CHAPTER 5

THE PAL8 ASSEMBLER

5.1 INTRODUCTION

PAL8, the OS/78 Operating System assembler, generates binary object files from source (ASCII) programs written in the PAL8 assembly language.

PAL8 is a two-pass assembler. During pass 1, the source program is read and an internal symbol table is produced that contains the PAL8 permanent symbols and any new symbols that you define. During pass 2, the assembler reads the source file again, generates the binary code using the symbol table definitions created during pass 1, and continues defining symbols as well. The binary file that is output may be loaded into memory as the "current" executable program by the LOAD command. Absolute binary format consists of 8-bit bytes, containing field setting commands, address setting commands, and sequential data words. An optional third pass will produce a program listing if one is desired. During pass 3, the assembler reads the source file a final time and generates the assembly listing as an ASCII (character string) file. The assembly listing consists of the source statement together with its current location counter and the generated code in octal. The first 40 (decimal) characters of the first line of each page of the listing contain a title, the assembler version number, the date and the listing page number.

Use the OS/78 command PAL to call the assembler. You can also use the commands CREF and EXECUTE as explained in this chapter.

The PAL command specifies the binary and listing output devices and file names, the input devices and file names, and any options that you select. From one to nine input files may be specified. The typical way to assemble, load, and then run a program called PROG is as follows:

```
·PAL PROG          -Assemble the program
·LOAD              -Load the program into memory
·SAVE SYS PROG     -Save the program
·R PROG            -Run the program
```

The long form of the command string is

```
PAL dev:binary,dev:listing,dev:crefls<dev:input,.../options
```

If the extension to the file name is omitted, the following extensions are assumed.

```
.PA for input files.
.BN for binary output file.
.LS for listing output file.
.TM for intermediate CREF file.
```

If an assembly or CREF listing is not desired, omit the listing file or CREF file, respectively.

For example, to assemble, load, and run a PAL8 program named SAMPLE, which is stored on diskette unit 1, type

```
.PAL RXA1:SAMPLE/G-T
```

After assembly the program is loaded and run (since the /G was specified) with the starting address assumed to be location 0200 in field 0; the binary file is stored on the DSK: device as SAMPLE.BN. The -T General-Purpose Dash Option displays the assembled program listing on the terminal (see Table 2-3).

If a binary file is not desired, specify the -NB option at the end of the command line (NB stands for No Binary). For example, to get a listing only, type

```
.PAL SAMPLE-LS-NB
```

The -LS option indicates that a listing should be produced.

The assembler displays any error messages encountered in the program on the terminal, even when a listing is not produced. Typing CTRL/O at the keyboard during an assembly suppresses the display of error messages. However, messages are still printed in the listing file (if any) and occur immediately before the line that is in error.

For example, the command line

```
.PAL SAMPLE/S-LS
```

causes PAL8 to assemble SAMPLE.PA (or SAMPLE), generating DSK:SAMPLE.BN and putting the listing into the file SAMPLE.LS on the default device DSK. The /S option suppresses listing of the symbol table.

The command line

```
.PAL BIN<SAMPLE.PA/G=600
```

assembles SAMPLE.PA, creates a binary output file named BIN.BN, loads the file BIN.BN, and starts it at location 600. The construction =600 is an option that specifies the starting address.

Assembly can be terminated at any time by typing CTRL/C on the keyboard, and any output files being stored will be deleted. Otherwise, PAL8 always returns to the monitor upon completion of assembly.

A source program may consist of a number of source file modules to be assembled together. You do this by specifying a string of input device and file names separated by commas. For example,

```
.PAL PART1,RXA1:PART2,RL0A:PART3
```

assembles a three-part program. This technique is useful when it is desired to assemble two programs that are identical except for a few lines at the beginning of the programs. Different lines can be broken

out into a "prefix file". For example, two different file assemblies may be generated by

```
.PAL PRFX1,FILE
```

and

```
.PAL PRFX2,FILE
```

You can enter up to nine input files to be treated as one source input in a command line.

If more than one input file is specified, and output files are desired but not explicitly specified, the name of the first input file is used for the output file names. For example,

```
.PAL A,B
```

produces the binary file A.BN.

If a file name other than the first input file is desired for the binary name, use the -NB General-Purpose Dash Option after the last input file name not desired as the binary file name. For example,

```
.PAL A-NB,B
```

produces DSK:B.BN and

```
.PAL A,B,C-NB,D,E,F
```

produces DSK:D.BN.

If a -LS option is specified, it must appear immediately after an input file name. This is the name that will be used for the name of the listing file. For example,

```
.PAL A,B-LS
```

produces DSK:B.LS while

```
.PAL A-LS,B
```

produces DSK:A.LS

The -L or -T General-Purpose Dash Options used with a PAL or COMPILE command send the listing output file to the line printer and terminal respectively.

Note that the PAL command normally produces a binary file even when a name is not given. Thus, typing

```
.PAL ,LPT:<file
```

produces a binary file.

If you do not specify an extension, PAL assumes that the input file extension is .PA. Thus, the command

```
.PAL TEST
```

causes the assembler to search for a file namel DSK:TEST.PA. If no file with .PA is found on DSK:, the assembler then searches for a file named TEST with no extension. It is good practice when creating a PAL8 source file to include a .PA extension to remind you what type of source file it is.

The COMPILE and EXECUTE commands may also be used to invoke PAL8. These commands search the directory of the specified device for the file given with the command, and if one is found with a .PA extension, PAL8 is invoked. For example,

```
*COMPILE TEST
```

will run PAL8 if TEST.PA is found. An unusual extension may be explicitly specified by typing

```
*PAL TEST.XX
```

which will assemble DSK:TEST.XX. To specify PAL8 as the processor in the COMPILE command, use the -PA General-Purpose Dash Option in the command line as follows:

```
*COMPILE TEST.XX-PA
```

The EXECUTE command is similar to the COMPILE command except that the EXECUTE command is supported by the /G option.

If an argument is not given with a PAL or COMPILE or EXECUTE command, the argument used with the last such command is assumed when that command is used again.

5.2 CREATING AND RUNNING A PAL8 PROGRAM

The following steps demonstrate the procedure for creating and running a PAL8 program.

5.2.1 Creating a Program

Create the assembly language source file by calling the Editor as follows:

```
*CREATE SAMPLE.PA
```

Since a new program is being created, only a single file name need be specified. The OS/78 Editor will then display a number sign (#) to indicate it is ready to accept a command. (See Chapter 4 for a detailed discussion of the OS/78 Editor.)

Type the A (Append) command to allow the Editor to accept text. Then type in the program, one line at a time. Press the RETURN key after each line.

```
#A

/ROUTINE TO TYPE A MESSAGE
        *200
        MONADR=7600
START,  CLA CLL          /CLEAR ACCUMULATOR AND LINK
        TLS              /CLEAR TERMINAL FLAG
        TAD BUFADR       /SET UP POINTER
        DCA PNTR         /FOR GETTING CHARACTERS
```

```
NEXT,   TSF             /SKIP IF TERMINAL FLAG SET
        JMP .-1         /NO: CHECK AGAIN
        TAD I PNTR      /GET A CHARACTER
        TLS             /PRINT A CHARACTER
        ISZ PNTR        /DONE YET?
        CLA CLL         /CLEAR ACCUMULATOR AND LINK
        TAD I PNTR      /GET ANOTHER CHARACTER
        SZA CLA         /JUMP ON ZERO AND CLEAR
        JMP NEXT        /GET READY TO PRINT ANOTHER
        JMP I MON       /RETURN TO MONITOR
BUFADR, BUFF            /BUFFER ADDRESS
PNTR,   BUFF            /POINTER
BUFF,   215;212;"H;"E;"L;"L;"O;"!;0
MON,    MONADR          /MONITOR ENTRY POINT
```

Now type a CTRL/L to terminate input. This command returns you to the Editor command mode.

Type the L (List) command in response to the Editor's number sign (#) to list the text that was inserted into the text buffer.

When you are satisfied that the input is correct, type the E (Exit) command to store the file and return to the monitor.

5.2.2 Assembling a Program

Now assemble the source program just created. Use the command:

```
.PAL SAMPLE-LS
```

This command creates two files, a binary file called SAMPLE.BN, and a listing file (-LS option) called SAMPLE.LS. Use the TYPE command to display the listing on the terminal or the LIST command to print the listing on a line printer.

The assembly listing produced by PAL appears as follows:

```
/ROUTINE TO TYPE A MESSAGE          PAL8-V13A  14-MARCH-79 PAGE 1

                    /ROUTINE TO TYPE A MESSAGE
          0200      *200
          7600      MONADR=7600
000200    7300 START, CLA CLL         /CLEAR ACCUMULATOR AND LINK
000201    6046        TLS             /CLEAR TERMINAL FLAG
000202    1216        TAD BUFADR      /SET UP POINTER
000203    3217        DCA PNTR        /FOR GETTING CHARACTERS
000204    6041 NEXT,  TSF             /SKIP IF TERMINAL FLAG SET
000205    5204        JMP .-1         /NO: CHECK AGAIN
000206    1617        TAD I PNTR      /GET A CHARACTER
000207    6046        TLS             /PRINT A CHARACTER
000210    2217        ISZ PNTR        /DONE YET?
000211    7300        CLA CLL         /CLEAR ACCUMULATOR AND LINK
000212    1617        TAD I PNTR      /GET ANOTHER CHARACTER
000213    7640        SZA CLA         /JUMP ON ZERO AND CLEAR
000214    5204        JMP NEXT        /GET READY TO PRINT ANOTHER
000215    5631        JMP I MON       /RETURN TO MONITOR
000216    0220 BUFADR, BUFF    /BUFFER ADDRESS
000217    0220 PNTR,   BUFF    /POINTER
000220    0215 BUFF,   215;212;"H;"E;"L;"L;"O;"!;0
```

```
000221  0212
000222  0310
000223  0305
000224  0314
000225  0314
000226  0317
000227  0241
000230  0000
000231  7600  MON,    MONADR          /MONITOR ENTRY POINT

/ROUTINE TO TYPE A MESSAGE          PAL8-V13A  14-MARCH-79 PAGE 2

BUFADR  0216
BUFF    0220
MON     0231
MONADR  7600
NEXT    0204
PNTR    0217
START   0200

ERRORS DETECTED: 0
LINKS GENERATED: 0
```

The first column of the listing gives the field number and octal address. The second column is the assembled object code. The symbol table is printed at the end followed by the number of errors detected and number of links generated. Link generation is described in Section 5.12. Each error generates an error message (see Section 5.14).

If errors have been detected, the program has been written or typed incorrectly. Check it again.

The COMPILE command may also be used to assemble the program by typing

```
.COMPILE SAMPLE
```

Several options are available with the PAL command. The options are described in Section 5.3.

5.2.3 Loading and Saving a Program

Load the binary file generated by assembling SAMPLE.PA into memory by typing

```
.LOAD SAMPLE
```

The SAMPLE program is now the "current" memory image.

Since programs in memory image format can be executed directly, it is desirable to save this format of your program. Do this with the SAVE command by typing

```
.SAVE SYS SAMPLE
```

The memory image format of SAMPLE is now both in memory and on the system device as a new file called SAMPLE.SV.

5.2.4 Executing the Program

Since the program now resides on SYS and in main memory, you can execute it by typing

```
.START
```

Otherwise, you can load the file SAMPLE.SV into memory from SYS: and run by typing

```
.R SAMPLE
```

As the program runs, it displays the message HELLO!

You can also use the EXECUTE command to assemble, load and run the program.

```
.EXECUTE SAMPLE
```

This command produces the binary file SAMPLE.BN, loads it into memory, and starts it running.

Another load-and-go method that is available with the PAL command is the /G option. Typing

```
.PAL SAMPLE/G
```

assembles the input file SAMPLE.PA, loads the binary file, and executes the program. Also, the command

```
.LOAD SAMPLE/G
```

will load the binary file SAMPLE.BN and execute it.

5.2.5 Getting and Using a Cross-Reference Listing

The Cross-Reference Program (CREF) aids in debugging assembly language programs by pinpointing all references to a particular symbol.

Generate the CREF listing by using the CREF command or the PAL command with the /C option. Typing

```
.PAL SAMPLE/C-LS
```

will produce a binary file and the CREF listing as a file called SAMPLE.LS. Using the TYPE or LIST command will display the listing on the terminal or print it on a line printer, respectively. Further information on CREF is given with the discussion of the CREF command in Chapter 3.

The output of CREF is identical to the PAL8 assembler output except that the CREF program numbers each line in decimal and generates after the listing a cross-reference table that has the following format:

```
BUFADR     6    18#
BUFF      18    19     20#
MON       17    29#
MONADR     3#   29
NEXT       8#   16
PNTR       7    10     12     14     19#
START      4#
```

The cross-reference table contains every user-defined symbol and literal, sorted alphabetically. If literals are used, each literal is indicated by an underline followed by the field and address at which it occurs. For each symbol and literal there appears a list of numbers that specify the lines in which each is referenced. The symbol # follows the number of the line where the symbol is defined.

5.2.6 Obtaining a Memory Map

Many times it is desirable to obtain a map of a program showing memory locations used by the given binary file. Generate the map by using the MAP command. Typing

```
.MAP SAMPLE-L
```

will print the map on a line printer from SAMPLE.BN, and typing

```
.MAP SAMPLE
```

will display the map on the terminal (in effect, equivalent to the command MAP SAMPLE-T).

The input file must always be a binary file (.BN extension). Use the command

```
.MAP MAPFIL<SAMPLE
```

to place the map in the file MAPFIL.MP. Display the map on the terminal or print it on a line printer by using the commands TYPE and LIST, respectively. The map for the program SAMPLE is shown below.

```
BITMAP V4  FIELD 0

      0000000011111111222222223333333344444444555555556666666677777777
      0123456701234567012345670123456701234567012345670123456701234567

00000
00100

00200 11111111111111111111111110000000000000000000000000000000000000
00300

00400
00500

00600
00700
.
.
.
```

The output is a series of lines that are made up of a string of digits. Each digit, which represents a single memory location, can have a value of 0 to 3. A 0 means that the location is empty while a 1 means that the location was loaded into once. The appearance of a 2 means that a location was loaded into two times. A 3 means that the location was loaded into three or more times. The appearance of a 2 or 3 may imply a programming error in that two or more separate routines are each trying to load values into the same location. The example program shows memory locations 0200 through 0231 being loaded into once which is correct. Further information on the MAP command is given in Chapter 3.

5.3 PAL8 OPTIONS

The command string typed for the PAL command may include several options. The options are listed in Table 5-1.

Table 5-1
PAL8 Options

Option	Meaning
/B	This option makes the operator ! a 6-bit left shift instead of an inclusive OR (A!B equals A^100 B). This allows you to pack two 6-bit ASCII characters into a 12-bit word. This effect applies for the entire assembly.
/C	Create a symbol cross-reference listing (runs CREF.SV program) after assembly. The third output file specified (optional) is the temporary output file passed to CREF. The second output file is the listing file to be produced. If no third output file is given, SYS:CREFLS.TM is assumed and will be deleted after use. The /C option supersedes the /G and /L options if specified in the same command string.
/E	Enable error messages if a link is generated. The LG error message is generated as well as the link being flagged.
/F	Disable extra zero fill in TEXT pseudo-op. If the text in the TEXT pseudo-op contains an even number of characters, no word of zeros will be added to the end.
/G	Load the binary file into memory and begin execution at the indicated starting address. If no starting address is indicated, start at 200.
/H	Generate nonpaginated output. Headers (including page numbers and page format) are suppressed.
/J	Do not list lines of unassembled conditional source code.
/K	Used in assembling very large programs; allows more space in field 1 to be used for symbol table storage.
/L	Load the resulting binary file into memory but do not start it.

(continued on next page)

Table 5-1 (Cont.)
PAL8 Options

Option	Meaning
/N	Generate the symbol table but not the rest of the listing.
/O	Disable output of default (200) current location counter (CLC) setting after a FIELD pseudo-op. The CLC remains unchanged.
/S	Omit the symbol table normally generated with the listing.
/T	Output a carriage return/line feed in place of form feed character(s) in the program listing.
/W	Do not remember the number of literals that were previously stored on a page after changing the current location counter to an off page value and then back again.

When the /L or /G option is specified, you can also include any option in the command line for the LOAD command, such as = starting address option. If no address is specified, 00200 is assumed. If no binary output file is specified (by using the -NB option) with a /L or /G, a temporary file SYS:PAL8BN.TM is created and loaded.

5.4 CHARACTER SET

PAL8 programs are composed of physical lines containing assembly language mnemonics that indicate processor instructions, user-defined symbols, comments, listing control characters and pseudo-operators (assembler directives). The following characters (see Appendix A also) are used to specify these components.

1. The alphabetic characters A through Z
2. The numeric characters 0 through 9
3. The characters special characters and operators described below
4. Characters that are ignored during assembly, such as LINE FEED and FORM FEED

All other characters are illegal (except when used in a comment) and cause the following error message to be printed during passes 1 and 2:

IC nnnn

where:

nnnn represents the octal location at which the illegal character occurred.

As assembly proceeds, each instruction is assigned a location determined by the current location counter. When an illegal character or any other error is encountered during assembly, the value of the current location counter is displayed in the error message.

NOTE

You cannot use lower case characters in labels, instruction mnemonics and operands.

5.5 STATEMENTS

A PAL8 source program is prepared at the terminal using the Editor program (EDIT command) to enter a sequence of statements. You must enter each statement on a single line and terminate with a carriage return. PAL8 statements have four elements. They are identified by the order of their appearance in the statement and by the separating (or delimiting) character that follows or precedes the element. These elements are:

1. label
2. instruction
3. operand
4. comment

A statement must contain at least one of these elements and may contain all four. The assembler interprets and processes the statements, generating one or more binary instructions or data words, or performing an assembly process.

5.5.1 Labels

A label is the symbolic name created by the programmer to identify the location of a statement in the program. If present, the label is written first in a statement. It must begin with an alphabetic character, contain only alphanumeric characters, and be terminated by a comma. There must be no intervening spaces between any of the characters and the comma. A label may be of any length, but only the first six characters are significant. If a label is the only element on a line, it identifies the location of the next program location.

For example,

```
*200
A,          Defines A as 00200
B,0         Defines B as 00200 and stores 0000 at location 00200
```

5.5.2 Instructions

An instruction may be one or more of the mnemonic machine instructions or a pseudo-operation that directs assembly processing. (Assembly pseudo-ops are described in Section 5.11.) Instructions are terminated with zero or more spaces (or tabs) followed by a semicolon, slash, or the end of the line.

5.5.3 Operands

Operands are the octal or symbolic addresses of an assembly language instruction or the argument of a pseudo-operator, and can be any legal expression. In each case, interpretation of an operand depends on the instruction or the pseudo-op. Operands are terminated by a semicolon, slash, or the end of the line.

5.5.4 Comments

Comments are arbitrary strings of any character in the ASCII set (see Appendix A) that begin with a slash (/). Comments do not affect assembly processing or program execution but are useful in the program listing to record information for later analysis or debugging. The assembler ignores all characters between the slash and the next carriage return.

It is possible to have only a carriage return on a line, resulting in a blank line in the final listing. Such a line is ignored, and the current location counter is not incremented.

5.6 FORMAT CHARACTERS

The following characters are useful in controlling the format of an assembly listing to improve readability. They allow a neat readable listing to be produced by providing a means of spacing through the program.

5.6.1 Form Feed

The form feed character causes the assembler to output blank lines (or a form feed character if listing on the line printer) in order to skip to a new page in the output listing during pass 3; this feature is useful in creating a page-by-page listing. The form feed is generated by the Editor P (Page) command. The pseudo-op EJECT may also be used to form pages in the assembly listing (see Section 5.11.7).

5.6.2 Tab

Tabs are used in the body of a source program to separate fields into columns. For example, a line written

```
GO, TAD TOTAL/MAIN LOOP
```

is much easier to read if tabs are inserted to form

```
GO,  TAD     TOTAL  /MAIN LOOP
```

Each occurrence of a tab character causes PAL8 to output enough spaces to move to the next text column. Each text column is 8 characters wide.

5.6.3 Statement Terminators

Each statement is terminated by the carriage return/line feed character combination produced by the Editor when the RETURN key was pressed during the Insert or Append modes. The semicolon (;) may also be used as a statement terminator and is considered identical to a carriage return except that it will not terminate a comment. For example,

```
TAD A  /THIS IS A COMMENT; TAD B
```

The entire expression between the slash and the end of the line is considered a comment. Thus in this case the assembler ignores the TAD B. If, for example, a sequence of instructions to rotate the contents of the accumulator and link six places to the right is desired, it can be written as follows:

```
RTR
RTR
RTR
```

However, as an alternative, all three instructions can be placed on a single line by separating them with the special character semicolon and terminating the entire line with a carriage return. The above sequence of instructions can then be written

```
RTR;RTR;RTR
```

These multistatement lines are particularly useful when setting aside a section of data storage. For example, a 4-word block of data could be reserved by specifying either of the following:

```
LIST,     1;2;3;4
```

or

```
LIST,     1
          2
          3
          4
```

5.7 NUMBERS

Any sequence of digits delimited by a SPACE, TAB, semicolon, or the end of a line forms a number. PAL8 initially interprets numbers in octal (base 8). This can be changed to decimal using the pseudo-op DECIMAL (Section 5.11.10). Numbers are used in expressions.

5.8 SYMBOLS

A symbol is a string of alphanumeric characters beginning with a letter and delimited by a nonalphanumeric character. Although a symbol may be any length, only the first six characters are significant. Since additional characters are ignored, symbols which are identical in their first six characters are considered identical.

5.8.1 Permanent Symbols

The assembler symbol table initially contains definitions of the symbols for all computer instructions and PAL8 psuedo-ops. These symbols are permanently defined by PAL8 and need no further definition by the user; they are summarized in Section 5.15. For example,

```
HLT  This is a symbolic instruction assigned the value 7402 in
     its permanent symbol table.
```

5.8.2 User-Defined Symbols

All desired symbols not defined by the assembler (in its permanent symbol table) must be defined within the source program. User symbols may be defined in two ways:

1. As a statement label. Labels are assigned a value equal to the current location counter.

2. As an explicitly defined symbolic value (for example, A = 33).

Permanent symbols (instructions, special characters, and pseudo-ops) may not be redefined as a label or symbolic value. The following examples are legal labels:

```
ADDR,
TOTAL,
SUM,
AL,
```

The following labels are illegal:

```
AD>M,      (contains an illegal character)
7ABC,      (first character not alphabetic)
LA BEL,    (contains embedded spaces)
D+TAG,     (contains a legal but non-alphanumeric character)
LABEL      (a comma does not follow immediately after)
TAD,       (instruction mnemonic)
```

5.8.3 Current Location Counter

As source statements are processed, PAL8 assigns consecutive memory addresses to the instructions and data words of the object program (binary and listing) being produced.

The current location counter contains the address in which the next word of object code will be assembled and is automatically incremented each time a memory location is assigned. A statement that generates a single object program storage word increments the location counter by one. Another statement might generate six storage words, incrementing the location counter by six.

The location counter is set or reset by typing an asterisk followed by an expression giving the address in which the next program word is to be stored. The expression may include symbols, but every such symbol must have been defined at some previous point in the current source file(s) being assembled. If the origin is not set by the user, PAL8 begins assigning addresses at location 200.

The symbol TAG in the following example is assigned a value of 0300, the symbol B a value of 0302, and the symbol A a value of 0303.

```
            *300        /SET CURRENT LOCATION COUNTER TO 300
TAG,        CLA
            JMP A
B,          0
A,          DCA B
```

If a symbol is defined more than once as a label, the assembler will display the "illegal definition" error message:

ID address

where:

address is the octal value of the location counter at the second occurrence of the symbol definition. The symbol is not redefined. PAL8 error conditions are described in Section 5.14.

Example:

```
            *300
START,      TAD A
            DCA COUNTER
CONTIN,     JMS LEAVE
            JMP START
A,          -74
COUNTER,    0
START,      CLA CLL
            .
            .
            .
```

The symbol START would have a value of 0300; the symbol CONTIN would have a value of 0302; the symbol A would have a value of 0304; and the symbol COUNTER (considered COUNTE by the assembler, because the assembler uses only the first six characters of a symbol) would have a value of 0305. When the assembler processes the next line, it will display the error message:

ID COUNTE+0001

PAL8 will also display an error message if you refer to an undefined symbol. For example,

```
            *7170
A,          TAD C
            CLA CMA
            HLT
            JMP A1
C,          0
```

This would produce the "undefined symbol" error message

US A+0003

since the symbol A1 has not been defined.

5.8.4 Symbol Table

Initially, the assembler's symbol table contains the definitions of the computer instructions and PAL8 pseudo-ops; these are PAL8's permanent symbols. As the source program is processed, user-defined symbols and their 12-bit binary values are added to the symbol table. Entries in the symbol table are listed in alphabetic order at the end of the assembly listing file.

During pass 1, if PAL8 detects that the symbol table is full (in other words, there is no more memory space in which to store symbols and their values), the "symbol table exceeded" error message is displayed as follows:

```
SE address
```

and control returns to the monitor. The number of symbols defined in the program may be reduced by using relative addressing for example, JMP START+3).

You can also segment a program and assemble the segments separately, taking care to define correct links between the segments. PAL8's symbol capacity is 2621 (decimal) symbols of which 96 are permanent. Where PAL8 is run under BATCH, 2357 symbols can be defined. (Use of the /K option expands these to 2971 and 2719 respectively, but assembly time is slower.)

Instructions for altering the permanent symbol table are in Section 5.11.8.

5.8.5 Direct Assignment Statements

New symbols and their assigned values may be inserted directly into the symbol table by using a direct assignment statement.

Format:

```
SYMBOL=value
```

where:

value is a number or an expression.

No spaces or tabs may appear between the symbol to the left of the equal sign and the equal sign itself but they may appear (and are ignored) after the equal sign. The following are examples of direct assignment statements:

```
A=6
EXIT=JMP I 0
C=A+B
CO=JMS I [.]
```

All symbols to the right of the equal sign must already be defined, except that symbols are allowed to be undefined during pass 1. For example,

```
A=B
...
B=3
```

During pass 1, A will equal 0, since it is undefined thus far. During pass 2, A will equal 3, since B is given the value 3 at the end of pass 1. The use of the equal sign does not increment the location counter; it is an instruction to the assembler itself rather than a data value.

A direct assignment statement may also equate a new symbol to the value assigned to a previously defined symbol. For example,

```
BETA=17
GAMMA=BETA
```

The new symbol GAMMA is entered into the user's symbol table with the value 17. The value assigned to a symbol may be changed as follows:

```
ALPHA=5
ALPHA=7
```

The second line of code shown changes the value assigned to ALPHA from 5 to 7.

Symbols defined by use of the equal sign may be used in any valid expression. For example,

```
        *200
A=100           /DOES NOT UPDATE CURRENT LOCATION COUNTER
B=400           /DOES NOT UPDATE CURRENT LOCATION COUNTER
A+B             /THE VALUE 500 IS ASSEMBLED AT LOC 200
TAD A           /THE VALUE 1100 IS ASSEMBLED AT LOC 201
```

If the symbol to the left of the equal sign is in the permanent symbol table, the "redefinition" diagnostic

```
RD address
```

will be displayed as a warning (address is the value of the location counter at the point of redefinition). The new value will be stored in the symbol table. For example,

```
CLA=7600
```

will cause the diagnostic

```
RD+200
```

Whenever CLA is used after this point, it will have the value 7600.

5.8.6 Symbolic Instructions

Symbols used as instructions must be predefined by the assembler or defined in the assembly by the programmer. If a statement has no label, the instructions may appear first in the statement and must be terminated by a space, tab, semicolon, slash, or carriage return. The following are examples of legal instructions:

```
TAD       (a mnemonic machine instruction)
PAGE      (an assembler pseudo-op)
ZIP       (an instruction defined by the user)
```

5.8.7 Symbolic Operands

Symbols used as operands normally have a value defined by the user. The assembler allows symbolic references to instructions or data defined elsewhere in the program. Operands may be numbers or expressions. For example,

```
TOTAL, TAD ACI+TAG
```

The values of the two symbols ACI and TAG (previously defined in the program) are combined by a two's complement add. (See Section 5.9.1 on Operators.) This value is then used as the operand address.

5.9 EXPRESSIONS

Expressions are formed by the combination of symbols, numbers, and certain characters called operators, which cause specific arithmetic operations to be performed. An expression is terminated by either a comma, carriage return, or semicolon. Expressions are evaluated by a left-to-right scan.

5.9.1 Operators

Seven characters in PAL8 act as operators:

+	Two's complement addition
-	Two's complement subtraction
^	Multiplication (unsigned, 12-bit integer)
%	Division (unsigned, 12-bit integer)
!	Boolean inclusive OR
&	Boolean AND
Space (or TAB)	Treated as a Boolean inclusive OR except in a memory reference instruction

No checks for arithmetic overflow are made during assembly, and any overflow bits are lost from the high-order end. For example,

```
7755+24
```

will give a result of 1.

The operators plus (+) and minus (-) may be used freely as unary (prefix) operators.

Multiplication is accomplished by repeated addition. No checks for sign or overflow are made. All 12 bits of each factor are considered as magnitude. For example,

```
3000^2
```

will give a result of 6000.

Division is accomplished by repeated subtraction. The quotient is the number of subtractions that are performed. The remainder is not saved and no checks are made for sign. Division by 0 will arbitrarily yield a result of 0. For example,

```
7000%1000
```

will yield a result of 7. This example could be written as:

```
-1000%1000
```

The answer might be expected to be -1 (7777), but all 12 bits are considered as magnitude and the result is still 7.

Use of the multiplication and division operators requires more attention to sign (on the part of the programmer) than is required for simple addition and subtraction. Table 5-2 contains examples of expressions using arithmetic operators.

Table 5-2
Use of Arithmetic Operators

Expression	Also Written as	Result
7777+2	-1+2	+1
7776-3	-2-3	7773 or -5
0^2		0
2^0		0
1000^7		7000 or -1000
0%12		0
12%0		0
7777%1	-1%1	7777 or -1
7000%1000	-1000%1000	7
1%2		0

The ! operator causes a Boolean inclusive OR to be performed bit by bit between the left-hand term and the right-hand term. Giving the /B option changes the memory of "!" throughout the assembly to become a 6-bit left shift of the left term prior to the inclusive OR of the right. According to this interpretation,

```
If A=1 and B=2
```

then

```
A!B=0102
```

Under normal conditions A!B would be 0003.

The & operator causes a Boolean AND to be performed bit by bit between the left and right values.

SPACE is an operator that has special significance depending on the context in which it is used. When the symbol preceding the space is not a memory reference instruction as in the following example

```
SMA CLA
```

it causes an inclusive OR to be performed between them. In this case, SMA=7500 and CLA=7600. The expression SMA CLA is assembled as 7700. When SPACE is used following pseudo-operators, it merely delimits the symbol. When it is used after memory reference operators, it has a special function explained below.

User-defined symbols are treated as non-memory reference instructions. For example,

```
A=1234
B=77
A B
```

stores a data value of 1277 (octal), the same as A!B.

If data values are generated, the current location counter is incremented. For example,

```
B-7;A+4;A-B
```

produces three words of information; the current location counter is incremented after each expression. The statement

```
HLTCLA=HLT CLA
```

produces no information to be loaded (it produces a value for "HLTCLA" in the symbol table) and hence does not increment the current location counter.

In the program

```
          *4271
TEMP,
TEM2,     0
```

the location counter is not incremented after the line TEMP,; the two symbols TEMP and TEM2 are assigned the same value, in this case 4721.

Since a CPU instruction has an operation code of three bits as well as one indirect bit, one page bit, and seven address bits, the assembler must combine memory reference instructions in a manner somewhat differently from the way in which it combines operate or IOT instructions. The assembler differentiates between memory reference instructions and user-defined symbols. The following symbols are the memory reference instructions:

```
AND  0000  Logical AND
TAD  1000  Two's complement addition
ISZ  2000  Increment and skip if zero
DCA  3000  Deposit and clear accumulator
JMS  4000  Jump to subroutine
JMP  5000  Jump
```

When the assembler has processed one of these symbols, the space or tab following it acts as an address field delimiter. In the example,

```
          *4100
          JMP A
A,        CLA
```

A has the value 4101, JMP has the value 5000, and the space acts as a field delimiter. These symbols are represented in binary as follows:

```
A:   100 001 000 001
JMP: 101 000 000 000
```

The seven address bits of A are taken, for example,

```
000 001 000 001
```

The remaining bits of the address are tested to see if they are zeros (page zero reference); if they are not, the current page bit is set:

```
000 011 000 001
```

The operation code is then ORed into the JMP value to form

```
101 011 000 001
```

or, in octal

```
5301
```

In addition to the above tests, the page bits of the address field are compared with the page bits of the current location counter. If the page bits of the address field are nonzero and do not equal the page bits of the current location counter, an out-of-page reference is being attempted, and the assembler will take action as described in Section 5.12, on Link Generation and Storage.

5.9.2 Special Characters

In addition to the operators described in the previous section, PAL8 recognizes several special characters that serve specific functions in the assembly process:

=	equal sign
,	comma
*	asterisk
.	period
"	double quote
()	parentheses
[]	square brackets
/	slash
;	semicolon
<>	angle brackets
$	dollar sign

The equal sign, comma, asterisk, slash, and semicolon have been previously described. The remaining special characters are described in the following sections.

5.9.2.1 **Period (.)** - The period character (.) represents the value contained in current location counter. It may be used in any expression (except to the left of an equal sign). It must be separated from other symbols by a space or other operator. For example,

```
*200
JMP .+2
```

is equivalent to JMP 0202. Also,

```
*300
.+2400
```

will produce in location 0300 the quantity 2700. Consider

```
*140
PRINT=JMS I.
2200
```

The second line (PRINT=JMS I.) does not increment the current location counter; therefore, 2200 is placed in location 140 and PRINT is placed in the user's symbol table with an associated value of 4540 (the octal equivalent of JMS I.). This technique is useful in creating "global" subroutine calls.

Large buffers may be defined by using a format such as the following:

```
        *1200    /BUFFER LOCATION
BUFFER, 0        /FIRST WORD OF BUFFER
        *.+400   /DEFINE A 401 WORD BUFFER
NEXT,   0        /PROGRAM CONTINUES
```

5.9.2.2 **Double Quote (")** - When a double quote (") precedes an ASCII character, PAL8 assembles the 8-bit ASCII equivalent of the character. (ASCII codes are listed in Appendix A.) For example,

```
CLA
TAD       ("A)
```

The constant 0301 is placed in the accumulator when these two instructions are eventually executed. The character must not be a carriage return or one of the characters that are ignored on input (discussed at the end of this section).

5.9.2.3 **Parentheses () and Brackets[]** - Left and right parentheses, (), enclose a current page literal. The right parenthesis is optional.

```
*200

      .
      .
      CLA
      TAD INDEX
      TAD (2)
      DCA INDEX
      .
      .
```

The left parenthesis is a signal to the assembler that the expression that follows is to be evaluated and assigned a word in the literal area of the current page. This is the same area in which the indirect address linkages are stored. In the above example, the quantity 2 is stored in a word in the literal area beginning at the end of the current memory page. The instruction in which the literal appears is given the address of the literal. A literal is assigned to storage the first time it is encountered; subsequent references to the same literal from the current page are made to the same location. The use

of literals frees symbol storage table and makes programs much more readable. Literal allocation starts with the last location on the page and works towards the first location. If the literal area reaches the instruction/data area, a PE (Page Exceeded) error message is generated and assembly continues.

If square brackets ([and]) are used in place of parentheses, the literal is assigned to page zero rather than the current page. This enables a value to be referenced from any address within the field. For example,

```
*200
     TAD[2]
     .
     .
     .
*500
     TAD[2]
     .
     .
     .
```

The closing member is optional. Literals may contain any expression.

NOTE

Literals can be nested, for example:

```
*200
     TAD (TAD (30))
```

This type of nesting may be continued to as many as six levels, depending on the number of other literals on the page and the complexity of the expressions within the next. If the limits of the assembler are reached, the error message BE (too many levels of nesting) or PE (too many literals) will result.

5.9.2.4 **Angle Brackets** (<>) - Angle brackets (<>) are used as conditional delimiters. The code enclosed in the angle brackets is assembled or ignored, depending on the definition of the symbol or value of the expression preceding the angle brackets. (The IFDEF, IFNDEF, IFZERO, and IFNZRO pseudo-operators are used with angle brackets and are described in Section 5.11.9.)

NOTE

Programs that use conditionals should avoid angle brackets in comments as they will be interpreted as beginning or terminating the conditional.

5.9.2.5 **Dollar Sign($)** - The dollar sign ($) character is optional at the end of a program and is interpreted as an unconditional end-of-pass. It may, however, occur in a text string, comment, or double quote (") term, in which case it is interpreted in the same manner as any other character. This feature is provided for compatibility with older PDP-8 assemblers, and its use is not recommended.

5.9.3 Other Characters

The following characters are handled by the assembler for the pass 3 program listing but are otherwise ignored:

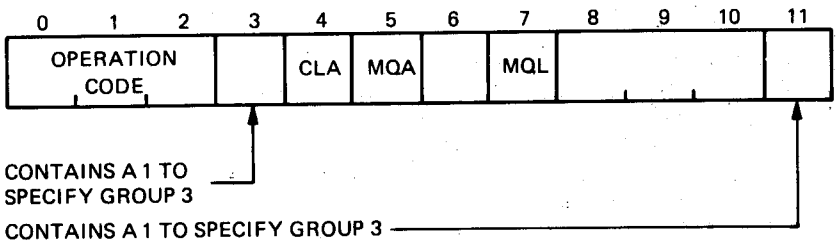
FORM FEED	Skips to a new page
LINE FEED	Creates a line spacing without causing a carriage return
SPACE	Spaces to next character position
TAB	Spaces to next "tab column" of 8 characters
RETURN	Terminates each line
BELL (CTRL/G)	Sounds the terminal buzzer

5.10 INSTRUCTION SET

The instruction set for PAL8 includes processor instructions, input/output (I/O) instructions, and assembler instructions (pseudo-ops). The processor instructions are further divided into two basic groups of instructions: memory reference and microinstructions. (See Section 5.15 for detailed listing of instructions.)

5.10.1 Memory Reference Instructions

Memory reference instructions have the following format:



Memory Reference Bit Instructions

Bits 0 through 2 contain the operation code of the instruction to be performed. Bit 3 indicates whether the memory reference is indirect. Bit 4 indicates whether the instruction is referencing the current page rather than page zero. Bits 5 through 11 (7 bits) specify an address. Using these seven bits, 200 octal (128 decimal) locations can be directly specified; the page bit increases accessible locations to 400 octal or 256 decimal. A list of the memory reference instructions and their codes is given at the end of this chapter.

In PAL8, a memory reference instruction must be followed by one or more spaces and tabs, an optional I and/or Z designation, and any valid expression.

When the character I appears in a statement between a memory reference instruction and an operand, the operand, is interpreted as the address (or location) containing the address of the operand to be used in the current instruction. Consider:

```
TAD 40
```

which is a direct address statement, where 40 is interpreted as the location on page zero containing the quantity to be added to the accumulator. References to locations on the current page and page zero may be done directly. For compatibility with older PDP-8 assemblers, the symbol Z is also accepted as a way of indicating a page zero reference, as follows:

```
TAD Z 40
```

This is an optional notation, not differing in effect from the previous example. Thus, if location 40 contains 0432, then 0432 is added to the accumulator when the code is executed. Now consider:

```
TAD I 40
```

which is an indirect address statement, where 40 is interpreted as the address containing the address of the quantity to be added to the accumulator. Thus, if location 40 contains 0432, and location 432 contains 0456, then 456 is added to the accumulator when the instruction is eventually executed.

NOTE

Because the letter I is used to indicate indirect addressing, it may not be redefined as a label or variable. Likewise the letter Z, which is sometimes used to indicate a page zero reference, may not be redefined.

5.10.2 Microinstructions

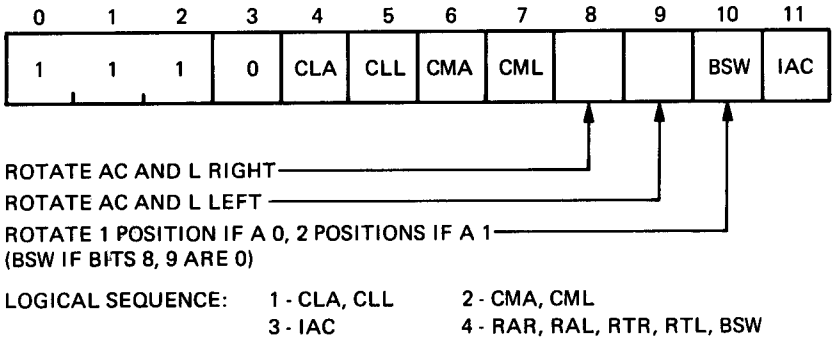
Microinstructions are divided into two classes: operate and Input/Output Transfer (IOT) microinstructions. Operate microinstructions are further subdivided into Group 1, Group 2, and Group 3.

NOTE

If an illegal combination of microinstructions is specified, the assembler will perform an inclusive OR between them, resulting in an unexpected operation. For example,

```
CLL SKP is interpreted as SPA
(7100)(7410)                (7510)
```

5.10.2.1 **Operate Microinstructions** - Operate instructions are divided into three groups of micro instructions. Although it is possible to combine instructions within a group, it is not logically possible to combine instructions from different groups. Group 1 microinstructions perform clear, complement, rotate and increment operations on the Accumulator and Link registers, and are designated by the presence of a 0 in bit 3 of the machine instruction word.



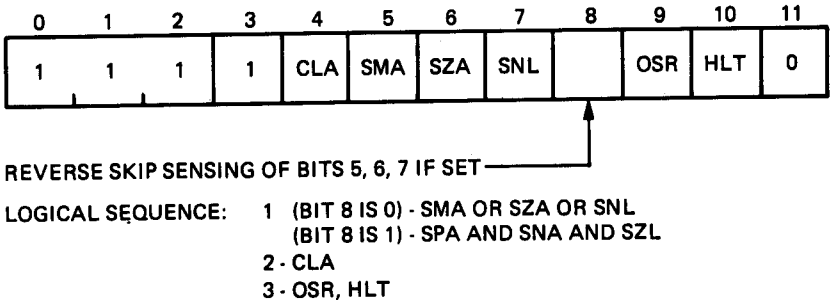
Group 1 Operate Microinstruction Bit Assignments

The following constants can be produced in the accumulator by a single instruction:

Constant	Instruction
0	CLA
1	CLA IAC
2	CLA CLL CML RTL
3*	CLA CLL CML IAC RAL
4*	CLA CLL IAC RTL
6*	CLA CLL CML IAC RTL
100*	CLA IAC BSW
2000	CLA CLL CML RTR
3777	CLA CLL CMA RAR
4000	CLA CLL CML RAR
5777	CLA CLL CMA RTR
6000*	CLA CLL CML IAC RTR
7775	CLA CLL CMA RTL
7776	CLA CLL CMA RAL
7777	STA (=CLA CMA)

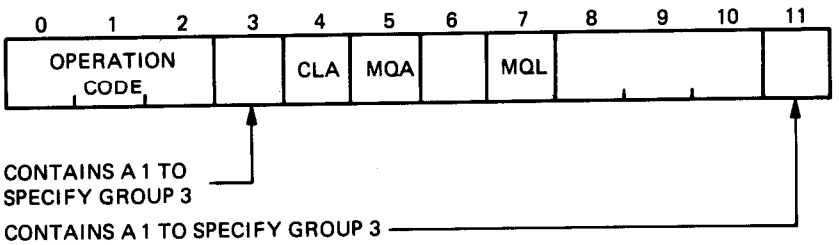
Instructions that are starred (*) must not be used on software to be transported onto old (non-omnibus) PDP-8 computers.

Group 2 microinstructions check the contents of the Accumulator and Link and, based on the check, continue to or skip the next instruction. Group 2 microinstructions are identified by the presence of a 1 in bit 3 and a 0 in bit 11 of the machine instruction word.



Group 2 Operate Microinstruction Bit Assignments

Group 3 microinstructions reference the MQ register. They are differentiated from Group 2 instructions by the presence of a 1 in bits 3 and 11 of the machine instruction word.



Group 3 Operate Microinstruction Bit Assignments

Group 1 and Group 2 microinstructions cannot be combined since bit 3 determines either one or the other. Group 2 has two groups of skip instructions. They can be referred to as the OR group and the AND group.

OR Group	AND Group
SMA	SPA
SZA	SNA
SNL	SZL

The OR group is designated by a 0 in bit 8, and the AND group by a 1 in bit 8. OR and AND group instructions cannot be combined with each other since bit 8 determines either one or the other.

If skip instructions are combined, it is important to note the conditions under which a skip may occur.

1. OR Group -- If these skips are combined in a statement, the inclusive OR of the conditions determines the skip. For example:

   ```
   SZA SNL
   ```

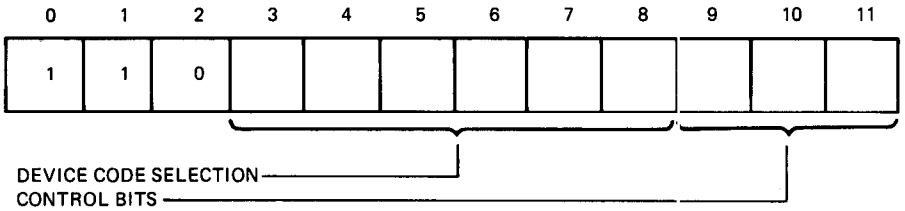
   The next statement is skipped if the Accumulator contains 0000 or the link is a 1 or both.

2. AND Group -- If the skips are combined in a statement, the logical AND of the conditions determines the skip. For example:

```
SNA SZL
```

The next statement is skipped only if the accumulator differs from 0000 and the link is 0.

5.10.2.2 **Input/Output Transfer Microinstructions** - Input/output transfer microinstructions initiate operation of peripheral equipment and effect an information transfer between the central processor and the input/output device(s).



IOT Instruction Bit Assignments

5.10.3 **Autoindexing**

Consecutive address references are often necessary for obtaining data values when processing large amounts of data. Autoindex registers (locations 10-17 of each memory field) are used for this purpose. When one of the absolute locations from 10 through 17 (octal) is indirectly addressed, the contents of the location are incremented and then used as an indirect operand address. This allows consecutive memory locations to be addressed, using a minimum of instructions. It must be remembered that initially these locations (10 through 17 on page 0 of each field) must be set to one less than the first desired address. No incrementation takes place when locations 10 to 17 are addressed directly. For example, if the instruction to be executed next is in location 300 and the data to be referenced is on the page starting at location 5000, autoindex register 10 can be used to address the data as follows:

```
0276    1377    TAD (4777)    /=5000-1
0277    3010    DCA 10        /SET UP AUTO INDEX
0300    1410    TAD I 10      /INCREMENT TO 5000
.       .       .             /BEFORE USE OF AN INDIRECT
.       .       .             /ADDRESS
.       .       .
0377    4777    (Literal Area of Page 1)
```

Note that the Data Field must be set to the field of the data being referenced, in this case, Field 0.

When the instruction in location 300 is executed, the contents of location 10 will be incremented to 5000, and the contents of location 5000 will be added to the contents of the accumulator. If the instruction TAD I 10 is executed again, the contents of location 5001 will be added to the accumulator, and so on.

5.11 PSEUDO-OPERATORS

Pseudo-operators are used to direct the assembler to perform certain processing operations or to interpret subsequent coding in a certain manner. Some pseudo-ops generate storage words in the object program, other pseudo-ops direct the assembler on how to proceed with the assembly. The pseudo-ops are defined in the permanent symbol table.

5.11.1 Indirect and Page Zero Addressing

The pseudo-operators I and Z specify the type of addressing to be performed. These are discussed in Section 5.10.1.

5.11.2 Extended Memory

5.11.2.1 **FIELD Pseudo-Operator** - The pseudo-op FIELD instructs the assembler to output a field setting so that it may assemble code into more than one memory field. This field setting is output during pass 2 in the object binary file and is recognized by the LOAD command which in turn causes all subsequent information to be loaded into the field specified by the expression.

Format:

```
FIELD ff
```

where:

ff — is an integer, a previously defined symbol, or an expression whose terms have been defined. The value must be in the range 0 to 37.

This field setting is output to the binary file during pass 2 along with a default current location counter setting of 200. These settings are read by the LOAD command when it is executed to begin loading information into the new field.

The field setting is never remembered by the assembler except as the high-order digit of the Location Counter on the listing. A binary file produced without field settings will be loaded into field 0 when using the LOAD command.

A symbol in one field may be used to reference the same location in any other field. The field to which it refers is determined by the use of the CDF and CIF instructions. CDF and CIF instructions must be used prior to any instruction referencing a location outside the current field, as shown in the following example:

```
        *200
        TAD P301
        CDF 00
        CIF 10
        JMS PRINT
        CIF 10
        JMP NEXT
P301,   301
        FIELD 1
        *200
```

```
NEXT,   TAD P302
        CDF 10
        JMS PRINT
        HLT
P302,   302
PRINT,  0
        TLS
        TSF
        JMP .-1
        CLA
        RDF
        TAD PCDIF
        DCA .+1
        000
        JMP I PRINT
PCDIF,  CDF CIF 0
```

When FIELD is used, the assembler follows the new FIELD setting with an origin at location 200. For this reason, to assemble code at location 400 in field 1, it would be necessary to write

```
FIELD1      /CORRECT EXAMPLE
*400
```

The following is incorrect and will not generate the desired code:

```
*400        /INCORRECT
FIELD1
```

Specifying the /O option to PAL8 inhibits the output of the default current location counter setting of 200 after a FIELD pseudo-op. This leaves the current location counter at its previous value.

5.11.2.2 **Specifying Data and Instruction Fields** - The PDP-8's memory addresses are specified by the contents of the Memory Reference Instruction modified by the Data Field and Instruction Field Registers. Direct addressing, specified by bit 3=0, causes reference to the address given in bits 5-11 in page 0 of the current field, if bit 4=0, or to the current page, if bit 4=1. Indirect addressing, specified by bit 3=1, causes reference to the indirect address contained in the location specified by bits 4-11, used as above. The indirect address for AND, TAD, ISZ, and DCA refers not to the current field, but to the field specified in the Data Field Register. The JMP and JMS instructions refer to locations in the field specified in the Instruction Field Register.

The Data Field Register and Instruction Field Register can be set under program control by means of the CIF and CDF instructions. The CIF instruction causes the Instruction Field Buffer to be set to the specified field. The CDF instruction causes the Data Field Register to be changed immediately. Other instructions allow the program to read, save, and restore the Data Field and Instruction Field Registers. Completion of execution of a JMP or JMS instruction causes the Instruction Field Register to be set to the contents of the Instruction Field Buffer. This procedure permits a program to choose a new field, then execute a jump from the current field to a chosen address in the new field.

The CDF and CIF instructions let you specify fields 0 to 37 as data and instruction fields. Entering the argument requires knowledge of the bit arrangement of these two instructions.

Mnemonic	IOT	Bit Arrangement
CDF	6201	110 01a cdeb01
CIF	6202	110 01a cdeb10

Bits a cde b indicate the data or instruction field. (The positioning of the bits is eccentric as to maintain compatibility with older PDP-8 systems.)

To specify a field from 0 to 7, use bits c, d, and e only. The format of the instructions are:

```
CDF n0
CIF n0
```

where:

n0 is an octal number that PAL8 ORs with the instruction code

n is an octal digit from 0 to 7 (bits cde)

For example, this instruction

```
CDF     60
```

specifies field 6 by causing PAL8 to do the following OR.

```
                                    a cde b
Instruction code        6201    110 010 000 001
Argument                  60    000 000 110 000

                        6261    110 010 110 001
```

Keep in mind that to call for fields above field 7 (above 32K) with CDF and CIF, you must first load the KT8A Extended Mode Register with the LXM instruction (see the KT8A Memory Management Control User's Guide). For example, the following code deposits 7777 in field 12, location 1000.

```
LXM
CDF 24
TAD (7777
DCA I (1000
```

KT8A users must also ensure that their programs and device handlers do not contain the following combination of instruction steps.

```
CIF       /Change instruction field
IOT       /Any PDP8 IOT instruction
JMP I     /The instruction that does the CIF
```

If you enable the KT8A and turn on the interrupts, the KT8A hardware will return to the wrong place on traps between the CIF and JMP I instructions.

To specify a field from 10 to 17, use bits cde and set bit b. The format of the instructions are:

```
CDF n4
CIF n4
```

where:

n4	is an octal number that PAL8 ORs with the instruction code
n	is an octal value from 0 to 7 (bits cde)
4	is an octal value indicating a field range of 10 to 17 (sets bit b)

For example, this instruction

```
CDF 64
```

indicates field 16.

To specify a field from 20 to 27, use bits cde and set bit a. The formats are:

```
CDF ln0
CIF ln0
```

where:

ln0	is an octal number that PAL8 ORs with the instruction
l	is an octal value indicating field range 20 to 27 (sets A)
n	is a value from 0 to 7 (bits CDE)

For example, this instruction

```
CDF 160
```

indicates field 26.

To specify a field from 30 to 37, use bits CDE and set bit A and B. The formats are:

```
CDF ln4
CIF ln4
```

where:

ln4	is an octal number that PAL8 ORs with the instruction
l...4	are octal values indicating a field range of 30 to 37 (set bits A and B)
n	is an octal digit in the range 0 to 7 (bits CDE)

For example, this instruction

```
CDF 164
```

specifies field 36

One way to avoid confusion with this unusual bit configuration is to define high fields with convenient mnemonics. For example:

```
F36=164
CDF F36
```

5.11.3 Resetting the Location Counter

The PAGE n pseudo-op resets the location counter to the first address of page n, where n is an integer, a defined symbol, or a symbolic expression whose terms have been defined previously and whose value is from 0 to 37 inclusive. If n is not specified, the location counter is reset to the beginning of the next page of memory. For example,

```
PAGE 2 sets the location counter to 00400
PAGE 6 sets the location counter to 01400
```

If the pseudo-op is used without an argument and the current location counter is at the first location of a page, the current location counter will not be reset. In the following example, the code TAD B is assembled into location 00400:

```
*377
JMP .-3
PAGE
TAD B
```

If several consecutive PAGE pseudo-ops are given, the first will cause the current location counter to be reset as specified. The rest of the PAGE pseudo-ops will be ignored.

5.11.4 Reserving Memory

ZBLOCK instructs the assembler to reserve n words of memory containing zeros, starting at the address indicated by the current location counter. It is of the form

```
ZBLOCK n
```

For example,

```
ZBLOCK 40
```

causes the assembler to reserve 40 (octal) words and store zeros in them. The n may be an expression. If n=0, no locations are reserved. A ZBLOCK statement may have a label.

5.11.5 Relocation Pseudo-Operator

It is sometimes desirable to assemble code at a given location and then move it at run time to another location for execution. This may result in errors unless the relocated code is assembled in such a way that the assembler assigns symbols their execution-time addresses rather than their load-time addresses. The RELOC pseudo-op establishes a virtual location counter without altering the actual location counter. The line

```
RELOC expr
```

sets the virtual location counter to expr. The line

```
RELOC
```

resets the virtual location counter back to the actual location counter value and terminates the relocation section.

For example, the following program causes the assembler to load the word at CODE into location 204, but assembles it as if it were loaded into 1371. The asterisks after the location values indicate that the virtual and the actual location counters differ for that line of code. Only the virtual location counter is listed. Do not use current page literals in code that is affected by a RELOC.

```
                0200                    *200
000200          1205                    TAD CHAR
000201          7402                    HLT
                1367                    RELOC 1367
001367*         1371                    TAD CODE
001370*         7402                    HLT
001371*         0000            CODE,   0
                0205                    RELOC
000205          0000            CHAR,   0
```

5.11.6 Suppressing the Listing

The portions of the source program enclosed by XLIST pseudo-ops will not appear in the listing file; the assembled binary will be output, however.

Two XLIST pseudo-ops may be used to enclose the code to be suppressed in which case the first XLIST with no argument will suppress the listing, and the second will allow it again. XLIST may also be used with an expression as an argument. The listing will be inhibited if the expression is not equal to zero, or allowed if the expression is equal to zero. XLIST pseudo-ops never appear in the assembly listing.

5.11.7 Controlling Page Format

The EJECT pseudo-op causes the listing to skip to the top of the next page. A page eject is done automatically every 55 lines; EJECT is useful if more frequent paging is desired. If this pseudo-op is followed by a string of characters, the first 40 (decimal) characters of that string will be used as a new title at the top of each page of the listing.

5.11.8 Altering the Permanent Symbol Table

If your DECsystem includes one or more optional devices that you want to program directly whose instruction sets are not defined in the permanent symbol table, then you would have to alter the symbol table to include the IOT instructions for these devices.

In another situation, programmed-defined symbols might require more space than is available in the symbol table. Again, the symbol table would have to be altered by removing all definitions not needed in the program being assembled. PAL8 has three pseudo-ops that can be used to alter the permanent symbol table. These pseudo-ops are recognized by the assembler only during pass 1. During either pass 2 or pass 3, the assembler ignores them, and they have no effect.

EXPUNGE deletes the entire permanent symbol table except pseudo-ops.

FIXTAB appends all currently defined symbols to the permanent symbol table. All symbols defined before the occurrence of FIXTAB are made part of the permanent symbol table for the current assembly.

To append the following instructions to the symbol table, you have to generate an ASCII file called SYM.PA. This file contains the following:

```
CLSK=6131   /SKIP ON CLOCK INTERRUPT
FIXTAB      /SO THAT THIS WON'T BE
            /PRINTED IN THE SYMBOL TABLE
```

The ASCII file is then entered in the PAL8 input designation. By placing the definitions at the beginning of the source file, you can avoid loading an extra file. Each time the assembler is loaded, the PAL8's initial permanent symbol table is restored.

The third pseudo-op used to alter the permanent symbol table in PAL8 is FIXMRI. FIXMRI defines a memory reference instruction and is of the form:

```
FIXMRI name=value
```

The letters FIXMRI must be followed by one space, the symbol for the instruction to be defined, an equal sign, and the value of the symbol. The symbol will be defined and stored in the symbol table as a memory reference instruction. The pseudo-op must be repeated for each memory reference instruction to be defined. For example,

```
EXPUNGE
FIXMRI TAD=1000
FIXMRI DCA=3000
CLA=7200
FIXTAB
```

When the preceding program segment is read by the assembler during pass 1, all symbol definitions are deleted, and the three symbols listed are added to the permanent symbol table. Notice that CLA is not a memory reference instruction. This process can be performed to alter the assembler's symbol table so that it contains only the symbols used at a given installation or by a given program.

5.11.9 Conditional Assembly Pseudo-Operators

The IFDEF pseudo-op takes the form

```
IFDEF symbol <source code>
```

If the symbol indicated is previously defined, the code contained in the angle brackets is assembled; if the symbol is undefined, this code is ignored. Any number of statements or lines of code may be contained in the angle brackets. The format of the IFDEF statement requires a single space before and after the symbol.

Example:

```
IFDEF A <TAD A
         DCA B>
```

The IFNDEF pseudo-op is similar in form to IFDEF and is expressed as:

```
IFNDEF symbol <source code>
```

If the symbol indicated has not been previously defined, the source code in angle brackets is assembled. If the symbol is defined, the code in the angle brackets is ignored.

The IFZERO pseudo-op is of the form

```
IFZERO expression <source code>
```

If the evaluated expression is equal to zero, the code within the angle brackets is assembled; if the expression is nonzero, the code is ignored. Any number of statements or lines of code may be contained in the angle brackets. The expression may not contain any embedded spaces and must have a single space preceding and following it.

IFNZRO is similar in form to the IFZERO pseudo-op and is expressed as

```
IFNZRO expression <source code>
```

If the evaluated expression is not equal to zero, the source code within the angle brackets is assembled; if the expression is equal to zero, this code is ignored.

Pseudo-ops can be nested. For example,

```
IFDEF SYM <IFNZRO X2<...>>
```

The evaluation and subsequent inclusion or deletion of statements are done by evaluating the outermost pseudo-op first.

Conditional code that is not assembled can be deleted from the listing output by the /J option.

5.11.10 Radix Control

Numbers used in a source program are initially considered to be octal numbers. However, the program may change the radix interpretation by the use of the pseudo-operators DECIMAL and OCTAL. The DECIMAL pseudo-op interprets all following numbers as decimal until the occurrence of the pseudo-op OCTAL. The OCTAL pseudo-op resets the radix to octal.

5.11.11 Entering Text Strings

The TEXT pseudo-op allows a string of text characters to be entered as data and stored in 6-bit ASCII. The format required is the pseudo-op TEXT followed by one or more spaces or tabs, a delimiting character (must be a printing character), the string of text, and the same delimiting character. If the number of characters in a specified string is odd, the last word will contain zero in its right half. If the number of characters is even, a final word of zero will be appended. Either way, a final 6-bit character of zeros is generated providing a convenient "end-of-string" indication. Note that the /F option prevents the extra word of zero when the number of characters is even. Note also that six bits are sufficient to encode only the printing characters. For example:

```
TAG,        TEXT"123*"
```

The string would be stored as

```
6162
6352
0000
```

The /F option inhibits the generation of the extra 6-bit zero word. Alternatively, the statement "*.-1" may be used to eliminate the extra zero word (when the number of characters is even).

5.11.12 End-of-File Signal

PAUSE signals the assembler to stop processing the file being read. The current pass is not terminated, and processing continues with the next file. The PAUSE pseudo-op is present only for compatibility with paper tape assemblers, and its use is not recommended.

5.11.13 Use of DEVICE and FILENAME Pseudo-Operators

The pseudo-operators DEVICE and FILENAME may be used by calls to the User Service Routine (see Appendix C), or may be used for other purposes. They store 6-bit ASCII strings at the current location. The form for these pseudo-ops is

```
DEVICE name
FILENAME name.extension
```

The name used with DEVICE can be from 1 to 4 alphanumeric characters. These are trimmed to 6-bit ASCII and packed into two words, filled-in with zeros on the right if necessary. With FILENAME (FILENA is also acceptable), the name (or name.extension) may be from 1 to 6 alphanumeric characters and the optional extension may be 1 or 2 characters. The characters are trimmed to 6-bit ASCII and packed two to a word. Three words are allocated for the file name, filled with zeros on the right if fewer than 6 characters are specified, followed by one word for the extension. The example

```
L,    FILENAME ABC.DA
```

is equivalent to the following coding:

```
L,   0102
     0300
     0000
     0401
```

The symbols DEVICE and FILENAME may not be used as labels since they are predefined pseudo-ops.

5.12 LINK GENERATION AND STORAGE

In addition to handling symbolic addressing on the current page of memory, PAL8 automatically generates "links" for off-page references. If a direct memory reference is made to an address not on the page where an instruction is located or on page 0, the assembler sets the indirect bit (bit 3), and an indirect address literal (a "link") will be stored on the current memory page. If the specified reference is already an indirect one, the error diagnostic II (Illegal Indirect) will be generated. In the following example,

```
          *2117
A,        CLA
          .
          .
          .
          *2600
          JMP A
```

the assembler will recognize that the register labeled A is not on the current page and will generate a link to it as follows:

1. In location 2600 the assembler will place the word 5777 (equivalent to JMP I 2777).

2. In address 2777 (the last available location on the current page), the assembler will place the word 2117 (the actual address of A).

In the listing, the octal code for the instruction will be followed by a single quote (') to indicate that a link was generated.

Although the assembler will recognize and generate an indirect address linkage when necessary, the program may indicate an explicit indirect address by the pseudo-op I, for example:

```
     *2117
A,   CLA
     .
     .
     .
     *2600
     JMP I (A)
```

The assembler cannot generate a link for an instruction that is already specified as being an indirect reference, since the computer supports no such instruction format. For the example shown below, the assembler will print the error message II (Illegal Indirect).

```
        *2117
A,      CLA
        .
        .
        .
        *2600
        JMP I A
```

NOTE

The option /E makes link generation a condition that produces the LG (Link Generated) error message.

The above coding will fail because A is not defined on the page where JMP I A is attempted, and the indirect bit is already set.

Literals and links are stored on each page starting at page address 177 (relative) and extending toward page address 0 (relative). Whenever the origin is then set to another page, the literal area for the current page is output. There is room for 160 (octal) literals and links on page zero and 100 (octal) literals on each other page of memory. Literals and links are stored only as far down as the highest instruction on the page. Further attempts to define literals will result in a PE (Page Exceeded) or ZE (page Zero Exceeded) error message.

5.13 TERMINATING ASSEMBLY

PAL8 will terminate assembly and return to the monitor under any of the following conditions:

1. Normal exit: The end of the source program was reached on pass 2 (or pass 3 if a listing is being generated).

2. Fatal error: One of the following error conditions was found and flagged (Section 5.14):

   BE DE DF PH SE

3. CTRL/C: If typed, control returns to the monitor. Any partial output files are deleted.

5.14 DIAGNOSTIC ERROR MESSAGES

PAL8 will detect and flag error conditions and display error messages both on the console terminal and in the program listing. The format of the error message is:

code address

where:

code — is a 2-letter code that specifies the type of error.

address — is either the absolute octal address where the error occurred or the address of the error relative to the last label (if there was one) on the current page.

For example, the instruction sequence:

```
BEG,    TAD LBL
        %TAD LBL
```

would produce the error message

```
IC BEG+0001
```

since % is an illegal character because of its placement.

In the listing, error messages are output as 2-character messages on the line just prior to the line in which the error occurred. Table 5-3 lists the PAL8 error codes. Fatal errors cause PAL8 to terminate the assembly immediately (deleting any output files produced so far) and return to the monitor.

Table 5-3
PAL8 Diagnostic Error Codes

Error Code	Meaning
BE	A PAL8 internal table has overflowed. This situation can usually be corrected by decreasing the level of literal nesting or the number of current page literals used prior to this point on the page. Fatal error: assembly cannot continue.
CF	Chain to CREF error. CREF.SV was not found on SYS.
DE	Device error. An error was detected when trying to read or write a device. Fatal error: assembly cannot continue.
DF	Device full. Fatal error: assembly cannot continue.
IC	Illegal character. The character is ignored and the assembly is continued.
ID	Illegal redefinition of a symbol. An attempt was made to give a previous symbol a new value by means other than the equal sign. The symbol is not redefined.
IE	Illegal equals. An attempt was made to equate a variable to an expression containing an undefined term. The variable remains undefined.
II	Illegal indirect. An off-page reference was made; a link could not be generated because the indirect bit was already specified.

(continued on next page)

Table 5-3 (Cont.)
PAL8 Diagnostic Error Codes

Error Code	Meaning
IP	Illegal pseudo-op. A pseudo-op was used in the wrong context or with incorrect syntax.
IZ	Illegal page zero reference. The pseudo-op Z was found in an instruction which did not refer to page zero. The Z is ignored.
LD	The /L or /G options have been specified but the Absolute Loader system program is not present.
LG	Link generated. This code is displayed only if the /E option was specified to PAL8.
PE	Current nonzero page exceeded. An attempt was made to 1. Override a literal with an instruction. 2. Override an instruction with a literal. 3. Use more literals than the assembler allows on that page. This can be corrected by decreasing either the number of literals on the page or the number of instructions on the page.
PH	A conditional assembly bracket is still in effect at the end of the input stream. This is caused by nonmatching < and > characters in the source file.
RD	Redefinition. A permanent symbol has been defined with =. The new and old definitions do not match. The redefinition is allowed.
SE	Symbol table exceeded. Too many symbols have been defined for the amount of memory available. Fatal error: assembly cannot continue.
UO	Undefined origin. An undefined symbol has occurred in an origin statement.
US	Undefined symbol. A symbol has been processed during pass 2 that was not defined before the end of pass 1.
ZE	Page 0 exceeded. This is the same as PE except occurs in page 0.

5.15 PAL8 PERMANENT SYMBOL TABLE

The following mnemonics represent the central processor's instruction set found in the permanent symbol table within the PAL8 Assembler. For additional information on these instructions, refer to the DECstation 78 User's Guide.

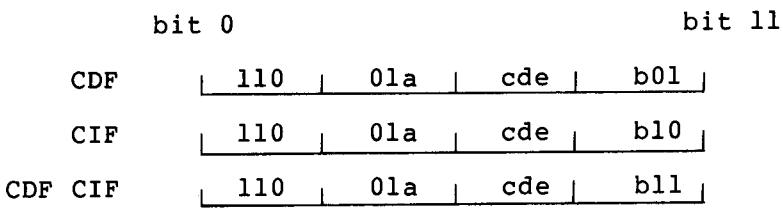
Mnemonic	Code	Operation
Memory Reference Instructions		
AND	0000	Logical AND
TAD	1000	Two's complement add
ISZ	2000	Increment and skip if zero
DCA	3000	Deposit and clear AC
JMS	4000	Jump to subroutine
JMP	5000	Jump
IOT	6000	In/Out transfer
OPR	7000	Operate
Group 1 Operate Microinstructions		
NOP	7000	No operation
IAC	7001	Increment AC
BSW	7002	Byte swap
RAL	7004	Rotate AC and Link left one
RTL	7006	Rotate AC and Link left two
RAR	7010	Rotate AC and Link right one
RTR	7012	Rotate AC and Link right two
CML	7020	Complement the link
CMA	7040	Complement the AC
CLL	7100	Clear Link
CLA	7200	Clear AC
Group 2 Operate Microinstructions (1 cycle)		
HLT	7402	Halts the computer
SKP	7410	Skip unconditionally
SNL	7420	Skip on nonzero Link
SZL	7430	Skip on zero Link
SZA	7440	Skip on zero AC
SNA	7450	Skip on nonzero AC
SMA	7500	Skip on minus AC
SPA	7510	Skip on positive AC (zero is positive)
Group 3 Operate Microinstructions		
MQL	7421	Load MQ, clear AC
MQA	7501	MQ OR into AC
SWP	7521	Swap AC and MQ
Combined Operate Microinstructions		
CIA	7041	Complement and increment AC
STL	7120	Set Link to 1
GLK	7204	Get Link (put Link in AC, bit 11)
STA	7240	SET AC to -1 (7777)
Internal IOT Microinstructions		
SKON	6000	Skip with interrupts on and turn them off
ION	6001	Turn interrupt facility on
IOF	6002	Turn interrupt facility off
GTF	6004	Get flags
RTF	6005	Restore flag, ION
CAF	6007	Clear all flags

Memory Extension IOT Instructions

CDF	62n1	Change to Data Field n (n=00 to 07)
	62n5	Change to Data Field n (n=10 to 17)
	63n1	Change to Data Field n (n=20 to 27)
	63n5	Change to Data Field n (n=30 to 37)
CIF *	62n2	Change to Instruction Field n (n=00 to 07)
	62n6	Change to Instruction Field n (n=10 to 17)
	62n2	Change to Instruction Field n (n=20 to 27)
	63n6	Change to Instruction Field n (n=30 to 37)
CDI	62n3	Change to Data and Instruction Fields n (n=00 to 07)
	62n7	Change to Data and Instruction Fields n (n=10 to 17)
	63n3	Change to Data and Instruction Fields n (n=20 to 27)
	63n7	Change to Data and Instruction Fields n (n=30 to 37)
CDF CIF	62n3	Combine CDF and CIF

Bit assignments for n are as follows.

Field Number (0-37) = abcde



RDF	6214	Read (OR) the Data Field into bits 6-8 of AC
RIF	6224	Read (OR) the Instruction Field into bits 6-8 of AC
RSB	6234	Read Instruction Save Field into bits 7-8 and Data Save Field into bits 10-11 of AC
RMF	6244	Restore memory fields to state prior to last interrupt by loading the Data Save Field into DF register and the Instruction Save Field into the IB register and inhibiting interrupts. At the next JMP or JMS, IB is loaded into the IF register and the interrupt inhibit is removed.

* Executed at next JMP or JMS instruction.

Keyboard (1 cycle)

KCF	6030	Clear keyboard flag
KSF	6031	Skip on keyboard flag
KCC	6032	Clear keyboard flag and AC
KRS	6034	OR keyboard buffer into AC
KIE	6035	Set/clear interrupt enable
KRB	6036	Clear AC, read keyboard buffer Clear keyboard flag

Terminal (1 cycle)

TSF	6041	Skip on terminal flag
TCF	6042	Clear terminal flag
TSK	6045	Skip on keyboard or terminal flag
TLS	6046	Load terminal, display character, and clear terminal flag

CHAPTER 7

FORTRAN IV

7.1 OVERVIEW

The FORTRAN (or FORmula TRANSlator) programming language enables you to express mathematical operations in a form similar to standard mathematical notation as well as perform a variety of such applications as process control, information retrieval, and commercial data processing.

This chapter describes the components of the FORTRAN system--the Compiler, Loader, Run-Time System, and Library--and the elements of the language.

OS/78 FORTRAN IV conforms generally to the specifications for American National Standard FORTRAN. Several enhancements have been added, and these are described in Section 7.2.

To create and run a FORTRAN program, use the following procedure.

1. Write the program in source code, using the statements and other features of the FORTRAN language as they are described in this chapter. Divide the program into logical units--one for the main program and one for each subroutine you need. Use the OS/78 Editor to create a file for each program unit.

2. Use the COMPILE command to compile and assemble the FORTRAN source program. The Compiler (which chains automatically to the assembler) accepts one program unit--a main program or a subroutine--and produces one module of relocatable code; that is, assembled code which has not yet been assigned permanent addresses in memory. The Compiler also produces an optional listing file. (For complete details on the FORTRAN Compiler, see Section 7.1.1.)

3. Use the LOAD command to link the program units and assign permanent memory locations to the complete program. The FORTRAN Loader accepts up to 128 modules of relocatable code. It links subroutines to the main program, assigns permanent addresses, and produces a loader-image file, which contains the complete program in linked and relocated form. In addition, it determines if the program requires any functions from the FORTRAN library and copies them in relocated form into the loader-image file. The Loader also produces an optional loader symbol map, showing the areas in memory that the program will use. (For details on the Loader, see Section 7.1.2.)

4. Load the program into memory and run it with the EXECUTE command. The EXECUTE command summons the Run-Time System, which accepts a loader-image file, places it in memory, determines the I/O requirements of the program, and starts execution. (For details on the Run-Time System, see Section 7.1.3.)

The Compiler and Loader provide options that enable you to compile, load, and run a program with a single command. In addition, the EXECUTE command accepts source code or relocatable code, performs the necessary operations, and causes execution.

For example, assume you have written a short program called POWER that you want to enter as a file on your system device and run. To enter the file, summon the Editor with the CREATE command, naming the file in the command line and adding the extension FT, the standard OS/78 extension for a file containing a FORTRAN source program.

```
.CREATE POWER.FT
```

As soon as the Editor displays its prompt (#) to indicate that it is ready to receive your first instruction, type I (to put the Editor into text mode) and a carriage return and enter the program.

```
#I
 C    FORTRAN DEMONSTRATION
 C    COMPUTE AND PRINT POWERS OF TWO
      DIMENSION A(16)
      WRITE(4,15)
15    FORMAT(1H ,'POWER OF TWO')
      DO  20 N=1,16
      A(N)=2.**N
20    CONTINUE
      WRITE (4,25) (N,A(N),N=1,16)
25    FORMAT(1H ,'2** ',I2,'=',F10.1)
      STOP
      END
```

Now type CTRL/L to put the Editor in command mode and issue the EXIT instruction. The Editor will write file POWER.FT on your system device and return control to the monitor.

To compile and assemble the source program, use the COMPILE command, entering POWER.FT as the input file. If you wish to generate an annotated listing file and send it to the lineprinter, enter LPT: as the second output specification.

```
.COMPILE POWER,LPT:<POWERS.FT
```

The Compiler will send the binary file to SYS: (which it uses as a default device), adding an RL extension to indicate that the file contains relocatable FORTRAN code, and produce the following annotated listing on the lineprinter.

```
        C       FORTRAN DEMONSTRATION
        C       COMPUTE AND PRINT POWERS OF TWO
0002            DIMENSION A(16)
0003            WRITE (4,15)
0004    15      FORMAT(1H ,'POWER OF TWO')
0005            DO 20 N=1,16
0006            A(N)=2.**N
0007    20      CONTINUE
0010            WRITE (4,25) (N,A(N),N=1,16)
0011    25      FORMAT(1H ,'2** ',I2,'=',F10.1)
0012            STOP
0013            END
```

To create a loader-image file--containing the program in absolute binary form and a copy of the exponentiation library function--use the LOAD command, specifying the RL file as input. To send a loader-symbol map--showing the areas of memory the program uses--to the lineprinter, enter LPT: as the second output specification.

```
.LOAD POWER,LPT:<POWER.RL
```

The Load command creates a loader-image file (adding the LD extension) on SYS and the following map.

```
LOADER V24A   17-APR-79

SYMBOL VALUE LVL OVLY

ARGERR 00204  0  00
EXIT   00223  0  00
#MAIN  10000  0  00
 11000  = 1ST FREE LOCATION


LVL OVLY LENGTH

 0  00   10601
```

The optional loader symbol map lists all symbols defined in the loader image file. The LVL and OVLY entries apply only to OS/8, a superset of OS/78 FORTRAN.

Following the alphabetical list of symbols, the loader prints the address of the first free memory location and the length, in octal words. This information is useful for optimizing memory requirements.

To execute this program, call the Run-Time System with the EXECUTE command, specifying POWER.LD for input.

```
.EXECUTE POWER.LD
```

The program output will appear on the terminal screen.

```
 POWER OF TWO
2**  1=        2.0
2**  2=        4.0
2**  3=        8.0
2**  4=       16.0
2**  5=       32.0
2**  6=       64.0
2**  7=      128.0
2**  8=      256.0
2**  9=      512.0
2** 10=     1024.0
2** 11=     2048.0
2** 12=     4096.0
2** 13=     8192.0
2** 14=    16384.0
2** 15=    32768.0
2** 16=    65536.0
```

FORTRAN programs are usually saved as loader-image files and then run with the EXECUTE command. To produce a loader-image file with a single command--that is, to instruct the Compiler/Assembler to chain automatically to the Loader--use the Compiler /L option.

```
COMPILE POWER.FT/L
```

The FORTRAN system consists of the following components: the Compiler (plus Assembler), the Loader, the Run-Time system, and the FORTRAN library of functions. These components are described fully in Sections 7.1.1 through 7.1.4.

7.1.1 The COMPILER

The OS/78 FORTRAN IV Compiler/Assembler accepts one FORTRAN source language program or subroutine as input, examines each FORTRAN statement for validity, and produces a list of error messages plus a relocatable assembly-language version of the source program, along with an optional annotated source listing, as output.

If your program contains one or more subroutines, compile and assemble the main program and each subroutine separately, then use the Loader to link them together.

The FORTRAN Compiler terminates a compilation by chaining automatically to the Assembler.

To summon the FORTRAN compiler, use the COMPILE command. The Compiler accepts 1 to 3 output file specifications and 1 to 9 files for input, along with several run-time options. If you omit the device name, the Compiler assumes SYS. If you omit the extensions on the output filenames, the Compiler adds .RL and .LS to the binary file and the listing.

The format is

```
COMPILE out:file.RL,out:file.LS,out:file.MP<in:file1.FT...,file9.F
```

where:

file.RL	is the relocatable binary code
file.LS	is an optional listing file
file.MP	is an optional loader symbol map (to obtain it, you must chain to the Loader with the /L option)
file1.FT...9	is a single program unit--a main program or a subroutine--written as 1 to 9 files

The Compiler assigns an internal statement number (ISN) to each FORTRAN IV statement sequentially, in octal notation, starting with ISN 2 at the first FORTRAN statement. When the Compiler encounters an error during compilation, it prints a 2-character error code, followed by the ISN of the offending statement, on the terminal. An extended error message is printed below every erroneous statement in the annotated listing. Certain errors causes the Compiler to return immediately to the monitor, thereby preventing the output of a listing file.

7.1.1.1 **Compiler Options** - Compiler options are described in Table 7-1. If you chain automatically to the Loader (with the /L option) you can also add Loader options to the command line.

Table 7-1
FORTRAN IV Compiler Run-Time Options

Option	Operation
/G	Chain to the loader when assembly is complete and chain to the run-time system following creation of a loader image file (equivalent to the EXECUTE command) (see Section 7.1.3).
/L	Chain to the loader when assembly is complete to create a loader-image file. If the /L option is not specified, the system will return to the monitor upon completion.
/N	Suppress compilation of ISNs. This option reduces program memory requirements by two words per executable statement; however, it also prevents full error traceback at run time.
/Q	Optimize cross-statement subscripting during compilation. This option should not be requested when any variable that appears in a subscript is modified either by referencing a variable equivalent to it or via a SUBROUTINE or FUNCTION call (whether as an argument or through COMMON).

7.1.1.2 **Compiler Error Messages** - During pass 2, the Compiler displays error messages on the terminal as a 2-character message followed by the ISN of the erroneous statement. To suppress the messages, type CTRL/O at the terminal. If you request a listing, it will contain an extended error message following the erroneous statement. Except as noted, errors located by the Compiler do not halt processing. Error messages are described in Table 7-2.

Table 7-2
Compiler Error Messages

Error Code	Meaning
AA	More than six subroutine arguments are arrays.
AS	Bad ASSIGN statement.
BD	Bad dimensions (too big or syntax) in DIMENSION, COMMON or type declaration.
BS	Illegal in BLOCK DATA Program.
CL	Bad COMPLEX literal.
CO	Syntax error in COMMON statement.
DA	Bad syntax in DATA statement.
DE	This type of statement illegal as end of DO loop (that is, GO TO, another DO).

(continued on next page)

Table 7-2 (Cont.)
Compiler Error Messages

Error Code	Meaning
DF	Bad DEFINE FILE statement.
DH	Hollerith field error in DATA statement.
DL	DATA list and variable list are not same length.
DN	DO-end missing or incorrectly nested. This message is not printed during pass 3, if it is followed by the statement number of the erroneous statement, rather than the ISN.
DO	Syntax error in DO or implied DO.
DP	DO loop parameter not integer or real.
EX	Syntax error in EXTERNAL statement.
GT	Syntax error in GO TO statement.
GV	Assigned or computed GO TO variable must be integer or real.
HO	Hollerith field error.
IE	Error reading input file. Control returns to the monitor.
IF	Logical IF statement cannot be used with DO, DATA, INTEGER, etc.
LI	Argument of logical IF is not type Logical.
LT	Input line too long, too many continuations.
MK	Misspelled keyword.
ML	Multiply defined line number.
MM	Mismatched parenthesis.
MO	Expected operand is missing.
MT	Mixed variable types (other than integer and real).
OF	Error writing output file. Control returns to the monitor.
OP	Illegal operator.
OT	Type / operator use illegal (for example, A.AND.B where A and/or B not typed Logical).
PD	Compiler stack overflow; statement too big and/or too many nested loops.
PH	Bad program header line.
QL	Nesting error in EQUIVALENCE statement.
QS	Syntax error in EQUIVALENCE statement.
RD	Attempt to redefine the dimensions of a variable.
RT	Attempt to redefine the type of a variable.
RW	Syntax error in READ/WRITE statement.
SF	Bad arithmetic statement function.
SN	Illegal subroutine name in CALL.
SS	Error in subscript expression, that is, wrong number, syntax.
ST	Compiler symbol table full, program too big. Causes an immediate return to the monitor.
SY	System error, that is, PASS2O.SV or PASS2.SV missing, or no room on system for output file. Causes an immediate return to the monitor.
TD	Bad syntax in type declaration statement.
US	Undefined statement number. This message is not printed during pass 3. It is followed by the statement number of the erroneous statement, rather than the ISN.
VE	Version error. One of the compiler programs is absent from SYS or is present in the wrong version.

7.1.2 The LOADER

The FORTRAN IV Loader accepts up to 128 relocatable binary modules as input. It links all modules, including any routines from the FORTRAN library that the program may require, to form a loader-image file of the complete program--that is, an image of the binary code with absolute addresses assigned. This loader-image file can be passed directly to the Run-time system, which loads it into memory and executes it.

In addition to the loader-image file, the Loader generates an optional symbol map, which shows the areas the program occupies in memory.

You can call the Loader automatically by adding the /L option to the COMPILE command line.

The format of the Load command is

```
LOAD out:file.LD,out:map.LS<in:file1.RL...,file9.RL
```

where:

file.LD	is a loader image file
map.LS	is an optional symbol map
file1...,9.RL	are relocatable binary modules

OS/78 permits a maximum of nine input files in a command line. If you wish to specify more than nine relocatable modules for input, use the Loader /C (continue) option. This option enables you to add additional file names on the following line.

7.1.2.1 **Specifying I/O Devices** - The LOAD /G option causes the Loader to chain directly to the FORTRAN Run-Time System. If you use the /G option and terminate the LOAD command with the RETURN key, execution begins immediately. If you use the /G option and terminate the command with the ESCAPE key, the system calls a special program called the Command Decoder (described in Appendix D). The Command Decoder prints an asterisk (*) to indicate that it is ready to accept your special device and file I/O specifications. This feature makes it possible for you to change the devices used in a FORTRAN program, making the program device-independent.

For example, if the POWER.FT program were written with 3 as the output unit designation, that is,

```
WRITE (3,15)
```

the output generated by program execution would be sent to the line printer. However, if your system does not have a line printer available, you could change the output unit designation by typing

```
.LOAD POWER.LD<POWER.RL/G (ESC)
```

Press the ESCape key (which echoes as a dollar sign) to call up the Command Decoder, which prompts with an asterisk. Then type /3=4 which assigns I/O unit 3 to the console terminal instead of the line printer. Pressing the ESCape key again executes the program and program output is sent to the terminal. The command line will appear as follows:

```
.LOAD POWER.LD<POWER.RL/G (ESC) */3=4 (ESC)
```

The Command Decoder program also allows you to store the output of an executed program in a file that has not been created at load time.

For example,

```
.LOAD POWER.LD<POWER.RL/G (ESC) *RXA1:HOLD.TM</4 (ESC)
```

will output the results of the program into a file called HOLD.TM on RXA1. Typing

```
.TYPE RXA1:HOLD.TM
```

will display the contents of this file, that is, the results of the program POWER.FT on the terminal screen.

For further information on I/O specifications at run-time, see the FORTRAN compiler.

7.1.2.2 **Running Subprograms** - The LOAD command is especially useful to link subprograms. Consider the program shown in Figure 7-1, which computes the volume of a regular polyhedron when given the number of faces and the length of one edge. It consists of a main program and a subroutine. The subprogram does the required computation, using a computed GO TO statement to determine whether the polyhedron is a tetrahedron, cube, octahedron, dodecahedron, or icosahedron, then transfers control to the proper arithmetic expression for performing the calculation.

MAIN PROGRAM:

```
C       COMPUTE THE VOLUME OF A REGULAR POLYHEDRON GIVEN
C       THE NUMBER OF FACES AND LENGTH OF ONE EDGE
        COMMON NFACES,EDGE
5       WRITE(4,10)
10      FORMAT(1H ,'TYPE IN NO.OF FACES AND ONE LENGTH EDGE')
        READ(4,20) NFACES,EDGE
20      FORMAT(I2,F8.5)
        CALL SOLVE
        GO TO 5
        STOP
        END
```

Figure 7-1 Main Program and Subprogram for Calculating the Volume of a Regular Polyhedron

SUBPROGRAM SOLVE:

```
C       SUBROUTINE TO SOLVE PROBLEM AND PRINT ANSWER
C       CALLED SOLVE.FT
        SUBROUTINE SOLVE
        COMMON NFACES,EDGE,VOLUME
        IF(NFACES.GT.20)GOTO 8
        CUBED=EDGE**3
GO TO(6,6,6,1,6,2,6,3,6,6,6,4,6,6,6,6,6,6,6,5,6),NFACES
1       VOLUME=CUBED*0.11785
        GO TO 20
2       VOLUME=CUBED
        GO TO 20
3       VOLUME=CUBED*0.47140
        GO TO 20
4       VOLUME=CUBED*7.66312
        GO TO 20
5       VOLUME=CUBED*2.18170
        GO TO 20
6       WRITE(4,10) NFACES
10      FORMAT(1HO,'NO REGULAR POLYHEDRON HAS',I3,1X,'FACES')
        RETURN
20      WRITE(4,30) VOLUME
30      FORMAT(1HO,'VOLUME=',F10.2)
        RETURN
8       WRITE(4,10)
40      FORMAT (1HO,"DO NOT SPECIFY MORE THAN 20 FACES)
        RETURN
        END
```

Figure 7-1 (Cont.) Main Program and Subprogram for Calculating the Volume of a Regular Polyhedron

If the number of faces of the solid is other than 4, 6, 8, 12, or 20, or if more than 20 faces are specified, the subroutine displays an error message on the terminal. If the correct input parameters are typed in at the terminal keyboard, the calculation is performed, and the answer is displayed on the terminal screen.

When subprograms are used, the main program and the subprograms must be individually compiled. For the example program, both the main program and subprogram are compiled as follows:

```
.COMPILE MAIN.FT
```

and

```
.COMPILE SOLVE.FT
```

which create relocatable binary files.

The modules must now be loaded to make a single image file that can be executed. This is done as follows:

```
.LOAD POLY.LD<MAIN.RL,SOLVE.RL
```

This command creates a file POLY.LD and returns to the monitor. Execute the program by typing

```
.EXECUTE POLY.LD
```

which results in the program calling for the required input parameters as follows:

```
TYPE IN NO. OF FACES AND ONE LENGTH EDGE
```

Typing a "6" to represent a cube (preceded by a space since the field specification is I2) and "20" as one length edge in accordance with the field specification requirement as follows:

```
b6 20.0
```

results in the answer being displayed on the terminal screen, and a prompt for the next set of input parameters as follows:

```
VOLUME = 8000.00
TYPE IN NO. OF FACES AND ONE LENGTH EDGE
```

Do not specify blanks or zeroes for the NFACES and EDGES variables. Type CTRL/C to return to the monitor.

7.1.2.3 **LOADER Options** - Loader options are described in Table 7-3. The examples following the table illustrate their use.

Table 7-3
Loader Run-Time Options

Option	Operation
/C	Continue the current line of input on the next line of input. When specifying input files to the loader, there may be more than nine files in a command line. The /C option permits the additional files to be put on the following line. Terminate each continuation line (except the last) with a RETURN. Terminate the last line with an ESC.
/G	Treat the current line as the last line of input, and chain to the FORTRAN IV run-time system when finished (see Section 7.1.3).
/S	Include system symbols in the loader symbol map. System symbols are identified by an initial number sign (#). This option is only valid when a symbol map output file was specifically defined.

Examples:

```
.LOAD POWER.RL
```
Loads POWER.RL and produces a loader image file POWER.LD.

```
.LOAD RXA1:POWER.RL/G
```
Loads POWER.RL, produces a loader image file POWER.LD, chains to the run-time system, and executes the program.

```
.LOAD PROG.LD,RLOB:MAP<MAIN.RL,SUBA.RL
```
Loads and links MAIN.RL and SUBA.RL to produce PROG.LD, and produces MAP.LS, a loader symbol map output file, on disk RLOB. Using the type command

```
.TYPE RLOB:MAP
```

will display the MAP file on the screen. Typing .EXE PROG.LD will execute the program.

```
.LOAD PROG.LD<PROG.RL,PROG1.RL(ESC)*(more file specifications)(ESC)
```
Loads and links two input files to produce PROG.LD, then calls a special system program for accepting file I/O specifications (see Section 7.2.3). The second ESCape executes the entire command.

7.1.2.4 **Loader Error Messages** - The Loader displays error messages on the terminal during generation of a loader-image file. Except where indicated in Table 7-4, Loader errors are fatal. When it encounters a fatal error condition, the Loader returns control to the monitor.

Table 7-4
Loader Error Messages

Error Message	Meaning
BAD INPUT FILE	An input file was not a relocatable binary module.
BAD OUTPUT DEVICE	The loader image file device was not a directory device, or the symbol map file device was a read-only device. The entire line is ignored.
EX	The symbol is referenced but not defined.
INVALID OS/78 OPTION	Attempt to use an option that is not available.
ME	Multiple entry. The symbol is multiply defined.
MIXED INPUT	The L option was specified on a line that contained some file other than a library file. The library file (if any) is accepted. Any other input file specification is ignored.

(continued on next page)

Table 7-4 (Cont.)
Loader Error Messages

Error Messages	Meaning
NO MAIN	No relocatable binary module contained the section #MAIN.
OVER CORE	The loader image requires more than 16K of memory.
OVER IMAG	Output file overflow in the loader image file.
OVER SYMB	Symbol table overflow. More than 253 (decimal) symbols in one FORTRAN job.
TOO MANY RALF FILES	More than 128 input files were specified.

7.1.3 The Run-Time System (FRTS)

The run-time system reads, loads, and executes a loader image file produced by the loader. It evaluates arithmetic and logical operations. It also configures a software I/O interface between the FORTRAN IV program and the OS/78 operating system, and then monitors program execution to direct I/O processes and identifies certain types of run-time errors.

The run-time system is automatically called to load and execute the loader image file produced by the loader whenever the /G option is specified to the loader. When chained to the loader, the run-time system reacts in one of two ways. If the LOAD command line was terminated by pressing the RETURN key, the program is executed. If the LOAD command line was terminated with an ESCape, a system program called the Command Decoder is called and indicates that it is running by printing an asterisk (*).

In response to the asterisk, you can supply file specifications to the run-time system. This allows a source program to be written that refers to I/O devices as integer constants or variables. Such a program may be compiled, assembled, and loaded into an image file. This image file may be run any number of times, each time specifying different actual peripheral devices. Thus logical unit 8 may refer in one run to the console terminal and in another run to a diskette file.

Of the nine I/O unit numbers available under FORTRAN IV, two are initially assigned to FORTRAN internal device handlers by the system:

I/O Unit	Internal Handler	Comments
3	Line printer	LA78 only.
4	Console terminal	Double buffered output, single character input.

The FORTRAN internal handlers listed above are not the same as the OS/78 device handlers. The FORTRAN internal handlers are designed for ASCII text only and will not transfer noncharacter data.

Additional unit numbers may be assigned, in addition to those listed above, to the FORTRAN internal device handlers by typing (in response to the asterisk generated by the Command Decoder):

```
/n=m
```

where:

n — is a different unit number (1 to 9) that is also to be assigned to that internal handler; and

m — is the I/O unit number (3 or 4) of the internal handlers.

This specification causes all program references to logical unit n to perform I/O to device m. For example,

/6=3 — Assigns the FORTRAN internal line printer handler as I/O unit number 6, in addition to unit number 3.

/3=4 — Assigns I/O unit number 3 to the FORTRAN console terminal handler instead of the internal line printer handler.

I/O unit numbers may be assigned to OS/78 device handlers for nondirectory devices by typing (in response to the asterisk generated by the Command Decoder):

```
dev:/n
```

where:

dev: — is the standard or assigned designation for any supported nondirectory device; and

n — is an I/O unit number (1 to 9).

Example:

LQP:/3 — Specifies the OS/78 LQP line printer handler to be used as device #3 instead of the FORTRAN internal line printer handler.

Existing directory device files may be assigned I/O unit numbers by typing (in response to the asterisk generated by the Command Decoder):

```
dev:file.ex/n
```

where:

dev:file.ex — is the standard OS/78 designation for an existing directory device file; and

n — is an I/O unit number (1 to 9).

Example:

RXA1:FORIO.TM/4 Assigns unit number 4 to Diskette file FORIO.TM rather than to the FORTRAN internal console terminal handler, where FORIO.TM is an existing file on Diskette unit 1.

A directory device file that does not presently exist may be assigned a FORTRAN I/O unit number in the same manner by entering it as an output file on the specification line; however, only one such file may be created on any particular device. For example:

FORIO.TM</9 Assigns unit number 9 to file DSK:FORIO.TM, which has not been created at load time.

In any case, only one device or file specification is permitted on each line, and no more than six directory device files may be created by the FORTRAN program. Excess files after the sixth are accepted and written, but they will not be closed. If a file created by the program has the same file name and extension as a pre-existing file, the old file is automatically deleted when the new file is closed.

The Command Decoder "[n]" specification may be used to optimize storage allocation when assigning files that do not yet exist, where n is a decimal number that indicates the maximum expected length of the file, in blocks.

Each time a run-time I/O specification is terminated by pressing the RETURN key, the Command Decoder is recalled to accept another specification. When a specification is terminated by the ESCape key, the program is run.

The following examples illustrate the use of device and file specifications.

```
C     WRITE.FT PROGRAM
      DIMENSION ILT(200)
      INTEGER ILT
C     SPECIFY LOGICAL UNIT NUMBER FOR TTY AS 2 INSTEAD OF 4
1     WRITE (2,10)
10    FORMAT (1X,'WRITING AND READING ASCII SEQ. DATA FILE')
      DO 3 I=1,200
3     ILT(I)=I**2
      WRITE (8,20) (ILT(I),I=1,200)
20    FORMAT (1H,//,20(10I7,/))
      REWIND 8
C     NOW READ DATA FILE DATA.DA FROM MASS STORAGE DEVICE
      READ (8,20) ILT
C     OUTPUT FILE TO TTY
      WRITE (4,20) ILT
      END
```

The above program raises 200 sequential numbers to the power of two. The following sequence of specifications are typed using the COMPILE command as follows:

```
.COMPILE WRITE.FT/G (ESC) */2=4
*RXA1:DATA.DA</8 (ESC)
```

The /G option in the command line will call the run-time system to execute the program. Pressing the ESCape key after the command calls the Command Decoder, allowing device and file specifications to be declared. In this case, the terminal (4) is assigned to unit number 2 while the load image file is assigned number 8. A file DATA.DA is created on diskette 1 that contains the results of the program. The contents of this file are then displayed on the terminal screen. The command and the specification strings are executed by the second ESCape.

The second example shows an output file RAY.DA being created on the diskette by PROG1.FT, and then being read from the diskette by PROG2.FT.

```
C     THIS PROGRAM WRITES A RECORD OF 400 VARIABLES
C     INTO A FILE CALLED RAY.DA
      DIMENSION RAY(400)
      INTEGER RAY
      DEFINE FILE 1 (1,400,U,J)
      J=1
      DO 5 I=1,400
5     RAY(I)=2*I
      WRITE (1/J) (RAY(I),I=1,400)
      CALL EXIT
      END
```

The above program writes a record into file RAY.DA. The following command and specification strings are typed to accomplish this.

```
.COMPILE PROG1.FT/G (ESC) *RAY.DA</1 (ESC)
```

```
C     READ DIRECT ACCESS FILE RAY.DA FROM MASS STORAGE
C     DEVICE CREATED BY PREVIOUS PROGRAM
      DIMENSION RAY(400)
      INTEGER RAY DEFINE FILE 1(1,400,U,J)
      J=1
      READ (1'J) (RAY(I),I=1,400)
100   FORMAT (1H,//,40(10I6,/))
C     DUMP CONTENTS OF DATA FILE ONTO TTY
      WRITE (4,100) RAY
      CALL EXIT
      END
```

The above program then reads out the results of PROG1.FT, and displays the contents of file RAY.DA on the terminal screen. This is done by typing the following command line.

```
.COMPILE PROG2.FT/G (ESC) *RAY.DA/1 (ESC)
```

Although existing files are specified as though they were input files and nonexistent files are specified as though they were output files, any file that has been assigned a unit number may be used for either input or output.

7.1.3.1 **Runtime System Options** - Run-time system option specifications are described in Table 7-5.

Table 7-5
Run-Time System Options

Option	Operation
/C	Carriage control switch. The first character on every output line is processed as a carriage control character by all FORTRAN internal handlers and also by the OS/78 handlers TTY and LPT. The first character on every output line is processed as data, in the same manner as any other character, by all OS/78 handlers except TTY and LPT. Entering a C option specification on the command line that assigns an I/O unit number to a particular handler reverses the processing of carriage control characters for that device. Thus, TEMP(2C) assigns file DSK:TEMP. as I/O unit 2. The /C option causes the first character of every output line to be processed as a carriage control character. If C were not specified, these characters would be processed as data. /C/6=3 assigns the FORTRAN internal line printer handler as I/O unit 6, as well as unit 3. The first character of every line will be processed as a carriage control character on unit 3, and as a character of data on unit 6.
/E	Ignore the following run-time system errors, any of which indicates that an error was detected earlier in the compilation loading process: 1. Illegal subroutine call. 2. Reference to an undefined symbol.

The console terminal serves as FORTRAN I/O unit 4 for both input and output. Terminal input is automatically echoed on the console screen. In addition, the run-time system monitors the keyboard continually during execution of a FORTRAN program. Typing CTRL/C at any time causes an immediate return to the monitor. Typing CTRL/B branches to the system traceback routine, and then exits to the monitor. This traceback routine generates a printout, similar to the error traceback, including the current subroutine, the line number in the next higher level subroutine from which it was called, and so forth, to the main program. This facilitates locating infinite loops when debugging a program. The following additional special characters are recognized by the console terminal handler and processed as shown:

DELETE Deletes last character accepted.

CTRL/U Deletes current line of input.

CTRL/Z Signals end-of-file on input.

Tentative output files (that is, files created by the FORTRAN program) are closed automatically upon successful completion of program execution provided that one of the following conditions occurs:

1. An END FILE statement referencing the file was executed. FRTS assigns a file length equal to the actual length of the file.

2. The last operation performed on the file was a write operation. FRTS proceeds as though an END FILE statement had been executed.

3. A DEFINE FILE statement referencing the file was executed but an END FILE statement was not executed. Upon completion of program execution, FRTS assigns a file length equal to the length specified in the DEFINE FILE statement.

Execution of a REWIND statement does not close a tentative file, nor does it modify the tentative file length.

7.1.3.2 **Run-Time System Error Messages** - The run-time system generates two classes of error messages. Messages listed in Table 7-6 identify errors that may occur during execution of a FORTRAN program and errors that may be encountered when the run-time system is reading a loader image file into memory in preparation for execution, or accepting I/O unit specifications. Except where indicated, all run-time system errors cause full traceback and an immediate return to the monitor. Nonfatal errors cause partial traceback, sufficient to locate the error, and execution continues.

The run-time system error traceback feature provides automatic printout of statement numbers ISNs corresponding to the sequence of executable statements that terminated in an error condition. At least one statement number is always printed. This number identifies the erroneous statement or, in certain cases, the last correct statement executed prior to the error. When a statement was compiled under the /N option, however, the system cannot generate meaningful statement numbers during traceback. When a statement is reached through any form of GOTO, the line number for error traceback is not reset. Thus, an error in such a line will give the number of the last executed line in the error traceback.

Table 7-6
Run-Time System Error Messages

Error Message	Meaning
BAD ARG	Illegal argument to library function.
CAN'T READ IT!	I/O error on reading loader image file.
D.F. TOO BIG	Random access file requirements exceed available storage.
DIVIDE BY 0	Attempt to divide by zero. The resulting quotient is set to zero and execution continues.

(continued on next page)

Table 7-6 (Cont.)
Run-Time System Error Messages

Error Messages	Meaning
EOF ERROR	End of file encountered on input.
FILE ERROR	Any of the following conditions occurred: 1. A file specified as an existing file was not found. 2. A file specified as a nonexistent file would not fit on the designated device. 3. More than one nonexistent file was specified on a single device. 4. The file specification contained an asterisk (*) as name or extension.
FILE OVERFLOW	Attempt to write outside file boundaries.
FORMAT ERROR	Illegal syntax in FORMAT statement.
INPUT ERROR	Illegal character received as input.
I/O ERROR	Error reading or writing a file, tried to read from an output device, or tried to write on an input device.
MORE CORE REQUIRED	The space required for the program, the I/O device handlers, the I/O buffers, and the monitor exceeds the available memory.
NO DEFINE FILE	Direct access I/O attempted without a DEFINE FILE statement.
NO NUMERIC SWITCH	The referenced FORTRAN I/O unit was not specified to the run-time system.
NOT A LOADER IMAGE	The first input file specified to the run-time system was not a loader image file.
OVERFLOW	Result of a computation exceeds upper bound for that class of variable. The result is set equal to zero and execution continues.
PARENS TOO DEEP	Parentheses nested too deeply in FORMAT statement.
SYSTEM DEVICE ERROR	I/O failure on the system device.
TOO MANY HANDLERS	Too many I/O device handlers are resident in memory, or files have been defined on too many devices.
UNIT ERROR	I/O unit not assigned, or incapable of executing the requested operation.
UNKNOWN INTERRUPT	A hardware interrupt occurred from a device that the run-time system is not using.
USER ERROR	Illegal subroutine call, or call to undefined subroutine. Execution continues only if the E option was requested.

7.1.4 The Library

The OS/78 FORTRAN IV system contains a general purpose FORTRAN library named FORLIB.RL. FORLIB.RL contains mathematical functions and miscellaneous subroutines.

Library functions and subroutines are called in the same manner as user written functions and subroutines. Section lists the library components that are available to FORTRAN programs and illustrates calling sequences, where necessary.

7.2 THE FORTRAN IV SOURCE LANGUAGE

A FORTRAN source program consists of statements using the language elements and the syntax described in this chapter. A statement performs one of the following functions:

- Causes operations such as multiplication, division, and branching to be carried out
- Specifies the type and format of data being processed
- Specifies the characteristics of the source program

FORTRAN statements are composed of keywords (that is, words that the FORTRAN compiler recognizes) that you use with elements of the language set. These elements are constants, variables and expressions. There are two basic types of FORTRAN statements: executable and nonexecutable.

Executable statements specify the action of the program; nonexecutable statements describe the characteristics and arrangement of data, editing information, statement functions, and subprograms that you may include in the program. The compilation of executable statements results in the creation of executable code. Nonexecutable statements provide information only to the compiler; they do not create executable code.

The OS/78 FORTRAN IV language generally conforms to the specifications for American National Standard FORTRAN X3.9-1966. The following enhancements are included in OS/78 FORTRAN:

- You may use any arithmetic expression as an array subscript. If the expression is not of integer type, FORTRAN converts it to integer form.
- You may use alphanumeric literals (character strings delimited by apostrophes or quotation marks) in place of Hollerith constants.
- The statement label list in an ASSIGNed GO TO statement is optional.
- The following Input/Output (I/O) statements have been added:

DEFINE FILE	Device-oriented I/O
READ (u'r) WRITE (u'r)	Unformatted Direct Access I/O

- You may use any arithmetic expression as the initial value, increment, or limit-parameter in the DO statement, or as the control parameter in the COMPUTED GO TO statement.

- OS/78 FORTRAN permits constants and expressions in the I/O lists of WRITE statements.

All FORTRAN statements are listed in Section 7.13.

In this chapter, the FORTRAN language statements are grouped into eight categories, each of which is described in a separate section. The name, definition, and section references for each statement category are given in Table 7-7.

Table 7-7
FORTRAN Statement Categories

Category	Function	Section
Assignment Statement	Assign values to named variables and array elements.	7.5
Specification Statement	Declare the properties of variables, arrays, and functions.	7.6
DATA Statements	Assign initial values to variables and array elements.	7.7
Control Statements	Determine order of execution of the object program and terminate its execution.	7.8
Subprogram Statements	Define functions and subroutines.	7.9
Input/Output Statements	Transfer data between internal storage and specified input/output devices.	7.10
FORMAT Statements	Specify formats for data on input/output.	7.11

7.2.1 The FORTRAN Character Set

The FORTRAN character set consists of:

- The upper case letters A through Z
- The numerals 0 through 9
- The special characters in Table 7-8

Table 7-8
FORTRAN Special Characters

Character	Name	Character	Name
	Space	( )	Parentheses
-	Tab	,	Comma
=	Equals	.	Decimal Point
+	Plus	'	Apostrophe
-	Minus	"	Quote
*	Asterisk	$	Dollar Sign
/	Slash		

You may type other printable characters such as %, _, and @ only as part of Hollerith constants, alphanumeric literals, or comments.

7.2.2 Elements of a FORTRAN Program

A FORTRAN program consists of FORTRAN statements and optional comments. You group the statements into logical units called program units (a program unit being a sequence of statements which you terminate with an optional END statement).

A program unit can be either a main program or a subprogram. One main program and possibly one or more subprograms form the executable program.

7.2.2.1 **Statements** - Statements are grouped into two general classes: executable and nonexecutable. Executable statements are the action statements of the program; nonexecutable statements describe data arrangement and data characteristics. Nonexecutable statements may also contain editing and data conversion information.

A program consists of a series of statements, written one statement to a line. (A line is a string of up to 72 characters.) If a statement is too long to fit on one line, you may continue it on up to five additional lines (called continuation lines). (For further information, see Section 7.2.3.4, Continuation Indicator Field.)

A statement can refer to another statement. FORTRAN refers to such a statement by an integer number (called a label) ranging from 1 to 99999. Such a statement is most often referenced for the information it may contain or so that program execution can continue at that statement.

7.2.2.2 **Comments** - Comments are lines of text which document program action, indicate program sections and processes, and provide greater ease in reading the source program listing by identifying variables.

The FORTRAN compiler ignores comments; the comments exist only so that you can document what the program is doing.

7.2.3 FORTRAN Lines

A FORTRAN line consists of four fields:

- Statement Label Field
- Continuation Indicator Field
- Statement Field
- Identification Field

You may skip any of these fields when entering statements, but, except for the identification field, the spaces allotted to each field must remain present. In the case of the identification field, you may type a carriage return before reaching it.

Each printing space represents a single character. The following sections describe the entering of the source program and the information contained in each field.

7.2.3.1 **Using a Text Editor** - When creating a source program with the OS/78 text editor, you type the lines on a "character-per-column" basis. You may also use the (TAB) character to format lines.

NOTE

The text editor and terminal advance the terminal print carriage to a predefined print position when you type a (TAB). This action, however, is not related to FORTRAN's interpretation of the (TAB) character. The FORTRAN system interprets a (TAB) as one character, not the number of characters (up to eight) that it will print.

For example, you may format the following lines in either of the ways shown:

```
C- INITIALIZE ARRAYS      or      C     INITIALIZE ARRAYS

10- W=3                   or      10    W=3

- SEL(1)=111200022D0      or            SEL(1)=111200022D0
```

where:

- - represents a (TAB), and
  represents a space character.

Use space characters in a FORTRAN statement to improve the legibility of a line. The compiler ignores all spaces in a statement field except those within a Hollerith constant or alphanumeric literal.

Example:

```
GO TO and GOTO are equivalent.
```

The compiler also ignores a (TAB) in a statement field; it regards a (TAB) to be the same as a space. However, in the compiler generated source listing, FORTRAN prints the character following the (TAB) at the next tab stop (located at columns 9,17,25,33, etc.).

7.2.3.2 **Statement Label Field** - A statement label is a number which FORTRAN uses to reference one statement from another statement.

A statement label (sometimes also called a statement number) consists of from one to five decimal digits ranging from 1 through 99999. Place this label in the first five positions of a statement's first line. Any source program statement that is referenced by another statement must have a statement number.

FORTRAN ignores spaces and leading zeros preceding the statement label, e.g., FORTRAN interprets the following as statement label 105:

```
105
00105
  105
```

You may assign statement numbers in any order; however, each statement number must be unique in the program or subprogram. In contrast, a main program and a subprogram may contain identical statement numbers. In this case, FORTRAN understands that reference to these numbers means the numbers in the program unit in which the reference is made.

Example:

Assume that the main program and a subprogram both contain statement number 105. A GOTO statement in the main program will refer to statement number 105 in the main program, not to statement 105 in SUB1. A GOTO in SUB1 will transfer control to 105 in SUB1.

An all-zero statement label is illegal.

You cannot label non-executable statements other than FORMAT statements.

When you type a source program with a terminal, an initial <TAB> skips over the label and continuation field.

7.2.3.3 **Comment Indicator** - A comment indicator tells FORTRAN that the text on a line is a comment and that, therefore, it should not process that line.

Type the letter C in column one to indicate that the line is a comment. The compiler will print the contents of that line in the source program listing. However, it ignores the line when it compiles the program.

The following are restrictions on comments.

- All comment lines must begin with the letter C in column one.
- You cannot continue comment lines; consequently each comment line must begin with a C.
- Unlike other statements, the text of a comment can begin in the second space of a line.
- Comment lines must not intervene between a statement's initial line and its continuation line (or lines), or between successive continuation lines.

7.2.3.4 **Continuation Indicator Field** - A continuation indicator tells FORTRAN that the text on that line is part of the same statement as the preceding line.

You must reserve column six of a FORTRAN line for the continuation indicator even if you do not type a continuation indicator.

FORTRAN defines any character except a space in column 6 to be a continuation indicator.

The following are rules for using continuation indicators:

- You may divide a statement into distinct lines at any point.
- You may precede the continuation indicator with space characters only; you may not precede it with a (TAB) as an initial (TAB) skips over the continuation field.
- The characters beginning in column seven of a continuation line are considered to follow the last character of the previous line as if there were no break at that point.
- You may enter no more than 5 continuation lines for one statement.
- You cannot continue comment lines.
- A comment line must not intervene between a statement's initial line and its continuation line (or lines), or between successive continuation lines.
- You cannot assign statement numbers to continuation lines.

7.2.3.5 **Statement Field** - Type the text of a FORTRAN statement in columns 7 through 72. A (TAB) may precede the statement field rather than spaces. Note that because the compiler ignores <TAB>s and spaces (except in Hollerith constants and alphanumeric literals), you can space the text of the statement in any way desired for maximum legibility.

7.2.3.6 **Identification Field** - Type a sequence number or other such identifying information in columns 73-80 of any line in a FORTRAN program. FORTRAN ignores the characters in this field.

NOTE

The FORTRAN compiler ignores text in these positions. Moreover, FORTRAN does not print a warning message if you accidently type text in this field. This is sometimes the source of inexplicable errors.

You might use this feature when typing punched card input. It is seldom used with terminals.

7.2.4 Blank Lines

You may insert lines consisting only of blanks, (TAB)s, or no characters anywhere in your source program except immediately preceding a continuation line. You would use a blank line to improve the readability of a source listing; the FORTRAN compiler ignores them.

7.2.5 Line Format Summary

The fields and the columns in which they may appear are listed in Table 7-9.

Table 7-9
Field Summary

Field	Column
Statement Label	1 through 5
Continuation Indicator	6
Statement	7 through 72
Identification	73 through 80

This example shows the placement of fields (the numbers represent column numbers):

```
                                                    7
1     67                                            3
      DIMENSION A(12),B(10,10,10),C(13,13),D(17,00000001
     121,5)
10    READ (1,10005) (A,B,C,D)                      00000002
C THE DATA IS STORED ON DECTAPE; USE THE FORTRAN RUN  03
C TIME SYSTEM TO ASSIGN LUN 1 TO DTAx:              00000004
      CALL UPDATE(A,D)                              00000005
      IF (.NOT. END) GO TO 10                       00000006
```

7.3 FORTRAN STATEMENT COMPONENTS

The elements of FORTRAN statements are:

- Constants

  A constant is a fixed, self-describing value.

- Variables

  A variable is a symbolic name that represents a stored value.

- Arrays

  An array is a group of variables that you may refer to individually or collectively. The individual values are called array elements. Use a symbolic name to refer to the array.

- Expressions

  An expression can be a constant, variable, array element, or function reference. It may also be a combination of those components and certain other elements (called operators). The operators indicate computations which FORTRAN will perform on the values represented by those components. The result of the computation is a single value.

- Function References

  A function reference is the name of a function (often followed by a list of arguments). After FORTRAN performs the computation indicated by the function definition, it substitutes the computed value in place of the function reference.

7.3.1 Symbolic Names

You use symbolic names to identify many entities within a FORTRAN program unit. Symbolic names consist of a combination of from one to six alphanumeric characters. If you use more than six characters in a symbolic name, FORTRAN reads only the first six.

The first letter of a symbolic name must be a letter. The special characters listed in Table 7-8 may not appear in symbolic names.

Examples of valid and invalid symbolic names are:

Valid	Invalid	
NUMBER	5Q	(Begins with a numeral)
K9	B.4	(Contains a special character)

Table 4-1 indicates the types of variables which FORTRAN identifies by symbolic names.

Except as specifically mentioned in this manual, you may not use the same symbolic name to identify more than one FORTRAN entity.

Each variable indicated as "Typed" in Table 7-10 has a data type. The means of specifying the data type of a name are presented in Sections 7.3.2, Data Types, and 7.6.1, Type Declaration Statements.

Within a subprogram, you may use symbolic names as dummy arguments. A dummy argument may represent a variable, array, array element, constant, expression, or subprogram. However, all subprograms must be uniquely named.

Table 7-10
Classes of Symbolic Names

Entity	Typed
Variables	yes
Arrays	yes
Arithmetic statement functions	yes
Processor-defined functions	yes
FUNCTION subprograms	yes
SUBROUTINE subprograms	no
Common blocks	no
Block data subprograms	no

7.3.2 Data Types

The data type of a FORTRAN element may be inherent in its construction, implied by convention, or you may declare it explicitly. The data types available in FORTRAN, and their definitions, are listed in Table 7-11.

Table 7-11
FORTRAN Data Types

Data Type	Meaning
INTEGER	A whole number.
REAL	A decimal number; it can be a whole number, a decimal fraction, or a combination of the two.
LOGICAL	The logical value "true" or "false".
OCTAL	An integer number in radix 8.

7.3.3 Constants

A constant represents a fixed value; that is, a constant can represent numeric values, logical values, or character strings. You can specify five types of constants in an OS/78 FORTRAN program: integer, real, octal, logical, and Hollerith.

7.3.3.1 **Integer Constants** - An integer constant is a whole number with no decimal point. It may have a leading sign.

Format:

```
snn
```

where:

nn is a string of from 1 to 7 decimal digits, and
s is an optional algebraic sign.

In OS/78 FORTRAN, an integer constant is a whole signed or unsigned number which contains no more than 7 decimal digits. Integer constants must fall within the range -2**23 to 2**23-1 (-8,388,608 to 8,338,607). When you use integer constants as subscripts, FORTRAN uses them at modulo 2**12 (4,096 decimal).

FORTRAN ignores leading zeros in integer constants.

Precede a negative integer constant by a minus symbol. A plus symbol is optional before a positive number because FORTRAN assumes an unsigned constant to be positive; e.g., +27 and 27 are identical.

With the exception of a plus or minus sign, an integer constant cannot contain any character other than the numerals 0 through 9. Specifically, embedded commas and decimal points are not allowed.

Examples:

Valid Integer Constants	Invalid Integer Constants	
0	99999999999	(Too large)
-127	3.14	(Decimal point and
+32123	32,767	comma not allowed)

7.3.3.2 **Real Constants** - A decimal real constant is a string of decimal digits with a decimal point. An exponential real constant is a decimal real constant followed by an exponent.

A Decimal Real Constant is a decimal number. It may have a leading sign.

Format:

```
s.nn
snn.nn
snn.
```

where:

nn is a string of numeric characters.
. is a decimal point.
s is an optional algebraic sign.

A decimal real constant is a string of decimal digits with a decimal point. Note that you do not always have to type a number following the decimal point, but you must always type the decimal point. The decimal point can appear anywhere in the digit string.

FORTRAN does not limit the number of digits in a decimal real constant, but only the leftmost six digits are significant. For example, in the constant 0.000012345678, all of the non-zero digits are significant (note that FORTRAN only stores 0.000012). However, in the constant 000507, the first three zeros are not significant.

You must precede a negative constant with a minus sign. The plus sign is optional preceding a positive real constant.

Except for algebraic signs and a decimal point, a real decimal constant cannot contain any character other than the numerals 0 through 9.

Examples:

Valid Real Constants	Invalid Real Constants
3.14159	1,234,567 (Commas not allowed)
71712.	879877399. (Too large)
-.00127	100 (Decimal point missing)
0.0	

An exponential real constant is a decimal real constant followed by a decimal exponent.

Format:

mmEsnn

where:

- mm is an integer or real constant,
- nn is a 1- to 3-digit integer constant,
- E indicates that the constant is an exponential real constant, and
- s is an algebraic sign.

An exponential real constant is a decimal number which you type in scientific notation, that is, in powers of ten. The number, nn, represents a power of 10 by which the preceding real or integer constant is to be multiplied (e.g., 1E6 represents the value 1.0 x 10**6). The magnitude of a real constant cannot be smaller than 10**-615 nor greater than 10**615. The number mm is an integer or real constant.

A real constant occupies three words (i.e., six bytes) of storage. FORTRAN interprets this number as having a degree of precision slightly greater than seven decimal digits.

In OS/78 FORTRAN, a real exponential constant need not contain a decimal point.

A minus symbol must appear between the letter E and a negative exponent; a plus symbol is optional for a positive exponent.

Except for algebraic signs, a decimal point, and the letter E, a real exponential constant cannot contain any character other than the numerals 0 through 9. However, you may omit the decimal point if the number does not have a fractional part.

Examples:

Valid Real Constants	Invalid Real Constants	
2E-3	-47.E645	(Too large)
+5.0E3	325E-801	(Too small)
	5E3.2	(decimal point misplaced)

7.3.3.3 **Logical Constants** - A logical constant specifies a logical value, that is, "true" or "false". Therefore, there are only two possible logical constants. They are:

.TRUE.

and

.FALSE.

NOTE

You may abbreviate .TRUE. and .FALSE. as .T. and .F.

You must type the delimiting periods as they are part of each constant.

Only logical operators can operate on logical constants.

7.3.3.4 **Octal Constants** - An octal constant is a string of octal digits (0-7 only) preceded by the letter O.

Format:

DATA/Onum/

where:

num is an octal number, and
O identifies the number as an Octal constant.

An octal constant is a digit string (0-7 only) which you may use only in DATA statements to enter numbers in radix eight. An octal constant may be of any length, but the FORTRAN compiler uses only the 12 low order digits.

You generally use octal constants to set bits for masking purposes.

Examples:

```
DATA JOB/O1032/
DATA BASE /O7777/
```

NOTE

The character following the first / in each of these examples is the letter O, not a zero.

7.3.3.5 **Hollerith Constants and Alphanumeric Literals** - A Hollerith constant is a string of ASCII characters preceded by 1) a character count, and 2) the letter H.

Format:

```
nHccc...c
```

where:

- n is an unsigned, non-zero integer constant indicating the number of characters in the string (including spaces and tabs),
- c is any ASCII character, and
- H identifies this as a Hollerith constant.

A Hollerith constant is a string of alphanumeric and/or special characters preceded by a number which states how many characters are in the constant and the letter H. You may use any ASCII character (including those which are not part of the FORTRAN character set).

Hollerith constants have no data type. They assume the data type of the context in which they appear.

Examples:

Valid Hollerith Constants	Invalid Hollerith Constants
16HTODAY'S DATE IS:	3HABCD (wrong number of characters. This will be stored as ABC.)
1H	

An alphanumeric literal is a string of ASCII characters delimited with apostrophes or quotation marks.

Format:

```
'ccc...c'
"ccc...c"
```

where:

- c is a printable ASCII character, and you must type both delimiting apostrophes or quotes.

An Alphanumeric literal is an alternate form of Hollerith constant. Like Hollerith constants, you may use any ASCII character (including those which are not part of the FORTRAN character set).

Alphanumeric literals have no data type. They assume the data type of the context in which they appear.

If you need to type an apostrophe within an alphanumeric literal, type it as two consecutive apostrophes.

Examples:

```
'CHANGE PRINTER PAPER TO PREPRINTED FORM NO. 721'

'TODAY''S DATE IS: '
```

You may use a quotation mark (") instead of an apostrophe. However, you may not mix quotation marks and apostrophes. For example, the following literal is not allowed:

```
"THIS IS A MIXED LITERAL'
```

but you may type

```
"THIS ISN'T A MIXED LITERAL"
```

7.3.4 Variables

A variable is a symbolic name that FORTRAN associates with a storage location. (The FORTRAN compiler assigns the storage locations.) The value of the variable is the value currently stored in that location; you can only change that value by assigning a new value to the variable with an assignment statement.

FORTRAN classifies variables by data type, in the same manner as constants. The data type of a variable indicates

- The type of data it represents,
- Its precision, and
- Its storage requirements.

You may specify the data type of a variable either by type declaration statements (see Section 7.6.1), or by FORTRAN default typing rules (Section 7.3.4.2).

FORTRAN associates two or more variables with each other when each variable uses the same storage location; or, partially associates variables when part (but not all) of the storage which one variable uses is the same as part or all of the storage which another variable uses. You create associations and partial associations with:

- COMMON statements,
- EQUIVALENCE statements, and
- Actual and dummy arguments in subprogram references.

A variable is defined if the storage with which it is associated contains a datum of the same type. You can define a variable prior to program execution by typing a DATA statement or during execution by means of assignment or input statements.

Before you assign a value to a variable, it is an undefined variable, and you should not reference it except to assign a value to it. If you reference an undefined variable, an unknown value (garbage) will be obtained.

If you associate variables of differing types with the same storage location, then defining the value of one variable (for example, by assignment) causes the value of the other variable to become not defined.

7.3.4.1 **Data Type Specification** - Declaration statements (Section 7.6.1) associate given variables with specified data types. For example:

```
INTEGER VAR1
```

This statement indicates that FORTRAN will associate the integer variable VAR1 with a 3-word storage location.

You can only explicitly declare the data type of a variable once in a program unit.

An explicit specification takes precedence over default specification.

7.3.4.2 **Default Data Types** - FORTRAN assumes all variables having names beginning with I, J, K, L, M, or N represent integer data; variables having names beginning with any other letter are real variables. For example:

Real Variables	Integer Variables
ALPHA	KOUNT
BETA	ITEM
TOTAL	NTOTAL

7.3.5 **Arrays**

An array is a group of contiguous storage locations which you reference with a single symbolic name, the array name. You reference the individual storage locations, called array elements, by a subscript appended to the array name.

An array can have from one to seven dimensions.

The following FORTRAN statements establish arrays:

- Type declaration statements (Section 7.6.1),
- DIMENSION statements (Section 7.6.2), and
- COMMON statements (Section 7.6.3).

Each of these statements defines

- The name of the array,
- The number of dimensions in the array, and
- The number of elements in each dimension.

7.3.5.1 **Array Declarations** - Use an array declaration to instruct FORTRAN to reserve storage for an array.

Format:

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

where:

[[typ]]	is a data type declaration,
a	is the array name, and
d	is a number specifying the number of elements in that part of the array.

An array is a group of variables that have the same symbolic name; you address the elements of the array by means of a subscript.

Declare a variable to be an array by specifying the symbolic name which identifies the array within a program unit and which indicates the properties of that array. The number of dimension declarators d indicates the number of dimensions in the array. The minimum number of dimensions is one and the maximum number is seven.

You must declare the size (that is, the number or elements) of an array in order to reserve the needed amount of locations in which to store the array. The value of a dimension declarator specifies the number of elements in that dimension. For example, a dimension declarator value of 50, for example, TABLE(50), indicates that the dimension contains 50 elements. The dimension declarators can be constant or variable.

The rules governing the dimensioning of arrays are as follows (characters enclosed within parentheses represent subscripted characters). In the equation:

```
L(n)=M(1)[1+M(2)+M(2)M(3)+M(2)M(3)M(4)...M(n-1)m(n)]
```

let:

L	= length of the entire array
n	= total number of dimensions in the array
M(i)	= maximum subscript for each dimension in the array, where i specifies which dimension in the array is being referenced.

In the above equation, L must not exceed 4095 in any case.

For example,

```
L(1) = M(1)<4096
L(2) = M(1)[1+M(2)]<4096
L(3) = M(1)[1+M(2)+M(2)M(3)]<4096
etc.
```

In the above equation, L must not exceed 2047 when transmitting arrays, individual arrays, elements, or subportions of an array to subprograms.

For example,

```
L(1) = M(1)<2047
L(2) = M(1)[1+M(2)]<2047
L(3) = M(1)[1+M(2)M(3)]<2047
etc.
```

The number of elements in an array is always equal to the product of the number of elements in each dimension. More specifically, the array IAB dimensioned as (3,4) has 12 elements (3 x 4 = 12) and takes 48 words of storage. Although FORTRAN stores arrays as a series of sequential storage locations, you may best visualize and reference arrays as if they were a single or multi-dimensional rectilinear matrices, dimensioned on a row, column, and plane basis. For example, Figure 7-2 represents a 3-row, 3-column, 2-plane array.

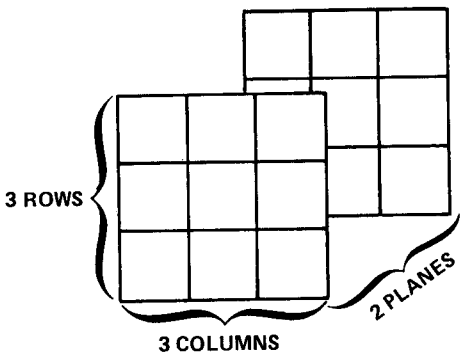


Figure 7-2 Array Representation

Specify the size of an array by an array declaration written as a subscripted array name. In an array declaration, however, each subscript quantity is a dimension of the array and must be either an integer variable or an integer constant.

An array name can appear in only one declaration statement within a program unit.

Use variable dimension declarations to define adjustable arrays (see Section 7.3.5.6).

7.3.5.2 **Array Storage (Order of Subscript Progression)** - OS/78 FORTRAN always stores arrays in memory as a linear sequence of values. For example, FORTRAN stores a 1-dimensional array with its first element in the first storage location and its last element in the last storage location of the sequence. FORTRAN stores a multi-dimensional array such that the leftmost subscripts vary most rapidly. For example, in the array ARRAY(3,2,2) the progression is:

```
ARRAY(1,1,1)
ARRAY(2,1,1)
ARRAY(3,1,1)
ARRAY(1,2,1)
ARRAY(2,2,1)
ARRAY(3,2,1)
ARRAY(1,1,2)
ARRAY(2,1,2)
ARRAY(3,1,2)
ARRAY(1,2,2)
ARRAY(2,2,2)
ARRAY(3,2,2)
```

This is called the "order of subscript progression". For example, consider in Figure 7-3 the array declarators and the arrays that they create.

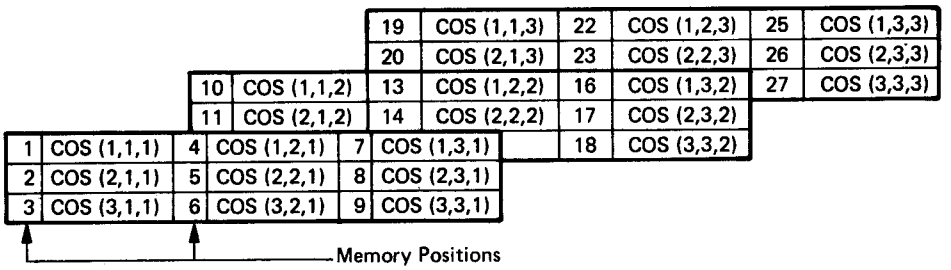


Figure 7-3 Array Storage

The arrows labeled "memory positions" show the order in which FORTRAN stores information in memory. This order is critically important when you use an unsubscripted array name in a READ or WRITE statement as this is the order in which FORTRAN fills memory or prints data.

7.3.5.3 **Subscripts** - A subscript is the means by which you address individual elements in an array.

Format:

(s [[,s]] ...)

where:

s is an integer subscript expression.

Use a subscript following the array to specify which element in the array FORTRAN will reference.

In any subscripted array reference, you must type one subscript expression for each dimension you define for that array (i.e., one for each dimension declaration). For example, you could use the following entry to refer to the element located in the first row, third column, second level of the array TEMP in Figure 7-2 (which is the element occupying memory position 16).

```
TEMP(1,3,2)
```

Note, however, that an array reference such as TEMP(1,3) would be illegal because the third subscript is not indicated.

Each subscript expression can be any valid integer expression. If the value of a subscript expression is not an integer, FORTRAN converts it to integer before using it.

A subscript can be a compound expression, that is,

- Subscript quantities may contain arithmetic expressions that involve addition, subtraction, multiplication, division, and exponentiation. For example, (I+J,K*5,L/2) and (I**3,(J/4+K)*L,3) are valid subscripts.
- A subscript may contain function references. For example, TABLE(IABS(N)*KOUNT,2,3) is a valid array element identifier.
- Subscripts may contain nested array element identifiers as subscripts. For example, in the subscript (I(J(K(L)),M+N,ICOUNT), the first subscript quantity given is a nested 3-level subscript.

7.3.5.4 **Data Type of an Array** - Specify the data type of an array in the same way as the data type of a variable; that is, implicitly by the initial letter of the name, or explicitly by a type declaration statement. (See Section 7.6.1.)

All of the values in an array are of the same data type. FORTRAN converts any value you assign to an array element to the data type of the array.

7.3.5.5 **Array References without Subscripts** - In the following statements, you may type an array name without a subscript to specify that you wish to use the entire array.

Type Declaration Statements

COMMON statement

DATA statement

EQUIVALENCE statement

FUNCTION statement

SUBROUTINE statement

CALL statement

Input/Output statements

Using unsubscripted array names in any other statement is illegal.

7.3.5.6 **Adjustable Arrays** - Use an adjustable array in a subprogram so that the subprogram can process arrays of different sizes. Do this by passing the bounds as well as the array name as subprogram arguments or dummy arguments.

An adjustable array declarator differs from a standard array declarator in that the adjustable declarator has variable dimension declarators (which are simply integer variables). In such an array declaration, each dimension declarator must be either an integer constant or an integer dummy argument. This array name must also appear as a dummy argument. (Consequently, you may not use adjustable array declarators in main program units.)

Upon entry to a subprogram containing adjustable array declarators, FORTRAN associates each dummy argument in a dimension declarator with an integer actual argument. FORTRAN uses these values to form the actual array declaration. These integer variables determine the size of the adjustable array for that single execution of the subprogram.

You must not change the values of the dummy adjustable array declarator arguments within subprogram.

The effective size of the dummy array must be equal to or less than the actual size of the associated array.

The function in the following example computes the sum of the elements of a two-dimensional array. Note the use of the integer variables M and N to control the iteration.

```
      FUNCTION SUM(A,M,N)
      DIMENSION A(M,N)
      SUM = 0.
      DO 10, I = 1,M
      DO 10, J = 1,N
10    SUM = SUM + A(I,J)
      RETURN
      END
```

Following are sample calls on SUM:

```
      DIMENSION A1(10,35), A2(3,56)
      SUM1 = SUM(A1,10,35)
      SUM2 = SUM(A2,3,56)
      SUM3 = SUM(A1,10,10)
```

If there are more dimensions in the adjustable array than in the array being passed to the subroutine, you must indicate a value of 1 for that dimension declaration.

7.4 EXPRESSIONS

An expression is a combination of elements which represents a single value. FORTRAN relates an element in an expression to another element in the same expression by operators and parentheses. The expression can be a single basic component, such as a constant or variable, or a combination of basic components with one or more operators. Operators specify computations to be performed (using the values of the basic components) to obtain a single value.

Expressions can be classified as arithmetic, relational, or logical. Arithmetic expressions yield numeric values; relational and logical expressions produce logical values.

7.4.1 Arithmetic Expressions

Form arithmetic expressions with arithmetic elements and arithmetic operators. The evaluation of such an expression yields a single numeric value.

An arithmetic expression element may be any of the following:

- A numeric constant,
- A numeric variable,
- A numeric array element,
- An arithmetic expression within parentheses,
- An arithmetic function reference.

Arithmetic operators specify a computation which FORTRAN will perform using the values of arithmetic elements; they produce a numeric value as a result. The operators and their meanings are listed in Table 7-12.

Table 7-12
Arithmetic Operators

Operator	Function
**	Exponentiation
*	Multiplication
/	Division
+	Addition and Unary Plus
-	Subtraction and Unary Minus

The operators listed in Table 7-12 are called binary operators, because you would use each in conjunction with two elements. You can also use the + and - symbols as unary operators because, when you write them immediately preceding an arithmetic element, they indicate a positive or negative value.

7.4.1.1 **Rules for Writing Arithmetic Expressions** - Observe the following rules in structuring compound arithmetic expressions:

- An expression cannot contain two adjacent and unseparated operators. For example, the expression A*/B is not permitted.
- You must include all operators; no operation is implied. For example, the expression A(B) does not specify multiplication although this is implied by standard algebraic notation. You must type A*(B) to obtain a multiplication of the elements.
- When you use exponentiation, the base quantity and its exponent may be of different types. For example, the expression ABC**13 involves a real base and an integer exponent. The permitted base/exponent type combinations and the type of the result of each combination are given in Table 7-13.
- You must assign a value to a variable or array element before you use it in an arithmetic expression. If you do not, the elements are undefined.

Table 7-13
Base/Exponent Combinations

BASE	EXPONENT	
	Integer	Real
Integer	Yes	No
Real	Yes	Yes

In addition, you can only exponentiate a negative element by an integer element; you cannot exponentiate an element having a value of zero by another zero-value element.

In any valid exponentiation, the result is of the same data type as the base element.

7.4.1.2 **Evaluation Hierarchy** - FORTRAN evaluates arithmetic expressions in an order determined by a precedence it associates with each operator. The precedence of the operators is listed in Table 7-14.

Table 7-14
Binary Operator Evaluation Hierarchy

Operator	Precedence
**	First
* and /	Second
+ and -	Third
=	Fourth

Whenever two or more operators of equal precedence (such as + or -) appear, FORTRAN evaluates them from left to right. However, FORTRAN evaluates exponentiation from right to left. For example, A**B**C is evaluated as A**(B**C) where FORTRAN computes the parenthetical subexpression (B**C) first.

7.4.1.3 **Date Type of an Arithmetic Expression** - The way in which OS/78 FORTRAN determines the data type of an expression is as follows:

- Integer operations - FORTRAN performs integer operations only on integer elements. (When you use octal constants and logical entities in an arithmetic context, FORTRAN treats them as integers.) In integer arithmetic, any fraction that results from a division is truncated, not rounded. For example, the value of the expression in integer arithmetic

  ```
  1/3 + 1/3 + 1/3
  ```

  is zero, not one.

- Real operations - FORTRAN performs real operations on real elements or a combination of real and integer elements. FORTRAN converts integer elements to real by giving each a fractional part equal to zero. It then evaluates the expression using real arithmetic. Note, however, that in the statement Y = (I/J)*X, FORTRAN performs an integer division operation on I and J and then performs a real multiplication on the result and X.

7.4.2 Relational Expressions

A relational expression consists of two arithmetic expressions which you separate by a relational operator. The value of the expression is either true or false, depending on whether or not the stated relationship exists.

A relational operator tests for a relationship between two arithmetic expressions. These operators are listed in Table 7-15.

Table 7-15
Relational Operators

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

The delimiting periods preceding and following a relational operator are part of the operator and must be present.

In a relational expression, FORTRAN evaluates the arithmetic expressions first to obtain their values. It then compares those values to determine if the relationship stated by the operator exists. For example, the expression:

```
APPLE+PEACH .GT. PEAR+ORANGE
```

tests the relationship, "The sum of the real variables APPLE and PEACH is greater than the sum of the real variables PEAR and ORANGE." If this relationship does exist, the value of the expression is true; if not, the expression is false.

All relational operators have the same precedence. Thus, if two or more relational expressions appear within an expression, FORTRAN evaluates the relational operators from left to right. Note that arithmetic operators have a higher precedence than relational operators.

Use parentheses to alter the evaluation of arithmetic expressions in a relational expression exactly as in any other arithmetic expression. However, as FORTRAN evaluates arithmetic operators before relational operators, it is unnecessary to enclose an arithmetic expression preceding or following a relational operator in parentheses.

7.4.3 Logical Expressions

A logical expression may be a single logical element, or it may be a combination of logical elements and logical operators. A logical expression yields a single logical value, either true or false.

A logical element can be any of the following:

- A logical constant,
- A logical variable,
- A logical array element,
- A relational expression,
- A logical expression enclosed in parentheses,
- A logical function reference (functions and function references are described in Chapter 8).

The logical operators are listed in Table 7-16.

Table 7-16
Logical Operators

Operator	Example	Meaning
.AND.	A .AND. B	Logical conjunction. The expression is true if, and only if, both A and B are true.
.OR.	A .OR. B	Logical disjunction (inclusive OR). The expression is true if, and only if, either A or B, or both, is true.
.XOR.	A .XOR. B	Logical exclusive OR. The expression is true if A is true and B is false, or vice versa. It is false if both elements have the same value.
.EQV.	A .EQV. B	Logical equivalence. The expression is true if, and only if, both A and B have the same logical value, whether true or false.
.NOT.	.NOT. A	Logical negation. The expression is true if, and only if, A is false.

NOTE

A and B can be expressions or constants.

You must type the delimiting periods of logical operators.

A logical expression, like an arithmetic expression may consist of basic elements.

For example:

```
.TRUE.
X .GE. 3.14159
```

or

```
TVAL .AND. INDEX
BOOL(M) .OR. K .EQ. LIMIT
```

where:

BOOL — is either a logical function with one argument or a one-dimensional logical array.

You may enclose logical expressions within parentheses, e.g.,

```
A .AND. (B .OR. C)
```

or

```
(A .AND. B) .OR. C
```

Note that these expressions evaluate differently, e.g., if A is false and C is true, then the first yields a false value while the second yields a true.

7.4.3.1 **Logical Operator Hierarchy** - A summary of all operators that may appear in a logical expression, and the order in which FORTRAN evaluates them is listed in Table 7-17.

Table 7-17
Logical Operator Hierarchy

Operator	Precedence
**	First
*,/	Second
+,-	Third
Relational Operators	Fourth
.NOT.	Fifth
.AND.	Sixth
.OR.	Seventh
.XOR.,.EQV.	Eighth

7.4.4 Use of Parentheses

In an expression, FORTRAN evaluates all subexpressions you place within parentheses first. When you nest parenthetic subexpressions (that is, one subexpression is contained within another) the most deeply nested subexpression is evaluated first, the next most deeply nested subexpression is evaluated second, and so on, until FORTRAN computes the entire parenthetical expression.

When you type more than one operation within a parenthetical subexpression, FORTRAN performs the required computations according to a hierarchy of operators.

Parentheses do not imply multiplication. For example, (A+B)(C+D) is illegal.

The following illustrates a typical numeric expression using numeric operators and a function reference is the familiar formula for obtaining one of the roots of a quadratic equation.

$$\frac{-b + \sqrt{b**2 - 4ac}}{2a}$$

which might be coded

```
(-B + SQRT(B**2-4*A*C))/(2*A)
```

Note how the parentheses affect the order or evaluation. Also note that one parentheses pair is required by the SQRT function. An example of the effect of parentheses is shown below (the numbers below the operators indicate the order in which FORTRAN performs the operations).

```
4 + 3 * 2 - 6 / 2 = 7
    2   1   4   3

(4 + 3) * 2 - 6 / 2 = 11
  1   2   4   3

(4 + 3 * 2 - 6) / 2 = 2
   2   1   3   4

((4 + 3) * 2 - 6) / 2 = 4
   1   2   3   4
```

Evaluation of expressions within parentheses takes place according to the normal order of precedence.

Nonessential parentheses, such as in the expression

```
4 + (3*2) - (6/2)
```

have no effect on the evaluation of the expression.

The use of parentheses to specify the evaluation order is often important where evaluation orders that are algebraically equivalent might not be computationally equivalent when carried out on a computer.

FORTRAN evaluates operators of equal rank from left to right.

7.5 ASSIGNMENT STATEMENTS

Assignment statements evaluate expressions and assign their values to variables or elements in an array.

There are three types of assignment statements:

- Arithmetic assignment statement,
- Logical assignment statement,
- ASSIGN statement (see Section 7.2.3.1).

7.5.1 Arithmetic Assignment Statement

The arithmetic assignment statement assigns a numerical value to a variable or array element.

Format:

```
v = e
```

where:

v is a variable or array element name.
e is an expression.

The arithmetic assignment statement assigns the value of the expression on the right of an equal sign to the variable or array element on the left of the equal sign. If you had previously assigned a value to the variable, an assignment statement replaces it with the value on the right side of the equals sign.

Note that the equal sign does not mean "is equal to", as in mathematics. It means "is replaced by". Thus, the statement:

```
KOUNT = KOUNT + 1
```

means, "Replace the current value of the integer variable KOUNT with the sum of that current value and the integer constant 1".

Although the symbolic name to the left of the equal sign can be undefined, you must previously have assigned values to all symbolic references in an expression (i.e., the right side of the equals sign).

An expression must yield a value that conforms to the requirements of the variable or array element to which you assign it (for example, a real expression that produces a value greater than 8,338,608 is illegal if the entity on the left of the equal sign is an INTEGER variable).

If the data type of the variable or array element on the left of the equal sign is the same as that of the expression on the right, FORTRAN assigns the value directly. If the data types are different, FORTRAN converts the value of the expression to the data type of the entity on the left of the equal sign before it is assigned.

Examples:

Valid Statements

```
BETA = -1./(2.*X)+A*A/(4.*(X*X))

PI = 3.14159

SUM = SUM+1.
```

Invalid Statements

```
3.14 = A-B                  (Entity on the left must be a variable
                             or array element.)

-J = I**4                   (Entity on the left must not be signed.)

ALPHA = ((X+6)*B*B/(X-Y)    (Invalid; left and right parentheses do
                             not balance.)
```

7.5.2 Logical Assignment Statements

Use a logical assignment statement to assign a true or false value to a logical variable.

Format:

```
v = e
```

where:

v is a variable or array element of type logical, and
e is a logical expression.

The logical assignment statement is similar to the arithmetic assignment statement, but it operates on logical data. The logical assignment statement evaluates the expression on the right side of an equal sign and assigns the resulting logical value, either true or false, to the variable or array element on the left.

The variable or array element on the left of the equal sign must be of type LOGICAL; its value can be undefined before the assignment.

You must have previously assigned values, either numeric or logical, to all symbolic references that appear in an expression. The expression must yield a logical value.

Examples:

```
PAGEND = .FALSE.

PRNTOK = LINE .LE. 132 .AND. .NOT. PAGEND

ABIG = A .GT. B .AND. A .GT. C .AND. A .GT. D
```

7.6 SPECIFICATION STATEMENTS

This section discusses FORTRAN specification statements. Specification statements are nonexecutable statements which provide information necessary for the proper allocation and initialization of variables and names you use in a program.

7.6.1 Type Declaration Statements

Type declaration statements explicitly define the data type of symbolic names.

Format:

```
typ v [[,v]] ...
```

where:

typ is one of the following data type specifiers:

```
LOGICAL
INTEGER
REAL
```

v is a typed variable or array.

A type declaration statement causes the specified symbolic names to have the specified data type; it overrides the data type implied by the initial letter of a symbolic name.

A type declaration statement can define arrays by including array declarators in the list. In each program unit, an array name can appear only once in an array declarator. Note, however, that

```
DIMENSION ISUM(3,4)
INTEGER ISUM
```

is legal.

Type declaration statements should precede all executable statements and all specification statements. You must precede the first use of any symbolic name with its declaration statement if you do not use the default type declaration.

You can explicitly declare the data type of a symbolic name only once.

You must not label type declaration statements. The FORTRAN entities which you may type are:

```
Arithmetic Statement Functions
Arrays
functions
Variables
```

Examples:

```
INTEGER COUNT, MATRIX(4,4), SUM
REAL MAN,IABS
LOGICAL SWITCH
```

7.6.2 DIMENSION Statement

The DIMENSION statement defines the number of dimensions in an array and the number of elements in each dimension.

Format:

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

where:

a is the symbolic name of an array
d is the dimension declarator

Example:

```
DIMENSION ARRAY(6,7,4)
```

The DIMENSION statement allocates storage locations, one for each element in each dimension, for each array in the DIMENSION statement. You may declare any number of arrays in one dimension statement. Each storage location is six or twelve bytes in length as determined by the data type of the array. The amount of storage FORTRAN assigns to an array is equal to 6 or 12 times the product of all dimension declarators in the array declarator for that array. For example,

```
DIMENSION ARRAY(4,4), MATRIX(5,5,5)
```

defines ARRAY as having 16 real elements of 6 words each, and MATRIX as having 125 integer elements, also of 6 words each.

You cannot declare more than 7 dimensions to an array. There is also a limit of 4095 elements to any array. Each size specification must be a non-zero positive integer constant.

For further information concerning arrays and the storage of array elements, see Section 2.6.

Array declarators can also appear in type declaration and COMMON statements; however, in each program unit, an array name can appear in only one array declarator.

You must not label DIMENSION statements.

Examples:

```
DIMENSION BUD(12,24,10)

DIMENSION X(5,5,5),Y(4,85),Z(100)

DIMENSION MARK(4,4,4,4,4)
```

7.6.3 EXTERNAL Statement

The EXTERNAL statement permits the use of external procedure names (functions, subroutines, and FORTRAN library functions) as arguments to other subprograms.

Format:

EXTERNAL v [[,v]] ...

where:

v is the symbolic name of a subprogram or the name of a dummy argument which is associated with a subprogram.

Example:

```
EXTERNAL SIN, COS, ABS
```

Any subprogram which you use as an argument to another subprogram must appear in an EXTERNAL statement in the calling subprogram. Thus, the purpose of the EXTERNAL statement is to declare names to be subprogram names. This distinguishes the external name v from other variable or array names.

The subprogram may be ones that you write or those which are part of the FORTRAN library. The EXTERNAL statement declares each name v to be the name of a procedure external to the program unit. Such a name can then appear as an actual argument to a subprogram.

NOTE

If you use a complete function reference such as a call to the SQRT external function in a reference such as CALL SORT(A,SQRT(B),C), the function reference is a value (the square root of B) and you do not need to define it as an external statement. You would only have to define it if you were passing the function name, i.e., CALL SORT(A,SQRT,C).

FORTRAN reserves the names you declare in an external statement throughout the compilation of the program; you cannot use it in any other declaration statement, with the exception of a type statement.

Example:

Main Program

```
EXTERNAL SIN,COS,TAN
   .
   .
CALL TRIG (ANGLE,SIN,SINE)
   .
   .
CALL TRIG (ANGLE,COS,COSINE)
   .
   .
CALL TRIG (ANGLE,TAN,TANGNT)
   .
   .
```

Subprograms

```
SUBROUTINE TRIG (X,F,Y)
Y = F(X)
RETURN
END

FUNCTION TAN (X)
TAN = SIN(X) / COS(X)
RETURN
END
```

The CALL statements pass the name of a function to the subroutine TRIG. The function is subsequently invoked by the function reference F(X) in the second statement of TRIG. Thus, the second statement becomes in effect:

```
Y = SIN(X)
Y = COS(X)
Y = TAN(X)
```

depending upon which CALL statement invoked TRIG. The functions SIN and COS are examples of trigonometric functions supplied in the FORTRAN Library.

7.6.4 COMMON Statement

Use a COMMON statement so that programs and subprograms can share information.

Format:

COMMON [[/ [[cb]] /]] nlist / [[cb]] / nlist]] ...

where:

cb — is a symbolic name or is blank. If the first cb is blank, you can omit the first pair of slashes, and

nlist — is a list of variable names, array names, and array declarators separated by commas.

Example:

```
COMMON /AREA1/A,B //C,D
```

The COMMON statement enables you to establish storage that two or more programs and/or subprograms may share and to name the variables and arrays that will occupy the common storage. The use of common storage conserves storage and provides a means to implicitly transfer arguments between a calling program and a subprogram. The transfer is implicit because no actual tranferral takes place; instead, the program unit references the common storage area.

FORTRAN determines the length of a COMMON block by the number of components and the amount of storage each component requires. COMMON blocks may be of any length, subject to the limitations of available memory.

nlist, which appears after each common name cb, lists the names of the variables and arrays that will occupy the common area cb. FORTRAN places the items for a common within common storage area in the order in which you list them in the COMMON statement or statements.

Elements you place into common storage in one program unit should agree in data type with elements reference in a second. This is because assignment of storage is on a storage unit-for-storage unit basis, not variable-for-variable.

Either label COMMON storage areas or leave them blank (unlabeled). If you label the common area, type a symbolic name within slashes immediately before the list of items that will occupy the cb area.

For example, the statement

```
COMMON/AREA1/A,B,C/AREA2/TAB(13,3,3)
```

establishes two labeled common areas (i.e., AREA1 and AREA2).

If you are declaring a common storage area to be blank common, then you may omit the double slashes (//) if and only if it is the first declaration of any common statement. Unlabeled common area is called "blank common". If the blank common declaration is not the first declaration in a COMMON statement, then the double slashes are mandatory.

For example, the statement

```
COMMON/AREA1/A,B,C//TAB(3,3,3)
```

establishes one labeled area (AREA1) and one unlabeled common area.

A given labeled common name may appear more than once in the same COMMON statement and in more than one COMMON statement within the same program or subprogram.

During compilation of a source program, FORTRAN will bring together all items you list for each labeled and blank common area in the order in which the items appear in the source program statements.

For example, the series of source program statements:

```
COMMON/ST1/A,B,C/ST1/TAB(2,2)//C,D,E
.
.
COMMON/ST1/TST(3,4)//M,N
.
.
COMMON/ST2/X,Y,Z//O,P,Q
```

has the same effect as the single statement

```
COMMON/ST1/A,B,C,TST(3,4)/ST2/TAB(2,2),X,Y,Z//C,D,E,M,N,O,P,Q
```

FORTRAN treats each labeled common area as a separate, specific storage area. You assign the contents of a common area, i.e., variables and arrays, initial values by DATA statements in a BLOCK DATA subprogram. Declarations of a given common area in different subprograms must contain the same number, size, and order of variables and arrays as the reference array.

Common block names must be unique with respect to all subroutine and function names.

The largest definition of a given common area must be loaded first.

Storage allocation for blocks of the same name begins at the same location for all program units FORTRAN executes together. For example, if a program contains:

```
COMMON A,B,C/R/X,Y,Z
```

as its first COMMON statement, and a subprogram has:

```
COMMON /R/U,V,W //D,E,F
```

as its first COMMON statement, the values represented by X and U are stored in the same location. A similar correspondence holds for A and D in blank common.

If one program unit references a part of a common block, then you must use dummy variables to establish the proper correspondence. For example, if you declare a common block to contain:

```
A,B,C,D,E,F,G,H,I,J,K
```

and a subprogram wishes to reference the storage location pointed to by K, then you must declare a common block as follows in the subprogram:

```
COMMON A,B,C,D,E,F,G,H,I,J,K
```

The declaration COMMON K in the subprogram would cause a correspondence between variable A in the main program and variable K in the subprogram. (Note that any other sequence of variables names would also be correct.)

Instead of declaring each variable contained in the COMMON block, you may substitute a dummy array (provided that you are careful to match-up proper storage lengths).

You may also define an array in a COMMON statement. You may not otherwise subscript array names. Also, you cannot assign individual array elements to COMMON.

7.6.5 EQUIVALENCE Statement

Use an EQUIVALENCE statement to associate different variables with the same storage.

Format:

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

where:

nlist — is a list of variables and array elements, separated by commas. At least two components must be present in each list.

Example:

```
EQUIVALENCE (A,B),(C,D(16),E,F)
```

The EQUIVALENCE statement declares two or more entities to be associated (either totally or partially) with the same storage location.

NOTE

EQUIVALENCE differs from COMMON in that EQUIVALENCE associates different variable names with the same storage area in a program unit. COMMON may associate different variable names with the same storage area but it always makes the association between program units.

The EQUIVALENCE statement causes FORTRAN to allocate all of the variables or array elements contained in one parenthesized list beginning at the same storage location.

You can also use the EQUIVALENCE statement to equate variable names. For example, the statement

```
EQUIVALENCE (FLTLEN, FLENTH, FLIGHT)
```

causes FLTLEN, FLENTH, and FLIGHT to have the same value provided they are also of the same data type.

An EQUIVALENCE statement in a subprogram must not contain dummy arguments.

Examples:

```
EQUIVALENCE (A,B), (B,C)        (has the same effect as EQUIVALENCE
                                (A,B,C))

EQUIVALENCE (A(1),X), (A(2),Y), (A(3),Z)
```

7.6.5.1 **Making Arrays Equivalent** - When you make an element of an array equivalent to an element of another array, the EQUIVALENCE statement also sets equivalences between other elements of the two arrays. Thus, if you make the first elements of two equal-sized arrays equivalent, both arrays share the same storage space. Moreover, if you make the third element of a 5-element array equivalent to the first element of another array, the last three elements of the first array overlap the first three elements of the second array.

The EQUIVALENCE statement must not attempt to assign the same storage location to two or more elements of the same array, nor to assign memory locations in any way that is inconsistent with the normal linear storage of array elements (for example, making the first element of an array equivalent with the first element of another array, then attempting to set an equivalence between the second element of the first array and the sixth element of the other).

In the EQUIVALENCE statement only, it is possible to identify an array element with a single subscript (that is, the linear element number), even though you have defined one as being multi-dimensional.

For example, the statements:

```
DIMENSION TABLE (2,2), TRIPLE (2,2,2)
EQUIVALENCE (TABLE(4), TRIPLE(7))
```

result in the entire array TABLE sharing a portion of the storage space FORTRAN allocates to array TRIPLE as illustrated in Figure 7-4. In Figure 7-4, the elements with asterisks are those explicitly mentioned in the above EQUIVALENCE statement.

Array TRIPLE		Array TABLE	
Array Element	Element Number	Array Element	Element Number
TRIPLE(1,1,1)	1		
TRIPLE(2,1,1)	2		
TRIPLE(1,2,1)	3		
TRIPLE(2,2,1)	4	TABLE(1,1)	1
TRIPLE(1,1,2)	5	TABLE(2,1)	2
TRIPLE(2,1,2)	6	TABLE(1,2)	3
TRIPLE(1,2,2)	7*	TABLE(2,2)	4*
TRIPLE(2,2,2)	8		

Figure 7-4 Equivalence of Array Storage

Figure 7-4 also illustrates that the following two statements

```
EQUIVALENCE (TABLE(1),TRIPLE(4))
EQUIVALENCE (TRIPLE(1,2,2), TABLE(4))
```

result in the same alignment of the two arrays.

7.6.5.2 **EQUIVALENCE and COMMON Interaction** - When you make components equivalent to entities in common, it can cause FORTRAN to extend the common block beyond its original boundaries.

An EQUIVALENCE statement can only extend common beyond the last element of the previously established common block. It must not attempt to increase the size of common in such a way as to place the extended portion before the first element of existing common. (See Figure 7-5.)

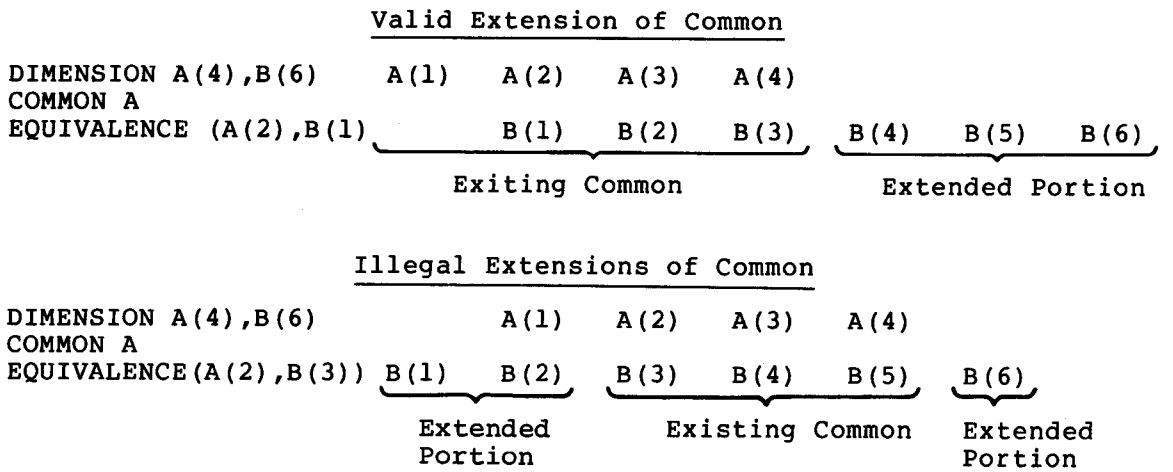


Figure 7-5 Legal and Illegal Common Extensions

If you assign two components to the same or different common blocks, you must not make them equivalent to each other.

7.7 DATA STATEMENTS AND BLOCK DATA SUBPROGRAMS

The DATA initialization statement permits the assignment of initial values to variables and array elements prior to program execution.

Format:

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

where:

nlist — is a list of one or more variable names, array names, or array element names separated by commas,

is an optional separator, and

clist — is a list of constants.

Example:

```
DATA A,B,C(3),C(7)/4.0,8.1,16.0,28.0/
```

The DATA statement causes FORTRAN to assign the constant values in each clist to the entities in the preceding nlist. FORTRAN assigns values in a one-to-one manner in the order in which they appear, from left to right. [[IS COMMA OPTIONAL OR MANDATORY]]

When an unsubscripted array name appears in a DATA statement, FORTRAN assigns values to every element of that array. The associated constant list must therefore contain enough values to fill the array. FORTRAN fills array elements in the order of subscript progression. (See Section 2.6.1.)

When you assign Hollerith data to a variable or array element, the number of characters that you can assign depends on the data type of the component. If the number of characters in a Hollerith constant or alphanumeric literal is less than the capacity of the variable or array element, the constant is padded on the right with spaces. If the number of characters in the constant is greater than the maximum number that the variable can hold, it ignores the rightmost excess characters.

When you assign the same value to more than one item in nlist, you may use a repeat specification. Write the repeat specification as N*D where N is an integer that specifies how many times the value of item D is to be used. For example, a DATA specification of /3*20/ specifies that the value 20 is to be assigned to the first three items named in the preceding list. Also, the statement

```
DATA M,N,L /3*20/
```

assigns the value 20 to the variables M, N, and L. The number of constants in a constant list must correspond exactly to the number of entities specified in the preceding name list. The data types of the data elements and their corresponding symbolic names must agree.

FORTRAN IV converts the constant to the type of the variable being initialized.

Example:

```
INTEGER A(10),BELL,K(5,5,5)
DATA A,BELL,STARS/10*0,7, '****'/K/25*0,25*1,25*2,25*3,25*4,25*5/
```

The DATA statement assigns zero to all ten elements of array A, the value 7 to the variable BELL, and four asterisks to the real variable STARS. The 125 element array, K, is initialized so that each of the five planes (i.e., the third dimension declarator) has a different value.

When you initialize an array, you must initialize the entire array, e.g., the DATA statement in the following

```
DIMENSION K
DATA K /10*1/
```

is illegal.

You could effect the same thing as follows:

```
DIMENSION I(30),K(10)
EQUIVALENCE (I,K)
DATA K/10*1/
```

The values you assign with a DATA statement may also be assigned with a BLOCK DATA subprogram. However, note that initial values for variables in COMMON storage may not be specified in subprograms which may be overlaid at execution time. If a subprogram will be overlaid, then you should only initialize these variables in a BLOCK DATA subprogram. (DIGITAL recommends that you only initialize variables in COMMON storage with BLOCK DATA subprograms.)

Use a BLOCK DATA to initialize variables you place into COMMON storage.

Format:

```
BLOCK DATA
```

Use the BLOCK DATA subprogram to assign initial values to entities in common blocks, at the same time establishing and defining those blocks. It consists of a BLOCK DATA statement followed by a series of specification statements.

The statements FORTRAN allows in a BLOCK DATA subprogram are:

```
Type Declaration
DIMENSION
COMMON
EQUIVALENCE
DATA
```

The specification statements in the BLOCK DATA subprogram establish and define common blocks, assign variables and arrays to those blocks, and assign initial values to those components.

A BLOCK DATA statement must be the first statement of a BLOCK DATA subprogram. You must not label the BLOCK DATA statement.

A BLOCK DATA subprogram must not contain any executable statements.

If you initialize any entity in a common block in a BLOCK DATA subprogram, you must enter a complete set of specification statements to establish the entire block, even though some of the components in the block do not appear in a DATA statement. You can define initial values for more than one common area with the BLOCK DATA subprogram.

7.8 CONTROL STATEMENTS

FORTRAN normally executes statements in the order in which you write them. However, it is frequently desirable to change the normal program flow by transferring control to another section of the program or to a subprogram. Transfer of control from a given point in the program may occur every time that point is reached in the program flow, or may be based on a decision made at that point.

Transfer of control, whether within a program unit or to another program unit, is performed by control statements. These statements also govern iterative processing, suspension of program execution, and program termination. The types of control statements discussed in this chapter are:

```
ASSIGN
CONTINUE
DO
END
IF
GO TO
PAUSE
STOP
```

A second kind of statement for transferring control, subprograms, are discussed in Chapter 8.

7.8.1 GOTO Statements

GOTO statements transfer control within a program unit, either to the same statement every time or to one of a set of statements, based on the value of an expression.

The three types of GOTO statements are:

- Unconditional GOTO statement,
- Computed GOTO statement, and
- Assigned GOTO statement.

7.8.1.1 **Unconditional GOTO Statement** - Transfers control to the same statement every time executed.

Format:

```
GOTO st
```

where:

st is the label of an executable statement in the same program unit as the GOTO statement.

Example:

```
GOTO 50
```

The unconditional GOTO statement transfers control to the statement identified by the specified label. The statement label must identify an executable statement in the same program unit as the GOTO statement.

Examples:

```
GOTO 7734

GOTO 99999

GOTO 27.5       (Invalid; the statement label is improperly
                formed.)
```

7.8.1.2 **Computed GOTO Statement** - Transfers control to a statement based on the value of an expression within the statement.

Format:

GOTO (slist) [[,]] e

where:

slist	is a list of one or more executable statement labels separated by commas,
,	is an optional separator, and
e	is an integer expression the value of which falls within the range 1 to n (where n is the number of statement labels in slist).

Example:

```
GOTO (10,200,25), NUMBER
```

Use the computed GOTO to transfer control to one statement out of a list of statements. The computed GOTO thus acts as a multi-directional switch.

The computed GOTO statement evaluates the integer expression e. The GOTO statement then transfers control to the e'th statement label in slist. That is, if the list contains (30,20,30,40), and the value of e is 2, the GOTO statement transfers control to statement 20, and so on.

You may include any number of statements in slist but you must use each number as a label within the program.

The comma following (slist) is optional.

If the value of the expression is less than 1, or greater than the number of labels in the slist, unpredictable results occur.

Examples:

```
GOTO (12,24,36),INCHES

GOTO (320,330,340,350,360)ISITU(J,K)+1
```

7.8.1.3 **ASSIGN and ASSIGNed GOTO Statement** - Use the ASSIGN statement to assign a statement label to a variable name.

Format:

```
ASSIGN st to v
```

where:

st	is the label of an executable statement in the same program unit as the ASSIGN statement, and
v	is an integer variable.

Example:

```
ASSIGN 50 TO NUMBER
```

Use the ASSIGN statement to associate a statement label with an integer variable. You can then use the variable as a transfer destination in a subsequent ASSIGNed GOTO statement.

NOTE

The statement number must be in the same program unit.

The statement label st must not be the label of a FORMAT statement.

The ASSIGN statement assigns the statement number to the variable in a manner similar to that of an arithmetic assignment statement, with one exception: the variable becomes defined for use as a statement label reference and becomes undefined as an integer variable.

FORTRAN must execute an ASSIGN statement before the ASSIGNed GOTO statement in which it will use the assigned variable. The ASSIGN statement and the ASSIGNed GOTO statement must occur in the same program unit.

For example, the statement

```
ASSIGN 100 TO NUMBER
```

associates the variable NUMBER with the statement label 100.

Arithmetic operations on the variable, such as in the statement

```
NUMBER = NUMBER + 1
```

then become invalid, as FORTRAN cannot alter a statement label. (This is because a statement refers to a location in memory and is not a number.) The statement:

```
NUMBER = 10
```

disassociates NUMBER from statement 100, assigns it an integer value 10, and returns it to its status as an integer variable. After you make such an assignment, you can no longer use it in an ASSIGNed GOTO statement.

Examples:

```
ASSIGN 10 TO NSTART

ASSIGN 99999 TO KSTOP

ASSIGN 250 TO ERROR      (ERROR must have been defined as an
                          integer variable.)
```

The ASSIGNed GOTO transfers control to a statement which is represented by a variable.

Format:

GOTO v [[[[,]] (slist)]]

where:

v	is an integer variable,
,	is an optional separator, and
slist	(when present) is a list of one or more executable statement labels separated by commas.

Example:

```
GOTO NUMBER,(10,35,15)
```

The ASSIGNed GOTO statement transfers control to the statement whose label was most recently assigned to the variable v by an ASSIGN statement. (See Section XXX.)

The variable v must be of integer type. In addition, you must have previously assigned to it a statement label number with an ASSIGN statement (not an arithmetic assignment statement).

The ASSIGNed GOTO statement and its associated ASSIGN statement must reside in the same program unit. Also, statements to which FORTRAN transfers control must be executable statements in the same program unit.

Examples:

```
ASSIGN 50 TO IGO
GOTO IGO

GOTO INDEX, (300,450,1000,25)
```

If the statement label value of v is not present in the list slist (and a list is specified), control transfers to the next executable statement following the ASSIGNed GOTO statement.

NOTE

You must label the statement following an ASSIGNed GOTO; otherwise, FORTRAN can never execute that statement.

7.8.2 IF Statements

An IF statement causes a conditional control transfer or the conditional execution of a statement. There are two types of IF statements:

- Arithmetic IF statements, and
- Logical IF statements.

7.8.2.1 **Arithmetic IF Statement** - Use the arithmetic IF as a three-way branching statement. The branching depends on whether the value of an expression is less than, equal to, or greater than zero.

Format:

```
IF (e) st1, st2, st3
```

where:

e	is an arithmetic expression, and
st1, st2, st3	are the labels of executable statements in the same program unit.

Example:

```
IF (I-K) 10, 20, 30
```

Use the arithmetic IF statement for conditional control transfers. This statement can transfer control to one of three statements, based on the value of an arithmetic expression.

You may use logical expressions in arithmetic IF statements. In such a case, FORTRAN first converts the logical expression value to an integer. If you use a complex expression, FORTRAN only uses the real portion.

Normal use of the arithmetic IF requires that all three labels, stl, st2, and st3, must be present. However, they need not refer to three different st. If desired, one or two labels can refer to the same statement.

OS/78 FORTRAN allows you to type less than three numbers. If you type either one or two numbers, then if a condition is not met (e.g., e is greater than zero), then control passes to the next statement.

Example:

```
IF (ALPHA) 10
STOP
```

In this statement, control transfers to statement number 10 if ALPHA is negative. If ALPHA is positive or equal to zero, execution stops.

The arithmetic IF statement first evaluates the expression in parentheses and then transfers control to one of the three statement labels that follow expression e. The values upon which FORTRAN makes the selection are listed in Table 7-18.

Table 7-18
Arithmetic IF Transfers

If the Value is:	Control Passes to:
Less than 0	Labels stl
Equal to 0	Label st2
Greater than 0	Label st3

Examples:

```
IF (THETA-CHI) 50,50,100
```

This statement transfers control to statement 50 if the real variable THETA is less than or equal to the real variable CHI. Control passes to statement 100 only if THETA is greater than CHI.

```
IF (NUMBER/2*2-NUMBER) 20,40
```

This statement transfers control to statement 40 if the value of the integer variable NUMBER is even and to statement 20 if it is odd.

7.8.2.2 **Logical IF Statement** - Use a logical IF statement for conditional execution of statements.

Format:

```
IF (e) st
```

where:

e is a logical expression, and
st is a complete FORTRAN statement. The statement can be any executable statement except a DO statement or another logical IF statement.

Example:

```
IF(X .EQ.  Y) Z=4
```

A logical IF statement causes a conditional statement execution. FORTRAN bases the decision to execute the statement on the value of a logical expression within the statement.

The logical IF statement first evaluates the logical expression. If the value of the expression is true, FORTRAN transfers control to the executable statement within the IF statement. If the value of the expression is false, control transfers to the next executable statement following the logical IF; in this case, FORTRAN does not execute statement st.

Examples:

```
IF (J .GT.  4 .OR.  J .LT.  1) GOTO 250

IF (REF(J,K) .NE.  HOLD) REF(J,K) = REF(J,K)*A(K,J)

IF (.NOT.  X) CALL SWITCH(S,Y)
```

7.8.3 **DO Statement**

Use the DO statement to repeatedly execute a block of statements.

Format:

```
DO st i=e1,e2 [[,e3]]
```

where:

st is the label of an executable statement which physically follows in the same program unit,
i is an unsubscripted real or integer variable,
e1 (the initial value of i) is an integer or real constant or expression,
e2 (the terminal value of i) is an integer or real constant or expression and must be greater than e1, and
e3 (the value by which i will be incremented each time it executes the statements in the range of the DO loops) an integer real constant or expression.

Example:

```
DO 10 I=1,10,2

DO 20 I=J,K,L
```

The DO statement causes FORTRAN to repeatedly execute the statements in its range a specified number of times.

The range of a DO statement is defined as the series of statements that follow the DO statement up to and including its specified terminal statement st, that is, the statements that follow the DO statement, up to and including the terminal statement are in the range of the DO loop.

The variable i is called the control (or index) variable of the DO and e1, e2, e3 are the initial, terminal, and increment parameters respectively.

The terminal statement of a DO loop is identified by the label st that appears in the DO statement. This terminal statement must not be a GOTO statement, an arithmetic IF statement, a RETURN statement, PAUSE statement, STOP statement, or another DO statement. A logical IF statement is acceptable as the terminal statement, provided it does not contain any of the above statements.

The DO statement first evaluates the expressions e1, e2, e3 to determine values for the initial, terminal, and increment parameters. FORTRAN then assigns value of the initial parameter to the control variable. FORTRAN then repeatedly executes the statements in the range of the DO loop.

The increment parameter must be positive, and the value of the terminal parameter must not be less than that of the initial parameter.

The value of the increment parameter must not be zero.

After each execution of the range of the DO loop, FORTRAN adds the increment value to the value of the index. It then compares the result to the terminal value. If the index value is not greater than the terminal value, FORTRAN reexecutes the range using the new value of the index i.

The number of executions of the DO range, called the iteration count, is given by

```
MAX(1,((e2-e1)/e3) + 1
```

FORTRAN always executes the range of a DO statement at least once.

7.8.3.1 **DO Iteration Control** - You can terminate the execution of a DO by a statement within the range that transfers control outside the loop. When you transfer out of the DO loop's range, the control variable of the DO remains defined with its current value.

When execution of a DO loop terminates, if other DO loops share the same terminal statement, control transfers outward to the next most enclosing DO loop in the DO nesting structure (Section 7.4.2). If no other DO loop share this terminal statement, or if this DO is the outermost DO, control transfers to the first executable statement following the terminal statement.

You may alter the values of i, e1, e2, and e3. If you alter the value of i, the loop will not be executed the number of times which you originally specified. If you alter the values of the expressions, you do not affect the looping as FORTRAN "remembers" these values. The control variable i is available for reference as a variable within the range.

The range of a DO loop can contain other DO statements, as long as those "nested" DO loops conform to certain requirements.

You can transfer control out of a DO loop, but you cannot transfer into a loop from elsewhere in the program. Exceptions to this rule are described in the following sections.

Examples:

```
DO 100 K=1,50,2      (25 iterations, K=49 during final iteration)

DO 25 IVAR=1,5       (5 iterations, IVAR=5 during final iteration)

DO NUMBER=5,40,4     (Invalid; statement label missing)

DO 40 M=2.10         (Invalid; decimal point instead of comma)
```

The last example illustrates a common clerical error. It is a valid arithmetic assignment statement in the FORTRAN language; i.e.,

```
DO40M = 2.10
```

7.8.3.2 **Nested DO Loops** - A DO loop may contain one or more complete DO loops. The range of an inner nested DO must lie completely within the range of the next outer loop. Nested loops may share the same terminal statement. (See Figure 7-6.)

Correctly Nested DO Loops

```
    DO 45 K=1,10
     .
     .
    DO 35 L=2,50,2
     .
     .
 35 CONTINUE
     .
     .
    DO 45 M=1,20
     .
     .
 45 CONTINUE
```

Incorrectly Nested DO Loops

```
    DO 15 K=1,10
     .
     .
    DO 25 L=1,20
     .
     .
 15 CONTINUE
     .
     .
    DO 30 M=1,15
     .
     .
     .
 25 CONTINUE
     .
     .
 30 CONTINUE
```

Figure 7-6 Nesting of DO Loops

In the correctly nested DO loops, note that the diagrammed lines do not cross. They do, however, share the same statement (45). In the incorrectly nested DO loops, the loop defined by DO 25 crosses the ranges of the other two DO loops.

Note that you may nest loops to a depth of (at least) 10 levels.

7.8.3.3 **Control Transfers in DO Loops** - Within a nested DO loop structure, you can transfer control from an inner loop to an outer loop. A transfer from an outer loop to an inner loop is illegal.

If two or more nested DO loops share the same terminal statement, you can transfer control to that statement only from within the range of the innermost loop, that is, the terminal statement belongs solely to the innermost DO statement. Any other transfer to that statement constitutes a transfer from an outer loop to an inner loop because the shared statement is part of the range of the innermost loop.

The following rules govern the transfer of program control from within the DO statements range or the ranges of nested DO statements.

- FORTRAN permits a transfer out of the range of any DO statement at any time. When such a transfer executes, the controlling DO statement's index variable retains its current value.

- FORTRAN permits a transfer into the range of a DO statement from within the range of any:

  1. DO loop;

  2. nested DO loop; or

  3. extended range loop (in which you leave the loop via a GOTO, execute statements elsewhere, and return to the original loop).

7.8.3.4 **Extended Range** - A DO loop is said to have an extended range if it contains a control statement that transfers control out of the loop and if, after the execution of one or more statements, another control statement returns control back into the loop. In this way, FORTRAN extends the range of the loop to include all of the executable statements between the destination statement of the first transfer and the statement that returns control to the loop.

Figure 7-7 illustrates valid and invalid control transfers.

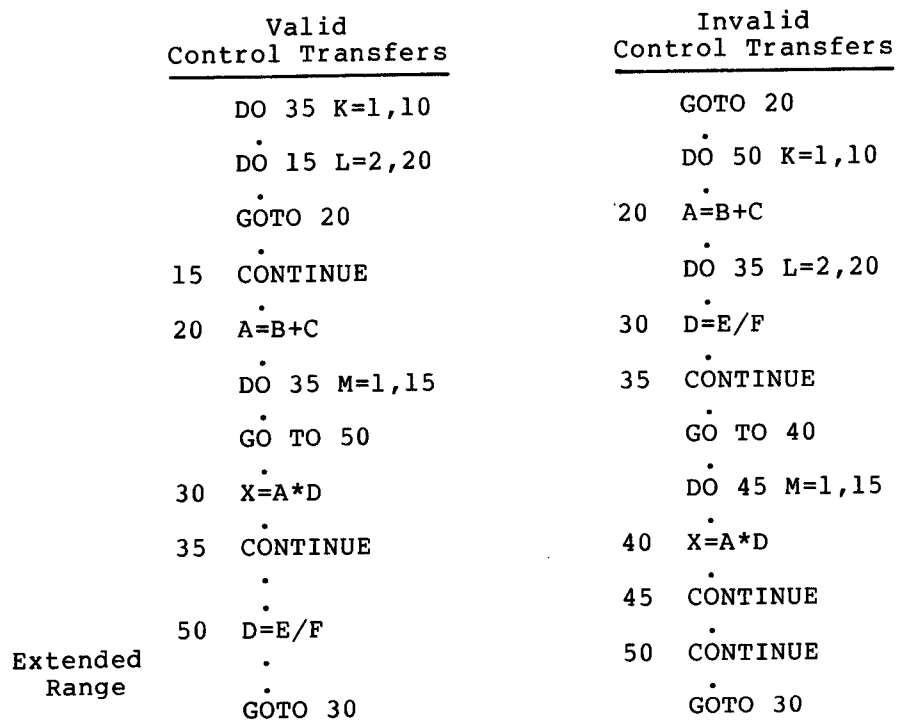


Figure 7-7 Control Transfers and Extended Range

The following rules govern the use of a DO statement extended range:

- The transfer out statement for an extended range operation must be contained by the most deeply nested DO statement that contains the location to which the return transfer is to be made.
- A transfer into the range of a DO statement is permitted only if the transfer is made from the extended range of that DO statement.
- The extended range of a DO statement must not contain another DO statement.
- The extended range of a DO statement cannot change the index variable or indexing parameters of the DO statement.
- You may execute subprograms within an extended range.

7.8.4 CONTINUE Statement

Insert a CONTINUE statement where you do not wish any statement to be executed.

Format:

```
st CONTINUE
```

where:

st is a statement label.

A CONTINUE statement is a statement that holds a place in the program without performing any operations.

You may place CONTINUE statements anywhere in the source program without affecting the program sequence of execution. CONTINUE statements are commonly used as the last statement of a DO statement range in order to avoid ending with a GOTO, PAUSE, STOP, RETURN, arithmetic IF, another DO statement, or a logical IF statement containing one of the previous statements. However, they are valid throughout a source program.

Note that you also use a CONTINUE as a transfer point for a GOTO statement within the DO loop that is intended to begin another repetition of the loop.

Example:

In the following sequence, the labeled CONTINUE statement provides a legal termination for the range of the DO loop.

```
          .
          .
          DO 45 ITEM=1,1000
          STOCK=NVNTRY(ITEM)
          IF (STOCK .EQ. TALLY) GO TO 45
          CALL UPDATE(STOCK,TALLY)
          IF (ITEM .EQ. LAST) GO TO 77
45        CONTINUE
          .
          .
          .
77        WRITE (4,20) HEADING, PAGENO
          .
          .
          .
```

7.8.5 PAUSE Statement

The PAUSE statement temporarily suspends program execution to permit some action on the part of the user.

Format:

PAUSE [[num]]

where:

num is an optional integer variable or expression containing one to five digits.

The PAUSE statement prints the display (if you have specified one) at your terminal, suspends program execution, and waits for you to type the RETURN key. This causes program execution to resume with the first executable statement following the PAUSE.

Examples:

```
PAUSE "13731

PAUSE 'MOUNT TAPE REEL #3'
```

7.8.6 STOP Statement

Use the STOP statement to terminate program execution.

Format:

```
STOP
```

The STOP statement terminates program execution and returns control to the operating system. If you do not type a STOP statement, a "stop" occurs when FORTRAN transfers control to an END statement in the main program unit.

A CALL EXIT statement is equivalent to STOP and closes any temporary files at the last block written on the file. Control returns to the OS/78 Monitor.

Examples:

```
        STOP

99999 STOP
```

7.8.7 END Statement

The END statement marks the end of every program unit and it must be the last source line of every program unit.

Format:

```
END
```

In a main program, if control reaches the END statement, execution of the program terminates; in a subprogram, a RETURN statement is implicitly executed.

In the main program, END is equivalent to STOP. In a subprogram, it is equivalent to RETURN.

A program cannot reference an END statement.

Control returns to the OS/78 Monitor after FORTRAN executes an END statement.

If you do not type an END statement as the last statement in your program, FORTRAN appends one.

7.9 SUBPROGRAMS

Procedures you use repeatedly in a program may be written once and then referenced each time you need the procedure. Procedures that you may reference are either internal (written and contained within the program in which they are referenced) or external (self-contained executable procedures that you may compile separately). The kinds of procedures that you may reference are:

- Arithmetic Statement Functions,
- External Functions,
- Subroutines, and
- Intrinsic functions (FORTRAN-defined functions).

7.9.1 Subprogram Arguments

Since you may reference subprograms at more than one point throughout a program, many of the values which the subprogram uses may be changed each time the subprogram is called. Dummy arguments in subprograms represent the actual values which the subprogram will use. The arguments are passed to the subprogram when FORTRAN transfers control to it.

Functions and subroutines use dummy arguments to indicate the type of the actual arguments they represent and whether the actual arguments are variables, array elements, arrays, subroutine names, or the names of external functions. You must use each dummy argument within a subprogram as if it were a variable, array, array element, subroutine, or external function identifier. You enter dummy arguments in an "argument list" which you associate with the identifier assigned to the subprogram; actual arguments are normally given in an argument list which you associate with a call made to the subprogram.

The position, number, and type of each dummy argument in a subprogram must agree with the position, number, and type of each argument in the argument list of the subprogram reference.

Dummy arguments may be:

- Variables,
- Array names,
- Subroutine identifiers, or
- Function identifiers.

When you reference a subprogram, FORTRAN replaces its dummy arguments by the corresponding actual arguments which you supply in the reference. All appearances of a dummy argument within a function or subroutine are related to the given actual arguments. Except for subroutine identifiers and literal constants, a valid association between dummy and actual arguments occurs only if both are of the same type; otherwise, the result of the subprogram will be unpredictable. Argument associations may be carried through more than one level of subprogram reference if a valid association is maintained through each level. FORTRAN terminates the dummy/actual argument associations which it establishes when you reference a subprogram. This occurs when FORTRAN completes the operations defined in the subprogram.

The following rules govern the use and form of dummy arguments:

- The number and type of the dummy arguments of a procedure must be the same as the number and type of the actual arguments given each time you reference the procedure.
- Dummy argument names may not appear in EQUIVALENCE, DATA, or COMMON statements.
- A variable dummy argument should have a variable, an array element identifier, an expression, or a constant as its corresponding argument.
- An array dummy argument should have either an array name or an array element identifier as its corresponding actual argument. If the actual argument is an array, the length of the dummy array should be less than or equal to that of the actual array. FORTRAN associates each element of a dummy array directly with the corresponding elements of the actual array.
- A dummy argument representing an external function must have an external function as its actual argument.
- A dummy argument representing a subroutine identifier should have a subroutine name as its actual argument.
- You may define (or redefine) a dummy argument in a referenced subprogram only if its corresponding actual argument is a variable. If dummy arguments are array names, then you may redefine the elements of the array.

7.9.2 User-written Subprograms

FORTRAN transfers control to a function by means of a function reference. It transfers control to a subroutine by a CALL statement. A function reference is the name of the function, together with its arguments, appearing in an expression. A function always returns a value to the calling program. Both functions and subroutines may return additional values via assignment to their arguments. A subprogram can reference other subprograms, but it cannot, either directly or indirectly, reference itself (that is, FORTRAN is not recursive).

7.9.2.1 Arithmetic Statement Functions (ASE)

- Use an Arithmetic Statement Function to define a one statement, self-contained computational procedure.

Format:

nam ([[a [[,a]] ...]])=e

where:

nam is the name you assign to the ASF,
a is a dummy argument, and
e is an expression.

Examples:

```
PROOT(A,B,C) = (-B+SQRT(B**2 - 4*A*C))/(2*A)
NROOT(A,B,C) = (-B-SQRT(B**2 - 4*A*X))/(2*A)
```

An arithmetic statement function is a computing procedure which you define by a single statement, similar in form to an arithmetic assignment statement. The appearance of a reference to the function within the same program unit causes FORTRAN to perform the computation and make the resulting value available to the expression in which the ASF reference appears.

The expression e is an arithmetic expression that defines the computation to be performed by the ASF.

You reference an ASF in the same manner as an external function.

Format:

nam ([[a [[,a]] ...]])

where:

nam is the name of the ASF, and
a is an actual argument.

NOTE

You must define all ASFs before you type any executable statements.

When a reference to an arithmetic statement function appears in an expression, FORTRAN associates the values of the actual arguments with the dummy arguments in the ASF definition. FORTRAN then evaluates the expression in the defining statement and uses the resulting value to complete the evaluation of the expression containing the function reference.

Specify the data type of an ASF either implicitly by the initial letter of the name or explicitly in a type declaration statement.

Dummy arguments in an ASF definition only indicate the number, order, and data type of the actual arguments. You may use the same names to represent other entities elsewhere in the program unit. Note also that with the exception of data type, FORTRAN does not associate declarative information (such as placement in COMMON or declaration as an array) with the ASF dummy arguments. Note that you cannot use the name of the ASF to represent any other entity within the same program unit.

The expression in an ASF definition may contain function references.

Any reference to an ASF must appear in the same program unit as the definition of that function. You cannot use an ASF name in an EXTERNAL statement.

An ASF reference must appear as, or be part of, an expression; you must not use it as the left side of an assignment statement.

Actual arguments must agree in number, order, and data type with their corresponding dummy arguments. You must assign values to actual argument before the reference to the arithmetic statement function.

Examples:

Definitions	
VOLUME(RADIUS) = 4.189*RADIUS**3	
SINH(X) = (EXP(X)-EXP(-X))*0.5	
AVG(A,B,C,3.) = (A+B+C)/3.	(invalid; constant as dummy argument not permitted)

ASF References	
AVG(A,B,C) = (A+B+C)/3.	(definition)
.	
.	
GRADE = AVG(TEST1,TEST2,XLAB)	
IF (AVG(P,D,Q).LT.AVG(X,Y,Z))	GOTO 300
FINAL = AVG(TEST3,TEST4,LAB2)	(Invalid; data type of third argument does not agree with dummy argument)

7.9.2.2 **FUNCTION Subprogram** - A FUNCTION is an external computing procedure that returns a value. You use this value as an expression or as part of an expression.

Format:

[[typ]] FUNCTION nam(a [[,a...]])

where:

typ is an optional data type specifier,
nam is a name of the function, and
a is one of a maximum of six dummy arguments.

A FUNCTION subprogram is a program unit that consists of a FUNCTION statement followed by a series of statements that define a computing procedure. FORTRAN transfers control to a FUNCTION subprogram by a function reference and returns to the calling program unit when it encounters a RETURN statement.

You must always specify at least one argument to a FUNCTION. You may specify other arguments explicitly or place them in COMMON.

A FUNCTION subprogram returns a single value to the calling program unit by assigning that value to the function's name. FORTRAN determines the data type of the returned value by the function's name unless you have explictly specified the data type.

A function reference that transfers control to a FUNCTION subprogram has the form:

nam ([[a [[,a]] ...]])

where:

nam is the symbolic name of the function, and
a is an actual argument.

When FORTRAN transfers control to a function subprogram, FORTRAN associates the values you supply by the actual arguments (if any) with the dummy arguments (if any) in the FUNCTION statement. FORTRAN then executes the statements in the subprogram.

NOTE

You may not pass an array to a subprogram if it contains more than 2047 elements. You must implicitly pass larger arrays in COMMON.

You must assign a value to the name of the function before FORTRAN executes a RETURN statement in that function. When FORTRAN returns control to the calling program unit, it makes the value you have assigned to the function's name available to the expression that contains the function reference; it then uses this value to complete the evaluation of the expression.

NOTE

You can store variables that a FUNCTION requires in COMMON rather than passing them explicitly.

You may specify the type of a function name implicitly, explicitly in the FUNCTION statement, or explicitly in a type declaration statement.

The FUNCTION statement must be the first statement of a function subprogram. You may not label a FUNCTION statement.

A FUNCTION subprogram must not contain a SUBROUTINE statement, a BLOCK DATA statement, or a FUNCTION statement (other than the initial statement of the subprogram). A function may, however, call another function or subroutine so long as the call is not directly or indirectly recursive.

7.9.2.3 **SUBROUTINE Subprograms** - A SUBROUTINE is an external computing procedure that you may repeatedly call from a program or subprogram.

Format:

SUBROUTINE nam [[([[a [[,a]] ...]])]]

where:

nam is the name of the subroutine, and
a is a dummy argument.

A SUBROUTINE subprogram is a program unit that consists of a SUBROUTINE statement followed by a series of statements that define a computing procedure. FORTRAN transfers control to a SUBROUTINE subprogram by a CALL statement and returns to the calling program unit by a RETURN statement.

When FORTRAN transfers control to a subroutine, it associates the values you supply with the actual arguments (if any) in the CALL statement with the corresponding dummy arguments (if any) in the SUBROUTINE statement. You may not specify more than six arguments in a subroutine call. FORTRAN then executes the statements in the subprogram.

The SUBROUTINE statement must be the first statement of a subroutine; it must not have a statement label.

A SUBROUTINE subprogram cannot contain a FUNCTION statement, a BLOCK DATA statement, or a SUBROUTINE statement (other than the initial statement of the subprogram).

Example:

```
C     MAIN PROGRAM
      COMMON NFACES, EDGE, VOLUME
      READ (4,65) NFACES, EDGE
65    FORMAT(I2,F8.5)
      CALL PLYVOL
      WRITE (4,66) VOLUME
66    FORMAT (' VOLUME=',F)
      STOP
      END

      SUBROUTINE PLYVOL
      COMMON NFACES, EDGE, VOLUME
      CUBED = EDGE**3
      GOTO (6,6,6,1,6,2,6,3,6,6,6,4,6,6,6,6,6,6,6,5,6),NFACES
1     VOLUME = CUBED * 0.11785
      RETURN
2     VOLUME = CUBED
      RETURN
3     VOLUME = CUBED * 0.47140
      RETURN
4     VOLUME = CUBED * 7.66312
      RETURN
5     VOLUME = CUBED * 2.18170
      RETURN
6     WRITE (4,100) NFACES
100   FORMAT(' NO REGULAR POLYHEDRON HAS ',I3,'FACES.')
      RETURN
      END
```

The subroutine in this example computes the volume of a regular polyhedron, given the number of faces and the length of one edge. It uses a computed GOTO statement to determine whether the polyhedron is a tetrahedron, cube, octahedron, dodecahedron, or icosahedron, and to transfer control to the proper procedure for calculating the volume. If the number of faces of the body is other than 4, 6, 8, 12, or 20, the subroutine transmits an error message to logical unit 4 as indicated in the WRITE statement.

7.9.3 CALL Statement

The CALL statement causes the execution of a SUBROUTINE subprogram; it can also specify an argument list for use by the subroutine.

Format:

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

where:

s is the name of a SUBROUTINE subprogram, a user-written assembly language routine, or a DEC-supplied system subroutine, or a dummy argument associated with one of the above.

a is an actual argument.

The CALL statement associates the values in the argument list (if the list is present) with the dummy arguments in the subroutine and then transfers control to the first executable statement of the subroutine.

The arguments in the CALL statement must agree in number, order, and data type with the dummy arguments in the subroutine definition. They can be variables, arrays, array elements, constants, expressions, alphanumeric literals, or subprogram names (if those names have been specified in an EXTERNAL statement, as described in Section 5.4). Note that an unsubscripted array name in the argument list refers to the entire array.

Examples:

```
CALL CURVE (BASE,3.14159+X,Y,LIMIT,R(LT+2))
CALL PNTOUT (A,N,'ABCD')
```

7.9.4 RETURN Statement

Use the RETURN statement to return control from a subprogram unit to the calling program unit.

Format:

```
RETURN
```

When FORTRAN executes a RETURN statement in a FUNCTION subprogram, it returns control to the statement that contains the function reference (see Section XXX). When FORTRAN executes a RETURN statement in a SUBROUTINE subprogram, it returns control to the first executable statement following the CALL statement which initiated execution of the subprogram.

A RETURN statement must not appear in a main program unit.

Example:

```
      SUBROUTINE CONVRT (N,ALPH,DATA,PRNT,K)
      DIMENSION DATA(N), PRNT(N)
      IF (N .LT. 10) GOTO 100
      DATA(K+2) = N-(N/10)*N
      N = N/10
      DATA(K+1) = N
      PRNT(K+2) = ALPH(DATA(K+2)+1)
      PRNT(K+1) = ALPH(DATA(K+1)+1)
      RETURN
100   PRNT(K+2) = ALPH(N+1)
      RETURN
      END
```

7.9.5 FORTRAN Library Functions

The FORTRAN library functions are listed and described in Section 7.12. You write function references to FORTRAN library functions in the same form as function references to user-defined functions. For example,

```
R = 3.14159 * ABS(X-1)
```

causes the absolute value of X-1 to be calculated, multiplied by the constant 3.14159, and assigned to the variable R.

The data type of each library function and the data type of the actual arguments is specified in Appendix B. Arguments you pass to these functions may not be array names or subprogram names.

Processor-defined function references are local to the program unit in which they occur and do not affect or preclude the use of the name for any other purpose in other program units.

7.10 INPUT/OUTPUT STATEMENTS

You specify input of data to a program by READ statements and output by WRITE statements. You use some form of these statements in conjunction with format specifications to control translation and editing of the data between internal representation and character (readable) form.

Each READ or WRITE statement contains a reference to the logical unit to or from which data transfer is to take place. You may associate a logical unit to a device or file.

READ and WRITE statements fall into the following three categories:

- Unformatted Sequential I/O
  Unformatted sequential READ and WRITE statements transmit binary data without translation.

- Formatted Sequential I/O
  Formatted sequential READ and WRITE statements transmit character data using format specifications to control the translation of data to characters on output, and to internal form on input.

- Unformatted Direct Access I/O
  Unformatted direct access READ and WRITE statements transmit binary data without translation to and from direct access files.

The auxiliary I/O statements, REWIND and BACKSPACE do not perform data transfer, but perform file positioning. The ENDFILE statement writes a special record that will cause an end-of-file condition when read by a READ statement. The BACKSPACE statement repositions a file to the previous record. The DEFINE FILE statement declares a logical unit to be connected to a direct access file and specifies the characteristics of the file.

7 10.1 Defining I/O Operations

FORTRAN I/O operations require knowledge of logical unit numbers, format specifiers, and record transmission.

7.10.1.1 **Input/Output Devices and Logical Unit Numbers** - OS/78 FORTRAN uses the I/O services of the operating system and thus supports all peripheral devices that are supported by the operating system. I/O statements refer to I/O devices by means of logical unit numbers which are integer constants or variables with a positive value.

The default logical unit numbers are:

3 Line Printer
4 Terminal

The logical unit number must be in the range 1 through 9. For more information, see Sections 7.1.2.1 and 7.1.3.

7.10.1.2 **Format Specifiers** - Use format specifiers in formatted I/O statements. A format specifier is the statement label of a FORMAT statement. Section 7.11 discusses FORMAT statements.

7.10.1.3 **Input/Output Record Transmission** - I/O statements transmit data in terms of records. The amount of information that one record can contain, and the way in which records are separated, depend on the medium involved.

For unformatted I/O, specify the amount of data which FORTRAN will transmit by an I/O statement. FORTRAN determines the amount of information it will transmit by the I/O statement and by specifications in the associated format specification.

If an input statement requires only part of a record, the excess portion of the record is lost. In the case of formatted sequential input or output, you may transmit one or more additional records by a single I/O statement.

7.10.2 Input/Output Lists

An I/O list specifies the data items to be manipulated by the statement containing the list. The I/O list of an input or output statement contains the names of variables, arrays, and array elements whose values FORTRAN will transmit. In addition, the I/O list of an output statement can contain constants and expressions.

Format:

s [[,s]] ...

where:

s is a simple list or an implied DO list.

The I/O statement assigns input values to, or outputs values from, the list elements in the order in which they appear, from left to right.

7.10.2.1 **Simple Lists** - A simple I/O list consists of a single variable, array, array element, constant, or expression.

When an unsubscripted array name appears in an I/O list, a READ statement inputs enough data to fill every element of the array; a WRITE statement outputs all of the values contained in the array. Data transmission starts with the initial element of the array and proceeds in the order of subscript progression, with the leftmost subscript varying most rapidly. For example, if the unsubscripted name of a 2-dimensional array defined as:

```
DIMENSION ARRAY(3,3)
```

appears in a READ statement, that statement assigns values from the input record(s) to ARRAY(1,1), ARRAY(2,1), ARRAY(3,1), ARRAY(1,2), and so on, through ARRAY(3,3).

If, in a READ statement, you input the individual subscripts for an array, you must input the subscripts before their use in the array. If, for example, FORTRAN executes the statement:

```
      READ (1,1250) J,K,ARRAY(J,K)
1250 FORMAT (I1,X,I1,X,F6.2)
```

and the input record contains the values:

```
1,3,721.73
```

FORTRAN assigns the value 721.73 to ARRAY(1,3). FORTRAN assigns the first input value to J and the second to K, thereby establishing the actual subscript values for ARRAY(J,K). Variables that you use as subscripts in this way must appear to the left of their use in the array subscript.

You may use any valid expression in an output statement I/O list. However, the expression must not cause FORTRAN to attempt further I/O operations. A reference in an output statement I/O list expression to a FUNCTION subprogram that itself performs input/output is illegal.

You must not include an expression in an input statement I/O list except as a subscript expression in an array reference.

7.10.2.2 **Implied DO Lists** - Use an implied DO list to specify iteration within an I/O list.

Format:

```
(list,i=e1,e2)
```

where:

```
list is an I/O list,
i    is a control variable definition,
e1   is the initial value of i, and
e2   is the terminal value of i.
```

You use an implied DO list to specify iteration within an I/O list, to transmit only part of an array, or to transmit array elements in a sequence other than the order of subscript progression. The implied DO list functions as though it were a part of an I/O statement that resides in a DO loop.

When you use nested implied DO lists, the first control variable definition is equivalent to the innermost DO of a set of nested loops, and therefore varies most rapidly. For example, the statement:

```
      WRITE (5,150) ((FORM(K,L), L=1,10), K=1,10)
150   FORMAT (F10.2)
```

is similar to:

```
      DO 50 K=1,10
      DO 50 L=1,10
      WRITE (5,150) FORM(K,L)
150   FORMAT (F10.2)
50    CONTINUE
```

Since the inner DO loop is executed ten times for each iteration of the outer loop, the second subscript, L, advances from one through ten for each increment of the first subscript. This is the reverse of the order of subscript progression.

The implied DO uses the control variable of the imaginary DO statement to specify which value or values are to be transmitted during each iteration of the loop.

i, el, and e2 have the same form as that used in the DO statement. The rules for the control, initial, and terminal variables of an implied DO list are the same as those for the DO statement. Note, however, an implied DO loop cannot use an increment parameter. The list may contain references to the control variable as long as the value of the control variable is not altered. There is no extended range for an implied DO list.

Examples:

```
WRITE (3,200) (A,B,C,  I=1,3)

WRITE (6,15) L,M,(I,(J,P(I),Q(I,J),J=1,L),I=1,M)

READ (1,75) (((ARRAY(M,N,I), I=2,8), N=2,8), M=2,8)
```

FORTRAN transmits the entire list of the implied DO before the incrementation of the control variable. For example:

```
READ (3,999) (P(I), (Q(I,J), J=1,10), I=1,5)
```

assigns input values to the elements of arrays P and Q in the order:

```
P(1), Q(1,1), Q(1,2), ... , Q(1,10),
P(2), Q(2,1), Q(2,2), ... , Q(2,10),
 .      .       .            .
 .      .       .            .
P(5), Q(5,1), Q(5,2), ... , Q(5,10)
```

When processing multi-dimensional arrays, you may use a combination of a fixed subscript and subscript or subscripts that varies according to an implied DO. For example:

```
READ (3,5555) (BOX(1,J), J=1,10)
```

assigns input values to BOX(1,1) through BOX(1,10), then terminates without affecting any other element of the array.

It is also possible to output the value of the control variable directly, as in the statement:

```
WRITE (6,1111) (I, I=1,20)
```

which simply prints the integers 1 through 20.

7.10.3 Input/Output Forms

7.10.3.1 **Unformatted Sequential Input/Output** - FORTRAN provides two types of I/O--unformatted and formatted. Unformatted input and output is the transfer of data in internal (binary) format without conversion or editing. Use unformatted I/O when data output by a program is to be subsequently input by the same program (or a similar program). Unformatted I/O saves execution time because it eliminates the data conversion process, preserves greater precision in the external data, and usually conserves file storage space.

7.10.3.2 **Formatted Sequential Input/Output** - Use formatted input and output statements in conjunction with FORMAT statements to translate and edit data on output for ease of interpretation, and, on input, to convert data from external format to internal format.

7.10.3.3 **Unformatted Direct Access Input/Output** - Use unformatted direct access READ and WRITE statements to perform direct access I/O with a file on a direct access device. Use the DEFINE FILE statement to establish the number of records, and the size of each record, in a file to which FORTRAN will perform direct access I/O. Each direct access READ or WRITE statement contains an integer expression that specifies the number of the record to be accessed. The record number must not be less than one nor greater than the number of records you define for the file.

In OS/78 FORTRAN, the expression that specifies the record number can be of any type. FORTRAN converts it to integer type if necessary.

7.10.4 READ Statements

FORTRAN provides the following READ statements.

7.10.4.1 **Unformatted Sequential READ Statement** - Use unformatted sequential read statements to assign fields to a record without translating stored information into external form.

Format:

```
READ (u) [[list]]
```

where:

u is a logical unit number from 1 to 9, and
list is an I/O list.

The unformatted sequential READ statement inputs one unformatted record from a logical unit and assigns the fields of the record without translation to the I/O list elements in the order in which they appear, from left to right.

An unformatted sequential READ statement transmits exactly one record. If the I/O list does not use all of the values in the record, FORTRAN discards the remainder of the record. If FORTRAN exhausts the contents of the record before the I/O list is satisfied, an error condition results.

You must only use the unformatted sequential READ statement to read records that were created by unformatted sequential WRITE statements.

If you use an unformatted WRITE statement that does not contain an I/O list, FORTRAN skips the next record.

Examples:

READ (1) FIELD1, FIELD2	Read one record from logical unit 1; assign values to variables FIELD1 and FIELD2.
READ (8)	Advance logical unit 8 one record.

7.10.4.2 **Formatted Sequential READ Statement** - Use formatted sequential read statements to transmit information in external format.

Format:

```
READ (u,f) [[ list ]]
```

where:

```
u     is a logical unit number from 1 to 9,
f     is a format statement number, and
list  is an I/O list.
```

The formatted sequential READ statement transfers data from the indicated logical unit. FORTRAN converts transmitted characters to internal format as specified by the format specification. FORTRAN assigns the resulting values to the elements of the I/O list.

If the FORMAT statement associated with a formatted input statement contains a Hollerith constant or alphanumeric literal, input data will be read and stored directly into the format specification. For example, the statements

```
      READ (5,100)
100   FORMAT (5H DATA)
```

cause five characters to be read and stored in the Hollerith format descriptor. If the character string were HELLO, statement 100 would become:

```
100   FORMAT (5HHELLO)
```

If there is no H field, the record is skipped.

If the number of elements in the I/O list is less than the number of fields in the input record, the excess portion of the record is discarded. If the number of elements in the list exceeds the number of input fields, an error condition results unless the format specifications state that one or more additional records are to be read (see Section 10.8).

If no I/O list is present, data transfer is between the record and the format specification.

Examples:

```
      READ (1,300) ARRAY
300   FORMAT (20F8.2)
```

Read a record from logical unit 1, assign fields to ARRAY.

```
      READ (5,50)
50    FORMAT (25H PAGE HEADING GOES HERE)
```

Read 25 characters from logical unit 5, place them in the FORMAT statement.

The CHKEOF subroutine returns a non-zero value if the logical end of a file is encountered during a formatted READ operation.

CHKEOF accepts one real, integer, or logical argument. After the next formatted READ operation, this argument will be set to a non-zero value if the logical end-of-file was encountered. Otherwise, it will be set to zero.

Only use CHKEOF when reading one record from the logical unit.

The fol lowing is an example of the use of CHKEOF.

```
      .
      .
      .
      CALL CHKEOF(EOF)
      READ (N,101)DATA
      IF (EOF .NE. 0) GO TO 9999
      .
      .
      .
```

7.10.4.3 **Unformatted Direct Access READ Statement** - Use an unformatted direct access read statement to transmit a value or values to a direct access device in internal format.

Format:

```
READ (u'r) [[list]]
```

where:

```
u    is a logical unit number from 1 to 9,
r    is the record number, and
list is an I/O list.
```

The unformatted direct access READ statement positions the input file to a specified record and transfers the fields in that record to the elements in the I/O list without translation.

u may be an unsigned integer constant or a positive integer variable. r may also be a variable. If there are more fields in the input record than elements in the I/O list, FORTRAN discards the excess portion of the record. If there is insufficient data in the record to satisfy the requirements of the I/O list, an error condition results.

The unit number in the unformatted direct access READ statement must refer to a unit that you have previously

Examples:

READ (1'10) LIST(1),LIST(8)	Read record 10 of a file on logical unit 1, assign two INTEGER values to specified elements of array LIST.
READ (4'58) (RHO(N),N=1,5)	Read record 58 of a file on logical unit 4, assign five real values to array RHO.

7.10.5 WRITE Statements

7.10.5.1 **Unformatted Sequential WRITE Statement** - FORTRAN includes the following formatted and unformatted WRITE statements. Use as unformatted sequential write statement to transmit values in their internal representation to a logical unit.

Format:

```
WRITE (u) [[ list ]]
```

where:

u is a logical unit number from 1 to 9, and
list is an I/O list.

The unformatted sequential WRITE statement transmits the values of the elements in the I/O list to the specified logical unit, without translation, as one unformatted record.

The logical unit specifier is an integer variable or an integer constant from 1 to 9.

If an unformatted WRITE statement contains no I/O list, one null record is output to the specified unit.

A record may hold 85 single precision variables. If the list elements fill more than one record, FORTRAN writes successive records until the list is completed. Thus, if there are 100 variables on the list, FORTRAN uses two records; one record contains 85 variables and the second contains 15 variables. For example

```
DIMENSION X(200)
WRITE (6) X
```

will produce three records on logical unit 6, the first containing X(1) to X(85), the second X(86) to X(170), and the third X(171) to X(200). If the amount of data FORTRAN will transmit exceeds the record size, an error condition results. If the WRITE statement does not completely fill the record with data, FORTRAN zero fills the unused portion of the record.

Examples:

```
WRITE (1) (LIST(K),K=1,5)      Output the contents of elements 1
                               through 5 of array LIST to logical
                               unit 1.

WRITE (4)                      Write a null record on logical unit
                               4.
```

7.10.5.2 **Formatted Sequential WRITE Statement** - Use a formatted sequential write statement to translate a value from its internal representation to character format and then transmit it to a logical unit.

Format:

WRITE (u,f) [[list]]

where:

u is a logical unit number from 1 to 9,
f is a format statement number, and
list is an I/O list.

The formatted sequential WRITE statement transfers data to the specified logical unit. The I/O list specifies a sequence of values which FORTRAN converts to characters and positions as specified by a format specification.

The logical unit specifier may be an integer variable.

If no I/O list is present, data transfer is entirely between the record and the format specification.

The data FORTRAN transmits by a formatted sequential WRITE statement normally constitutes one formatted record. The format specification can, however, specify that additional records are to be written during the execution of that same WRITE statement.

FORTRAN rounds numeric data output under format control during the conversion to external format. (If such data is subsequently input for additional calculations, loss of precision may result. In this case, unformatted output is preferable to formatted output.)

The records FORTRAN transmits by a formatted WRITE statement must not exceed the length that the specified device can accept. For example, a line printer typically cannot print a record that is longer than 132 characters. [If longer, run to next line]

Examples:

```
     WRITE (6, 650)                 (Output the contents of the
650  FORMAT (' HELLO, THERE')       FORMAT statement to logical
                                    unit 6.)

     WRITE (1,95) AYE, BEE, CEE     (Write one record of three
95   FORMAT (F8.5, F8.5, F8.5)      fields to logical unit 1.)

     WRITE (1,950) AYE, BEE, CEE    (Write three separate records
950  FORMAT (f8.5)                  of one field each to logical
                                    unit 1.)
```

In the last example, format control arrives at the rightmost parenthesis of the FORMAT statement before all elements of the I/O list have been output. Each time this occurs, FORTRAN terminates the current record and initiates a new record. Thus, FORTRAN writes three separate records. (See Section 10.5.)

7.10.5.3 **Unformatted Direct Access WRITE Statement** - Use an unformatted direct access write statement to transmit a value in its internal representation to a specific record on a direct access device.

Format:

```
WRITE (u'r) [[list]]
```

where:

```
u     is a logical unit number from 1 to 9,
r     is the record number, and
list  is an I/O list.
```

The unformatted direct access WRITE statement transmits the values of the elements in the I/O list to a particular record position on a direct access file. The data is written in internal format without translation.

The logical unit specifier r may be an unsigned integer constant or integer variable.

A record may hold a maximum of 85 single precision variables. If the list elements fill more than one record, FORTRAN writes suçcessive records until the list is completed. Thus, if there are 100 variables on the list, FORTRAN uses two records; one record contains 85 variables and the second contains 15 variables. For example

```
DIMENSION X(200)
WRITE (6) X
```

will produce three records on unit 6, the first containing X(1) to X(85), the second X(86) to X(170), and the third X(171) to X(200). If the amount of data FORTRAN will transmit exceeds the record size, an error condition results. If the WRITE statement does not completely fill the record with data, FORTRAN zero fills the unused portion of the record.

Examples:

`WRITE (2'35) (NUM(K),K=1,10)`	(Output ten integer values to record 35 of the file connected to logical unit 2.)
`WRITE (3'J) ARRAY`	(Output the entire contents of ARRAY to the file connected to logical unit 3 into the record indicated by the value of J.)

7.10.6 Auxiliary Input/Output Statements

You use statements in this category to perform file management functions.

7.10.6.1 **BACKSPACE Statement** - Use the BACKSPACE statement to reposition a file to the previous record accessed.

Format:

```
BACKSPACE u
```

where:

u is a logical unit number from 1 to 9.

The BACKSPACE statement repositions a currently open sequential file backward one record and repositions it to the beginning of that record. On the execution of the next I/O statement for that unit, that record is available for processing.

The unit number must refer to a directory-structured device (e.g., disk). A file must be open on that device.

If the file is positioned at the first record, FORTRAN ignores the BACKSPACE statement.

Example:

```
BACKSPACE 4     (Reposition open file on logical unit 4 to
                beginning of the previous record.)
```

7.10.6.2 **DEFINE FILE Statement** - The DEFINE FILE statement establishes the size and structure of a file upon which FORTRAN will perform direct access I/O.

Format:

```
DEFINE FILE u (m,n,U,v) [[ ,u(m,n,U,v) ]] ...
```

where:

- u is an integer constant or variable that specifies the logical unit number,
- m is an integer constant or variable that specifies the number of records in the file,
- n is an integer constant or variable that specifies the length, in words, of each record,
- U specifies that the file is unformatted (binary) and the letter U is the only acceptable entry in this position, and
- v is an integer variable, called the associated variable of the file.

The DEFINE FILE is the means by which you specify the attributes of a direct access device. Once the file characteristics have been established, you should always specify them in the same manner.

At the conclusion of each direct access I/O operation, FORTRAN assigns the record number of the next higher numbered record in the file to v.

The DEFINE FILE statement specifies that a file containing m fixed-length records of n words each exists or is to exist, on logical unit u. The records in the file are sequentially numbered from 1 through m.

You must type the DEFINE FILE statement before the first direct access I/O statement that refers to the specified file.

The DEFINE FILE statement also establishes the integer variable v as the associated variable of the file. At the end of each direct access I/O operation, the FORTRAN I/O system places in v the record number of the record immediately following the one just read or written. Because the associated variable always points to the next sequential record in the file (unless you redefine it by an assignment or input statement), you can use direct access I/O statements to perform sequential processing of the file. The logical unit number u cannot be passed as a dummy argument to a DEFINE FILE statement in a subroutine.

If more than one program unit processes the file, or in an overlay environment, the associated variable should be placed in a resident common block.

Example:

```
DEFINE FILE 3 (1000,48,U,NREC)
```

This statement specifies that logical unit 3 is to be connected to a file of 1000 fixed-length records, each record of which is 48 words long. The records are numbered sequentially from 1 through 1000, and are unformatted. After each direct access I/O operation on this file, the integer variable NREC will contain the record number of the record immediately following the one just processed.

7.10.6.3 **ENDFILE Statement** - The ENDFILE statement writes an end-file record to the specified sequential unit.

Format:

```
ENDFILE u
```

where:

u is a logical unit number from 1 to 9.

Use the ENDFILE statement to write an end-of-file mark on a directory-structured device. [Note that you cannot write additional information to that device after the ENDFILE statement.]

The ENDFILE statement must be written to a formatted output file.

No rewind occurs after this statement.

Example:

```
ENDFILE 2    (Output an end-file record to logical unit 2.)
```

7.10.6.4 **REWIND Statement** - The REWIND statement repositions a currently open sequential file to be repositioned to the beginning of the file.

Format:

```
REWIND u
```

where:

u is a logical unit number from 1 to 9.

Use the REWIND statement to position a directory-structured device to its first record.

If the file is already at its first record, FORTRAN ignores the REWIND statement.

The unit number in the REWIND statement must refer to a directory-structured device (e.g., disk). A file must be open on that device.

Example:

```
REWIND 3  (Reposition logical unit 3 to beginning of currently
           open file.)
```

7.11 FORMAT STATEMENTS

FORMAT statements are nonexecutable statements used in conjunction with formatted I/O statements. The FORMAT statement describes the format in which FORTRAN transmits data fields, and the data conversion and editing to be performed to achieve that format.

The FORMAT statement has the form:

st FORMAT (glflsl [[f2s2]] ... [[fnqn]])

where:

- f is a field descriptor, or a group of field descriptors enclosed in parentheses,
- s is a field separator (either a comma or slash),
- q is zero or more slash (/) record terminators
- st is a mandatory statement number.

including the parentheses is called the format specification. You must enclose the list in parentheses.

A field descriptor in a format specification has the form:

[[r]] cw [[.d]]

where:

- r represents a repeat count which specifies that FORTRAN is to apply the field descriptor to r successive fields. If you omit the repeat count, FORTRAN assumes it to be 1.
- c is a format code,
- w is the field width, and
- d is the number of characters to the right of the decimal point, and should be less than w.

The terms r, w, and d must all be unsigned integer constants less than or equal to 255.

The field separators are comma and slash. A slash can also be a record terminator. Use a slash to skip records or lines in a record.

The field descriptors used in format specifications are as follows:

1. Integer: Iw

2. Logical: Lw

3. Real: Fw.d, Ew.d, Dw.d, Gw.d, Bw.d

4. Literal and editing: Aw, nH, nP, nX, Tn, $, '...', /

(In the alphanumeric and editing field descriptors, n specifies the number of characters or character positions.)

You can precede the F, E, D, or G field descriptors by a scale factor of the form:

np

where:

n is an optionally signed integer constant in the range -127 to +127 the scale factor specifies the number of positions the decimal point is to be scaled to the left or right.

During data transmission, FORTRAN scans the format specification from left to right. FORTRAN then performs data conversion by correlating the values in the I/O list with the corresponding field descriptors. In the case of H field descriptors and alphanumeric literals, data transmission takes place entirely between the field descriptor and the external record.

For example, consider the following data for input (where b equals a blank space).

```
b10.2bb6732bb3967.61
```

To read this data, use the following FORMAT statement in conjunction with a READ statement.

```
20 FORMAT (1X,F3.1,2X,I4,2X,F6.2)
```

where the field descriptor:

1X	Indicates a blank space.
F3.1	Indicates a 3-digit real number with 1 decimal place.
2X	Indicates 2 blank spaces.
I4	Indicates a 4-digit integer number.
2X	Indicates 2 blank spaces.
F6.2	Indicates a 6-digit real number with 2 decimal places.

7.11.1 Field Descriptors

The individual field descriptors that can appear in a format specification are described in detail in the following sections. The field descriptors ignore leading spaces in the external field, but treat embedded and trailing spaces as zeros.

7.11.1.1 **I Field Descriptor** - The I field descriptor governs the translation of integer data.

Format:

Iw

Input

The I field descriptor causes an input statement to read w characters from an external record. FORTRAN then assigns the character as an integer value to the corresponding integer element of the I/O list. The external data must be an integer; it must not contain a decimal point or exponent field.

The I field descriptor interprets an all-blank field as a zero value.

If the value of the external field exceeds the range of the corresponding integer list element, an error occurs. If the first non-blank character of the external field is a minus symbol, the I field descriptor causes the field to be stored as a negative value; FORTRAN treats a field preceded by a plus symbol, or an unsigned field, as a positive value.

Examples:

Format	External Field	Internal Representation
I4	2788	2788
I3	-26	-26
I9	312	312
I9	3.12	not permitted; error
I3	-871	-87 (one is lost)

Output

On output, the I field descriptor transmits the value of the corresponding integer I/O list element, right justified, to an external field w characters in length. It also replaces any leading zeros with spaces. If the value does not fill the field, FORTRAN inserts leading spaces. If the value of the list element is negative, the field will have a minus symbol as its leftmost non-blank character. Space must therefore be included in w for a minus symbol if you expect one to be output. FORTRAN suppresses plus symbols and you need not account for them in w. If w is too small to contain the output value, FORTRAN fills the entire external field with asterisks.

Examples:

Format	Internal Value	External Representation
I3	284	284
I4	-284	-284
I5	174	174
I2	3244	**
I3	-473	***
I7	29.812	not permitted; error

7.11.1.2 **F Field Descriptor** - The F field descriptor specifies the data conversion and editing of real values.

Format:

Fw.d

Input

On input, the F field descriptor causes FORTRAN to read w characters from the external record and to assign the characters as a real value to the corresponding I/O list element. If the first non-blank character of the external field is a minus sign, FORTRAN treats the field as a negative value; FORTRAN assumes a field preceded by a plus sign (or an unsigned field) to be positive. FORTRAN considers an all-blank field to have a value of zero. In all appearances of the F field descriptor, w must be greater than or equal to d+ where the extra character is the decimal point.

If the field contains neither a decimal point nor an exponent, FORTRAN treats it as a real number of w digits, in which the rightmost d digits are to the right of the decimal point. If the field contains an explicit decimal point, the location of that decimal point overrides the location you specify in the field descriptor. If the field contains an exponent, FORTRAN uses the exponent to establish the magnitude of the value before it assigns the value to the list element.

Examples:

Format	External Field	Internal Representation
F8.5	123456789	123.45678
F8.5	-1234.567	-1234.56
F8.5	24.77E+2	2477.0
F5.2	1234567.89	123.45

Output

On output, the F field descriptor causes FORTRAN to round the value of the corresponding I/O list element to d decimal positions and to transmit an external field w characters in length, right justified. If the converted data consists of fewer than w characters, FORTRAN inserts leading spaces; if the data exceeds w characters, FORTRAN fiils the entire field with asterisks.

The field width must be large enough to accommodate 1) a minus sign, if you expect one to be output (FORTRAN suppresses plus signs), 2) at least one digit to the left of the decimal point, 3) the decimal point itself, and 4) d digits to the right of the decimal. For this reason, w should always be greater than or equal to (d+3).

Examples:

Format	Internal Value	External Representation
F8.5	2.3547188	2.35472
F9.3	8789.7361	8789.736
F10.4	-23.24352	-23.2435
F5.2	325.013	*****
F5.2	-.2	-0.20

7.11.1.3 **E Field Descriptor** - The E field descriptor specifies the transmission of real values in exponential format.

Format:

Ew.d

Input

The E field descriptor causes an input statement to input w characters from an external record. It interprets and assigns that data in exactly the same way as the F field descriptor.

Examples:

Format	External Field	Internal Representation
E9.3	734.432E3	734432.0
E12.4	1022.43E-6	1022.43E-6
E15.3	52.37596	52.37596

Output

The E field descriptor causes an output statement to transmit the value of the corresponding list element to an external field w characters in width, right justified. If the number of characters in the converted data is less than w, FORTRAN inserts leading spaces; if the number of characters exceeds w, FORTRAN fills the entire field with asterisks. The corresponding I/O list element must be of real type.

FORTRAN transmits data output under control of the E field descriptor in a standard form, consisting of

1. a minus sign if the value is negative (plus signs are suppressed),
2. a zero,
3. a decimal point,
4. d digits to the right of the decimal, and
5. a 3-character exponent of the form:

   E+nnn

   or

   E-nnn

where:

nn is a 2-digit integer constant.

The d digits to the right of the decimal point represent the entire value, scaled to a decimal fraction.

Because w must be large enough to include a minus sign (if any are expected), a zero, a decimal point, and an exponent, in addition to d digits, w should always be equal to or greater than d+7.

Examples:

Format	Internal Value	External Representation
E9.2	475867.222	0.48E+06
E12.5	475867.222	0.47587E+06
E12.3	0.00069	0.690E-03
E10.3	-0.5555	-0.556E+00
E5.3	56.12	*****

7.11.1.4 **G Field Descriptor** - The G field descriptor transmits real data in a form that is in effect a combination of the F and E field descriptors.

Format:

Gw.d

Input

On input, the G field descriptor functions identically to the F field descriptor.

Output

On output, the G field descriptor causes FORTRAN to transmit the value of the corresponding I/O list element to an external field w characters in length, right justified. The form in which the value is output is a function of the magnitude of the value, as described in Table 7-19.

Table 7-19
Effect of Data Magnitude on G Format Conversions

Data Magnitude	Effective Conversion
m < 0.1	Ew.d
0.1 < m < 1.0	F(w-4).d, 4X
1.0 < m < 10.0	F(w-4).(d-1), 4X
.	.
.	.
.	.
10d-2 < m < 10d-1	F(w-4).1, 4X
10d-1 < m < 10d	F(w-4).0, 4X
m > 10d	Ew.d

The 4X field descriptor is inserted by the G field descriptor for values within its range, and means that four spaces are to follow the numeric data representation.

The field width, w, must include

1. space for a minus sign, if any are expected (plus signs are suppressed),
2. at least one digit to the left of the decimal point,
3. the decimal point itself,
4. d digits to the right of the decimal, and
5. (for values that are outside the effective range of the G field descriptor) a 4-character exponent.

Therefore, w should always be equal to or greater than d+7.

Examples:

Format	Internal Value	External Representation
G13.6	0.01234567	0.123457E-01
G13.6	-0.12345678	-0.123457
G13.6	1.23456789	1.23457
G13.6	12.34567890	12.3457
G13.6	123.45678901	123.457
G13.6	-1234.56789012	-1234.57
G13.6	12345.67890123	12345.7
G13.6	123456.78901234	123457.
G13.6	-1234567.89012345	-0.123457E+07

For comparison, consider the following example of the same values output under the control of an equivalent F field descriptor.

Format	Internal Value	External Representation
F13.6	0.01234567	0.012346
F13.6	-0.12345678	-0.123457
F13.6	1.23456789	1.234568
F13.6	12.34567890	12.345679
F13.6	123.45678901	123.456789
F13.6	-1234.56789012	-1234.567890
F13.6	12345.67890123	12345.678901
F13.6	123456.78901234	123456.789012
F13.6	-1234567.89012345	**********

NOTE

Only the first 6 digits in external representation are accurate.

7.11.1.5 **L Field Descriptor** - The L field descriptor specifies the transmission of logical data.

Format:

Lw

Input

The L field descriptor causes an input statement to read w characters from external record. If the first non-blank character of that field is the letter T or the string .T, FORTRAN assigns the value .TRUE. to the corresponding I/O list element. (The corresponding I/O list element must be of logical type.) If the first non-blank character of the field is the letter F or the string .F, or if the entire field is blank, FORTRAN assigns the value .FALSE. . Any other value in the external field causes an error condition.

Output

The L field descriptor causes an output statement to transmit either the letter T, if the value of the corresponding list element is .TRUE. or the letter F, if the value is .FALSE., to an external field w characters wide. The letter T or F is in the rightmost position of the field, preceded by w-1 spaces.

Examples:

Format	Internal Value	External Representation
L5	.TRUE.	T
L1	.FALSE.	F

7.11.1.8 **A Field Descriptor** - The A field descriptor specifies the transmission of alphanumeric data.

Format:

Aw

Input

On input, the A field descriptor causes w characters to be read from the external record and stored in ASCII format in the corresponding I/O list element. (The corresponding I/O list element may be of any data type.) The maximum number of characters that FORTRAN can store in a variable or array element depends on the data type of that element, as listed in Table 7-20.

Table 7-20
Character Storage

I/O List Element	Maximum Number of Characters
Logical	6
Integer	6
Real	6

If w is greater than the maximum number of characters that FORTRAN can store in the corresponding I/O list element, only the rightmost six characters are assigned to that entity; the leftmost excess characters are lost. If w is less than the number of characters that FORTRAN can store, it assigns w characters to the list element, left justified, and adds trailing spaces to fill the variable or array element.

Examples:

Format	External Field	Internal Representation
A6	PAGE #	PAGE # (Integer)
A6	PAGE #	GE # (Real)

Output

On output, the A field descriptor causes FORTRAN to transmit the contents of the corresponding I/O list element to an external field w characters wide. If the list element contains fewer than w characters, the data appears in the field right-justified with leading spaces. If the list element contains more than w characters, FORTRAN transmits only the leftmost w characters.

Examples:

Format	Internal Value	External Representation
A5	OHMS	OHMS
A5	VOLTS	VOLTS
A5	AMPERES	AMPER

7.11.1.7 H Field Descriptor

Format:

nHccc...c

where:

- n specifies the number of characters that are to be transmitted, and
- c is an ASCII character.

Input

When the H field descriptor appears in a format specification, data transmission takes place between the external record and the field descriptor itself.

The H field descriptor causes an input statement to read n characters from the external record and to place them in the field descriptor, with the first character appearing immediately after the letter H. FORTRAN replaces any characters that had been in the field descriptor prior to input by the input characters.

Output

The H field descriptor causes an output statement to transmit the n characters in the field descriptor following the letter H to the external record. An example of the use of H field descriptors for input and output follows:

```
      WRITE (4,100)
100   FORMAT (41H ENTER PROGRAM TITLE, UP TO 20 CHARACTERS)
      READ (4,200)
200   FORMAT (20H  TITLE GOES HERE    )
```

The WRITE statement transmits the characters from the H field descriptor in statement 100 to the user's terminal. The READ statement accepts the response from the keyboard, placing the input data in the H field descriptor in statement 200. The new characters replace the string TITLE GOES HERE; if the user enters fewer than 20 characters, FORTRAN fills the remainder of the H field descriptor with spaces to the right.

7.11.1.8 **Alphanumeric Literals** - You may use an alphanumeric literal in place of an H field descriptor. For output, both types of format specifiers function identically. However, you cannot use an alphanumeric literal on input.

You write an apostrophe character within an alphanumeric literal as two apostrophes. For example:

```
50 FORMAT (' TODAY''S DATE IS:  ',I2,'/',I2,'/',I2)
```

FORTRAN treats a pair of apostrophes used in this manner to be a single character.

7.11.1.9 **X Field Descriptor** - The X field descriptor causes spaces to be skipped in a record.

Format:

```
nX
```

Input

The X field descriptor causes an input statement to skip over the next n characters in the input record.

Output

The X field descriptor causes an output statement to transmit n spaces to the external record. For example:

```
      WRITE (5,90) NPAGE
90    FORMAT (13H1PAGE NUMBER ,I2,16X,23HGRAPHIC ANALYSIS, CONT.)
```

The WRITE statement prints a record similar to:

```
PAGE NUMBER nn            GRAPHIC ANALYSIS, CONT.
```

where "nn" is the current value of the variable NPAGE. FORTRAN does not print the numeral 1 in the first H field descriptor, instead using it to advance the printer paper to the top of a new page. Printer carriage control is explained in Section 10.5.

7.11.1.10 **T Field Descriptor** - The T field descriptor is a tabulation specifier.

Format:

```
Tn
```

where:

n indicates the character position of the external record. The value of n must be greater than or equal to one, but not greater than the number of characters allowed in the external record.

Input

On input, the T field descriptor causes FORTRAN to position the external record to its nth character position. For example, if a READ statement inputs a record containing:

```
ABC   XYZ
```

under control of the FORMAT statement:

```
10   FORMAT (T7,A3,T1,A3)
```

the READ statement would input the characters XYZ first, then the characters ABC.

Output

On output to devices other than the line printer or terminal, the T field descriptor states that subsequent data transfer is to begin at the nth character position of the external record. For output to a printing device, data transfer begins at position n-1). This is because FORTRAN reserves the first position of a printed record for a carriage control character (see Section 10.5) which is never printed.

Example the statements:

```
      WRITE(4,25)
25    FORMAT (T51,'COLUMN 2',T21,'COLUMN 1')
```

would cause the following line to be printed:

```
              Position 20                    Position 50

                   COLUMN 1                       COLUMN 2
```

7.11.1.11 **$ Descriptor** - The character $ (dollar sign) appearing in a format specification modifies the carriage control specified by the first character of the record. The $ descriptor is intended primarily for interactive I/O and causes the terminal print position to be left at the end of the text written (rather than returned to the left margin) so that a typed response will appear on the same line following the output.

Example:

```
          A=5
          WRITE (4,100) A
          READ (4,200) B
100       FORMAT (' SAMPLE NO.', I2, ' IS: ',$)
200       FORMAT (A6)
          WRITE (4,200) B
          END
```

This program outputs

```
SAMPLE NO.  5 IS:  RED
RED
```

7.11.2 Scale Factor

You can alter the location of the decimal point in real values during input or output through the use of a scale factor.

Format:

nP

where:

n is a signed or unsigned integer constant in the range -127 to +127 specifying the number of positions the decimal point is to be moved to the right or left.

You may place a scale factor anywhere in a format specification, but it must precede the field descriptors with which it is to be associated. It has the forms:

nPFw.d nPEw.d nPGw.d

Data input under control of one of the above field descriptors is multiplied by 10**-n before FORTRAN assigns it to the corresponding I/O list element. For example, a 2P scale factor multiplies an input value by .01, moving the decimal point two places to the left; a -2P scale factor multiplies an input value by 100, moving the decimal point two places to the right. If the external field contains an explicit exponent, however, the scale factor has no effect.

Examples:

Format	External Field	Internal Representation
3PE10.5	37.614	.037614
3PE10.5	37.614E2	3761.4
-3PE10.5	37.614	37614.

The effect of the scale factor on output depends on the type of field descriptor with which it is associated. For the F field descriptor, FORTRAN multiplies the value of the I/O list element by 10**N before it transmits it to the external record. Thus, a positive scale factor moves the decimal point to the right; a negative scale factor moves the decimal point to the left.

FORTRAN adjusts values output under control of an E or D field descriptor with a scale factor by multiplying the basic real constant portion of each value by 10**N and subtracting n from the exponent. Thus a positive scale factor moves the decimal point to the right and decreases the exponent; a negative scale factor moves the decimal point to the left and increases the exponent.

FORTRAN suspends the effect of the scale factor while the magnitude of the data to be output is within the effective range of the G field descriptor, since G supplies its own scaling function. The G field descriptor functions as an E field descriptor when the magnitude of the data value is outside its range; the effect of the scale factor is therefore the same as described for that field descriptor.

Note that on input, and on output under control of an F field descriptor, a scale factor actually alters the magnitude of the data; on output, a scale factor attached to an E or G field descriptor merely alters the form in which the data is transmitted. Note also that on input a positive scale factor moves the decimal point to the left and a negative scale factor moves the decimal point to the right, while on output the effect is just the reverse.

If you do not attach a scale factor to a field descriptor, FORTRAN assumes a scale factor of zero. Once you specify a scale factor, however, it applies to all subsequent real field descriptors in the same format specification, unless another scale factor appears. You may only reinstate a scale factor of zero by an explicit 0P specification.

Some examples of scale factor effect on output are:

Format	Internal Value	External Representation
1PE12.3	-270.139	-2.701E+02
1PE12.2	-270.139	-2.70E+02
-1PE12.2	-270.139	-0.03E+04

7.11.3 Grouping and Group Repeat Specifications

You can apply any field descriptor (except H, T, P, or X) to a number of successive data fields by preceding that field descriptor with an unsigned integer constant, called a repeat count, that specifies the number of repetitions. For example, the statements:

```
20    FORMAT (E12.4,E12.4,E12.4,I5,I5,I5,I5)
```

and

```
20    FORMAT (3E12.4,4I5)
```

have the same effect.

Similarly, you may repeatedly apply a group of field descriptors to data fields by enclosing those field descriptors in parentheses, with an unsigned integer constant, called a group repeat count, preceding the left parenthesis. For example:

```
50    FORMAT (2I8,3(F8.3,E15.7))
```

is equivalent to:

```
50    FORMAT (I8,I8,F8.3,E15.7,F8.3,E15.7,F8.3,E15.7)
                              1          2          3
```

You can enclose an H or X field descriptor, which could not otherwise be repeated, in parentheses. FORTRAN then treats it as a group repeat specification, thus allowing it to be repeated a desired number of times.

If you omit a group repeat count, FORTRAN assumes it to be 1.

7.11.4 Carriage Control

FORTRAN never transmits the first character of a record to a printing device; instead, FORTRAN interprets this first character as a carriage control character. The FORTRAN I/O system recognizes certain characters for this purpose; the effects of these characters are shown in Table 7-21.

Table 7-21
Carriage Control Characters

Character	Effect
space	Advance one line
0 zero	Advance two lines
1 one	Advances to top of next page
+ plus	Do not advance (allows overprinting)

FORTRAN treats any character other than those described in Table 10-2 as though it is a space, and deletes it from the print line.

7.11.5 Format Specification Separators

You generally separate field descriptors in a format specification from one another by commas. You may also use the slash (/) record terminator to separate field descriptors. A slash causes FORTRAN to terminate the input or output of the current record and to initiate a new record.

You may omit the comma when using a slash. Also, you need not type a comma after a Hollerith constant.

Example:

```
      WRITE (5,40) K,L,M,N,O,P
40    FORMAT (3A6/I6,2F8.4)
```

is equivalent to:

```
      WRITE (5,40) K,L,M
40    FORMAT (3A6)
      WRITE (5,50) N,O,P
50    FORMAT (I6,2F8.4)
```

It is possible to bypass input records or to output blank records by the use of multiple slashes. If n consecutive slashes appear between two field descriptors, they cause FORTRAN to skip n-1 records on input or n-1 blank records to be output. (The first slash terminates the current record; the second slash terminates the first skipped or blank record, and so on.) If n slashes appear at the beginning or end of a format specification, however, they result in n skipped or blank records, because the initial and terminal parentheses of the format specification are themselves a record initiator and record terminator, respectively. An example of the use of multiple record terminators is:

```
      WRITE (5,99)
99    FORMAT ('1'T51'HEADING LINE'//T51'SUBHEADING LINE'//)
```

The above statements output the following:

```
                    Column 50, top of page

                                HEADING LINE
(blank line)
                                SUBHEADING LINE
(blank line)
(blank line)
```

7.11.6 Short Field Termination

A field descriptor such as fw.d specifies that an input statement is to read w characters from the external record. If the data field in question contains fewer than w characters, the input statement would read some characters from the following field unless the short field were padded with leading zeros or spaces. To avoid the necessity of doing so, you may terminate an input field containing fewer than w characters by a comma. The comma overrides the field descriptor's field width specification. This practice, called short field termination, is particularly useful when entering data from a terminal keyboard. You may also use it in conjunction with I, F, E, D, G, and L field descriptors.

Examples:

```
      READ (6,100) I,J,A,B
100   FORMAT (2I6,2F10.2)
```

If the external record input by the above statements contains:

```
1,-2,1.0,35
```

Then the following assignments take place:

```
I = 1

J = -2

A = 1.0

B = 0.35
```

Note that the physical end of the record also serves as a field terminator. Note also that the d part of a w.d specification is not affected as illustrated by the assignment to B.

You may only terminate fields of fewer than w characters by a comma. If you follow a field of w characters or greater by a comma, FORTRAN will consider the comma to be part of the following field.

Two successive commas, or a comma following a field of exactly w characters, constitutes a null (zero-length) field. Depending on the field descriptor in question, the resulting value assigned is 0, 0.0, 0D0, or .FALSE.

You cannot use a comma to terminate a field that is to be read under control of an A, H, or alphanumeric literal field descriptor. If FORTRAN encounters the physical end of the record before it has read w characters, however, short field termination is accomplished and FORTRAN assigns the characters that were input successfully. It also appends trailing spaces to fill the corresponding I/O list element or the field descriptor.

7.11.7 Format Control Interaction with Input/Output Lists

FORTRAN initiates format control with the beginning of execution of a formatted I/O statement. The action of format control depends on information provided jointly by the next element of the I/O list (if one exists) and the next field descriptor of the FORMAT statement. FORTRAN interprets both the I/O list and the format specification from left to right.

If the I/O statement contains an I/O list, at least one field descriptor of a type other than H, X, T, or P must exist in the format specification. An execution error occurs if this condition is not met.

When FORTRAN executes a formatted input statement, it reads one record from the specified unit and initiates format control; thereafter, additional records can be read as indicated by the format specification. Format control demands that a new record be input whenever a slash is encountered in the format specification, or when the last outer right parenthesis of the format specification is reached and additional I/O list elements remain.

Each field descriptor of types I, F, E, G, L, and A corresponds to one element in the I/O list. No list element corresponds to an H, X, P, T, or alphanumeric literal field descriptor. In the case of H and alphanumeric literal field descriptors, data transfer takes place directly between the external record and the format specification.

When format control encounters an I, F, E, G, L, or A field descriptor, it determines if a corresponding element exists in the I/O list. If so, format control transmits data, appropriately converted to or from external format, between the record and the list element, then proceeds to the next field descriptor (unless the current one is to be repeated). If there is no corresponding list element, format control terminates.

When FORTRAN reaches the last outer right parenthesis of the format specification, it determines whether or not there are more I/O list elements to be processed. If not, format control terminates. If additional list elements remain, however, FORTRAN terminates the current record, initiates a new one, and format control reverts to the right-most top-level group repeat specification (the one whose left parenthesis matches the next-to-last right parenthesis of the format specification). If no group repeat specification exists in the format specification, format control returns to the initial left parenthesis of the format specification. Format control then continues from that point.

7.11.8 Summary of Rules for Format Statements

The following is a summary of the rules pertaining to the construction and use of the format statement and its components, and to the construction of the external fields and records with which a format specification communicates.

General

1. You must always label a FORMAT statement.

2. In a field descriptor such as rIw or nX, the terms r, w, and n must be unsigned integer constants greater than zero. You may omit the repeat count and field width specification.

3. In a field descriptor such as Fw.d, the term d must be an unsigned integer constant. It must be present in F, E, D, and G field descriptors even if it is zero. The decimal point must also be present. The field width specification w must be greater than d. The w and d must either both be present or both omitted.

4. In a field descriptor such as nHcc...c, exactly n characters must be present following the H format code. Any ASCII character may appear in this field descriptor (an alphanumeric literal field descriptor follows the same rule).

5. In a scale factor of the form nP, n must be a signed or unsigned integer constant in the range -127 to 127 inclusive. Use of the scale factor applies to F, E, D, and G field descriptors only. Once you specify a scale factor, it applies to all subsequent real or double precision field descriptors in that format specification until another scale factor appears; FORTRAN requires an explicit 0P specification to reinstate a scale factor of zero.

6. FORTRAN does not permit a repeat count in H, X, T or alphanumeric literal descriptors unless you enclose those field descriptors in parentheses and treats them as a group repeat specification.

7. If an I/O list is present in the associated I/O statement, the format specification must contain at least one field descriptor of a type other than H, X, P, T or alphanumeric literal.

Input

1. You must precede an external input field with a negative value by a minus symbol; you may optionally precede a positive value by a plus sign.

2. An external field whose input conversion is governed by an I field descriptor must have the form of an integer constant. It cannot contain a decimal point or an exponent.

3. FORTRAN handles an external field whose input conversion by an F, E, or G field descriptor must have the form of an integer constant or a real or double precision constant. It can contain a decimal point and/or an E or D exponent field.

4. If an external field contains a decimal point, the actual size of the fractional part of the field, as indicated by that decimal point, overrides the d specification of the corresponding real field descriptor.

5. If an external field contains an exponent, it causes the scale factor (if any) of the corresponding field descriptor to be inoperative for the conversion of that field.

6. The field width specification must be large enough to accommodate, in addition to the numeric character string of the external field, any other characters that can be present (algebraic sign, decimal point, and/or exponent).

7. A comma is the only character that is acceptable for use as an external field separator. You use it to terminate input of fields that are shorter than the number of characters expected, or to designate null (zero-length) fields.

Output

1. A format specification must not demand the output of more characters than can be contained in the external record (for example, a line printer record cannot contain more than 133 characters including the carriage control character).

2. The field width specification w must be large enough to accommodate all characters that FORTRAN may generate by the output conversion, including an algebraic sign, decimal point, and exponent. (The field width specification in an E field descriptor, for example, should be large enough to contain d+7 characters.)

3. FORTRAN uses the first character of a record output to a line printer or terminal for carriage control; FORTRAN never prints it. The first character of such a record should be a space, 0,1,$, or +. FORTRAN treats any other character as a space and deletes it from the record.

7.12 LIBRARY FUNCTIONS AND SUBROUTINES

Library functions and subroutines are called in the same manner as user written functions and subroutines. This section lists the library components that are available to FORTRAN programs and illustrates calling sequences, where necessary. Arguments must be of the correct number and type, but need not have the same name as those shown in the illustrative examples. Certain library routines are used by the FORTRAN system programs and are not available to a user's FORTRAN program. These routines may be identified by a number sign (#) in the entry point or section name, and are not listed in the following section.

7.12.1 ABS (Single-Precision Absolute Value)

ABS calculates the absolute value of a real variable by leaving the variable unchanged if it is positive (or zero) and negating the variable if it is negative.

7.12.2 ACOS (Single-Precision Arc-Cosine Function)

ACOS calculates and returns the primary arc-cosine (in radians) of a real argument less than or equal to 1.0 according to the relation:

$$\text{If } x > 0.0,\ \text{ACOS}(x) = \text{ATAN}\ \frac{\text{SQRT}(1-X\hat{}2)}{x}$$

$$\text{If } x < 0.0,\ \text{ACOS}(x) = +\text{ATAN}\ \frac{\text{SQRT}(1-X\hat{}2)}{x}$$

$$\text{If } x = 0.0,\ \text{ACOS}(x) = /2.0$$

7.12.3 AINT (Single-Precision Floating-Point to Integer

AINT is a floating-point truncation function. Given a real argument, it truncates the fractional part of the argument and returns the integral part as an integer. This is accomplished by taking the absolute value of the argument, aligning and normalizing this result, then restoring the original sign. AINT, IFIX, and INT perform identical functions.

7.12.4 ALOG (Single-Precision Natural Logarithm)

ALOG calculates and returns the natural (Naperian) logarithm of a real argument greater than zero. Any negative or zero argument returns an error message and a value of 0.0. The algorithm used is an 8-term Taylor series approximation.

7.12.5 ALOG10 (Single-Precision Common Logarithm)

ALOG10 calculates and returns the common (base 10) logarithm of a real argument greater than zero. Any negative or zero argument returns an error message and a value of 0.0. The calculation is accomplished by calling ALOG to compute the natural logarithm and executing a change of base.

7.12.6 AMAX0 (Single-Precision Maximum Value)

AMAX0 accepts an arbitrary number of integer arguments and returns a real value equal to the largest of the arguments.

7.12.7 AMAX1 (Single-Precision Maximum Value)

AMAX1 accepts an arbitrary number of real arguments and returns a real value equal to the largest of the arguments.

7.12.8 AMIN0 (Single-Precision Minimum Value)

AMIN0 accepts an arbitrary number of integer arguments and returns a real value equal to the smallest of the arguments.

7.12.9 AMIN1 (Single-Precision Minimum Value)

AMIN1 accepts an arbitrary number of real arguments and returns a real value equal to the smallest of the arguments.

7.12.10 AMOD (Single-Precision A Modulo B)

AMOD accepts two real arguments and returns a real value equal to the remainder when the first argument is divided by the second argument. If the second argument is not sufficiently large to prevent overflow, an error message and a value of 0.0 are returned.

7.12.11 ASIN (Single-Precision Arc-Sine)

ASIN calculates and returns the arc-sine (in radians) of a real argument in the range [-1, 1] according to the relation:

```
ASIN(X) = ATAN(X/SQRT(1-X**2))
```

If the argument falls outside the range [-1, 1], an error message results.

7.12.12 ATAN (Single-Precision Arc-Tangent)

ATAN calculates and returns the primary arc-tangent (in radians) of a real argument. The argument is first reduced according to the relations:

```
(1) If x<2^-14,     atan(x) = x
(2) If x>2^-14,     atan(x) = 1/x
(3) If x>1.0,       atan(x) = /2 - atan(1/x)
(4) If x<0,         atan(x) = -atan(-x)
```

and the arc-tangent is then computed by a power series approximation.

7.12.13 ATAN2 (Single-Precision Arc-Tangent of Two Arguments)

ATAN2 accepts two real arguments, assumed to be an abscissa and an ordinate respectively, and calculates the arc-tangent of the quotient of the first argument divided by the second argument. This is accomplished by calling ATAN to find the principal arc-tangent of the quotient and then adjusting the result, depending upon the quadrant in which a point defined by the arguments falls, according to the relations:

```
argument in first quadrant          atan2(y,x) = atan(y/x)
argument in second quadrant         atan2(y,x) = atan(y/x)-
argument in third quadrant          atan2(y,x) = atan(y/x)-
argument in fourth quadrant         atan2(y,x) = atan(y/x)+
```

7.12.14 CGET (Character Get Subroutine)

The calling sequence:

```
CALL CGET (STRING,N,CHAR)
```

causes the Nth character to be unpacked from STRING and stored in CHAR as a variable in the range 0, 63, where STRING is a character string in A6 format.

7.12.15 CHKEOF (Check for End-of-File Subroutine)

CHKEOF accepts one real, integer or logical argument. After the next formatted read operation, this argument will be set to non-zero if the logical end-of-file was encountered, or to 0 if the logical end-of-file was not encountered. The following is an example of the use of CHKEOF:

```
      .
      .
      .
      CALL CHKEOF(EOF)
      READ (N,101)DATA
      IF (EOF.NE.0) GO TO 999
      .
      .
      .
```

7.12.16 CLOCK* (Initialize Clock Subroutine)

The purpose of the CLOCK subroutine is to initialize the KK8-B real-time clock. The calling sequence is:

```
CALL CLOCK (FUNCTN,RATE)
```

Where functn can have a value of 0 or 8, specifying multiple or single A/D channel input, respectively. The value of rate is preset to 100 Hz. Any value of rate other than 0 or 8 causes an error message; any value of rate other than 100 is ignored.

7.12.17 COS (Single-Precision Cosine Function)

COS calculates and returns the cosine of a real argument (in radians) by applying the identity:

```
COS(X) = SIN(X+ /2)
```

7.12.18 COSH (Single-Precision Hyperbolic Cosine Function)

COSH calculates and returns the hyperbolic cosine of a real argument according to the relations:

$$\text{If } |x| \leq 88.029$$
$$\text{COSH}(x) = 1/2\left(\text{EXP}(x) + \frac{1.0}{\text{EXP}(x)}\right)$$

$$\text{If } |x| > 88.028 \text{ and } |x| - \log_e 2 \leq 88.028$$
$$\text{COSH}(x) = \text{EXP}(|x| - \log_e 2)$$

$$\text{If } |x| - \log_e 2 > 88.028$$
$$\text{COSH}(x) = 377737777777_8$$

The third relation produces an error message.

7.12.19 CPUT (Character Put Subroutine)

The calling sequence:

```
CALL CPUT(STRING,N,CHAR)
```

causes CPUT to insert CHAR as the Nth character in STRING, where STRING is a character string stored in A6 format, and CHAR is a number in the range [0, 63] which is interpreted as a character. The following program illustrates the use of CGET and CPUT.

```
    DATA STR/'HEY!'/
    WRITE(4,100) STR
100 FORMAT('   HEY! IN ASCII  ',A6)
    WRITE(4,101)
101 FORMAT('   HEY! IN DECIMAL')
    DO 10 I=1,4
    CALL CGET(STR,I,ICHAR)
    WRITE(4,102) ICHAR
 10 CONTINUE
102 FORMAT(I6)
    DO 20 I=1,6
    J=2*I
    CALL CPUT(STR,I,J)
 20 CONTINUE
    WRITE(4,103) STR
103 FORMAT('   NEW STRING   ',A6)
    CALL EXIT
    END
```

```
.R F4
*TCHRC/G$
 HEY! IN ASCII  HEY!
 HEY! IN DECIMAL
    8
    5
   25
   33
 NEW STRING   BDFHJL
```

7.12.20 DATE (OS/78 Date Subroutine)

DATE accepts three integer arguments, accesses the current OS/78 system date, and returns an integer from 1 to 12 corresponding to the current month as the first argument, an integer from 1 to 31 corresponding to the current day as the second argument, and an integer from 1970 to 1977 corresponding to the current year as the third argument.

7.12.21 DIM (Single-Precision Positive Real Difference)

DIM calculates and returns the positive difference of two real arguments. That is, if the first argument is larger than the second argument, DIM returns the difference between the arguments; if the first argument is less than or equal to the second argument, DIM returns 0.0.

7.12.22 EXP (Single-Precision Exponential Function)

EXP calculates and returns the exponential function of a real argument. The algorithm uses a numerical method after Kogbetliantz (IBM Journal of Research and Development, April, 1957, pp 110-5). EXP3 (Base Raised to an Exponent) Exp3 accepts two real or integer arguments, that is, a base and an exponent and performs the calculation

a=b^e

If the first argument is outside the range [0, 12], the value returned in the second argument is unpredictable. If EXTLVL is called on a PDP-8, the second argument will always be set to zero.

7.12.23 FLOAT (Integer-to-Floating-Point Conversion)

FLOAT accepts an integer argument and returns a real variable equal to the argument.

7.12.24 IABS (Integer Absolute Value Function)

IABS calculates and returns the absolute value of an integer variable by leaving the variable unchanged if it is positive (or zero), and negating the variable if it is negative.

7.12.25 IDIM (Integer Positive Difference Function)

IDIM calculates and returns the positive difference of two integer arguments. That is, if the first argument is larger than the second argument, IDIM returns the difference between the arguments; if the first argument is less than or equal to the second argument, IDIM returns a value of 0.

7.12.26 IFIX (Single-Precision Floating-Point-to-Integer Function)

IFIX is a floating-point truncation function. Given a real argument, it truncates the fractional part of the argument and returns the integral part as an integer. IFIX, AINT and INT perform the same function.

7.12.27 INT (Single-Precision Floating-Point-to-Integer)

INT is a floating-point truncation function that performs the same function as AINT and IFIX.

7.12.28 ISIGN (Integer Transfer of Sign Function)

ISIGN accepts two integer arguments, calculates the absolute value of the first argument, and returns this value if the second argument is positive (or zero), or the negative of this value if the second argument is negative.

7.12.29 MAX0 (Single-Precision Maximum Value)

MAX0 accepts an arbitrary number of integer arguments and returns an integer result equal to the largest of the arguments.

7.12.30 MAX1 (Single-Precision Maximum Value)

MAX1 accepts an arbitrary number of real arguments and returns an integer result equal to the largest of the arguments.

7.12.31 MIN0 (Single-Precision Minimum Value Function)

MIN0 accepts an arbitrary number of integer arguments and returns an integer value equal to the smallest of the arguments.

7.12.32 MIN1 (Single-Precision Minimum Value Function)

MIN1 accepts an arbitrary number of real arguments and returns an integer value equal to the smallest of the arguments.

7.12.33 MOD (Integer A Modulo B Function)

MOD accepts two integer arguments and returns an integer value equal to the remainder when the first argument is divided by the second argument. If the second argument is not sufficiently large to prevent overflow, an error message and a value of 0 are returned.

7.12.34 SIGN (Single-Precision Transfer of Sign)

SIGN accepts two real arguments, calculates the absolute value of the first argument, and returns this value if the second argument is positive (or zero), or the negative of this value if the second argument is negative.

7.12.35 SIN (Single-Precision Sine Function)

SIN calculates and returns the sine of a real argument (in radians). The argument is reduced to the first quadrant, and the sine is then computed from a Taylor series expansion.

7.12.36 SINH (Single-Precision Hyperbolic Sign)

SINH calculates and returns the hyperbolic sine of a real argument according to the relations:

$$\text{If } 0.10<|x|<87.929, \text{SINH}(x) = 1/2\left[\text{EXP}(x) - \frac{1}{\text{EXP}(x)}\right]$$

$$\text{If } |x|\leq 0.10, \text{SINH}(x) = x+x^{3/6}+ x^{5/120}$$

$$\text{If } |x|>88.028, \text{SINH}(x) = [\text{EXP}(|x| - \log_e 2)] \cdot [\text{signum}(x)]$$

7.12.37 SQRT (Single-Precision Square Root Function)

SQRT calculates and returns the (positive) square root of a positive real argument. Any negative argument results in an error message.

7.12.38 TAN (Single-Precision Tangent Function)

TAN calculates and returns the tangent of a real argument (in radians). This is accomplished by computing the quotient of the sine of the argument divided by the cosine of the argument; thus, if the cosine of the argument is zero, an error message is returned.

7.12.39 TANH (Single-Precision Hyperbolic Tangent)

TANH calculates and returns the hyperbolic tangent of a real argument by computing the quotient of the hyperbolic sine of the argument divided by the hyperbolic cosine of the argument.

7.12.40 TIME (Read Time of Day)

TIME may be called as a subroutine with one real or integer argument, or as a function with a dummy argument. It returns the elapsed time since the clock was started. This result will be in seconds.

7.13 FORTRAN LANGUAGE SUMMARY

Statement	Form	Effect
Arithmetic	a=b	The value of expression b is assigned to the variable a.
Arithmetic Statement Function Definition	t nam(al...)=x	The value of expression x is assigned to f(al...) after parameter substitution.

Statement	Form	Effect
ASSIGN	ASSIGN n TO v	Statement number n is assigned as the value of integer variable v for use in an assigned GOTO statement.
BACKSPACE	BACKSPACE u	Peripheral device u is backspaced one record.
BLOCK DATA	BLOCK DATA	Identifies a block data subprogram.
CALL	CALL prog CALL prog(al...)	Invokes subroutine named prog, supply arguments when required.
COMMON	COMMON/blockl/a,b../..	Variables (a,b,...) are assigned to a common block.
CONTINUE	CONTINUE	No processing, target for transfers.
DATA	DATA varlist/var/...	Assigns initial or constant values to variables.
DEFINE FILE	DEFINE FILE a(b,c,U,v)	Describes a mass storage file for direct access I/O.
DIMENSION	DIMENSION array (vl...,v7)	Storage allocated according to dimensions specified for the array.
DO	DO st l-el,e2,e3	Statements following the DO up to statement st are iterated for values of integer variable i, starting at i=el, incrementing by e3, and terminating when i>e2.
END	END	Cease program compilation; equivalent to STOP in main program or RETURN in subprogram.
END FILE	END FILE u	Writes END-OF-FILE character in file u.
EQUIVALENCE	EQUIVALENCE (vl,v2,...,)	Identifies same storage location for variables within parentheses.

Statement	Form	Effect
EXTERNAL	EXTERNAL subprogram	Declares a subprogram for use by other subprograms.
FORMAT	FORMAT (spec1,spec2,.../...)	Specifies conversions between internal and external representations of data.
FUNCTION	FUNCTION name(a1,...)	Indicates an external function definition.
GO TO	(1) GO TO n (2) GO TO (n1,...nk),e (3) GO TO v GO TO v,(n1,...nk)	Transfers control to: (1) statement n (2) to statement n1 if e=1, to statement nk if e=k. (3) transfers control to state-number assigned to v optionally checking that v is assigned one of the labels n1,...nk.
IF	IF(arith expr)n1,n2,n3	Transfers control to n1 if expr<0, n2 if = 0, or n3 if > 0.
IF	IF(logical expr)st	Executes statement if expression has a value .TRUE., otherwise executes the next statement.
Logical Assignment	v=e	Value of expression E is assigned to variable V.
PAUSE	PAUSE [num]	Program execution interrupted and number printed, if given.
READ	READ(u,f) list READ(u,f) READ(u) list READ(a'r) list	Reads a record from a a peripheral device according to specifications given in the argument of the statement.
RETURN	RETURN	Returns control from a subprogram to the calling program.
REWIND	REWIND u	Repositions designated unit to the beginning of the file.
STOP	STOP	Terminate program execution.

Statement	Form	Effect
SUBROUTINE	SUBROUTINE nam[(al...)]	Declares name to be a subroutine subprogram and al,..., if supplied are dummy arguments.
WRITE	WRITE(u,f) list WRITE(u,f) WRITE(u) list WRITE(a'r) list	Writes a record to a peripheral device according to specifications given in the arguments of the statement.

Operators in each type are shown in order of descending precedence.

Type	Operator	Operates Upon
Arithmetic	** exponentiation * multiplication / division + addition - subtraction	arithmetic or logical constants, variables, and expressions
Relational	.GT. greater than .GE. greater than or equal to .LT. less than .LE. less than or equal to .EQ. equal to .NE. not equal to	arithmetic or logical constants variables, and expressions (all relational operators have equal priority)
Logical	.NOT. .NOT.A is true if and only if A is false	logical constants, variables, and expressions
	.AND. A.AND.B is true if and only if A and B are true	
	.OR. A.OR.B is true if and only if either A or B is true.	
	.EQV. A.EQV.B is true if and only if A and B are both true or A and B are both false.	(precedence same as .XOR.)
	.XOR. A.XOR.B is true if and only if A is true and B is false or B is true and A is false	(precedence same as .EQV.)