
00005000           CP="! BL "
00006000           PP="! WH "
00007000           DO 100 I=1,3,2
00008000           DO 90 J=2,8,2
00009000      90   CHEX(I,J)=CP
00010000      100  CONTINUE
00011000           DO 110 J=1,7,2
00012000      110  CHEX(2,J)=CP
00013000           DO 130 I=6,8,2
00014000           DO 120 J=1,7,2
00015000      120  CHEX(I,J)=PP
00016000      130  CONTINUE
00017000           DO 140 J=2,8,2
00018000      140  CHEX(7,J)=PP
00019000           DO 160 I=1,7,2
00020000           DO 150 J=1,7,2
00021000      150  CHEX(I,J)=IN
00022000      160  CONTINUE
00023000           DO 180 I=2,8,2
00024000           DO 170 J=2,8,2
00025000      170  CHEX(I,J)=IN
00026000      180  CONTINUE
00027000           DO 190 J=1,7,2
00028000      190  CHEX(4,J)=BL
00029000           DO 200 J=2,8,2
00030000      200  CHEX(5,J)=BL
00031000      300  FORMAT(T12,"1",4X,"2",4X,"3",4X,"4",4X,"5"
00032000          #,4X,"6",4X,"7",4X,"8")
00033000      400  FORMAT(T10,S)
00034000      500  FORMAT(T6,I2,T10,8S)
00035000           FRAME="!----!----!----!----!----!----!----!----!"
00036000           WRITE(6,300)
00037000           DO 600 I=1,8
00038000           WRITE(6,400)FRAME
00039000      600  WRITE(6,500)I,(CHEX(I,J),J=1,8)
00040000           WRITE(6,400)FRAME
00041000           RETURN
00042000           END


****       GLOBAL STATISTICS       ****
****    NO ERRORS,   NO WARNINGS   ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:00:06

 END OF COMPILE
```

Figure 12-4. Creating a USL File

```
:SEGMENTER

SEGMENTER SUBSYSTEM (C.Ø)
-BUILDRL RLPROC,3ØØ,1
-USL USL1
-ADDRL START
-LISTRL

RL FILE RLPROC.PUB.GOODWIN

* ENTRY POINTS *

START            3    5Ø      4ØØ     1    555    652

* EXTERNALS *

TFORM'           Ø
FMTINIT'         Ø
IIO'             Ø
SIO'             Ø

USED                 15ØØ          AVAILABLE           1113ØØ
-EXIT

 END OF SUBSYSTEM
```

Figure 12-5. Using the Segmenter to Build a Relocatable Library File

```
:FORTRAN FTRAN38

PAGE 0001    HP32102B.00.0


00001000           PROGRAM RELOCATABLE
00002000     C
00003000     C THIS PROGRAM CALLS THE RL LIBRARY PROCEDURE START
00004000     C
00005000      10   CALL START
00006000           STOP
00007000           END



****      GLOBAL STATISTICS        ****
****    NO ERRORS,   NO WARNINGS   ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:00:08

 END OF COMPILE
:BUILD PROG1;CODE=PROG
:PREP $OLDPASS,PROG1;RL=RLPROC

 END OF PREPARE
:RUN PROG1

               1    2    3    4    5    6    7    8
             !----!----!----!----!----!----!----!----!
        1    ! ** ! BL ! ** ! BL ! ** ! BL ! ** ! BL
             !----!----!----!----!----!----!----!----!
        2    ! BL ! ** ! BL ! ** ! BL ! ** ! BL ! **
             !----!----!----!----!----!----!----!----!
        3    ! ** ! BL ! ** ! BL ! ** ! BL ! ** ! BL
             !----!----!----!----!----!----!----!----!
        4    !    ! ** !    ! ** !    ! ** !    ! **
             !----!----!----!----!----!----!----!----!
        5    ! ** !    ! ** !    ! ** !    ! ** !
             !----!----!----!----!----!----!----!----!
        6    ! WH ! ** ! WH ! ** ! WH ! ** ! WH ! **
             !----!----!----!----!----!----!----!----!
        7    ! ** ! WH ! ** ! WH ! ** ! WH ! ** ! WH
             !----!----!----!----!----!----!----!----!
        8    ! WH ! ** ! WH ! ** ! WH ! ** ! WH ! **
             !----!----!----!----!----!----!----!----!
 END OF PROGRAM
```

Figure 12-6. Calling a Procedure from a Relocatable Library

Note: A FORTRAN/3000 *main program* or BLOCK DATA subprogram cannot be added to an RL file as a procedure. Such procedures must be subroutine or function *subprograms.*

Figure 12-6 demonstrates how to compile the text file FTRAN38 into a USL using the :FORTRAN command, prepare this USL ($OLDPASS) into the temporary program file PROG using the :PREP command (note the reference to the RL file *RLPROX* with the *RL = filename* keyword parameter), and execute the program using the :RUN command. Statement 10 in the program of figure 12-6 calls the procedure START (which is the procedure contained in the RL library).

See the *MPE Segmenter Reference Manual* for further discussions of Segmenter commands and user library management.

12-20. SEGMENTED LIBRARIES

Segmented libraries (SL's) are specially formatted files that are searched at *program run* time to satisfy references to external procedures. These libraries, like program files, contain procedures in segmented (prepared) form. An individual procedure may be the only procedure in its segment, or it may exist in a segment containing many other procedures. When a procedure is referenced, the segment containing it is loaded with your program. Since the segmentation is not altered when different programs reference procedures in an SL, these procedures may be shared concurrently by other programs.

To specify that an SL file be searched in your group account, add the keyword parameter *LIB = library* in the :RUN command as follows:

```
:RUN PROG1;LIB = G
```

12-21. CREATING AND MAINTAINING SEGMENTED LIBRARIES. To create and maintain segmented libraries, you first must access the Segmenter by entering the MPE/3000 :SEGMENTER command.

The form of the :SEGMENTER command is

:SEGMENTER *listfile*

where

listfile
is an ASCII file from the output set (formal designator SEGLIST) to which is written any listable output generated by the Segmenter subsystem commands. (The designator SEGLIST should *not* be used as the *actual* file designator.) If *listfile* is omitted, the standard job/session list device ($STDLIST) is assigned by default. (Optional parameter.)

If in interactive session, the Segmenter will prompt with a dash (−). Once the Segmenter is accessed, the following commands are used to create and maintain an SL:

−BUILDSL
creates a permanent, formatted SL file.

−SL
identifies an existing SL file.

−ADDSL
adds a procedure to the SL file being managed currently.

−PURGESL
purges an entry-point from a segment in an SL, or the entire segment from the SL.

−LISTSL
lists the procedures in the currently-managed SL file.

In addition, the −USL and −LISTUSL Segmenter commands can be used (see figure 12-7).

The form of a −BUILDSL command is

BUILDSL *filereference, records, extents*

where

filereference
is a file whose local name is SL (and optional group and account names).

Note: You can create an SL file with a local name other than SL, but such a file cannot be searched by the :RUN command. (Required parameter.)

records
is the total maximum file capacity, specified in terms of 128-word binary logical records. (Required parameter.)

extents
the total number of disc extents that can be dynamically allocated to the file as logical records are written to it. The size of each extent is determined by the *records* parameter value divided by the *extents* parameter value. The *extents* value must be an integer from 1 to 16. (Required parameter.)

The form of an −SL command is

−SL *filereference*

where

filereference
is the name of the SL to be modified (and optional group and account name). (Required parameter.)

The form of an −ADDSL command is

−ADDSL *name, PMAP*

where

name
is the name of the segment to be added to the SL. (Required parameter.)

PMAP
is an indication that a listing describing the prepared segment will be produced on the device specified in the :SEGMENTER command parameter *listfile*. If omitted, the prepared segment is not listed. (Optional parameter.)

The form of a −PURGESL command is

−PURGESL *unitspec, name*

where

unitspec
is ENTRY, to delete the entry-point identified by *name;* or SEGMENT, to delete the segment identified by *name.* (Optional parameter.)

The default parameter is ENTRY.

name
is the name of the entry point or segment to be deleted. (Required parameter.)

The form of a −LISTSL command is

−LISTSL

For further descriptions of these Segmenter commands, see the *MPE Segmenter Reference Manual.*

Figures 12-7 and 12-8 demonstrate how to build an SL file, add a function procedure to this file, and run a program that references the procedure. In figure 12-7, a USL file (USL1) is created and the function subprogram DISP is compiled into this file.

Note: A FORTRAN/3000 *main program* cannot be added to an SL file as a procedure. Such procedures must be subroutine or function *subprograms.* In addition, FORTRAN/3000 procedures in an SL library cannot contain DATA, COMMON, labeled COMMON, or TRACE variables, or references to FORTRAN/3000 logical units.

The Segmenter command −BUILDSL is used to create an SL file named SEARCH, with 300 records maximum and one disc extent. The −USL command is used to identify the USL file (USL1) containing the procedure which is to be added to the SL file.

The −LISTUSL command causes a listing which shows the segment name of the segment containing the procedure to be added to the SL file.

(Note: The −LISTUSL command was used for demonstrative purposes only inasmuch as there is only one segment (SEG') in USL file USL1.)

The segment is added to the SL file using the −ADDSL command.

In figure 12-8, a program file (PROG1) is created with a :BUILD command and the calling program (FTRAN40) is compiled and prepared into this file using the :FORTPREP command.

The :RENAME command is used to rename the SL file SEARCH to SL (recall that the SL file must have the local name SL in order to be searched by a :RUN command).

The LIB = G parameter appended to the :RUN command specifies that the Group library is to be searched first for the referenced procedure.

```
:BUILD USL1;CODE=USL
:FORTRAN FTRAN39,USL1

PAGE 0001    HP32102B.00.0


00001000          FUNCTION DISP(CYL,RAD,HGT,WEIGHT,DISTANCE,TRANSCHG)
00002000          DISP=CYL*(3.14159*(RAD**2)*HGT)
00003000          TRANSCHG=WEIGHT+DISTANCE*.0019
00004000          RETURN
00005000          END



****      GLOBAL STATISTICS        ****
****   NO ERRORS,   NO WARNINGS   ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:00:04

 END OF COMPILE
:SEGMENTER

SEGMENTER SUBSYSTEM (C.0)
-BUILDSL SEARCH,300,1
-USL USL1
-LISTUSL

USL FILE USL1.PUB.GOODWIN

SEG'
   DISP                 24  P  A C N R

FILE SIZE          377600
DIR. USED              37          INFO USED              24
DIR. GARB.              0          INFO GARB.              0
DIR. AVAIL.         37541          INFO AVAIL.        337554
-ADDSL SEG'
-EXIT

 END OF SUBSYSTEM
```

Figure 12-7. Adding a Procedure to an SL Library File

```
:BUILD PROG1;CODE=PROG
:FORTPREP FTRAN4Ø,PROG1

PAGE ØØØ1   HP321Ø2B.ØØ.Ø

             PROGRAM SL LIBRARY
       C
       C EXAMPLE PROGRAM TO CALL SL LIBRARY PROCEDURE
       C
        1ØØ  FORMAT(T5,"DISPLACEMENT OF THIS VEHICLE IS ",F14.5)
        2ØØ  FORMAT(T5,"TOTAL COST IS ",M12.2)
        3ØØ  FORMAT(T5,"REGISTRATION COST IS ",M12.2//)
             DISPLAY "WEIGHT?"
             ACCEPT A
             DISPLAY "DISTANCE?"
             ACCEPT B
             DISPLAY "COST?"
             ACCEPT COST
             DISPLAY "NO. CYLINDERS?"
             ACCEPT C
             DISPLAY "BORE?"
             ACCEPT D
             DISPLAY "STROKE?"
             ACCEPT E
             D=D/2
       C
       C THE NEXT STATEMENT CALLS THE SL
       C LIBRARY PROCEDURE "DISP"
       C
             DISPL=DISP(C,D,E,A,B,T)
             WHOLECOST=COST+T
             TAX=1.5*DISPL
             REGISTRATION=SQRT(DISPL)*6.5
             WRITE(6,1ØØ)DISPL
             WRITE(6,2ØØ)WHOLECOST
             WRITE(6,3ØØ)REGISTRATION
             STOP
             END


****      GLOBAL STATISTICS       ****
****   NO ERRORS,   NO WARNINGS   ****
TOTAL COMPILATION TIME  Ø:ØØ:Ø1
TOTAL ELAPSED TIME      Ø:ØØ:Ø3

 END OF COMPILE

 END OF PREPARE
:RENAME SEARCH,SL
:RUN PROG1;LIB=G


WEIGHT?
?4512

DISTANCE?
?3463

COST?
?7645.32

NO. CYLINDERS?
?6

BORE?
?4.647

STROKE?
?3.549

   DISPLACEMENT OF THIS VEHICLE IS      361.15375
   TOTAL COST IS   $12,163.9Ø
   REGISTRATION COST IS      $123.53


 END OF PROGRAM
```

Figure 12-8. Referencing an SL Library Procedure from a Program

NON-FORTRAN/3000 PROGRAM UNITS

Any non-FORTRAN/3000 language program unit may be used as part of an executable FORTRAN/3000 program, provided the program unit has a calling sequence and method of execution compatible with FORTRAN/3000. In addition, a FORTRAN/3000 subprogram can be used by a program written in some other language, as long as its use is compatible with the calling program's requirements.

All arguments of a subprogram written in FORTRAN/3000 are passed by *reference*. This means that the *address* where the value is located is passed instead of the actual value of the argument. Thus, a FORTRAN/3000 subprogram expects a list of addresses for the formal arguments passed to it, one for each argument and in the order given by the dummy argument list contained within the subprogram.

A function reference or CALL statement prepares a list of addresses for the actual arguments associated with the call. In a function subprogram, space is allocated immediately before the address list for the value associated (returned) with the function name after execution. A subprogram written in FORTRAN/3000 deletes the actual argument addresses from its data space after it is executed. Any FORTRAN/3000 program referencing a non-FORTRAN/3000 language subprogram expects that subprogram to delete its actual parameter addresses.

Although values normally are passed by reference, or *indirectly*, they can be passed by value, or *directly*, in order to facilitate invoking non-FORTRAN/3000 language procedures which allow passing of arguments by value. To accomplish this, the expressions used as actual arguments are enclosed in back slashes (\). This tells the FORTRAN/3000 compiler to pass the actual value instead of the address where the value is located.

For example,

```
     CALL SUBR (\6.4\,\8.7\)
     CALL SUBR (\A\,\B\,\C\)
```

In both examples above, the actual values are passed to the referenced procedure SUBR.

A-1. SPL/3000 PROGRAMS

SPL/3000 (Systems Programming Language for the HP 3000 Computer System) will be the language used most frequently for non-FORTRAN/3000 external procedures. SPL/3000 procedures may be invoked by FORTRAN/3000 in the same manner as function and subroutine subprograms written in FORTRAN/3000, except that actual arguments may be passed by value.

SPL/3000 programs do not accept complex values as do FORTRAN/3000 programs. Double precision real numbers in FORTRAN/3000 are called long in SPL/3000 although their use is compatible. FORTRAN/3000 statement labels are not useful to an SPL/3000 program, and SPL/3000 labels are not allowed in FORTRAN/3000; they cannot be passed as actual arguments between one language and the other.

Note that SPL/3000 supports one-dimensional arrays, so that multi-dimensional arrays passed by FORTRAN/3000 will be linearized (see Section V, paragraph 5-8, *Equivalence Between Arrays of Different Dimensions).* Arrays are passed between SPL/3000 and FORTRAN/3000 programs by supplying an array element as the actual parameter by reference (address). The address of the first element of an SPL/3000 array points to the zeroth value, while the address of the first element of a FORTRAN/3000 array points to the first value. Note also that a FORTRAN/3000 character value of length *n* corresponds to an SPL/3000 byte array of *n* elements.

The form of the SPL/3000 procedure SORTINITIAL is shown below:

```
PROCEDURE SORTINITIAL(INPUTFILE,
  OUTPUTFILE,OUTPUTOPTION,RECLEN,
  NUMRECS,NUMKEYS,KEYS,ERRORPROC,
  KEYCOMPARE,STATISTICS,FAILURE);
VALUE INPUTFILE,OUTPUTFILE,
  OUTPUTOPTION,RECLEN,NUMKEYS;
DOUBLE NUMRECS;
ARRAY KEYS,STATISTICS;
PROCEDURE ERRORPROC;
LOGICAL PROCEDURE KEYCOMPARE;
LOGICAL FAILURE;
OPTION VARIABLE,EXTERNAL;
```

Note: SORTINITIAL is part of the SORT/3000 subsystem and as such can be called using the SYSTEM INTRINSIC declaration as described in paragraph A-4. It is used here merely because it illustrates most of the problems encountered when calling SPL/3000 procedures from FORTRAN/3000.

The SORTINITIAL procedure above is specified as being *OPTION VARIABLE*. This means that some parameters need not be supplied. Since this feature does not exist in FORTRAN/3000, you must supply an extra logical value argument, which is appended to the complete argument list. This value consists of one or more 16-bit *parameter mask words.* This extra parameter serves as a "bit map," with each unique bit representing a parameter in the parameter list. The rightmost bit represents the last parameter in the list, the second rightmost bit the next-to-last parameter, etc. Each "on" bit (bit = 1) indicates a required parameter to SORTINITIAL and each "off" bit (bit = 0) represents a parameter that is being supplied a dummy value. For example, the following statement calls SORTINITIAL, which is OPTION VARIABLE, and whose parameter list contains eleven parameters.

```
CALL SORTINITIAL (\IN\, \OUT\, \0\, \0\,
\0.0\, \1\, KEYS, \0\, \0\, \0\, F,\%3061\)
```

Values are supplied for the first, second, sixth, seventh, and eleventh parameters and dummy values (zeros) are supplied for the third, fourth, fifth, eight, ninth, and tenth parameters. For one parameter (NUMRECS), a double value is required, so the dummy value \0.0\ is furnished. SORTINITIAL must be told which parameters are being passed and so the octal value 3061 is appended to the parameter list as the mask word. Thus, the mask word

Bits	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	0	0	0	0	0	1	1	0	0	0	1	1	0	0	0	1
						3			0			6			1	

informs the SORTINITIAL procedure that values are being passed for parameters INPUTFILE, OUTPUTFILE, NUMKEYS, KEYS, and FAILURE in the SORTINITIAL call and that dummy parameters (zeros) are being passed for the remaining parameters. Note that KEYS and FAILURE (F) are passed by reference (no back slashes).

If more than 16 arguments are specified by a procedure, then two or more words are required for this masking. Again, the last bit represents the rightmost argument.

Figure A-1 is an example of calling the sortinitial procedure to sort a file (MAILLIST) and write the sorted file into another file (NEWLIST).

Note: A composite number may be used to represent the bit map. All arguments are positional and space must be allocated in the argument list according to the data type required, i.e., a real variable needs two words, an integer one word. Arguments passed by reference require only one word.

A-2. FUNCTIONS WITHOUT PARAMETERS

When a non-FORTRAN/3000 language procedure is a function which does not require parameters, then its use in FORTRAN/3000 requires special consideration because FORTRAN/3000 requires that a parameter be included in a function call.

If a dummy parameter is specified, the function will execute. The result of executing this procedure, however, is that an extra value will be placed on the stack. If such a procedure is called from within a DO loop, the program can abort. To take care of this extra value on the stack, a function call to a procedure that does not require parameters should be made from a subroutine. The subroutine's return clears the extra stack value.

See figure A-2 for an example of calling an SPL function which does not require parameters. (Any non-FORTRAN/3000 language function procedure which does not require parameters should be called in the same manner.)

A-3. DATA TYPES

The following data types are available in FORTRAN/3000, with SPL/3000 correspondence, as follows:

FORTRAN/3000	SPL/3000
REAL	REAL
DOUBLE PRECISION	LONG
CHARACTER	BYTE
LOGICAL	LOGICAL
INTEGER	INTEGER
DOUBLE INTEGER	DOUBLE INTEGER
COMPLEX	TWO ELEMENT REAL ARRAY

It is important to note that the FORTRAN types CHARACTER and COMPLEX are not the same as the SPL types BYTE and TWO ELEMENT REAL ARRAY but the correspondence is possible since the compilers implement such types similarly. When passing parameters requiring the correspondence of CHARACTER to BYTE or COMPLEX to TWO ELEMENT REAL ARRAY from a FORTRAN procedure to an SPL procedure, do not use OPTION CHECK 3 in the called SPL procedure. This option directs the Segmenter to check the procedure type, number of parameters, and type of each parameter for exact correspondence. Since the types COMPLEX and CHARACTER are not available in SPL, the use of OPTION CHECK 3 results in a Segmenter error. To avoid this problem use a CHECK number of 2 or less.

A-4. SYSTEM INTRINSICS

The MPE file SPLINTR.PUB.SYS contains information concerning the attributes of a set of subprograms. These subprograms are usually, but not always, user-callable system subprograms, such as FOPEN. In particular, all intrinsics mentioned in the MPE manuals may be accessed through this facility. The information about a specific subprogram includes such items as the number and type of parameters, whether parameters are by value or by reference, and whether the subroutine is SPL OPTION VARIABLE. (See the *SPL Reference Manual* for a discussion of OPTION VARIABLE). FORTRAN/3000 will read the SPLINTR file for specially designated subprograms and will generate the indicated code sequences.

```
:FORTRAN FTRAN41

PAGE 0001    HP32102B.00.0

$CONTROL INIT,FILE=21,FILE=22
      PROGRAM SPL3000 CALL
C
C EXAMPLE PROGRAM TO CALL SPL/3000 PROCEDURE
C
      CHARACTER*72 BUF
      INTEGER KEYS(6),FNUM
      LOGICAL FAILURE
C
C SORT THE FILE MAILLIST (FTN21) INTO NEWLIST (FTN22)
C SORT ON PHONE NUMBERS WITHIN STATES
C
C ESTABLISH THE KEYS - MAJOR AT 52 FOR 2 BYTES (STATE)
C MINOR AT 61 FOR 12 BYTES (PHONE NO)
C
      KEYS(1)=52
      KEYS(2)=2
      KEYS(3)=0
      KEYS(4)=61
      KEYS(5)=12
      KEYS(6)=0
C
C CALL SORTINITIAL TO START SORT PROGRAM
C
      CALL SORTINITIAL(\FNUM(21)\,\FNUM(22)\,\0\,\0\,\0.0\,
     #\2\,KEYS,\0\,\0\,\0\,FAILURE,\%3061\)
      IF(FAILURE)STOP 10
C
C CALL SORTEND WHEN SORT IS COMPLETE
C
      CALL SORTEND
      IF(FAILURE)STOP 20
C
C READ AND DISPLAY OUTPUT FILE
C
      REWIND 22
 30   READ(22,END=100)BUF
 50   FORMAT(T2,S)
      WRITE(6,50)BUF
      GO TO 30
 100  STOP
      END


****       GLOBAL STATISTICS         ****
****    NO ERRORS,    NO WARNINGS    ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:00:03

 END OF COMPILE
:BUILD PROG1;CODE=PROG
:PREP $OLDPASS,PROG1;MAXDATA=4000

 END OF PREPARE


:FILE FTN21=MAILLIST,OLD
:FILE FTN22=NEWLIST,OLD
:RUN PROG1

SPACE     MANN     9999 GALAXY WAY     UNIVERSE    CA 61239 231-999-9999
JENNA     GRANDTR  493 TWENTIETH ST    PROGRESSIVE CA 61335 799-191-9191
KING      ARTHUR   329 EXCALIBUR ST    CAMELOT     CA 61322 812-200-0100
SWASH     BUCKLER  497 PLAYACTING CT   MOVIETOWN   CA 61497 NONE
ALI       BABA     40 THIEVES WAY      SESAME      CO 69142 NONE
KARISSA   GRANDTR  7917 BROADMOOR WAY  BIGTOWN     MA 21799 713-244-3717
JANE      DOE      3959 TREEWOOD LN    BIGTOWN     MA 21843 714-399-4563
JOHN      DOUGHE   239 MAIN ST         HOMETOWN    MA 26999 714-411-1123
JAMES     DOE      4193 ANY ST         ANYTOWN     MD 00133 237-408-7100
KNEE      BUCKLER  974 FISTICUFF DR    PUGILIST    ND 04321 976-299-2990
JOHN      BIGTOWN  965 APPIAN WAY      METROPOLIS  NY 20013 619-407-2314
LOIS      ANYONE   6190 COURT ST       METROPOLIS  NY 20115 619-732-4997
BLACK     BEAR     47 ALLOVER DR       ANYWHERE    US 00111 NONE
 END OF PROGRAM
```

Figure A-1. Calling an SPL/3000 Procedure from FORTRAN/3000

```
:FORTGO NOPARMS3

PAGE ØØØ1    HP321Ø2B.ØØ.Ø


      PROGRAM NOPARMS1
C
C THIS PROGRAM CALLS THE SPL PROCEDURE GETJCW
C WHICH DOES NOT REQUIRE ANY PARAMETERS.  THE
C PROCEDURE IS CALLED FROM A SUBROUTINE SO THAT
C AN EXTRA VALUE IS NOT LEFT ON THE STACK.
C
      LOGICAL JCW
      JCW=.TRUE.
      DISPLAY "JCW IS ",JCW
      CALL FORTGETJCW(JCW)
      DISPLAY "JCW IS ",JCW
      STOP
      END



      SUBROUTINE FORTGETJCW(JCW)
      LOGICAL JCW,GETJCW
C
C THE \Ø\ IN THE NEXT STATEMENT IS A
C DUMMY PARAMETER USED TO FAKE OUT THE
C FORTRAN COMPILER.  THE PROBLEM IS THAT
C FORTRAN REQUIRES ALL FUNCTIONS TO HAVE PARAMETERS
C
      JCW=GETJCW(\Ø\)
      RETURN
      END





****      GLOBAL STATISTICS       ****
****   NO ERRORS,   NO WARNINGS   ****
TOTAL COMPILATION TIME  Ø:ØØ:Ø1
TOTAL ELAPSED TIME      Ø:ØØ:43

 END OF COMPILE

 END OF PREPARE


JCW IS      -1
JCW IS       Ø

 END OF PROGRAM
```

Figure A-2. Calling an SPL Function that does not Require Parameters

The appearance of "SYSTEM INTRINSIC" followed by a list of system intrinsics separated by commas implies the SPLINTR file is to be searched for names appearing in the list. If no such name can be found, an error message is displayed. The SYSTEM INTRINSIC statement must appear before any executable statements in a program.

Use of this facility provides three main conveniences over the usual way of accessing external subprograms:

1. Convenient access to SPL OPTION VARIABLE routines is provided. For each such subprogram, the list of actual parameters need not be complete and may not include the mask word(s) that signify which parameters are present. Missing parameters are indicated by commas or a right parenthesis. The occurrence of a right parenthesis before the formal parameter list is exhausted implies the rest of the parameters are missing. The parameter presence mask word(s) are automatically generated by the compiler from the actual parameter list.

2. The value or reference attribute of a formal parameter is recognized and the appropriate code is generated automatically to stack actual parameters for the call. The use of "/" to specify value parameters will be ignored if the parameter is actually passed by value. If the parameter is actually passed by reference, the "/" is ignored and a warning message is generated. Parameter checking is done at the highest level.

3. Automatic typing of SPLINTR file functions. Thus, SYSTEM INTRINSIC FOPEN, BINARY will result in FOPEN being typed as integer and BINARY being typed as logical.

   For example,

   The source

   ```
   SYSTEM INTRINSIC FCHECK
   INTEGER ERRCODE
   CALL FCHECK(0,ERRCODE)
   ```

   would result in the following code

   ```
   ZERO
   LRA ERRCODE
   DZRO,ZERO
   LDI %30 <<OPTION VARIABLE MASK>>
   PCAL FCHECK
   ```

   The source

   ```
   SYSTEM INTRINSIC BINARY
   LOGICAL RESULT
   INTEGER LENGTH
   CHARACTER*8 STRING
         .
         .
         .
   RESULT= BINARY(STRING,LENGTH)
   ```

   would result in the following code

   ```
   ZERO
   LRA STRING
   LOAD LENGTH
   PCAL BINARY
   STOR RESULT
   ```

Compare the examples shown in figures A-3 and A-4. Note that the back slashes were not necessary for the call to PRINTOP in figure A-4. The FORTRAN/3000 compiler still passed the value parameters by value.

Now compare the examples shown in figures A-5 and A-6. Note that in figure A-5, GETJCW was automatically typed logical and the stack was adjusted automatically by the compiler to take into account the fact that function GETJCW has no parameter.

In figure A-6, note that FOPEN is typed integer; parameter 2, parameter 4 and all succeeding parameters are missing; and three of FCHECK'S optional parameters are missing.

A-5. VALUE ARGUMENTS

Whenever an argument is specified by value, and the argument is not an argument of a system intrinsic, the value or variable must be enclosed in back slashes, e.g., \ VAR\ or \ 25\. The argument must also be of the correct length according to the argument's type.

A-6. CONDITION CODES

Frequently, condition codes are returned to a FORTRAN/3000 program by system intrinsics. These condition codes have the following general meanings. (Specific meanings depend on each individual intrinsic. Refer to the *MPE Intrinsics Reference Manual* for condition codes for specific intrinsics.)

```
:FORTGO FTRAN42

PAGE ØØØ1   HP321Ø2B.ØØ.Ø

        PROGRAM PRINTOP
C
C EXAMPLE PROGRAM TO CALL SYSTEM INTRINSIC PRINTOP
C
        CHARACTER MESSAGE*14
        LOGICAL LMESSAGE(7)
        EQUIVALENCE(LMESSAGE,MESSAGE)
        DATA MESSAGE/"THIS IS A TEST"/
        CALL PRINTOP(LMESSAGE,\-14\,\Ø\)
        IF(.CC.)2Ø,1Ø,2Ø
  1Ø    STOP
  2Ø    DISPLAY "INTRINSIC RETURNED BAD CONDITION CODE"
        STOP
        END


****      GLOBAL STATISTICS       ****
****    NO ERRORS,   NO WARNINGS  ****
TOTAL COMPILATION TIME  Ø:ØØ:Ø1
TOTAL ELAPSED TIME      Ø:ØØ:Ø4

 END OF COMPILE

 END OF PREPARE


 END OF PROGRAM
```

Figure A-3. Example of Condition Code Check

```
PAGE 0001   HP32102B.00.0


        PROGRAM PRINTOP2
C
C EXAMPLE PROGRAM TO CALL SYSTEM INTRINSIC PRINTOP
C
        CHARACTER MESSAGE*14
        LOGICAL LMESSAGE(7)
        SYSTEM INTRINSIC PRINTOP
        EQUIVALENCE(LMESSAGE,MESSAGE)
        DATA MESSAGE/"THIS IS A TEST"/
        CALL PRINTOP(LMESSAGE,-14,0)
        IF (.CC.)20,10,20
10      STOP "SUCCESSFUL WRITE"
20      STOP "INTRINSIC RETURNED BAD CONDITION CODE"
        END




****      GLOBAL STATISTICS        ****
****    NO ERRORS,   NO WARNINGS   ****
TOTAL COMPILATION TIME   0:00:03
TOTAL ELAPSED TIME       0:00:50

 END OF PROGRAM
:PREPRUN $OLDPASS;LIB=G

 END OF PREPARE

STOP  SUCCESSFUL WRITE

 END OF PROGRAM
```

Figure A-4. SYSTEM INTRINSIC Statement Example

```
:FORTGO NOPARMS2

PAGE 0001    HP32102B.00.0


00001000          PROGRAM NOPARMS2
00002000   C
00003000   C   EXAMPLE OF A PROGRAM CALLING A SYSTEM
00004000   C   INTRINSIC WITH NO PARAMETERS.
00005000   C
00006000          SYSTEM INTRINSIC GETJCW
00007000          LOGICAL JCW
00008000          JCW=.TRUE.
00009000          DISPLAY "JCW IS ",JCW
00010000          JCW=GETJCW
00011000          DISPLAY "JCW IS ",JCW
00012000          STOP
00013000          END




****       GLOBAL STATISTICS        ****
****     NO ERRORS,    NO WARNINGS   ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:00:13

 END OF COMPILE

 END OF PREPARE


JCW IS       -1
JCW IS        0

 END OF PROGRAM
```

Figure A-5. Calling a SYSTEM INTRINSIC Function that does not Require Parameters

```
PAGE 0001   HP32102B.00.0


00001000  $CONTROL MAP
00002000        PROGRAM OPENFILE
00003000  C
00004000  C THIS EXAMPLE SHOWS THE USE OF FOPEN IN A SYSTEM
00005000  C INTRINSIC STATEMENT.  IN PARTICULAR THIS ILLUSTRATES
00006000  C THE HANDLING OF MISSING PARAMETERS AND THE AUTOMATIC
00007000  C TYPING OF SPL FUNCTION NAMES.
00008000  C
00009000        SYSTEM INTRINSIC FCHECK
00010000        SYSTEM INTRINSIC FOPEN
00011000        INTEGER*2 FNUM
00012000        CHARACTER*8 FNAME
00013000        DATA FNAME/"TEMPFILE"/
00014000        FNUM=FOPEN(FNAME,,%3L)
00015000        IF(.CC.)10,20,10
00016000  10    DISPLAY "OPEN FAILED"
00017000        CALL FCHECK(FNUM,IERRCODE)
00018000        DISPLAY "FCHECK ERROR CODE WAS ",IERRCUDE
00019000        STOP
00020000  20    DISPLAY "MPE FILE NUMBER WAS",FNUM
00021000        STOP 'OPEN SUCCESSFUL'
00022000        END


  SYMBOL MAP

NAME                TYPE        STRUCTURE    ADDRESS

FCHECK                          SUBROUTINE
FNAME               CHARACTER   SIMPLE VAR   Q+   1,I
FNUM                INTEGER     SIMPLE VAR   Q+   3
FOPEN               INTEGER     FUNCTION
IERRCODE            INTEGER     SIMPLE VAR   Q+   2




****       GLOBAL STATISTICS          ****
****    NO ERRORS,    NO WARNINGS     ****
TOTAL COMPILATION TIME    0:00:04
TOTAL ELAPSED TIME        0:01:59

 END OF PROGRAM
:PREPRUN $OLDPASS

 END OF PREPARE


MPE FILE NUMBER WAS        1
STOP  OPEN SUCCESSFUL
 END OF PROGRAM
:
```

Figure A-6. Option Variable SYSTEM INTRINSIC Example

CONDITION CODE	MEANING
CCE	Condition code is zero. This generally indicates that the request was granted.
CCG	Condition code is greater than zero. A special condition occurred but may not have affected the execution of the request. (For example, the request was executed, but default values were assumed as intrinsic call parameters.)
CCL	Condition code is less than zero. The request was *not* granted, but the error condition may be recoverable. Beyond this condition code, some instrinsics return further error information to the calling program through their return values.

The condition code is checked by an arithmetic IF statement in the FORTRAN/3000 calling program. The special argument, .CC., is used. Branching occurs to the appropriate statement label on the conditions less than, equal, or greater than.

Note: If the procedure is a function and the returned value is to an array element, the condition code will not be valid because of the intermediate instruction necessary to handle the array subscripts. This invalidity applies also if the value is used in an arithmetic expression.

The intrinsic PRINTOP was shown in figure A-3. This intrinsic called through FORTRAN/3000 uses arguments passed by value. The sample program also illustrates the condition code check.

Note that in the PRINTOP intrinsic procedure, the parameter MESSAGE is a *word* (16 bit) array. When the parameter MESSAGE is used in the FORTRAN/3000 calling program, it is declared as Type CHARACTER with a length of 14, which assigns it a length of 14 *bytes*. The statement,

```
LOGICAL LMESSAGE(7)
```

declares a logical variable with a length of 7 words, and this variable and MESSAGE are equivalenced so that a *word value* of the proper length will be passed to the intrinsic.

Note: Most MPE/3000 intrinsics do not check the type or number of arguments.

A-7. OPTION VARIABLE

When OPTION VARIABLE (meaning that certain parameters in an intrinsic are optional) is specified by an intrinsic, an extra logical value argument is appended to the complete argument list. See paragraph A-1 for a description of this argument.

A-8. SUMMARY

The following items should be checked, then, when calling system intrinsics from FORTRAN/3000:

DATA TYPES
VALUE LENGTHS
OPTION VARIABLE
FUNCTIONS WITHOUT ARGUMENTS
CONDITION CODE SETTINGS

Figure A-7 shows a FORTRAN/3000 program which calls the system intrinsic COMMAND.

See the *MPE Intrinsics Reference Manual* for more information about the COMMAND intrinsic.

```
:FORTGO FTRAN44

PAGE 0001    HP32102B.00.0

             PROGRAM COMMAND
      C
      C THIS IS AN EXAMPLE OF USING THE COMMAND INTRINSIC
      C TO SET UP A FILE COMMAND
      C
             INTEGER RECNUM
             CHARACTER*72 BUFFER,UPDATE
             BUFFER="FILE FTN20=MAILLIST,OLD;ACC=UPDATE"
             BUFFER[35:1]=%15C
             CALL COMMAND(BUFFER,IERR,IPARM)
             IF(.CC.)100,10,100
        10   DISPLAY "INPUT RECORD NUMBER TO START UPDATE"
        20   ACCEPT RECNUM
             DISPLAY "INPUT UPDATE INFORMATION"
             ACCEPT UPDATE
             IF(UPDATE[1:3].EQ."END")GOTO 150
             WRITE(20@RECNUM,ERR=40,END=50)UPDATE
             READ(20@RECNUM,ERR=60,END=70)UPDATE
             DISPLAY "THE NEW RECORD IS:"
             DISPLAY UPDATE
             DISPLAY "NEXT RECORD TO UPDATE?"
             GOTO 20
        40   DISPLAY "WRITE ERROR"
             STOP
        50   DISPLAY "FILE FULL"
             STOP
        60   DISPLAY "READ ERROR"
             STOP
        70   DISPLAY "END OF FILE"
             STOP
        100  DISPLAY "INTRINSIC RETURNED BAD CONDITION CODE"
        150  STOP
             END


****       GLOBAL STATISTICS        ****
****    NO ERRORS,   NO WARNINGS    ****
TOTAL COMPILATION TIME  0:00:01
TOTAL ELAPSED TIME      0:00:04

  END OF COMPILE

  END OF PREPARE


INPUT RECORD NUMBER TO START UPDATE
?1

INPUT UPDATE INFORMATION
?FREDDIE   JOHNSONE

THE NEW RECORD IS:
FREDDIE   JOHNSONE

NEXT RECORD TO UPDATE?
?5

INPUT UPDATE INFORMATION
?MARY        MEEK

THE NEW RECORD IS:
MARY        MEEK

NEXT RECORD TO UPDATE?
?1

INPUT UPDATE INFORMATION
?END

  END OF PROGRAM
```

Figure A-7. Calling the COMMAND Intrinsic

FORTRAN/3000 AND ANSI STANDARD FORTRAN

APPENDIX B

FORTRAN/3000 conforms to the American National Standard Institute's (ANSI) Standard for FORTRAN (X3.9 - 1966). To provide a more powerful programming tool, FORTRAN/3000 extends beyond the Standard and, in some minor cases, places restrictions on the Standard to conform with the HP 3000 Computer System architecture. A brief description of each extension or restriction appears below. Numbers in the "Standard Reference" column are references to the appropriate text in the Standard (X3.9 - 1966).

Standard Reference	Comments
3.	Program preparation can occur in a-free-field format as well as a fixed-field format.
3.1	FORTRAN/3000 uses a 128-character USACII 8-bit standard character set. All printing characters can appear in Hollerith and string values. Some of the control characters are reserved for special purposes (such as carriage return or line feed).
3.2	End lines can appear with a statement label preceding.
3.5	Symbolic names consist of as many as 15 characters instead of just 6.
4.	Character-type data can be used in FORTRAN/3000 programs to facilitate string manipulation. A double integer data type also is available.
4.2	Logical data can be manipulated as 16-bit binary masks in addition to their function as true/false data.
5.1.1	Constants of all types can be specified in more than one way by using octal values, partial-word designators, etc. Character constants in the form of string or Hollerith values can also be used. Double integer constants may be specified.
5.1.3	FORTRAN/3000 allows arrays of up to 255 dimensions instead of just three dimensions allowed by the Standard. Subscript expressions are any linear expressions.

Standard Reference	Comments
5.3	The IMPLICIT statement can be used to generalize the data type associated with the first letter of an identifier to include integer, double integer, real, double precision, complex, or character. Function subprograms can determine their type through a Type statement within the subprogram defining unit.
6.	Expressions of type character can be used to facilitate the use of character data.
6.1	Expressions can be created using primaries of different types. In assignment statements, the resulting expression value type is converted to the type of the identifier on the left side of the assignment indicator. In exponentiation, constructs such as A**B**B are allowed (without the need for parentheses) and can use powers and bases of differing types. No base can be raised to a complex power, however. Partial-word designators allow manipulation of the subparts of integer or logical values.
6.3	FORTRAN/3000 includes an "exclusive OR" operator. The other relational operators are generalized. Expressions of type integer, double integer, real, or double precision can appear on one side of a relational operator with an expression of type integer, double integer, real, or double precision on the other side. Complex expressions can appear between equal (.EQ.) or not equal (.NE) signs only.
7.1.1	The identifier to the left of the assignment operator in an assignment statement need not be of the same type as the expression on the right of the operator. The expression value type is converted to the identifier type prior to

Standard Reference	Comments
	assignment. Partial-word designators can be used to assign parts of integer or logical variables. Character-type assignment statements can be used providing the left and right-hand parts are of type character.
	Label data and integer data are mutually exclusive. A variable of the same name can be assigned values of both types without ambiguity.
7.1.2	Additional control statements are the TRAP statement and the ENTRY statement.
7.1.2.1	The assigned GO TO statement does not require a list of labels. The computed GO TO can use a linear expression for its index for selecting the transfer statement.
7.1.2.4	A label can be used as an actual argument in a CALL statement to allow alternative return points following execution of the subroutine referenced by CALL.
7.1.2.5	An optional exit label can be included in the RETURN statement to return to one of the calling program unit's statements whose label appears as an actual argument to the subroutine containing the RETURN.
7.1.2.7	STOP or PAUSE statements use a decimal integer or a character string for identification rather than an octal integer.
7.1.2.8	FORTRAN/3000 supports an extended concept of DO ranges as discussed in this manual.
7.1.3	Direct-access files can be referenced in FORTRAN/3000 input/output statements. These statements allow extended format and error recovery capabilities.
7.2.11	Adjustable array declarators can be used for local arrays in subprograms to select different size arrays for each activation of a subprogram.
7.2.1.3	Since FORTRAN/3000 executes on a 16-bit word machine, integer and logical values require one word of computer memory, double integer values two, real values two, double precision four, complex four. Character data uses half-word storage units. The maximum number of pointers allowed for referencing variables (simple variables and/or arrays) allowed in COMMON is 254. However, with the $CONTROL MORECOM option, this limit can be increased to 254 *common blocks.* The number of variables in each common block has no upper limit except for the user's stack constraints.
7.2.1.6	Type statements for types character and double integer are available.
7.2.2	The DATA statement in FORTRAN/3000 extends beyond the Standard as described in this manual. The FORTRAN/3000 PARAMETER statement does not exist in ANSI FORTRAN.
7.2.3	Additional editing types other than those described in the Standard are available in FORMAT statements.
8.	FORTRAN/3000 provides secondary entry point statements.
8.1	The defining statement of a statement function can be any expression of the appropriate type. The expression can include array elements.
8.2	FORTRAN/3000 includes a larger set of intrinsic functions than listed in the Standard. FORTRAN/3000 supports generic naming of most intrinsics.
8.3	Recursion in function subprogram definition is allowed. The type of actual arguments in a function reference has been expanded. Arguments are all passed by reference rather than value.
8.3.3	The HP 3000 includes a larger set of basic external functions than listed in the standard. FORTRAN/3000 supports generic naming of most basic external functions.

Standard Reference	Comments
8.4	Subroutines can be defined recursively and can be called with the same actual argument types as function subprograms.
8.5	Block data subprograms can be given a name.
9.	Main program units can be given a name.
10.2	A variable that is defined is always available on the first and second level. For instance, an integer simple variable that is used for both label values and integer values. FORTRAN/3000 never confuses the two values. Any variable appearing in a DATA or COMMON statement remains defined until it is explicitly redefined.

FORTRAN/3000 AND HP 2100 FORTRAN

APPENDIX C

FORTRAN/3000 attempts to correspond to other versions of Hewlett-Packard FORTRAN whenever possible. Differences between the 2100 family and the 3000 family of computers, however, require that some differences exist in these two versions of FORTRAN. The following differences are deletions of certain aspects of HP 2100 FORTRAN.

- Octal constants are no longer represented by a B suffix following the constant but are prefixed with %.
- Array variables must explicitly reference all subscripts. Previously, 2100 FORTRAN filled in any omitted subscripts with 1's.
- An array declarator for the same array may not appear in both a DIMENSION and COMMON statement within the same program unit.
- An arithmetic IF statement must always include three statement labels, not just two.
- The logical IF statement cannot be followed by a pair of statement labels in place of an executable statement.
- An END line can contain no other non blank characters other than a statement label followed by the characters E, N, and D.
- Index expressions such as subscripts and computed GO TO indices cannot evaluate to a complex value.
- Hollerith constants cannot be used in place of integer constants or expressions.
- Statement function names and intrinsic function names cannot be passed as actual arguments for a dummy function name.
- Comments cannot separate a continuation line from its predecessor.

The following differences constitute modifications of various 2100 FORTRAN aspects.

- Intrinsics cannot be passed as actual arguments while Basic External Functions can.
- The @ and K format editing phrases are supplanted by the O editing phrase.
- Character strings appearing as free-field data are enclosed in double quotes (") or apostrophes (').

ERROR AND WARNING MESSAGES

APPENDIX D

During compilation of FORTRAN/3000 source programs, the compiler prints error messages to indicate conditions such as illegal syntax, or warning messages to warn of marginal conditions which may cause improper execution of the source program. When one of the above conditions occurs, the compiler prints a message which includes the error or warning number and a brief explanatory message.

D-1. ERROR MESSAGES

If an error condition occurs, the compiler outputs a message of the following form:

    ERROR *nnn message text*

where *nnn* is the message number and *message text* is a brief description of the error condition (the program continues to compile). Error conditions (as opposed to warnings) are fatal to the program unit. The compiler deletes any object code generated for the current program unit and attempts to compile the next program unit. If a :FORTPREP or :FORTGO command is being used, the preparation (and execution stage for :FORTGO) stage is suppressed.

D-2. WARNING MESSAGES

If a warning condition occurs, the compiler outputs a message of the following form:

    WARNING *nnn message text*

where *nnn* is the message number and *message text* is a brief description of the warning condition. Warnings do not inhibit successful compilation of a program unit or the generation of object code by the compiler. If not corrected, however, the conditions indicated by the message can cause program execution errors. The warning condition should be corrected and the program unit should be recompiled.

D-3. ERROR POSITION INDICATION

Error and warning messages relate to the source code in several ways, depending on the type of message at the time the compiler prints the message.

During syntax scanning, the compiler usually prints an up arrow (↑) or caret (∧) at or to the right of the error position in the line. If the compiler is not producing a source listing, the last line examined is printed before the error message is printed.

Some error messages do not refer to any particular part of the program (e.g., TOO MANY TRACE SYMBOLS) and have no error indication other than the message. (The solution is not tied to one symbol or statement.) In some cases, such as solving equivalences, the compiler prints an error message after reading the entire program unit. The compiler indicates which statement is being processed by printing the message:

    AT *statement number + offset*

where *statement number* is the label of the offending statement or the closest preceding statement label if the offending statement is not labeled. *Offset* is a count of the number of statements the offending statement is from the label preceding it. These messages also include the name of the variable being processed when the compiler notices the error.

Error messages such as TRACE SYMBOL NOT FOUND include a variable name or a label name to indicate the area of the program in error since the error depends upon omitted statements or statement parts.

Table D-1 lists errors and warnings, indicated by E and W, and the compiler action.

D-4. UNDEFINED VARIABLE DETECTION

A warning message is displayed if a variable appears in one or more of the following:

1. On the right hand side of an assignment statement.
2. As a value parameter to a subprogram.
3. As a list element in a WRITE or DISPLAY statement.

and if the same variable does not appear in one or more of the following:

1. On the left hand side of an assignment statement.
2. As a list element in a READ statement.
3. In a DATA statement.
4. As a formal parameter.
5. As a reference parameter to a subprogram.
6. In an EQUIVALENCE statement.
7. In a COMMON statement.

The warning will appear at the end of each program unit.

Figure D-1 is an example of undefined variable detection.

```
:FORTGO ENF2

PAGE ØØØ1   HP32102B.ØØ.Ø


ØØØØ1ØØØ  $CONTROL USLINIT
ØØØØ2ØØØ        PROGRAM SUBPRFUNCTION
ØØØØ3ØØØ  C     UNDEFINED VARIABLE DETECTION EXAMPLE
ØØØØ4ØØØ        INTEGER FACTORIAL
ØØØØ5ØØØ        ACCEPT L
ØØØØ6ØØØ        IFACT=FACTORIAL(L)
ØØØØ7ØØØ        STOP
ØØØØ8ØØØ        END



ØØØØ9ØØØ        INTEGER FUNCTION FACTORIAL(N)
ØØØ1ØØØØ        IF(N-1) 2Ø,2Ø,1Ø
ØØØ11ØØØ  1Ø    FACTORIAL=M*FACTORIAL(N-1)
ØØØ12ØØØ        RETURN
ØØØ13ØØØ  2Ø    FACTORIAL=1
ØØØ14ØØØ        RETURN
ØØØ15ØØØ        END
** WARNING  226 **  REFERENCED VARIABLE NOT DEFINED M



****  NO ERRORS,   1 WARNING   ****
PROGRAM UNIT FACTORIAL COMPILED



****       GLOBAL STATISTICS       ****
****   NO ERRORS,    1 WARNING    ****
TOTAL COMPILATION TIME  Ø:ØØ:Ø1
TOTAL ELAPSED TIME      Ø:ØØ:33

 END OF COMPILE

 END OF PREPARE

?
-15

 END OF PROGRAM
:
```

Figure D-1. Undefined Variable Detection Example

Table 3-3. FORTRAN Compiler Warning & Error Messages

NO.	TYPE	MESSAGE	MEANING	ACTION
0	E	COMPILATION TERMINATED	An error has occurred which aborts the compilation. Compiler appends cause of termination as part of message.	Examine listing. Fix all errors and recompile.
1	W	NON-DIGIT IN LABEL FIELD	Label can consist only of numbers. Compiler ignores label field.	Correct label and other references to it in program.
2	W	CONTINUATION LINE IN LABEL FIELD	Continuation line text entered in fixed format begins before column 7. Compiler ignores text in label field.	Correct position of text in line.
3	W	SYMBOLIC NAME EXCEEDS 15 CHARACTERS	A symbolic variable name exceeds 15 characters. Compiler truncates name to 15 characters.	Check all references to symbolic name and make sure they are consistent.
4	W/E	EXPECTED A COMMA	Compiler detected a missing comma. If warning, compilation continues as if comma included; if error, comma required.	None if warning; comma can be included for future compilations. If error, comma must be inserted.
5	W	EXTRANEOUS COMMA	Compiler detected an unnecessary comma and ignored it.	None required. Can correct for future compilations by deleting comma.
7	W	UNEXPECTED CHARACTER	Compiler detected an unexpected character and ignored it.	Examine statement format and delete extra character.
8	W	UNEXPECTED '-'	Compiler detected an unexpected hyphen or minus sign in a FORMAT statement. Unchanged statement is passed to the Formatter.	Check statement to determine whether '-' is necessary. If not, remove it.
9	W	UNEXPECTED COMMA	Compiler detected an unexpected comma in FORMAT statement. Unchanged statement is passed to the Formatter.	Check statement to determine whether comma is necessary If not, remove it.
10	W	UNEXPECTED ')'	Compiler detected an unexpected right parentheses in FORMAT statement. Unchanged statement is passed to Formatter.	Check statement to determine whether ')' is necessary. If not, remove.it.
11	W/E	EXPECTED AN INTEGER	Compiler expected to find an integer in FORMAT statement. Unchanged statement is passed to Formatter. If compiler control statement caused warning, unrecognized item ignored. If error, integer required.	Check statement to determine whether integer is necessary. If so, insert appropriate integer. If error, integer must be included and program recompiled.

Table 3-3. FORTRAN Compiler Warning & Error Messages (Continued)

NO.	TYPE	MESSAGE	MEANING	ACTION
12	W/E	EXPECTED A '.'	If warning, compiler expected decimal point in FORMAT statement; unchanged statement is passed to Formatter. If error reserved token such as .OR. requires trailing period.	Check statement to determine whether decimal point is necessary. If so, insert it.
13	W	EXPECTED A 'P'	Compiler expected a scale factor identifier in FORMAT statement. Unchanged statement is passed to Formatter.	Check statement to determine whether scale factor identifier is necessary. If so, insert it.
14	W	UNEXPECTED 'P'	Compiler encountered an unexpected scale factor identifier in FORMAT statement. Unchanged statement is passed to Formatter.	Check statement to determine whether scale factor identifier is necessary. If not, remove it.
15	W	NESTING EXCEEDS 5 LEVELS	A group of format and edit specifications in a FORMAT statement includes groups at a level greater than 4, making the total number of levels greater than 5.	Revise statement so that no group includes more than 4 levels of other groups.
16	E	EXTRANEOUS SOURCE	Compiler encountered symbol that is not logically part of statement. Since preceding part of statement is logically complete, compiler deleted symbol and following text, compiled text preceding symbol.	Examine statement format and make necessary change.
17	W	MULTIPLE SEGMENT NAMES	Segment names not the same when more than one $SEGMENT command without intervening program unit is specified. Compiler uses last observed segment name.	None.
18	W	SYMBOLIC NAME REDUNDANTLY TYPED	Symbolic name assigned the same type more than once. Compiler takes no action.	Remove redundant type declaration. Current compilation unaffected.
19	W	SYMBOLIC NAME REDUNDANTLY EXTERNALLED	Symbolic name appears in more than one EXTERNAL statement. Compiler takes no action	Remove name from extra EXTERNAL statement later. Current compilation unaffected.
20	W	EXTRANEOUS EQUATE GROUP	One of the list parameters of an EQUIVALENCE statement contains only one item.	Insert two or more variable names which share same storage in EQUIVALENCE list parameter (equate group).
22	W	EXTRA INITIAL VALUES	DATA statement contains more values than variables.	Check statement and either add missing variable or delete extra value.

Table 3-3. FORTRAN Compiler Warning & Error Messages (Continued)

NO.	TYPE	MESSAGE	MEANING	ACTION
23	W	EXTRANEOUS DATA ITEM	Compiler discovered variable name in a DATA statement in a BLOCK DATA subprogram which is not also in a COMMON block.	Check both DATA and COMMON statement for consistent variable names.
24	W	NO INITIAL VALUES	Compiler discovered a BLOCK DATA subprogram that did not specify initial values for a labeled COMMON block.	Check both DATA and COMMON statement for consistent variable names.
25	W	INITIAL VALUE TRUNCATED	Compiler truncated an initial value that was too large for its type.	Determine correct value and replace existing one in DATA statement.
26	W	STATEMENT FUNCTION DUMMY USED AS NON-SIMPLE VARIABLE	A dummy argument in a statement function is referenced elsewhere in subprogram as a non-simple variable, for example, as an array or procedure.	Review calling program unit to determine nature of inconsistency, make any necessary changes, and recompile.
27	W	EXPRESSION VS. NON-EXPRESSION ARGUMENT	Compiler discovered inconsistency between argument structure in this procedure call and previous procedure call in this program unit.	Review calling program unit to determine nature of inconsistency, make any necessary changes, and recompile.
28	W	SUBPROGRAM VS. NON-SUBPROGRAM ARGUMENT	Compiler discovered inconsistency between this and previous subprogram call in this program unit, such as subprogram name passed in one but not the other.	Review calling program unit to determine nature of inconsistency, make any necessary changes, and recompile.
29	W	SUBROUTINE VS. FUNCTION ARGUMENT	Compiler discovered inconsistency between actual arguments in two subprogram calls in this program unit.	Review calling program unit to determine nature of inconsistency, make any necessary changes, and recompile.
30	W	ARGUMENT TYPE INCONSISTENT	Calls made to the same subroutine reference the same dummy argument with actual arguments of different types.	Review calling program unit to determine nature of inconsistency, make any necessary changes, and recompile.
31	W	SUBPROGRAM NAME NOT EXTERNALLED	Compiler discovered a subprogram name not mentioned in an EXTERNAL statement.	Add EXTERNAL statement to calling program, and recompile.
32	W	ARGUMENTS OF NESTED REFERENCE NOT CHECKED	Compiler discovered a recursive call in the actual argument list of procedure call and informed user that only the actual arguments of initial call (not recursive call) are checked.	None.
33	W	INTRINSIC NAME CONVERTED TO SIMPLE VARIABLE	Compiler converted an intrinsic name to a simple variable. Name cannot refer to intrinsic in remainder of subprogram.	None.
34	W	STATEMENT CANNOT BE REACHED	Compiler discovered statement that cannot be reached (for example, an unlabeled statement following a RETURN statement).	Check program logic and add statement number if necessary.

Table 3-3. FORTRAN Compiler Warning & Error Messages (Continued)

NO.	TYPE	MESSAGE	MEANING	ACTION
36	W	RETURN OR STOP INSERTED	Compiler inserted a RETURN or STOP statement because an unlabeled END statement was not preceded by an unconditional transfer of control.	None. STOP or RETURN can be added if desired.
37	W	BLANK LINE ILLEGAL	Statement consisting of a blank line is illegal under ANSI standard. (All statements must have a keyword.) In general, compiler ignores blank line. If it follows END statement, compiler is reinitialized and ready to compile next program of which blank line is first statement.	None. Blank lines should be deleted.
38	E	TOO MANY CONTINUATION LINES	Statement consists of too many continuation lines. Compiler deletes entire statement.	Alter program logic as needed and recompile.
39	W	EXPECTED CONTINUATION LINE	Compiler expected a continuation line and did not find one.	Check source for possible missing text.
40	W	EXPECTED COMPILER CONTROL KEYWORD	Compiler discovered missing keyword, unrecognized keyword, or incomplete parameter. Compiler skipped past the next comma and continued to compile.	Make changes as necessary and recompile.
41	E	EXPECTED SYMBOLIC NAME	Compiler did not find symbolic name where one was required for proper statement form. Compiler deleted statement.	Alter source code and recompile.
42	W	NOT ACCEPTABLE AT THIS POINT	Compiler control command appeared too late in program; compiler ignored command.	Move compiler command if necessary and recompile.
43	E	IMPROPER STATEMENT LABEL	Compiler discovered improper statement label and ignored it.	Alter source code and recompile.
44	W	TRACE SYMBOL NOT FOUND	Compiler could not find symbol mentioned in TRACE command.	Alter $TRACE command and recompile.
45	W	INTRINSIC IN TRACE RECORD	Compiler discovered intrinsic name in TRACE command. Intrinsic will not be traced.	Alter $TRACE command if desired.
46	W	EXPECTED A '='	Compiler expected an = sign in compiler control command and skipped to next comma.	Alter command and recompile.
47	E	SEQUENCE FIELD TOO LONG	In free field format or in compiler control command, sequence number > 8 characters; compiler truncated to 8 characters.	None. Check source code for validity.

Table 3-3. FORTRAN Compiler Warning & Error Messages (Continued)

NO.	TYPE	MESSAGE	MEANING	ACTION
48	W	OUT OF SEQUENCE	Statement containing sequence number is out of sequence. Statement compiled as if it were next in sequence.	Check source code for validity and recompile if necessary.
49	W	TITLE TOO LONG	Title specified in $TITLE command exceeds 52 characters. Compiler truncated the title.	Check $TITLE command.
50	E	PROGRAM UNIT ABORTED – *message qualifier*	Compiler terminated the program unit being compiled but continues to compile succeeding program units.	Fix problem as indicated by *message qualifier* and recompile.
51	E	EXPECTED A '('	Compiler expected left parenthesis. Statement deleted.	Alter source code and recompile.
52	E	EXPECTED A ')'	Compiler expected right parenthesis. Statement deleted.	Alter source code and recompile.
53	E	EXTRANEOUS ')'	Compiler ignored an extra right parenthesis.	Alter source code and recompile.
55	E	EXPECTED A ']'	Compiler expected right bracket. Statement deleted.	Alter source code and recompile.
56	E	EXPECTED A '['	Compiler expected left bracket. Statement deleted.	Alter source code and recompile.
58	E	EXPECTED ASSIGNMENT OPERATOR	Compiler expected an = sign. Statement deleted.	Alter source code and recompile.
59	E	EXPECTED A '/'	Compiler expected a (/) and deleted the statement.	Alter source code and recompile.
60	E	EXPECTED A QUOTE	Compiler expected a string delimiter (') or ("). Statement deleted.	Alter source code and recompile.
61	E	EXPECTED A '\ '	Compiler expected a back slash (\ ) in a parameter list. Statement deleted.	Alter source code and recompile.
63	E	IMPOSSIBLE CONTEXT FOR '.'	Compiler found a period which did not logically fit into the statement syntax and ignored it.	Alter source code and recompile.
64	E	IMPOSSIBLE CONTEXT FOR '%'	Compiler found a percent sign (%) which did not logically fit into the statement syntax and ignored it.	Alter source code and compile.
65	E	IMPOSSIBLE CONTEXT FOR '$'	Compiler found a dollar sign ($) out of context and ignored it.	Alter source code and recompile.
67	E	SYMBOLIC NAME EXCEEDS 255 CHARACTERS	Compiler found a symbolic name exceeding 255 characters and deleted the statement containing the name.	Alter source code and recompile.
68	E	NUMBER EXCEEDS 255 CHARACTERS	Compiler found a number exceeding 255 characters and deleted the statement containing the number.	Alter source code and recompile.
69	E	STRING LITERAL EXCEEDS 255 CHARACTERS	Compiler found a string literal exceeding 255 characters and deleted the statement containing the string.	Alter source code and recompile.

Table 3-3. FORTRAN Compiler Warning & Error Messages (Continued)

NO.	TYPE	MESSAGE	MEANING	ACTION
70	E	HOLLERITH LITERAL EXCEEDS 255 CHARACTERS	Compiler found a Hollerith literal exceeding 255 characters and deleted the statement.	Alter source code and recompile.
71	E	HOLLERITH LITERAL TOO SHORT	Compiler found a Hollerith literal shorter than that specified by the length element of the literal and deleted the statement containing the literal.	Alter source code and recompile.
72	E/W	NON-OCTAL DIGIT	Compiler found a non-octal digit in an octal value and ignored the octal number.	Alter source code and recompile.
73	E	EXPECTED FIELD WIDTH	Compiler expected field width in composite number, and deleted the statement when no field width was found.	Alter source code and recompile.
74	E	EXPECTED FIELD VALUE	Compiler expected field value in composite number, and deleted the statement when no field value was found.	Alter source code and recompile.
75	E	PACKED NUMBER OVERFLOW	Composite or ASCII number overflowed; compiler used arbitrary value contained in the data space.	Check program logic, alter source code if necessary and recompile.
76	E	INTEGER CANNOT BE 0	Compiler encountered an integer value which cannot be 0 and changed it to 1.	Check program logic, alter source code if necessary and recompile.
77	E	INTEGER EXCEEDS CONTEXTUAL LIMITS	Absolute value of integer is too large in statement context; compiler sets value to 1.	Check program logic, alter source code if necessary and recompile.
78	E	INTEGER OVERFLOW	Integer overflow occurred and compiler used whatever arbitrary value was contained in the data space.	Check program logic, alter source code if necessary and recompile.
79	E	EXPECTED EXPONENT VALUE	Expected exponent value was incorrect or not supplied and compiler deleted the statement.	Alter source code - check data types and constants - recompile.
80	E	FLOATING LITERAL UNDER/OVERFLOW	Floating-point literal caused under or overflow and compiler used whatever arbitrary value was contained in the data space.	Change floating-point, literal to number within acceptable range, recompile.
81	E	IMPROPER COMPLEX LITERAL	Compiler found an improper complex literal and deleted statement containing it.	Change complex literal to correct format - recompile.
84	W	MISSING END LINE	Compiler expected an END statement at this point and supplied one.	Insert END statement before compile program again. Current compilation not affected.

Table 3-3. FORTRAN Compiler Warning & Error Messages (Continued)

NO.	TYPE	MESSAGE	MEANING	ACTION
85	E	UNEXPECTED CONTINUATION LINE	Compiler did not expect a continuation line to be used and deleted the entire statement up to (but not including) the next non-continuation line.	Alter source code and recompile.
87	E	RESERVED TOKEN NOT RECOGNIZED	Compiler expected a reserved symbol such as .OR. or .TRUE. and did not find one.	Alter source code and recompile.
88	E	CANNOT RECOGNIZE KEYWORD	Compiler expected FORTRAN/3000 keyword such as INTEGER or REAL or END and deleted statement when keyword was not recognized.	Alter source code and recompile.
89	E	CANNOT CLASSIFY STATEMENT	Statement possibly did not begin with a letter or is missing a right parenthesis so that compiler is totally unable to recognize statement.	Alter source code and recompile.
91	E	STATEMENT OUT OF POSITION	Compiler found declaration statement following an executable statement or declaration out of order relative to other declaration statements; statement deleted.	Alter source code and recompile.
92	E	DUMMY NAME NOT UNIQUE	Compiler found dummy parameter name not unique to program unit where it appears.	Alter source code and recompile.
93	E	IMPROPER DUMMY ARGUMENT	Compiler found improper dummy argument and stopped checking argument list.	Alter source code and recompile.
94	E	ARGUMENT ADDRESSIBILITY EXCEEDED	Compiler found more than 54 arguments in subroutine or function or too many arguments in statement function (in statement function, number of arguments allowed depends on type and complexity of expression).	Refer to Section XI of FORTRAN/3000 reference manual for rules; alter source code accordingly and recompile.
95	E	TOO MANY ALTERNATE RETURNS	Compiler found too many alternate return points specified in subprogram.	Refer to Section XI of FORTRAN/3000 reference manual for rules; alter source code accordingly and recompile.
96	E	IMPROPER TYPE CONSTRUCT	Compiler has found incorrect type construct and deleted statement.	Alter source code and recompile.
97	E	IMPROPER INITIAL LETTER CONSTRUCT	Format of letter construct in IMPLICIT statement is incorrect; statement is deleted.	Correct IMPLICIT statement and recompile.
99	E	SUBROUTINE CANNOT BE TYPED	Compiler has found subroutine subprogram with type explicitly declared.	Alter source code and recompile.

Table 3-3. FORTRAN Compiler Warning & Error Messages (Continued)

NO.	TYPE	MESSAGE	MEANING	ACTION
100	E	SYMBOLIC NAME TYPED INCONSISTENTLY	Compiler has found symbolic name whose type was declared in a manner inconsistent with a previous declaration.	Alter source code and recompile.
102	E	DYNAMIC BOUND DIMENSIONED	Compiler has found a dynamic bound defined in DIMENSION statement. Within subprogram, bound must be passed parameter.	Alter source code and recompile.
103	E	PROCEDURE DIMENSIONED	Compiler found procedure name in DIMENSION statement.	Alter source code and recompile.
104	E	ARRAY REDUNDANTLY DIMENSIONED	Compiler found an array whose dimensions were defined more than once in same program unit.	Alter source code and recompile.
105	E	EXPECTED BOUND	Compiler expected an array bound and deleted the statement.	Alter source code and recompile.
106	E	ARRAY EXCEEDS 32767 ELEMENTS	Compiler found an array defined with more than 32767 elements.	Alter source code and recompile.
107	E	NUMBER OF BOUNDS EXCEEDS 255	Compiler found an array with more than 255 bounds.	Alter source code and recompile.
108	E	DYNAMIC STRUCTURE IN COMMON	Compiler found a dynamically-defined array in COMMON statement.	Alter source code and recompile.
109	E	DUMMY NAME IN COMMON	Compiler found dummy parameter name in COMMON statement.	Alter source code and recompile.
110	E	ITEM IN COMMON TWICE	Compiler found item appearing in COMMON statement twice.	Alter source code and recompile.
111	E	COMMON BLOCK NAME ALSO PROCEDURE NAME	Compiler found name used as a COMMON block name and as name of a procedure.	Alter source code and recompile.
112	E	PROCEDURE NAME IN COMMON	Compiler found procedure name appearing in COMMON statement.	Alter source code and recompile.
113	E	CHARACTER FUNCTION HAS DYNAMIC LENGTH	Compiler found character function defined with dynamic length (using a variable length specification).	Alter source code and recompile.
114	E	SIMPLE VARIABLE OR ARRAY EXTERNALLED	Compiler found simple variable or array name used in an EXTERNAL statement.	Alter source code and recompile.
116	E	DYNAMIC STRUCTURE IN EQUATE	Compiler found a dynamic array (defined with variable memory allocation) used in EQUIVALENCE statement.	Alter source code and recompile.
117	E	DUMMY NAME IN EQUATE	Compiler found dummy parameter name in EQUIVALENCE statement.	Alter source code and recompile.
118	E	PROCEDURE NAME IN EQUATE	Compiler found procedure name in EQUIVALENCE statement.	Alter source code and recompile.

Table 3-3. FORTRAN Compiler Warning & Error Messages (Continued)

NO.	TYPE	MESSAGE	MEANING	ACTION
120	E	DYNAMIC STRUCTURE IN DATA	Compiler found a dynamic structure in DATA statement.	Alter source code and recompile.
121	E	DUMMY NAME IN DATA	Compiler found dummy parameter name in DATA statement.	Alter source code and recompile.
122	E	PROCEDURE NAME IN DATA	Compiler found procedure name in DATA statement.	Alter source code and recompile.
123	E	COMMON ITEM IN DATA ALLOWED ONLY IN BLOCK DATA	Initialization of COMMON data items is valid only in the BLOCK DATA. The item may not appear in the Data statement.	Alter source code and recompile.
124	E	EXPECTED INITIAL VALUE	Compiler expected an initial value for data element. Statement deleted when none was found.	Alter source code and recompile.
125	E	INITIAL VALUE TYPE IMPROPER	Compiler found initial value not coinciding with defined data element type.	Alter source code and recompile.
126	E	UNARY SIGN REQUIRES ARITHMETIC LITERAL	Compiler found minus or plus sign with no constant following.	Alter source code and recompile.
127	E	NUMBER OF SUBSCRIPTS <> NUMBER OF BOUNDS	Compiler found array element with a different number of elements than were specified for the array in the array declaration.	Alter source code and recompile.
128	E	ARRAY EXCEEDS 32767 WORDS	Compiler found array that occupies more than 32767 words of memory.	Alter source code and recompile.
129	E	SUBSCRIPT VALUE NOT IN ARRAY	Compiler found array element subscript that indicates memory location outside defined bounds of array.	Alter source code and recompile.
131	E	LOCAL ADDRESSIBILITY EXCEEDED	Compiler was unable to address all of the local variables in program unit.	Alter source code and recompile.
132	E	DYNAMIC BOUND NOT DUMMY INTEGER	Compiler found dynamic bound not represented by dummy integer.	Alter source code and recompile.
134	E	DATA BLOCK TOO LARGE	Compiler found data block larger than 32767 words.	Alter source code and recompile.
135	E	COMMON BLOCK TOO LARGE	Compiler found common block larger than 32767 words.	Alter source code and recompile.
136	E	COMMON EXTENDED FORWARD	Compiler found EQUIVALENCE statement that has attempted to extend common data space from beginning instead of from the end.	Alter source code and recompile.
137	E	EQUATE BLOCK TOO LARGE	Compiler found group of EQUIVALENCE statements which equivalence too large a block of data (more than 32767 words).	Alter source code and recompile.

Table 3-3. FORTRAN Compiler Warning & Error Messages (Continued)

NO.	TYPE	MESSAGE	MEANING	ACTION
140	E	WORD STRUCTURE ALIGNED ON BYTE BOUNDARY	Compiler found data value aligned by an EQUIVALENCE statement on a half-word (byte) boundary instead of full-word boundary.	Alter source code and recompile.

Table 3-3. FORTRAN Compiler Warning & Error Messages (Continued)

NO.	TYPE	MESSAGE	MEANING	ACTION
141	E	DATA BLOCK ITEM EQUATED TO COMMON BLOCK ITEM	Element defined in data block was illegally used in a common block; should have been in block data subprogram.	Alter source code and recompile.
142	E	TWO COMMON BLOCKS EQUATED	Compiler found two common blocks whose elements are equated through EQUIVALENCE statement.	Alter source code and recompile.
143	E	INCONSISTENT EQUATE	Array elements in EQUIVALENCE statement cause other elements of arrays to equate improperly; or two elements are equated that require unique data space (label values).	Alter source code and recompile.
144	E	SIMPLE VARIABLE HAS SUBSCRIPT	Compiler found simple variable with a subscript.	Alter source code and recompile.
145	E	EXPECTED STATEMENT LABEL	Compiler expected statement label and deleted statement when no label was found.	Alter source code and recompile.
147	E	DUPLICATE LABEL	Compiler found duplicate statement label and ignored the second occurrence.	Alter source code and recompile.
148	E	UNRESOLVED LABEL REFERENCE	Compiler found label referenced in statement but never found statement prefixed by that label. Referenced label ignored.	Alter source code and recompile.
149	E	FORMAT REFERENCE TO NON-FORMAT	Compiler found statement label reference to a non-FORMAT statement when a FORMAT statement was expected.	Alter source code and recompile.
150	E	EXECUTABLE REFERENCE TO NON-EXECUTABLE STATEMENT	Compiler found statement label reference to a non-executable statement.	Alter source code and recompile.
153	E	SUBROUTINE USED AS PRIMARY	Compiler found subroutine name used as a primary.	Alter source code and recompile.
154	E	EXPECTED ARITHMETIC PRIMARY	Compiler expected to find an arithmetic primary and deleted the statement when no primary was found.	Alter source code and recompile.
155	E	NON-ARITHMETIC PRIMARY WHERE ARITHMETIC EXPECTED	Compiler found non-arithmetic primary such as a logical variable when it expected an arithmetic primary and deleted statement.	Alter source code and recompile.
156	E	NON-LOGICAL OPERAND WHERE LOGICAL EXPECTED	Compiler expected to find a logical operand and deleted statement when non-logical operand was found.	Alter source code and recompile.
157	E	RELATIONAL OPERAND HAS LOGICAL TYPE	Compiler found logical operand in arithmetic relation. Statement deleted.	Alter source code and recompile.

Table 3-3. FORTRAN Compiler Warning & Error Messages (Continued)

NO.	TYPE	MESSAGE	MEANING	ACTION
158	E	CHARACTER VS. ARITHMETIC RELATION	Compiler expected arithmetic relation and deleted statement when character relation was found.	Alter source code and recompile.
159	E	ILLEGAL RELATION FOR COMPLEX OPERANDS	Compiler found illegal relational operator between two complex values.	Alter source code and recompile.
160	E	OPERAND OF .NOT. NOT LOGICAL	Compiler found non-logical operand following a .NOT. operator and deleted statement.	Alter source code and recompile.
161	E	IMPROPER STRING DESIGNATOR	Compiler discovered substring designator in improper form and deleted statement.	Alter source code and recompile.
162	E	COMPLEX POWER	Compiler found number raised to a complex power.	Alter source code and recompile.
163	E	COMPLEX BASE TO NON-INTEGER POWER	Compiler found complex number raised to non-integer power.	Alter source code and recompile.
164	E	STRING EXPRESSION IN PARENTHESIS	Compiler found string expression in parenthesis when it expected arithmetic expression and deleted statement.	Alter source code and recompile.
165	E	PARTIAL-WORD EXCEEDS 15 BITS	Compiler found partial-word designator that specifies more than 15 bits.	Alter source code and recompile.
166	E	IMPROPER TYPE FOR PARTIAL-WORD DESIGNATOR	Compiler found partial-word designator operating on data item of improper type.	Alter source code and recompile.
167	E	COMPLEX INDEX EXPRESSION	Compiler found index expression of type complex.	Alter source code and recompile.
168	E	COMPLEX SUBSCRIPT	Compiler found subscript value of type complex.	Alter source code and recompile.
169	E	RECURSIVE STATEMENT FUNCTION	Compiler found a recursively defined statement function and deleted the statement.	Alter source code and recompile.
170	E	SUBROUTINE MISSING ARGUMENTS	Compiler found subroutine call with with improper number of arguments.	Alter source code and recompile.
171	E	FUNCTION MISSING ARGUMENTS	Compiler found function definition with no arguments.	Alter source code and recompile.
172	E	REDEFINITION OF USED INTRINSIC	Compiler found intrinsic that had been redefined to a simple variable after being called; statement deleted.	Alter source code and recompile.
173	E	MISSING SUBSCRIPT	Compiler found missing subscript for an array name and deleted statement.	Alter source code and recompile.

Table 3-3. FORTRAN Compiler Warning & Error Messages (Continued)

NO.	TYPE	MESSAGE	MEANING	ACTION
174	E	ILLEGAL ARGUMENT FOR INTRINSIC	Compiler found illegal argument for intrinsic and deleted statement.	Alter source code and recompile.
176	E	TOO FEW ARGUMENTS	Compiler found procedure with too few arguments compared with previous usage.	Alter source code and recompile.
177	E	TOO MANY ARGUMENTS	Compiler found procedure with too many arguments compared with previous usage.	Alter source code and recompile.
178	E	VALUE VS. REFERENCE ARGUMENT	Argument passed by value when previous usage caused compiler to expect argument passed by reference.	Alter source code and recompile.
179	E	CHARACTER ARGUMENT BY VALUE	Compiler found character argument being passed by value.	Alter source code and recompile.
180	E	NO LIMIT PARAMETER	Compiler found missing limit parameter in DO statement or implied-DO in an I/O statement.	Alter source code and recompile.
181	E	TERMINAL LABEL PRECEDES DO STATEMENT	Compiler found DO-loop terminal label preceding DO-loop DO statement. DO statement skipped.	Alter source code and recompile.
182	E	IMPROPERLY NESTED DO STATEMENTS	Compiler found improperly nested (overlapping) DO statements. Current DO statement skipped or DO loop closed early.	Alter source code and recompile.
183	E	INTEGER SIMPLE VARIABLE EXPECTED	Compiler expected integer simple variable but did not find one.	Alter source code and recompile.
184	E	IMPROPER TERMINAL STATEMENT	Compiler found an improper termination statement in a DO loop.	Refer to manual for restrictions on DO loop termination statements. Alter source code and recompile.
185	E	UNDECLARED ARRAY NAME	Compiler found symbolic name used as array but not defined as such. Statement deleted.	Alter source code and recompile.
186	E	LEFT-HAND IS FUNCTION OR SUBROUTINE	Compiler found function or subroutine on left side of assignment operator and deleted statement.	Alter source code and recompile.
188	E	RIGHT AND LEFT-HAND TYPES INCOMPATIBLE	Compiler found incompatible types in left and right sides of assignment statement.	Alter source code and recompile.
189	E	DUMMY HAS TYPE CHARACTER	Compiler found dummy parameter to a statement function that is type character.	Alter source code and recompile.

Table 3-3. FORTRAN Compiler Warning & Error Messages (Continued)

NO.	TYPE	MESSAGE	MEANING	ACTION
191	E	CHARACTER STATEMENT FUNCTION	Compiler found character-type statement function and deleted statement.	Alter source code and recompile.
194	E	UNABLE TO CLASSIFY GOTO	Compiler was unable to classify a GO TO statement as one of the three allowable types and deleted statement.	Alter source code and recompile.
196	E	IMPROPER ASSIGN	Compiler found an improper ASSIGN statement and deleted statement.	Alter source code and recompile.
197	E	EXPECTED LOGICAL EXPRESSION	Compiler expected logical expression but none was found.	Alter source code and recompile.
198	E	IMPROPER LOGICAL CLAUSE	Compiler found improper logical clause and deleted statement.	Alter source code and recompile.
199	E	IMPROPER DEPENDENT STATEMENT	Compiler found improper dependent statement in IF statement.	Alter source code and recompile.
200	E	ALTERNATE RETURN IN NON-SUBROUTINE	Compiler found alternate RETURN statement in a function, main program, or block data subprogram.	Alter source code and recompile.
201	E	LABEL ARGUMENTS OUT OF POSITION	Compiler found label arguments appearing elsewhere than at the end of the subroutine argument list. Statement deleted.	Alter source code and recompile.
202	E	SYMBOL NOT SUBROUTINE NAME	Compiler encountered symbolic name where subroutine name expected. Statement deleted.	Alter source code and recompile.
203	E	EXPRESSION IN INPUT LIST	Compiler found expression in I/O input list.	Alter source code and recompile.
204	E	IMPROPER I/O LIST ITEM	Compiler found improper I/O list item in I/O statement and deleted statement.	Alter source code and recompile.
205	E	EXPECTED I/O LIST	Compiler expected I/O list and none was found. Statement deleted.	Alter source code and recompile.
206	E	IMPROPER UNIT REFERENCE	Compiler found improper unit reference in I/O statement. Statement deleted.	Alter source code and recompile.
207	E	EXPECTED CHARACTER VARIABLE	Compiler expected character variable for core-to-core I/O operation. Statement deleted.	Alter source code and recompile.
208	E	EXPECTED FORMAT REFERENCE	Compiler expected format reference in I/O statement and none was found. Statement deleted.	Alter source code and recompile.
209	E	EXPECTED ACTION LABEL	Compiler expected action label in I/O statement and none was found. Statement deleted.	Alter source code and recompile.

Table 3-3. FORTRAN Compiler Warning & Error Messages (Continued)

NO.	TYPE	MESSAGE	MEANING	ACTION
210	E	DUPLICATE ACTION LABEL	Compiler found duplicate action label.	Alter source code and recompile.
211	E	TOO MANY TRACE SYMBOLS	TRACE/3000 symbol table overflowed.	Alter source code and recompile.
212	E	DATA SPACE OVERFLOW	The program stack size required exceeds 32767 words.	Decrease array sizes, or variables used, and recompile.
213	E	CODE SPACE OVERFLOW	Program code space overflowed.	Alter source code and recompile.
214	E	OUT OF STACK FOR GLOBAL CROSS REFERENCE	Global cross reference processor cannot fit its tables in stack space available. No global cross reference is produced.	If system-wide MAX-DATA limit is less than 32767, have System Manager raise limit and re-compile.
215	E	EXPECTED AN INTEGER OR STRING	Argument to STOP or PAUSE statement could not be identified. Argument is ignored.	Alter source code and recompile.
216	E	TRAP TYPE KEYWORD NOT RECOGNIZED	String of characters following "ON" in trap statement did not match any of trap types allowed. Statement deleted.	Alter source code and recompile.
217	E	ENTRY STRUCTURE MIS-MATCH	ENTRY statement name used as function in subroutine or vice versa. Rest of statement ignored.	Alter source code and recompile.
218	E	REDUNDANT ENTRY	Previous ENTRY statement has already appeared with name identifier. Rest of statement ignored.	Alter source code and recompile.
219	E	DUMMY USED AS ENTRY	Formal parameter appears as entry name in ENTRY statement. Rest of statement is ignored.	Alter source code and recompile.
220	E	STATEMENT FUNCTION OR INTRINSIC NAME	Name used as entry is statement function or intrinsic. Rest of statement ignored.	Alter source code and recompile.
221	E	ENTRY IN RANGE OF DO	ENTRY statement appears in range of a DO loop. Rest of statement is analyzed by the compiler.	Alter source code and recompile.
222	E	NAME NOT UNIQUE	Name appearing in SYSTEM INTRINSIC statement has been specified with some other structure. Rest of statement is ignored.	Alter source code and recompile.
223	E	NAME NOT ON FILE	Name appearing in SYSTEM INTRINSIC statement cannot be found in the SPLINTR file. Rest of statement is ignored.	Alter source code and recompile.
224	E	INCORRECT DIGIT	INTEGER*2 or INTEGER*4 declaration has a number other than 2 or 4. INTEGER*2 is assumed.	Alter source code and recompile

Table 3-3. FORTRAN Compiler Warning & Error Messages (Continued)

NO.	TYPE	MESSAGE	MEANING	ACTION
225	E	EXPECTED DIGIT	INTEGER*2 or INTEGER*4 declaration has a non-numeric character following the *. INTEGER*2 is assumed.	Alter source code and recompile.
226	W	REFERENCED VARIABLE NOT DEFINED	Variable listed with the message is uninitialized. Compilation not affected.	Alter source code and recompile.
227	E	PARAMETER IDENTIFIER NOT UNIQUE	Name given to a parameter constant was not unique. Compiler attempted to analyze the rest of the statement.	Alter source code and recompile.
228	E	EXPECTED PARAMETER VALUE	Whatever was found following the equal signs in a parameter definition was not recognized as a valid constant. Compiler attempted to analyze the rest of the statement.	Alter source code and recompile.
229	E	ENTRY IN MAIN PROGRAM	ENTRY statement may only appear in subprogram. The rest of statement is ignored.	Alter source code and recompile.
230	E	ITEM IN DATA REDUNDANTLY INITIALIZED	The same variable is initialized twice through a DATA statement.	Alter source code and recompile.
231	W	COMPILER BRANCH GENERATION ERROR	The compiler has generated bad branch code for a very large or very complex conditional expression. Incorrect operation may result.	Try to break conditional statement into several less complex statements.
232	E	ATTEMPT TO STORE INTO PARAMETER	A symbolic name assigned a value in a PARAMETER statement has been assigned a value otherwise; it may only have value assigned in PARAMETER statement.	Alter source code and recompile.
233	E	COMPILER ERROR	Compiler detected an irrecoverable internal error.	Document error and submit with bug report.
234	W	LOCATION DISALLOWED FOR INTERACTIVE TEXT AND LIST FILES	When in interactive mode, the compiler cannot list code locations	Remove the LOCATION option from the $CONTROL compiler subsystem commands
235	W	COMMON VARIABLE MAY NOT BE TRACED WITH MORECOM	Cannot trace a common variable when the MORECOM option of the $CONTROL compiler subsystem command is set; the results are invalid	Remove any common variables from the TRACE statement

Table 3-3. FORTRAN Compiler Warning & Error Messages (Continued)

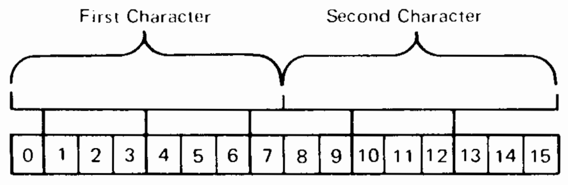
NO.	TYPE	MESSAGE	MEANING	ACTION
236	W	MORE THAN ONE OUTER BLOCK ACTIVE	The compiler encountered an END statement followed by executable statements and began to compile another main program unit	Take out the extra END statement

ASCII CHARACTER SET

ASCII Character	First Character Octal Equivalent	Second Character Octal Equivalent
A	040400	000101
B	041000	000102
C	041400	000103
D	042000	000104
E	042400	000105
F	043000	000106
G	043400	000107
H	044000	000110
I	044400	000111
J	045000	000112
K	045400	000113
L	046000	000114
M	046400	000115
N	047000	000116
O	047400	000117
P	050000	000120
Q	050400	000121
R	051000	000122
S	051400	000123
T	052000	000124
U	052400	000125
V	053000	000126
W	053400	000127
X	054000	000130
Y	054400	000131
Z	055000	000132
a	060400	000141
b	061000	000142
c	061400	000143
d	062000	000144
e	062400	000145
f	063000	000146
g	063400	000147
h	064000	000150
i	064400	000151
j	065000	000152
k	065400	000153
l	066000	000154
m	066400	000155
n	067000	000156
o	067400	000157
p	070000	000160
q	070400	000161
r	071000	000162
s	071400	000163
t	072000	000164
u	072400	000165
v	073000	000166
w	073400	000167
x	074000	000170
y	074400	000171
z	075000	000172
0	030000	000060
1	030400	000061
2	031000	000062
3	031400	000063
4	032000	000064
5	032400	000065
6	033000	000066
7	033400	000067
8	034000	000070
9	034400	000071
NUL	000000	000000
SOH	000400	000001
STX	001000	000002
ETX	001400	000003
EOT	002000	000004
ENQ	002400	000005

ASCII Character	First Character Octal Equivalent	Second Character Octal Equivalent
ACK	003000	000006
BEL	003400	000007
BS	004000	000010
HT	004400	000011
LF	005000	000012
VT	005400	000013
FF	006000	000014
CR	006400	000015
SO	007000	000016
SI	007400	000017
DLE	010000	000020
DC1	010400	000021
DC2	011000	000022
DC3	011400	000023
DC4	012000	000024
NAK	012400	000025
SYN	013000	000026
ETB	013400	000027
CAN	014000	000030
EM	014400	000031
SUB	015000	000032
ESC	015400	000033
FS	016000	000034
GS	016400	000035
RS	017000	000036
US	017400	000037
SPACE	020000	000040
!	020400	000041
"	021000	000042
#	021400	000043
$	022000	000044
%	022400	000045
&	023000	000046
'	023400	000047
(	024000	000050
)	024400	000051
*	025000	000052
+	025400	000053
,	026000	000054
-	026400	000055
.	027000	000056
/	027400	000057
:	035000	000072
;	035400	000073
<	036000	000074
=	036400	000075
>	037000	000076
?	037400	000077
@	040000	000100
[	055400	000133
\	056000	000134
]	056400	000135
Δ	057000	000136
—	057400	000137
`	060000	000140
{	075400	000173
|	076000	000174
}	076400	000175
~	077000	000176
DEL	077400	000177

← = <ESC>D



PROGRAM OPTIMIZATION

APPENDIX F

Optimizing the use of the FORTRAN compiler involves minimizing memory usage during compilation and execution of a program, and minimizing the total run-time. An understanding of the way the compiler uses memory helps you to make effective use of the system. Run-time efficiency can be improved by avoiding unnecessary calls to the COMPILER LIBRARY. The following recommendations may aid you in making the data area of a large program smaller, or help you in decreasing the CPU usuage of a program. These recommendations should not be necessary for the average FORTRAN program under normal use.

F-1. COMPILATION PHASE

The FORTRAN compiler uses a portion of the DATA stack as a working storage area, called the compiler symbol table. Control over memory usage during compilation can be accomplished by keeping the symbol table small. The compiler uses 13 bits to link entries within the symbol table. This limits the size of the symbol table to 8191 words. The compiler initializes (or reuses) this area in memory for each program unit (main program, function subprogram or subroutine).

Some of the uses of the symbol table are the following:

1. Each subroutine, system intrinsic, or function subprogram, whether in a program unit or in an external statement, requires a table entry (the length of each entry is 4 words + $\frac{\text{number of characters in variable name}}{2}$).
2. All simple variables and arrays require an entry, whether in Local Storage or in COMMON.
3. The table is also used for control entries (DO loops and STATEMENT labels).
4. It is used for compiler options such as CROSSREF, LOCATION, and MAP.

If a particular program unit is very large, and therefore uses a lot of symbol table storage, the compiler issues the message: SYMBOL TABLE OVERFLOW (compiler error #51).

Table overflow can be avoided if the program is laid out in a modular fashion, that is, using several subroutines and functions. Programs designed using standard structured programming techniques should not encounter symbol table overflow situations. However, if you already have a large program unit written and you get the symbol table overflow error message, the symbol table requirements can be reduced by applying the following techniques:

1. Minimize the length of the variable and array names. Less memory space is needed for storage if these symbolic names are short.
2. Combine several variables into an array since only one symbol table entry is required for an array, whereas separate variables require one entry each.
3. Avoid compiler command options such as CROSSREF, LOCATION, MAP, etc., as they require more storage space. For example, CROSSREF uses one extra word in each symbol table entry for linking purposes.
4. Divide larage program units into smaller units. It is easier to maintain smaller program units than large program units. Smaller program units improve the supportability and readability of the program.

Note: In many cases, the readability and the supportability of a particular program can be adversely affected by the use of short variable names. If a variable name is too short, it is difficult to know its meaning and distinguish it from other variables. For example, a single letter G does not properly describe and identify a variable which may otherwise be represented by the symbolic name GRADE. It is not advisable to combine variables to form a single array without giving due consideration to their individual characteristics. For example, if variable names such as SALARY, DEDUCTIONS, GROSS, NETINCOME are changed into names such as AIARR (1), AIARR (2), AIARR (3), AIARR (4), elements of the array AIARR, it is difficult to identify the variables which are represented by these names.

F-2. EXECUTION PHASE

Run-time optimization involves the minimizing of memory usage and total run-time. You can employ some or all of the following techniques to achieve these goals:

MINIMIZING MEMORY USAGE

1. Avoid using COMMON storage for those variables that are used only within a program unit. COMMON variables use a portion of the data stack throughout the program execution, whereas local storage is active only when the program unit (main program, function or subroutine) is active.

2. Save stack space by changing the order of COMMON variables. Each COMMON block element normally has one pointer established by the segmenter. The maximum number of pointers allowed is 254, unless the MORECOM option is used. A rearrangement of simple variables and arrays helps in reducing memory usage, provided the variables are of the same type. For example, a program with the following variables would have a stack assignment as shown in Fig. F-1:

```
COMMON C(3),A,B
INTEGER A,B,C,D,E
```

   However, if the order of the COMMON variables is changed to A,B,C(3) (the simple variables preceding the array variable), only two pointers are required (Fig. F-2) instead of three. Thus, it is possible to have more than 254 COMMON variables without having to use the less efficient MORECOM option. The fact that there are two variables, C(0) and B, having the same DB location and the same pointer may cause confusion when looking at the symbol map. Since the zero'th element of an array does not exist in FORTRAN, the two variables can use the same pointer, and save memory.

MINIMIZING EXECUTION TIME

1. Avoid mixed-mode expressions. For example, the assignment statement X=1+X, where X is real, requires conversion of the integer one to the real number one during execution, and the compiler must generate extra code for this conversion. A more efficient assignment statement would be X=1.+X.

2. Avoid using FORMAT statements whenever possible, as they cause more external calls to the FORMATTER. FORMAT statements also require storage in the code segment of a program. Figs. F-3 and F-4 demonstrate the difference in run-time when a FORMAT statement is not used, compared to when it is used. If a FORMAT statement is used (Fig. F-4), the program takes about twice as long to run as when the FORMAT statement is not used (Fig. F-3).

3. Avoid unnecessary external procedure calls (PCAL's). For example, the statement X=Y**2 explicitly requires a PCAL to an exponential function procedure. Rewriting this statement as X=Y*Y avoids the PCAL.

   Certain constructs can implicitly cause PCAL's. In Fig. F-5, the assignment X=Y, where X and Y are character variables, generates a call to BLANKFILL, which fills the remainder of X with blanks, starting with the third character from the left. However, if the assignment statement is changed to X [1:2] =Y (Fig. F-6), only the first two leftmost positions are assigned the value of Y and the rest of X remains undefined. This does not generate a PCAL to BLANKFIL and hence saves run-time.

4. Use SPL for certain specific tasks if it can save run-time. For example, an SPL MOVE operation can be used instead of a FORTRAN DO loop initialization. When the number of words to be moved exceeds 50, the FORTRAN routine requires more CPU time, compared to the time used by SPL.

Note: The programs shown in Figs. F-3 through F-6 were run on the same HP 3000 computer system operating under identical conditions. The run-time shown will vary among different machines and different operating conditions. They are included here for relative comparison purposes only.

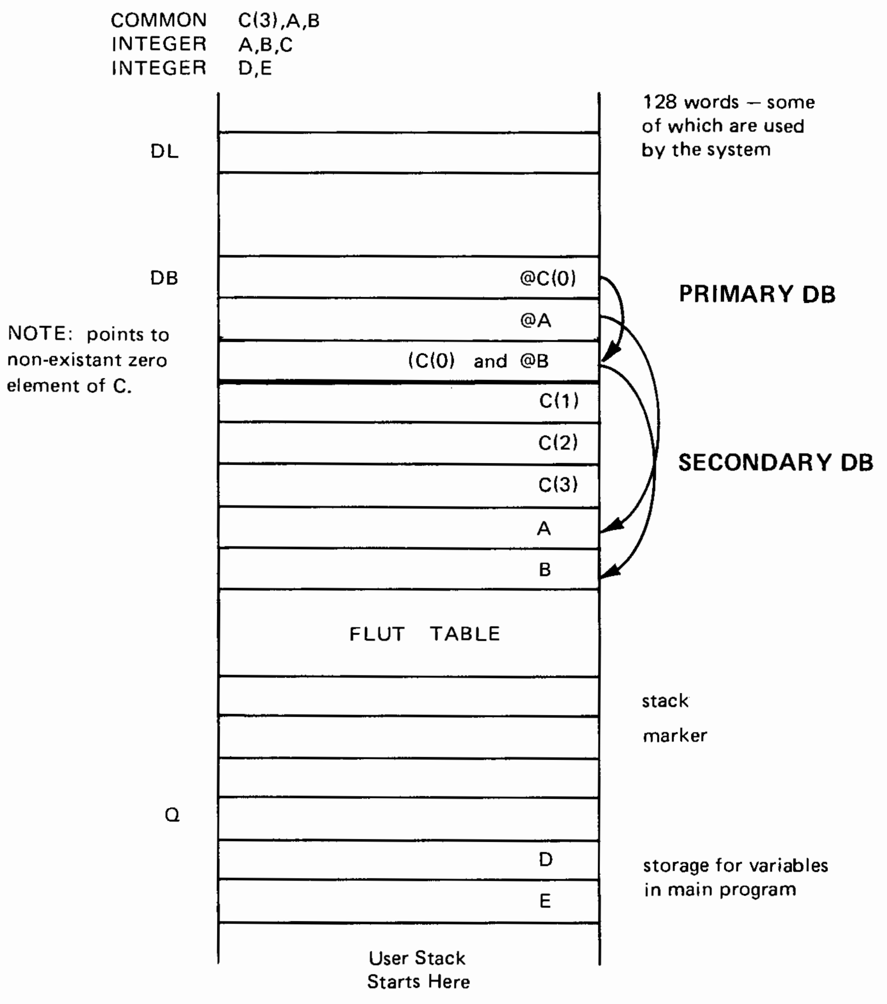


Figure F-1. COMMON Variables Stack Assignment in Non-Optimized Order

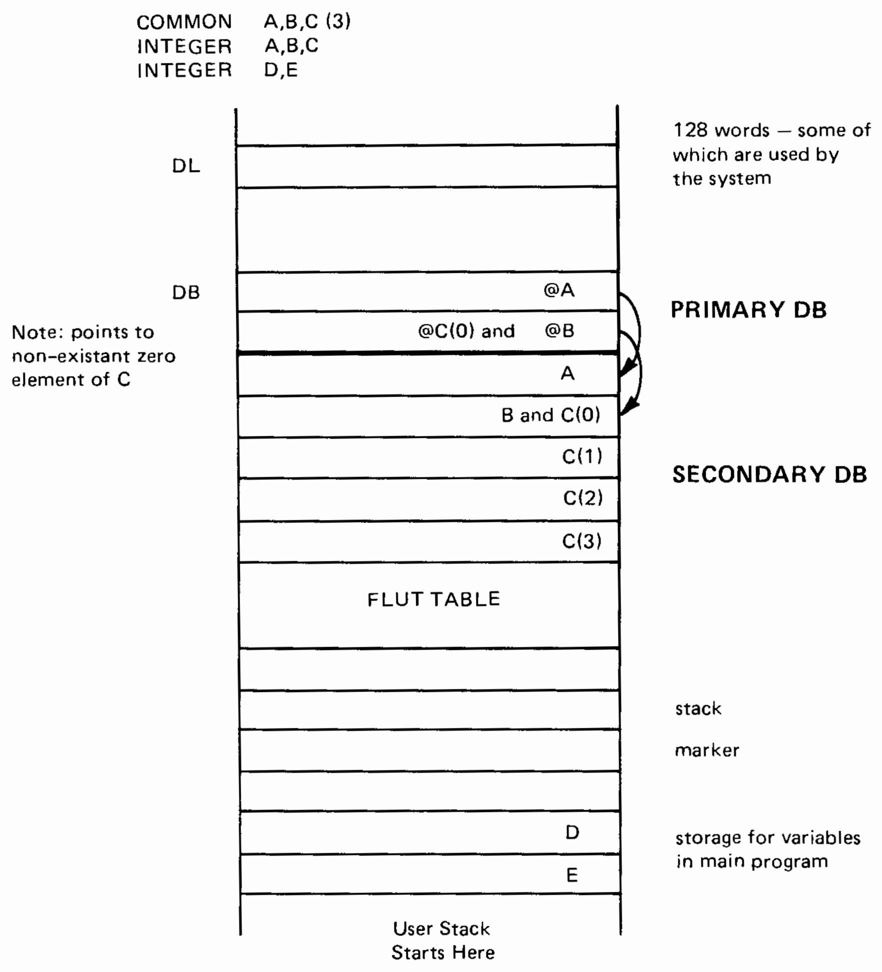
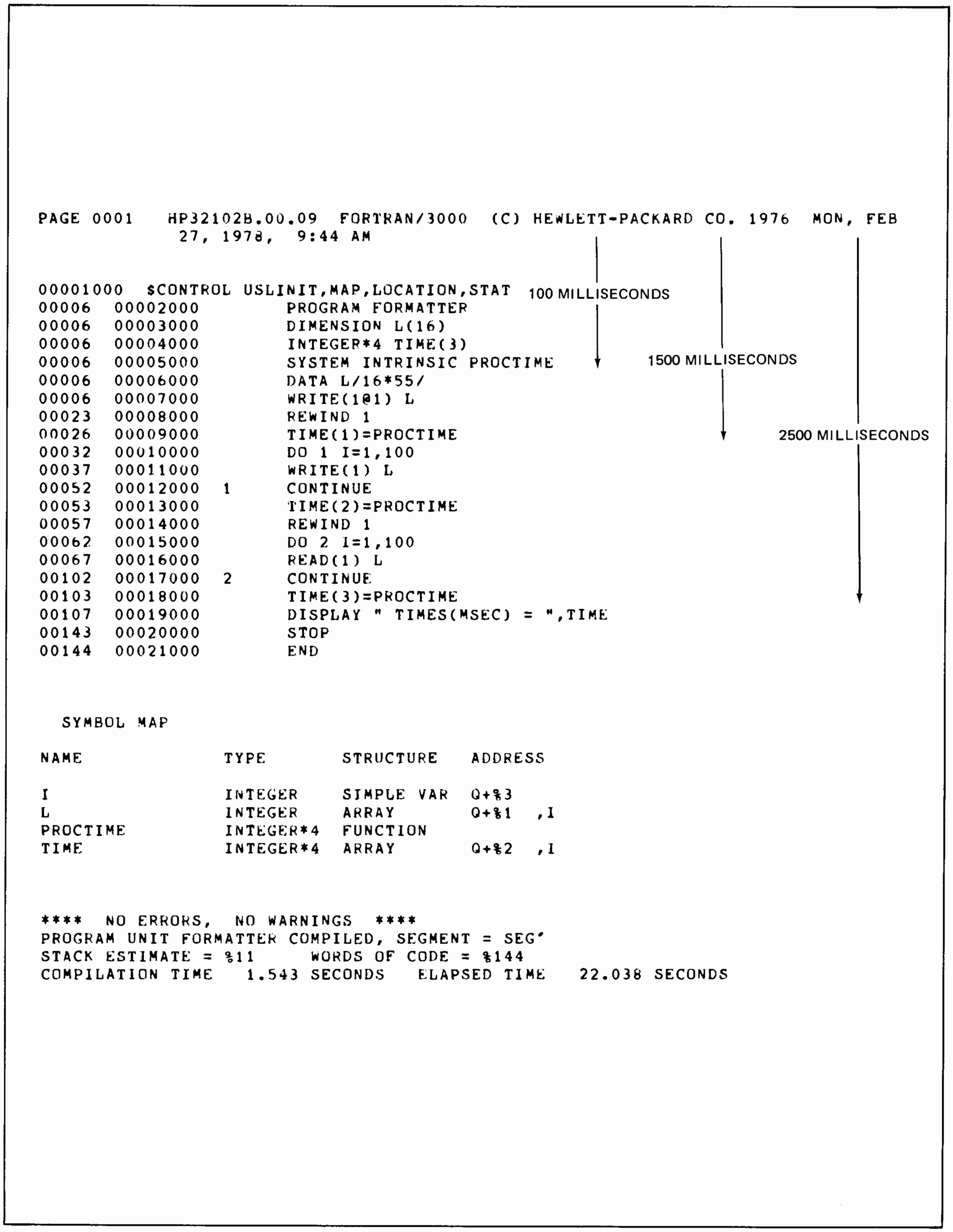
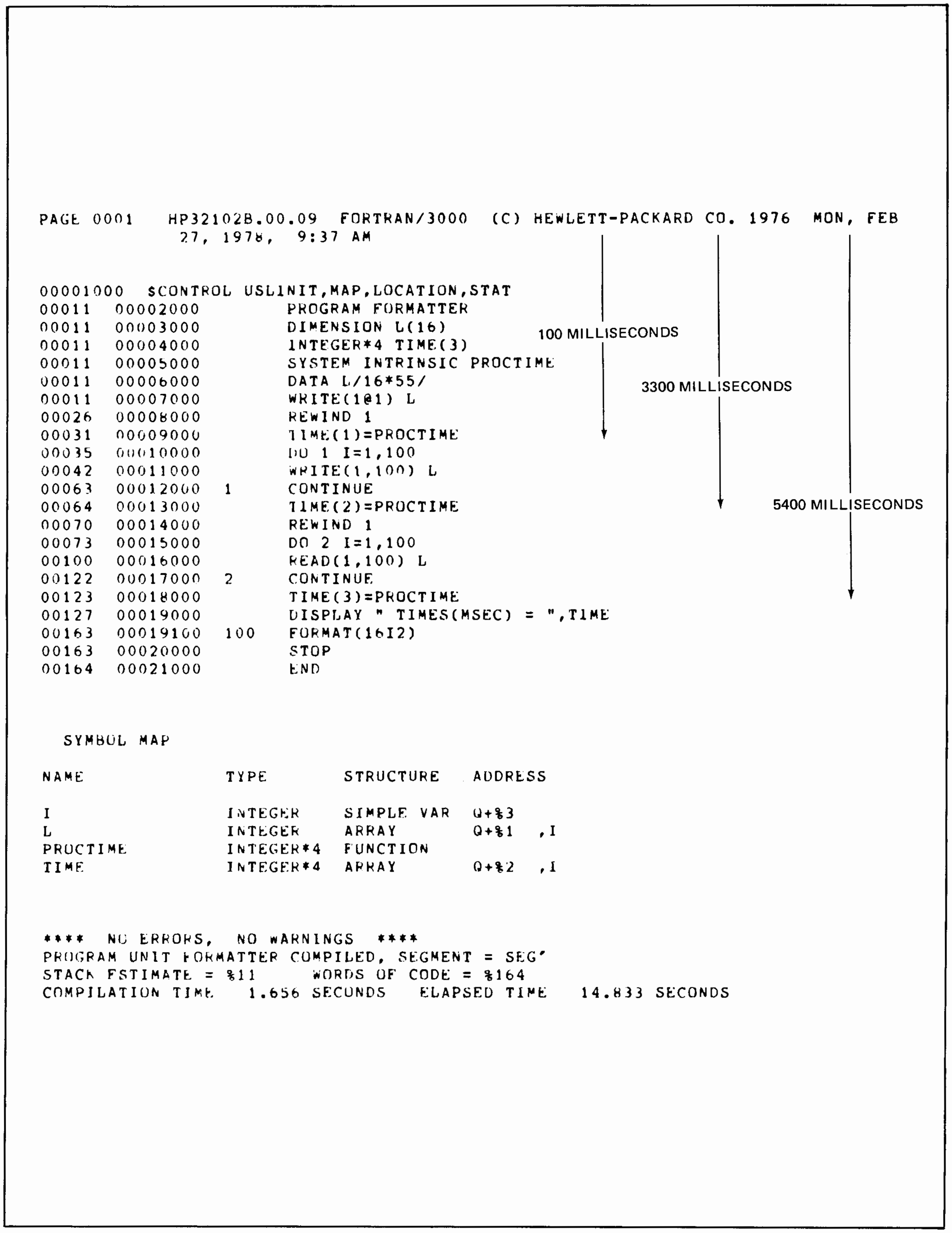


Figure F-2. COMMON Variables Stack Assignment in Optimized Order


100 MILLISECONDS
1500 MILLISECONDS
2500 MILLISECONDS

```
PAGE 0001    HP32102B.00.09  FORTRAN/3000  (C) HEWLETT-PACKARD CO. 1976  MON, FEB
              27, 1978,  9:44 AM


00001000  $CONTROL USLINIT,MAP,LOCATION,STAT
00006   00002000          PROGRAM FORMATTER
00006   00003000          DIMENSION L(16)
00006   00004000          INTEGER*4 TIME(3)
00006   00005000          SYSTEM INTRINSIC PROCTIME
00006   00006000          DATA L/16*55/
00006   00007000          WRITE(1@1) L
00023   00008000          REWIND 1
00026   00009000          TIME(1)=PROCTIME
00032   00010000          DO 1 I=1,100
00037   00011000          WRITE(1) L
00052   00012000  1       CONTINUE
00053   00013000          TIME(2)=PROCTIME
00057   00014000          REWIND 1
00062   00015000          DO 2 I=1,100
00067   00016000          READ(1) L
00102   00017000  2       CONTINUE
00103   00018000          TIME(3)=PROCTIME
00107   00019000          DISPLAY " TIMES(MSEC) = ",TIME
00143   00020000          STOP
00144   00021000          END


  SYMBOL MAP

NAME              TYPE        STRUCTURE   ADDRESS

I                 INTEGER     SIMPLE VAR  Q+%3
L                 INTEGER     ARRAY       Q+%1   ,I
PROCTIME          INTEGER*4   FUNCTION
TIME              INTEGER*4   ARRAY       Q+%2   ,I


****  NO ERRORS,  NO WARNINGS  ****
PROGRAM UNIT FORMATTER COMPILED, SEGMENT = SEG'
STACK ESTIMATE = %11      WORDS OF CODE = %144
COMPILATION TIME   1.543 SECONDS   ELAPSED TIME   22.038 SECONDS
```

Figure F-3. Less Execution Time — No Format Statement

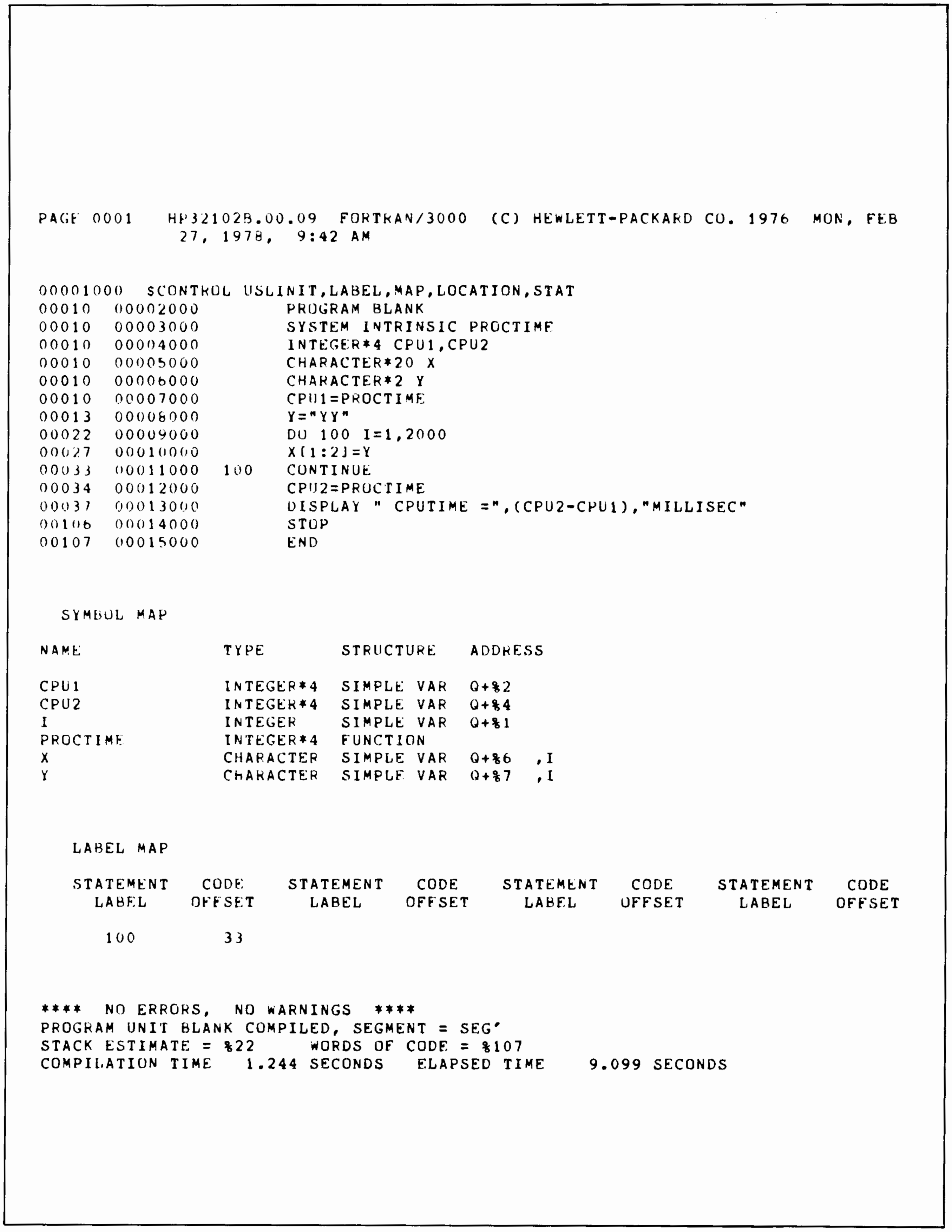


```
PAGE 0001    HP32102B.00.09  FORTRAN/3000  (C) HEWLETT-PACKARD CO. 1976  MON, FEB
              27, 1978,   9:37 AM


00001000  $CONTROL USLINIT,MAP,LOCATION,STAT
00011  00002000         PROGRAM FORMATTER
00011  00003000         DIMENSION L(16)
00011  00004000         INTEGER*4 TIME(3)
00011  00005000         SYSTEM INTRINSIC PROCTIME
00011  00006000         DATA L/16*55/
00011  00007000         WRITE(1@1) L
00026  00008000         REWIND 1
00031  00009000         TIME(1)=PROCTIME
00035  00010000         DO 1 I=1,100
00042  00011000         WRITE(1,100) L
00063  00012000   1     CONTINUE
00064  00013000         TIME(2)=PROCTIME
00070  00014000         REWIND 1
00073  00015000         DO 2 I=1,100
00100  00016000         READ(1,100) L
00122  00017000   2     CONTINUE
00123  00018000         TIME(3)=PROCTIME
00127  00019000         DISPLAY " TIMES(MSEC) = ",TIME
00163  00019100   100   FORMAT(16I2)
00163  00020000         STOP
00164  00021000         END


  SYMBOL MAP

NAME              TYPE         STRUCTURE    ADDRESS

I                 INTEGER      SIMPLE VAR   Q+%3
L                 INTEGER      ARRAY        Q+%1   ,I
PROCTIME          INTEGER*4    FUNCTION
TIME              INTEGER*4    ARRAY        Q+%2   ,I



****  NO ERRORS,  NO WARNINGS  ****
PROGRAM UNIT FORMATTER COMPILED, SEGMENT = SEG'
STACK ESTIMATE = %11      WORDS OF CODE = %164
COMPILATION TIME   1.656 SECONDS   ELAPSED TIME   14.833 SECONDS
```

Figure F-4. FORMAT Statement (100) Causes External Calls

```
PAGE 0001   HP32102B.00.09  FORTRAN/3000  (C) HEWLETT-PACKARD CO. 1976  MON, FEB
             27, 1978,  9:19 AM


00001000  $CONTROL USLINIT,LABEL,MAP,LOCATION,STAT
00010  00002000            PROGRAM BLANK
00010  00003000            SYSTEM INTRINSIC PROCTIME
00010  00004000            INTEGER*4 CPU1,CPU2
00010  00005000            CHARACTER*20 X
00010  00006000            CHARACTER*2 Y
00010  00007000            CPU1=PROCTIME
00013  00008000            Y="YY"
00022  00009000            DO 100 I=1,2000
00027  00010000            X=Y
00035  00011000   100      CONTINUE
00036  00012000            CPU2=PROCTIME
00041  00013000            DISPLAY " CPUTIME =",(CPU2-CPU1),"MILLISEC"
00110  00014000            STOP
00111  00015000            END



  SYMBOL MAP

NAME              TYPE        STRUCTURE   ADDRESS

CPU1              INTEGER*4   SIMPLE VAR  Q+%2
CPU2              INTEGER*4   SIMPLE VAR  Q+%4
I                 INTEGER     SIMPLE VAR  Q+%1
PROCTIME          INTEGER*4   FUNCTION
X                 CHARACTER   SIMPLE VAR  Q+%6   ,I
Y                 CHARACTER   SIMPLE VAR  Q+%7   ,I


   LABEL MAP

   STATEMENT   CODE    STATEMENT   CODE    STATEMENT   CODE    STATEMENT   CODE
     LABEL    OFFSET     LABEL    OFFSET     LABEL    OFFSET     LABEL    OFFSET

      100        35


****  NO ERRORS,  NO WARNINGS  ****
PROGRAM UNIT BLANK COMPILED, SEGMENT = SEG'
STACK ESTIMATE = %22     WORDS OF CODE = %111
COMPILATION TIME   1.330 SECONDS   ELAPSED TIME   26.853 SECONDS
```

Figure F-5. X=Y Generates an Implicit Call to BLANKFIL



```
PAGE 0001   HP32102B.00.09  FORTRAN/3000  (C) HEWLETT-PACKARD CO. 1976  MON, FEB
             27, 1978,  9:42 AM


00001000  $CONTROL USLINIT,LABEL,MAP,LOCATION,STAT
00010  00002000         PROGRAM BLANK
00010  00003000         SYSTEM INTRINSIC PROCTIME
00010  00004000         INTEGER*4 CPU1,CPU2
00010  00005000         CHARACTER*20 X
00010  00006000         CHARACTER*2 Y
00010  00007000         CPU1=PROCTIME
00013  00008000         Y="YY"
00022  00009000         DO 100 I=1,2000
00027  00010000         X[1:2]=Y
00033  00011000  100    CONTINUE
00034  00012000         CPU2=PROCTIME
00037  00013000         DISPLAY " CPUTIME =",(CPU2-CPU1),"MILLISEC"
00106  00014000         STOP
00107  00015000         END


  SYMBOL MAP

NAME                TYPE        STRUCTURE   ADDRESS

CPU1                INTEGER*4   SIMPLE VAR  Q+%2
CPU2                INTEGER*4   SIMPLE VAR  Q+%4
I                   INTEGER     SIMPLE VAR  Q+%1
PROCTIME            INTEGER*4   FUNCTION
X                   CHARACTER   SIMPLE VAR  Q+%6   ,I
Y                   CHARACTER   SIMPLE VAR  Q+%7   ,I


   LABEL MAP

   STATEMENT   CODE     STATEMENT   CODE     STATEMENT   CODE     STATEMENT   CODE
     LABEL    OFFSET      LABEL    OFFSET      LABEL    OFFSET      LABEL    OFFSET

      100        33


****  NO ERRORS,  NO WARNINGS  ****
PROGRAM UNIT BLANK COMPILED, SEGMENT = SEG'
STACK ESTIMATE = %22      WORDS OF CODE = %107
COMPILATION TIME   1.244 SECONDS   ELAPSED TIME    9.099 SECONDS
```

Figure F-6. Less Run-Time With the Changed Assignment Statement X[1:2]=Y

INDEX

Part No. 30000-90040
Printed in U.S.A. 6/76
Update #3 Incorporated 5/79
3FORTN.320.30000-90040

HEWLETT
PACKARD