FORTRAN-10
LANGUAGE MANUAL
Second Edition
The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this manual.

The software described in this document is furnished to the purchaser under a license for use on a single computer system and can be copied (with inclusion of DIGITAL's copyright notice) only for use in such system, except as may otherwise be provided in writing by DIGITAL.

Digital Equipment Corporation assumes no responsibility for the use or reliability of its software on equipment that is not supplied by DIGITAL.

Copyright © 1973, 1974, 1975 by Digital Equipment Corporation

The postage prepaid READER’S COMMENTS form on the last page of this document requests the user’s critical evaluation to assist us in preparing future documentation.

The following are trademark’s of Digital Equipment Corporation:

<table>
<thead>
<tr>
<th>CDP</th>
<th>DIGITAL</th>
<th>INDAC</th>
<th>PS/8</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPUTER LAB</td>
<td>DNC</td>
<td>KA10</td>
<td>QUICKPOINT</td>
</tr>
<tr>
<td>COMSyst</td>
<td>EDGRIN</td>
<td>K110</td>
<td>RAD-8</td>
</tr>
<tr>
<td>COMTEX</td>
<td>EDUSYSTEM</td>
<td>LAB-8</td>
<td>RSTS</td>
</tr>
<tr>
<td>DDT</td>
<td>FLIP CHIP</td>
<td>LAB-8/E</td>
<td>RSX</td>
</tr>
<tr>
<td>DEC</td>
<td>FOCAL</td>
<td>LAB-K</td>
<td>RTM</td>
</tr>
<tr>
<td>DCSYSTEM-10</td>
<td>GLC-8</td>
<td>OMNIBUS</td>
<td>RT-11</td>
</tr>
<tr>
<td>DECTAFE</td>
<td>IDAC</td>
<td>OS/8</td>
<td>TOPS10</td>
</tr>
<tr>
<td>DIBOL</td>
<td>IDACS</td>
<td>PDP</td>
<td>TYPESET 8</td>
</tr>
</tbody>
</table>

November 1975
LA TROBE MODIFICATIONS TO FORTRAN-10

Some of the information in this manual does not apply to the FORTRAN-10 system at La Trobe.

1. Chapter 10, page 10-4

The default Logical Device Assignments of TABLE 10-1 on this page have been changed as below:

<table>
<thead>
<tr>
<th>Device/Function</th>
<th>Default Filename</th>
<th>FORTRAN Logical Unit Number</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ø</td>
<td>FORxx.DAT</td>
<td>Ø0</td>
<td>Illegal</td>
</tr>
<tr>
<td>DSK</td>
<td></td>
<td>Ø1</td>
<td>DISK</td>
</tr>
<tr>
<td>Input Device*</td>
<td></td>
<td>Ø5</td>
<td>User's Terminal Card Reader</td>
</tr>
<tr>
<td>Output Device*</td>
<td></td>
<td>Ø6</td>
<td>User's Terminal Line Printer</td>
</tr>
<tr>
<td>DSK</td>
<td></td>
<td>Ø7</td>
<td>DISK</td>
</tr>
<tr>
<td>DSK</td>
<td></td>
<td>63</td>
<td>DISK</td>
</tr>
</tbody>
</table>

* For terminal use, the default input device is the user's terminal and the default output device is also the user's terminal. For BATCH use, the respective default devices are the card reader and line printer.

2. Chapter 15, pages 15-15 to 15-19

The FORTRAN-10 Library Subroutines of TABLE 15-3 are not all available at La Trobe. Specifically, the La Trobe routines for plotting are better described in Programming Description 13 (available from the data receptionist).
PREFACE

This manual is composed of two parts: PART I, Introduction to using FORTRAN-10 with SOS and PART II, FORTRAN-10 Language Manual.

The Introduction to Using FORTRAN-10 with SOS is a short guide to using the DECSYSTEM-10 Operating System, and it describes the minimum set of commands necessary to input, edit, and execute FORTRAN programs. It assumes that the reader has a rudimentary knowledge of—or is presently learning—FORTRAN programming. It is not an introduction to FORTRAN-10, but a guide to implementing FORTRAN on the DECSYSTEM-10. The complete set of Operating System commands is given in the DECSYSTEM-10 Operating Systems Commands Manual (DEC-10-OSCMA-A-D); the SOS text editor is described completely in the SOS Users Guide (DEC-10-USOSA-A-D).

The FORTRAN-10 Language Manual describes the FORTRAN language as implemented for the FORTRAN-10 Language Processing System (referred to as FORTRAN-10).

The language manual (PART II) is intended for reference purposes only; tutorial type text has been minimized. The reader is expected to have some experience in writing FORTRAN programs and to be familiar with the standard FORTRAN language set and terminology as defined in the American National Standard FORTRAN, X3.9-1966.

The descriptions of the FORTRAN-10 extensions and additions to the standard FORTRAN language set are printed in bold face italic type.

Operating procedures and descriptions of the DECSYSTEM-10 programming environment are included in the appendices.

May 1975
# MASTER TABLE OF CONTENTS

## PART I  INTRODUCTION TO USING FORTRAN-10 WITH SOS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>CONTENT</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LOGGING IN</td>
<td>INTRO 1-1</td>
</tr>
<tr>
<td>2</td>
<td>INPUTTING YOUR PROGRAM</td>
<td>INTRO 2-1</td>
</tr>
<tr>
<td>3</td>
<td>RUNNING YOUR PROGRAM</td>
<td>INTRO 3-1</td>
</tr>
<tr>
<td>4</td>
<td>CHANGING YOUR PROGRAM</td>
<td>INTRO 4-1</td>
</tr>
<tr>
<td>5</td>
<td>FORTRAN-10 INPUT AND OUTPUT OF DATA</td>
<td>INTRO 5-1</td>
</tr>
<tr>
<td>6</td>
<td>SOME HELPFUL COMMANDS</td>
<td>INTRO 6-1</td>
</tr>
<tr>
<td>7</td>
<td>SAYING GOODBYE TO THE COMPUTER</td>
<td>INTRO 7-1</td>
</tr>
<tr>
<td>8</td>
<td>EXAMPLES</td>
<td>INTRO 8-1</td>
</tr>
</tbody>
</table>

## PART II  FORTRAN-10 LANGUAGE MANUAL

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>CONTENT</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>INTRODUCTION</td>
<td>1-1</td>
</tr>
<tr>
<td>2</td>
<td>CHARACTERS AND LINES</td>
<td>2-1</td>
</tr>
<tr>
<td>3</td>
<td>DATA TYPES, CONSTANTS, SYMBOLIC NAMES, VARIABLES, AND ARRAYS</td>
<td>3-1</td>
</tr>
<tr>
<td>4</td>
<td>EXPRESSIONS</td>
<td>4-1</td>
</tr>
<tr>
<td>5</td>
<td>COMPILATION CONTROL STATEMENTS</td>
<td>5-1</td>
</tr>
<tr>
<td>6</td>
<td>SPECIFICATION STATEMENT</td>
<td>6-1</td>
</tr>
<tr>
<td>7</td>
<td>DATA STATEMENT</td>
<td>7-1</td>
</tr>
<tr>
<td>8</td>
<td>ASSIGNMENT STATEMENTS</td>
<td>8-1</td>
</tr>
<tr>
<td>9</td>
<td>CONTROL STATEMENTS</td>
<td>9-1</td>
</tr>
<tr>
<td>10</td>
<td>I/O STATEMENTS</td>
<td>10-1</td>
</tr>
<tr>
<td>11</td>
<td>NAMELIST STATEMENTS</td>
<td>11-1</td>
</tr>
<tr>
<td>12</td>
<td>FILE COL STATEMENTS</td>
<td>12-1</td>
</tr>
<tr>
<td>13</td>
<td>FORMAT STATEMENT</td>
<td>13-1</td>
</tr>
<tr>
<td>14</td>
<td>DEVICE COL STATEMENTS</td>
<td>14-1</td>
</tr>
<tr>
<td>15</td>
<td>SUBPROGRAM STATEMENTS</td>
<td>15-1</td>
</tr>
<tr>
<td>16</td>
<td>BLOCK DATA SUBPROGRAMS</td>
<td>16-1</td>
</tr>
<tr>
<td>A</td>
<td>ASCII-1968 CHARACTER CODE SET</td>
<td>A-1</td>
</tr>
<tr>
<td>B</td>
<td>SYMBOLIC NAMES</td>
<td>B-1</td>
</tr>
<tr>
<td>C</td>
<td>USING THE COMPILER</td>
<td>C-1</td>
</tr>
<tr>
<td>D</td>
<td>WRITING USER PROGRAMS</td>
<td>D-1</td>
</tr>
<tr>
<td>E</td>
<td>FOROTS</td>
<td>E-1</td>
</tr>
<tr>
<td>F</td>
<td>FORDDT</td>
<td>F-1</td>
</tr>
<tr>
<td>G</td>
<td>COMPILER MESSAGES</td>
<td>G-1</td>
</tr>
<tr>
<td>H</td>
<td>FORTRAN-10 REAL TIME SOFTWARE</td>
<td>H-1</td>
</tr>
</tbody>
</table>

---

*November 1975*
PART I

Introduction to Using FORTRAN-10 with SOS

May 1975
CONTENTS

CHAPTER 1 LOGGING IN ........................................ INTRO 1-1

CHAPTER 2 INPUTTING YOUR PROGRAM .................... INTRO 2-1
TO STOP ENTERING LINES INTO YOUR PROGRAM .......... INTRO 2-2
ENDING OR STORING YOUR PROGRAM (E) ............... INTRO 2-2
THE RUBOUT OR DELETE KEY ............................. INTRO 2-3
(CORRECTING TYPING MISTAKES) ....................... INTRO 2-3

CHAPTER 3 RUNNING YOUR PROGRAM ..................... INTRO 3-1
THE EXECUTE COMMAND ................................ INTRO 3-1
CTRL/C (^C) (GETTING THE MONITORS ATTENTION) .. INTRO 3-3
Stopping Your Programs Execution ..................... INTRO 3-3
Deleting a Command ...................................... INTRO 3-4
CTRL/U (^U) (CHANGING A LINE) ........................ INTRO 3-4

CHAPTER 4 CHANGING YOUR PROGRAM .................... INTRO 4-1
THE R SOS COMMAND ..................................... INTRO 4-1
(CORRECTING MISTAKES IN YOUR PROGRAM) ........... INTRO 4-1
SOS COMMANDS .......................................... INTRO 4-1
1 - Inserting Lines Into Your Program ................ INTRO 4-1
D - Deleting Lines From Your Program ................ INTRO 4-3
R - Replacing Lines In Your Program ................... INTRO 4-3
P - Printing Lines Of Your Program On The Terminal INTRO 4-5
N - Changing The Line Numbers ........................ INTRO 4-5
RETURNING TO THE MONITOR ............................ INTRO 4-5
E - End (Ends Editing and Stores the Program) ........ INTRO 4-5
EQ - Returning To the Monitor Without Storing Your Program INTRO 4-6
A FEW SOS CONVENTIONS ................................. INTRO 4-6
TAB (CTRL/I) ............................................. INTRO 4-7
CORRECTING MISTAKES ................................... INTRO 4-7

CHAPTER 5 FORTRAN-10 INPUT AND OUTPUT OF DATA 5-1
READ STATEMENT ......................................... 5-1
WRITE STATEMENT ....................................... 5-1
DEVICE UNIT NUMBERS .................................. 5-1
ACCEPT STATEMENT ...................................... 5-2
TYPE STATEMENT ........................................ 5-2
DATA FILES ............................................. 5-3
Letting FORTRAN Use a Predefined Filename .......... 5-3
Using Your Own Filename ................................ 5-3

CHAPTER 6 SOME HELPFUL COMMANDS ..................... INTRO 6-1
TYPE COMMAND .......................................... INTRO 6-1
(PRINTING OUT YOUR PROGRAM) ........................ INTRO 6-1
DIRECT COMMAND ....................................... INTRO 6-1
(LISTING ALL STORED PROGRAMS AND FILES) ........... INTRO 6-1

INTRO iii  May 1975
DELETE COMMAND
(ERASING A PROGRAM OR FILE) .................................. INTRO 6-2

RENAME COMMAND
(GIVING A PROGRAM OR FILE A NEW NAME) .............. INTRO 6-2
CTRL/O (SUPPRESSING PRINTED OUTPUT) .................. INTRO 6-2
GRIPES ............................................................. INTRO 6-3

CHAPTER 7 SAYING GOODBYE TO THE COMPUTER .......... INTRO 7-1
KJOB COMMAND (LOGGING OUT) .............................. INTRO 7-1
K/F Command (Fast Logout) ..................................... INTRO 7-2
HELP Command (Getting Assistance) ......................... INTRO 7-3
WHAT TO DO IF YOU ARE DISCONNECTED
FROM YOUR JOB (ATTACH) .................................... INTRO 7-3
FORGOT YOUR JOB NUMBER? (SYS) .......................... INTRO 7-4

CHAPTER 8 EXAMPLES ............................................. INTRO 8-1

INTRO iv

May 1975
CHAPTER 1

LOGGING-IN

To begin programming on the DECsystem-10 Timesharing System you need an account number and a password. You may also need to make a telephone connection to the computer; if so, you need the computer's telephone number. Write this information here:

Telephone Number: _____________________________
(if needed)

Account Number: _____________________________

Password: _____________________________

NOTE
Before logging-in be sure to read Chapter 7 on KJOB (logging-out). If you do not logout, but merely disconnect your terminal, the DECsystem-10 accounting system will not know you have finished and WILL CONTINUE TO CHARGE YOU FOR TERMINAL TIME.

First you must make sure that the terminal is turned on to LINE. If you are to make a telephone connection to the computer, turn on the acoustic coupler and then dial the telephone number to make the connection to the DECsystem-10.

The computer now may print a few lines identifying itself and will print

PLEASE LOGIN OR ATTACH

followed by a line beginning with a period (...). This period signifies the computer's readiness to accept your LOGIN command. If the computer does not type a period, you should type CTRL/C (^C). The ^ may appear as an Ü on some terminals. The computer will respond with a period.

NOTE
To type CTRL/C hold the Control (CTRL) key down while typing C. This causes the computer to type the characters ^ C on the terminal. In this book the symbols ^ C will mean that you are to type CTRL/C. In order to signal the computer system that you wish to give it a command, you can type ^ C. This is your way of getting the computer's attention so that you can give it your next command.

INTRO 1-1

May 1975
Monitor: In what follows we shall often call the computer system the *monitor:* this is the operating system or executive program which directs the execution of all the programs and performs the record-keeping duties for the computer.

You may now login by typing

```
LOGIN account number
```

Example: If your account number were 27,240 you would type

```
LOGIN 27,240
```

**NOTE**

We shall use the symbol `\( \)\) to show where you are to press the RETURN key. This key may also be labeled CR or CAR RET and is often referred to as a 'carriage return.' To distinguish between characters you type and those the computer types, underlining will be used for the characters you, the user, type.

The monitor will now respond with the lines

```
JOB job number system number TTYterminal line number
PASSWORD:
```

(The job number is assigned by the monitor.)

The monitor is now asking for your password. You should respond by typing your password and pressing the RETURN key. Since many users prefer to keep their passwords secret, the password is not printed. If your password were the word TROLL, and if everything you typed was printed, the output would appear as:

```
PASSWORD: TROLL
```

But what is actually there is

```
PASSWORD:
```

The monitor signals its acceptance of your account number and password by typing the time, date, and perhaps a message. Then it types a period (.) indicating that it is ready to accept your command. What you now have on the page will look something like this (remember that the underlined passages are those which *you* have typed).

Example:

```
LOGIN 27,240
JOB 25 R5725D SYS #40/2 TTY106
PASSWORD: 1242 30-JAN-75 THUR
```

INTRO 1-2  

November 1975
CHAPTER 2

INPUTTING YOUR PROGRAM

In order to input your program, you are going to use an editing program called SOS. You call SOS by giving the monitor command:

```
R SOS
```

SOS responds with:

```
FILE:
```

asking you for the name of your file. The computer stores your program in a disk file. You must give the file a name by which you and the computer can refer to it — you may think of this name as the name of your program. This name must be from one to six letters or digits. Since the computer can handle several different computer languages, you must also declare that this file will be used to store a program written in the FORTRAN language. This is done by extending the name of your file with the letters FOR. These letters will be separated from the filename (or program name) by a period. Some examples of filenames in which you may store programs written in FORTRAN are:

```
ASPEN.FOR
ABC123.FOR
INSPIR.FOR
```

WHENEVER YOU REFER TO YOUR PROGRAM USE ITS FULL EXTENDED NAME.

Now you should type in the name of your file/program.

Example: (Here the name of the file or program is ASPEN.FOR.)

```
R SOS

FILE: ASPEN.FOR
```

SOS will now print

```
INPUT: ASPEN.FOR
00100
```

INTRO 2-1

May 1975
and the carriage will move to the correct position (column 1) for you to begin typing your program. Remember that in a FORTRAN program, columns 1 through 5 are reserved for the statement number, column 6 is the continuation field, and columns 7 through 72 are for the FORTRAN statement. The number 00100 that SOS has printed is not part of your program, but is SOS's line number for the first statement of your program. If this first statement is not a numbered or comment statement you must skip 6 spaces (to column 7) before beginning to type in the statement. After you have typed in the first line of your program, press the RETURN key and SOS will print the next line number (in increments of 100), and you may enter the next line of your program. Thus, when SOS prints a line number, you know that it is ready to accept a line of your program. (For a fast way of skipping the label field see the section on TAB, page INTRO 4-7.)

**TO STOP ENTERING LINES INTO YOUR PROGRAM (ESCAPE)**

When you wish to stop entering lines into your program you should press the ESCape key (on some terminals labeled ESC, ALT, ALTMODE, or PREFIX). We shall refer to this key as ESCape. Pressing the ESCape key causes a $ to be printed on the terminal.

Example: (In this example, the first statement is a comment statement: the character C is in column 1.)

```
* C
R SOS /
FILE: ASPEN.FOR /
INPUT: ASPEN.FOR
00100 C THIS IS AN EXAMPLE.
00200 TYPE 10
00300 10 FORMAT (' ASPEN IS A NICE PLACE TO SKI')
00400 END
00500 $
*
```

Note that we have two programs which are already stored in the computer — the system monitor program and SOS. As you know the monitor indicates its readiness to accept your command by printing a period ( . ); SOS indicates its readiness to accept your command by printing an asterisk (*) . When you press the ESCape key, SOS returns with an asterisk (*) showing that it is ready to accept a command.

**ENDING OR STORING YOUR PROGRAM (E)**

It is very important that, when you have finished writing your program, you tell SOS that you are done and that it should store your program until you are ready to use it again. You respond to SOS's request for a command by typing the End command (the letter E) and the RETURN key.

INTRO 2-2

May 1975
Example:

\[ \text{DSKC: ASPEN.FOR} \]

In the above example, SOS tells us that the program ASPEN.FOR has been stored on the disk (named DSKC:). Then SOS turns control over to the monitor which signals its readiness to accept your next command by printing a period. PLEASE NOTE THAT THE SOS 'END' COMMAND (E) IS ESSENTIAL. IF YOU FAIL TO INSTRUCT SOS TO STORE YOUR FILE BEFORE YOU RETURN TO THE MONITOR, YOUR PROGRAM WILL BE LOST.

THE RUBOUT OR DELETE KEY (CORRECTING TYPING MISTAKES)

If you make a mistake while typing a line, the RUBOUT (DELETE or DEL) key allows you to correct your mistake without having to retype the entire line. Press the RUBOUT key once for each character you wish deleted. This causes the deleted characters to be printed with a backslash (\) before and after them. Then, type the correct characters.

Example:

\[ \text{FILE: ASPN\n\nEN.FOR} \]

In the above example, the character 'N' has been rubbed out.

Example:

\[ 00300 \ 10 \ \text{FORMAT ('APEN\n\nE\n\nS PEN IS A NICE PLACE TO SKI!')} \]

In this example, the RUBOUT key was pressed three times to erase the unwanted characters PEN. Note, also, that the deleted characters are printed in reverse order.

Think of the RUBOUT key as a "backspace plus erasure" key!

INTRO 2-3

May 1975
CHAPTER 3

RUNNING YOUR PROGRAM

THE EXECUTE COMMAND

To execute or cause the computer to follow the instructions given by the program, command the monitor to

\texttt{EXECUTE \textsc{filename.extension}}

Example:

\texttt{EXECUTE ASPEN.FOR}
\texttt{FORTRAN: ASPEN}
\texttt{MAIN.}
\texttt{LINK: LOADING}
\texttt{[LINKXCT ASPEN EXECUTION]}
\texttt{ASPEN IS A NICE PLACE TO SKI!}

\texttt{END OF EXECUTION}
\texttt{CPU TIME: 0.05 ELAPSED TIME: 0.15}
\texttt{EXIT}

EXECUTE may be abbreviated to EX.

ASIDE

In the above example you may have been puzzled at the occurrence of lines written by the monitor before the actual execution. They appear because before your FORTRAN source program can be executed it must be translated or compiled into a machine language program (the object program) that the computer can execute. This is done during the step labeled FORTRAN: \textsc{filename}. This object program, like the original source program, is stored in a disk file. Before the program can be executed, a copy of the compiled or object program must be placed (loaded) into the working memory of the computer — this copy is often called a core image of the object program. This is accomplished during the LINK: LOADING step. Finally the execution step is performed.
Few programs will complete execution the very first time you try to execute them. Do not be discouraged! Chances are that the compiler will find at least one mistake in your program. To help you find your mistake(s), it will type out a message to you. For example, suppose that you have made the following mistake in the program on page INTRO 2-2: in the FORMAT statement in line 300 the closing quote has been omitted. The program would look like this:

```
00100     C THIS IS AN EXAMPLE.
00200     TYPE 10
00300  10     FORMAT (' ASPEN IS A NICE PLACE TO SKI')
00400     END
```

An attempt to EXECUTE it will cause the following:

```
EX ASPEN.FOR

FORTRAN: ASPEN
00300  10     FORMAT (' ASPEN IS A NICE PLACE TO SKI')
?FTNCOL LINE:00300 NO CLOSING QUOTE IN LITERAL
?FTNFWE LINE:00300 FOUND END OF STATEMENT WHEN EXPECTING A "")"

UNDEFINED LABELS
10

?FTNFTL MAIN. 3 FATAL ERRORS AND NO WARNINGS
LINK: LOADING
[LNKNSA NO START ADDRESS]
EXIT
```

If the compiler has found errors in your program making execution impossible, you will again have to call on SOS to help you correct your program. Do this by using the R SOS command discussed in Chapter 4.
NOTE

The compiler will only print error messages for cases where the program is not clearly understandable. It is possible to have a program which consists of valid FORTRAN statements, but which gives the wrong answers. For example, if you meant to enter

\[
\text{TAX} = \text{RATE} \times \text{AMOUNT}
\]

but by mistake typed

\[
\text{TAX} = \text{RATE} + \text{AMOUNT}
\]

Since both are possibly valid formulas, the compiler cannot detect this as an error. Errors of this type (logic errors) are the most difficult to find. The program will run, but the answers will be wrong. Frequently the author of the program will read the incorrect statement and see instead what he meant to write. An extremely valuable method of finding errors of this kind is to attempt to explain to someone else why the program should work. The act of explaining will often highlight the error.

CTRL/C (\(^C\)) (GETTING THE MONITOR’S ATTENTION)

CTRL/C informs the monitor that you wish to give it a command. The monitor interrupts whatever the computer is doing and prints a period to indicate that it is ready to accept your command. To type CTRL/C hold the Control (CTRL) key down while typing C.

Stopping Your Program's Execution

CTRL/C interrupts a program during execution, returning control to the monitor. Sometimes it is necessary to type CTRL/C twice to interrupt a program.

Example:

```
  *EXECUTE ASPEN.FOR
  FORTRAN: ASPEN
  ^C
  ^C
```

INTRO 3-3  November 1975
Deleting a Command

You may also use CTRL/C to delete the line you are presently typing and return control to the monitor.

Example

```
.EXECUTE ASPEN.FOR+C
```

CTRL/U (^U) (CHANGING A LINE)

CTRL/U deletes the entire line you are typing. It moves the carriage to the beginning of the next line. You may then retype the line. Note that CTRL/U only deletes that part of the line which you have typed and not the part which the computer prints, i.e., in the following example the line number is not deleted. CTRL/U is typed by holding the Control (CTRL) key down while typing U.

Example:

```
01800 40 SROOT = SRT (DISC) \tU
40 SROOT = SQRT (DISC)
01900
```

In the above example, CTRL/U deletes your input line which you then reenter. CTRL/U does not delete the line number, 1800, printed by SOS.

If you wish to delete the line entirely, follow CTRL/U with the RETURN key.
CHAPTER 4

CHANGING YOUR PROGRAM

THE R S0S COMMAND (CORRECTING MISTAKES IN YOUR PROGRAM)

In order to correct a mistake in a program you must return to SOS. As we saw on page INTRO 2-1, we turn control over to SOS by commanding the monitor

\[R \text{ SOS}\]

SOS responds with

\[\text{FILE:} \quad \text{QUAD.FOR} \]

and we type the filename and extension, in this case QUAD.FOR. But now SOS recognizes that this program already exists and correctly assumes that instead of inputting a file you wish to edit it. SOS thus types

\[\text{EDIT:} \quad \text{QUAD.FOR}\]

\[*\]

The asterisk (*) indicates that SOS is at your command. The remainder of this section lists those SOS commands which are essential for inputting and editing simple FORTRAN programs.

SOS COMMANDS

I – Inserting Lines Into Your Program

To Insert lines into your program beginning with line 2700 you give SOS the command:

\[*I2700\]

SOS types cut each line number, and you respond by Inserting the line into the program. When you press the RETURN key after typing each line, SOS will type the next line number. (This is called 'Insert mode'.)
Example:

```
*12700
02700  IF (DISC) 20, 30, 40
02800  40  ROOT1 = (-B + SQRT (DISC))/(2*A)
02900
```

Terminate the Insert command by typing ESCape (ALTMODE/PREFIX); this causes a $ to be printed on the terminal.

Example:

```
*14000
04000  20  WRITE (5, 70)
04100  $
```

Note that in the above examples, SOS has numbered the lines in increments of 100. The reason for providing this increment is to allow you room to maneuver—suppose you have accidentally omitted lines which must now be inserted, or suppose you now find it necessary to change your original program. In the event that you have left out 2 lines which should have gone between lines 3200 and 3300, you may insert these lines by changing the increment size, say, to 20, using the command

```
*13210, 20
```

This allows you to insert lines 3210, 3230, 3250, 3270, and 3290 into your program. (The size of the increment doesn’t really matter, so long as it is small enough to accommodate all additional lines.) You may use the ESCape key to terminate the Insert command whenever necessary.

Example:

```
*13210, 20
03210  WRITE (5, 50) ROOT1, ROOT2
03230  50  FORMAT ('ROOTS ARE', F10.2, 'AND', F10.2)
03250  $
```

Each time you change the step size, the new size is kept until changed again.
NOTE

Trouble arises when the Insert command tries to insert a line that bears the same number as an existing line, and when an existing line falls between a line that has just been inserted and the next line determined by the increment size. In these cases SOS will either designate a different line number for the line to be inserted or will terminate the Insert command.

D — Deleting Lines From Your Program

To Delete line 500 from your program command SOS to

*D500 /

Example:

*D500 /
1 LINES (00500/1) DELETED
*

If you wish to Delete lines 1400 through 1600 from your program use

*D1400;1600 /

Example:

*D1400;1600 /
3 LINES (01400/1:01600) DELETED
*

R — Replacing Lines in Your Program

The Replace command is a combination of a Delete command followed by an Insert command. To instruct SOS to Delete line 1700 and to begin Inserting lines at line 1700, use the Replace command

*R1700 /

This is equivalent to the command D1700 followed by the command I1700.
Example:

*R1700
01700- 60 FORMAT (' ROOT IS', F10.2),
1 LINES (01700/1) DELETED
*

To Replace lines 500 through 700 use

*R500:700

This is equivalent to D500:700 commanding SOS to Delete lines 500 through 700, followed by the command 1500 instructing SOS to begin Inserting lines at 500.

Example:

*R500:700
00500 80 FORMAT ('GIVE COEFFICIENTS'),
00600 READ (5, 10) A, B, C,
00700 10 FORMAT (F10.2),
3 LINES (00500/1:00700) DELETED
*

If you also wish to change the increment size to 10, use the Replace command

*R1000:1100,10

This is equivalent to the command D1000:1100 followed by the command I1000,10.

Example:

*R1000:1100,10
01000 40 SROOT = SQRT (DISC)
01010 DENOM = 2*A,
01020 ROOT1 = (-B + SROOT) / DENOM,
01030 ROOT2 = (-B - SROOT) / DENOM,
01040 $,
2 LINES (01000/1:01100) DELETED
*

As with the Insert command to terminate the Replace command use the ESCape as in the above example.

INTRO 4-4

May 1975
P — Printing Lines of Your Program on the Terminal

If you wish to Print line 1800 of your program, type

\[ \texttt{P1800} \]

Example:

\[ \texttt{P1800} \]
\[ \texttt{01800} \]
\[ \texttt{40} \]
\[ \texttt{SROOT = SQRT (DISC)} \]

In order to Print lines 2700 through 3000 of your program use

\[ \texttt{P2700:3000} \]

Example:

\[ \texttt{P2700:3000} \]
\[ \texttt{02700} \]
\[ \texttt{30} \]
\[ \texttt{ROOT = -B / (2*A)} \]
\[ \texttt{02800} \]
\[ \texttt{WRITE (5, 60) ROOT} \]
\[ \texttt{02900} \]
\[ \texttt{60} \]
\[ \texttt{FORMAT (' ROOT IS ', F10.2)} \]
\[ \texttt{03000} \]
\[ \texttt{GO TO 100} \]

N — Changing the Line Numbers

The Number command instructs SOS to renumber your program beginning at line 100 in increments of 100. SOS does not print anything on the terminal. If you wish to see the renumbered program you must use the Print command.

Example:

\[ \texttt{N} \]

RETURNING TO THE MONITOR

E — End (Ends Editing and Stores the Program)

When you have completed editing your program, inform SOS that it should now store your program on the disk by typing E (End). If you do not instruct SOS to store your program, the editing you have just completed will be lost.

INTRO 4-5

May 1975
Example:

\[\text{DSKC:QUAD.FOR}[27,240]\]

This indicates that the program named QUAD.FOR has been stored on DSKC.
The End command turns control over to the monitor which prints a period to indicate its readiness to accept your next command.

**EQ** — Returning to the Monitor *Without* Storing Your Program

If you decide that the current editing session is worthless, you may return to the monitor *without storing* your program by using the Quit command.

Example:

\[\text{*EQ}\]

This does not destroy the program; it restores the original copy of the program as it was when you last typed R SOS. If the program is a new one, it is deleted since the original program did not exist.

**A FEW SOS CONVENTIONS**

(a) A range of lines is indicated by a colon between the first and last line numbers of the range, (i.e., 500:700)

(b) A period represents the current line. Thus, D. means delete the current line.

Example:

\[00700 \ 80 \ 	ext{FORMAT ("0GIVE COEFFICIENTS")}
\]

\[\text{*D*}
\]

\[\text{LINES (00700/1) DELETED} \ 	ext{*}\]

In the above example the current line is line 700 and it is deleted.

(c) An asterisk is used to represent the last line of the file. Thus, to instruct SOS to print out your entire file use:

\[\text{*P0:*}\]
TAB (CTRL/I)

The TAB or Horizontal Tab (sometimes labeled HT or — | ) is handy when you are entering lines into your program. The TAB, similar to that on a typewriter, is set at 8-character intervals. It moves the carriage to the next column that is a multiple of 8; no characters are output on the terminal. As you know, a FORTRAN statement must be located within columns 7 through 72, although it may appear at any point within this range. Using the TAB to skip over all or part of the label field will bring the carriage to column 8, enabling you to begin your FORTRAN statement in that column. If your terminal does not have a key labeled TAB, use CTRL/I instead. To type CTRL/I, hold down the Control (CTRL) key while typing I.

CORRECTING MISTAKES

To correct one or more characters use the RUBOUT key (see page INTRO 2-3).

To change an entire line use CTRL/U (see page INTRO 3-4).

Example:

```
*DI500 +U
12000
02000
```

In the above example, CTRL/U (^U) allows you to change the command ‘Delete line 1500’ to an Insert command.
Example:

```
R SOS.
```

FILE: ZELDA.FOR
INPUT: ZELDA.FOR

```
00100 C THIS PROGRAM DOES NOTHING.
00200 TYPE 10
00300 10 FORMAT ('IT'S WORKING!')
00400 TYPE EPE 20
00500 $
*P400
00400 TYPE 20
*I500
00500 20 FORMAT ('WHAT IS YOUR NAME?')
00600 ACCEPT 30, YORNAM TU
00700 ACCEPT 30, YORNAM
00800 30 FORMAT (A5)
00900 TYPE 40, YORNAM
01000 40 FORMAT ('0', 'HI', ', A5, ',DO YOU WANT')
01100 $
*R900
00900 40 FORMAT ('0HI', ', A5, ',DO YOU')
01000 TYPE 50
01100 50 FORMAT ('WANT TO BE FRIENDS?')
01200 END
01300 $
```

1 LINES (00900/1) DELETED
*R900:1100

```
00900 40 FORMAT ('0HI, ', A5, ', WANT TO BE FRIENDS?')
01000 $
```

3 LINES (00900/1:01100) DELETED
*N*

```
00100 C THIS PROGRAM DOES NOTHING.
00200 TYPE 10
00300 10 FORMAT ('IT'S WORKING!')
00400 TYPE 20
00500 20 FORMAT ('WHAT IS YOUR NAME?')
00600 ACCEPT 30, YORNAM
00700 30 FORMAT (A5)
00800 TYPE 40, YORNAM
00900 40 FORMAT ('0HI, ', A5, ', WANT TO BE FRIENDS?')
01000 END
*E*
```

[DSKC:ZELDA.FOR]

Let us look at the above example in detail.
R SOS

Command the monitor to turn control over to the editor program SOS.

FILE: ZELDA.FOR

SOS requests the name of the file you wish to edit. You respond with the name of your file or program: ZELDA.FOR.

INPUT: ZELDA.FOR
       00100 C THIS PROGRAM DOES NOTHING.

When SOS fails to find a file by this name, it concludes that you intend to input a new file. SOS then prints the name of the file and the first line number. Now you are ready to enter the first line of the program.

       00200 TYPE 10

Each time you finish typing a line and press the RETURN key, SOS prints out the next line number so that you may input that line. In typing line 200 the first character actually typed was a TAB (CTRL/I) which caused the label field to be skipped over; this avoids the necessity of counting spaces so that our FORTRAN statement would begin in the proper column. TAB is a non-printing character.

       00300 10 FORMAT ("IT'S WORKING!")

This statement is labeled. After typing the FORTRAN statement label (10), you type the non-printing character TAB (CTRL/I) to skip over the remainder of the label field. Remember that the first character of a printing FORMAT statement must be the carriage control (here a blank which means 'single space output'). Notice that because apostrophes are used to enclose literal fields, they are not allowable characters within a literal field but must instead be represented by two successive apostrophes. In other words, although line 300 appears in the program with two successive apostrophes (IT'S), in the execution it causes the word IT'S to be printed (see the EXECUTION which follows).

       00400 TYPE\E\PE 20

Again a non-printing TAB is used here to skip over the label field. The RUBOUT key erases the E.

       00500 $

The ESCape key terminates the inputting of lines into the program.

       *P400

SOS is now ready for a new command. You ask it to print line 400.
00400 TYPE 20

SOS prints line 400.

*1500 *

Your next step is to insert lines into your program beginning with line 500.

00500 20 FORMAT ('WHAT IS YOUR NAME?')

You type line 500 into your program.

00600 ACEPT 30, YORNAM U
     ACEPT 30, YORNAM /

After typing in line 600 but before pressing the RETURN key, you pause and notice that you have misspelled ACCEPT. CTRL/U (^U) deletes the line which you then retype beginning with the non-printing TAB.

00700 30 FORMAT (A5) /
00800 TYPE 40, YORNAM /
00900 40 FORMAT ('0', 'HI', ' ', A5, 'DO YOU WANT?') /
01000 $ /

You enter lines 700 through 900 into your program and terminate the insert. The carriage control 'O' in the FORMAT statement causes the output to be double spaced.

*R900 *

At this point, you decide to replace line 900. The R900 command causes it to be deleted and initiates an insert command beginning with line 900.

00900 40 FORMAT ('0HI', ' ', A5, 'DO YOU') /
01000 TYPE 50 /
01100 50 FORMAT ('WANT TO BE FRIENDS?') /
01200 END /
01300 $ /

1 LINES (00900/1) DELETED

When you use the ESCape key to terminate the insert command (initiated by the replace command), SOS informs you that 1 line (line 900) has been deleted.

*R900:1100 *

You decide to replace lines 900 through 1100.

INTRO 4-10 November 1975
00900  40   FORMAT ('0 HI, ', AS, ', WANT TO BE FRIENDS?')
01000  $ 
3 LINES (00900/1:01100) DELETED

Line 900 is replaced and the command is terminated. SOS confirms that 3 lines (900 through 1100) have been deleted.

*N

SOS is now asked to renumber the lines beginning with 100 and in steps of 100.

*P0:*

You instruct SOS to print out your entire program.

*E

To conclude the editing session, instruct SOS to store your program on the disk.

[DSKC: ZELDA.FOR]

Your program has been stored on DSKC:. The monitor is now in control.

The EXECUTION of the above program:

.EX ZELDA.FOR
FORTRAN: ZELDA
MAIN.
LINK: LOADING
[LNKXCT ZELDA EXECUTION]
IT'S WORKING!
WHAT IS YOUR NAME?
HAL

HI, HAL , WANT TO BE FRIENDS?

END OF EXECUTION
CPU TIME: 0.10  ELAPSED TIME: 10.20
EXIT

INTRO 4-11

May 1975
CHAPTER 5

FORTRAN-10 INPUT AND OUTPUT OF DATA

Although FORTRAN-10 is essentially the same as standard FORTRAN, a few minor differences do arise in statements that involve the input and output of data.

READ STATEMENT

The statement

READ (u, f) list

where u = device unit number and

f = FORMAT statement number

reads data from the device with unit number u (refer to the section on Device Unit Numbers, below) according to the specifications given by FORMAT statement f.

Example:

00800 READ (5, 35) IGRADE
00900 35 FORMAT (I3)

WRITE STATEMENT

This has the form

WRITE (u, f) list

where u = device unit number and

f = FORMAT statement number

Example:

01000 WRITE (1, 30) (STUDENT(I), I=1,8), IGRADE
01100 30 FORMAT (BA5, I3)

DEVICE UNIT NUMBERS

In READ and WRITE statements we must specify to which device (Disk, Line Printer, Terminal, etc.) we are referring. For the DECSystem-10 the device unit numbers, u, are uniform - they are the same on all DECSystem-10's. The most commonly used are:

INTRO 5-1

May 1975
Device | Device Unit Number, u
-------|-------------------
Disk   | 01
Card Reader | 02
Line Printer | 03
Terminal | 05

(For a complete list see FORTRAN-10 Language Manual, Table 10-1.)

Thus, WRITE (5,7) causes output to be printed on your terminal; READ (1,25) causes data to be read from the disk.

**ACCEPT STATEMENT**

To input data from the terminal you may use

\[ \text{ACCEPT } f, \text{ list} \]

where \( f = \) the FORMAT statement number.

Example:

\[
00500 \quad \text{ACCEPT 20, IGRADE}
00600 \quad 20 \quad \text{FORMAT (13)}
\]

Thus, "ACCEPT \( f, \) list" is equivalent to "READ \( (5, f) \) list".

**TYPE STATEMENT**

To have output typed on your terminal use

\[ \text{TYPE } f, \text{ list} \]

where \( f = \) the FORMAT statement number.

Example:

\[
00200 \quad \text{TYPE 10}
00300 \quad 10 \quad \text{FORMAT ('ASPEN IS A NICE PLACE TO SKI!')}\]

Thus, "TYPE \( f, \) list" is equivalent to "WRITE \( (5, f) \) list".

**NOTE**

To print something on your terminal, you must include a carriage control character similar to the way you do for a line printer. For example, to print the word HELLO on your terminal, use the format statement below:

\[
00200 \quad \text{TYPE 101}
00300 \quad 101 \quad \text{FORMAT ('HELLO')}
\]

The space before HELLO tells the system to start on a new line.
DATA FILES
You may use data files in one of two ways:

1. In the first method, you let FORTRAN use a predefined filename.

2. In the second method, you choose the filename by using the OPEN statement.

Letting FORTRAN Use A Predefined Filename
There are six Device Unit Numbers for disk files; whenever you use one of them, FORTRAN uses a predetermined filename. The device numbers and their filenames are listed below.

<table>
<thead>
<tr>
<th>Device Unit Number</th>
<th>Filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>FOR01.DAT</td>
</tr>
<tr>
<td>20</td>
<td>FOR20.DAT</td>
</tr>
<tr>
<td>21</td>
<td>FOR21.DAT</td>
</tr>
<tr>
<td>22</td>
<td>FOR22.DAT</td>
</tr>
<tr>
<td>23</td>
<td>FOR23.DAT</td>
</tr>
<tr>
<td>24</td>
<td>FOR24.DAT</td>
</tr>
</tbody>
</table>

Examples:

```
00200  WRITE (1,101) X
```
Writes the value of X in the file FOR01.DAT, according to FORMAT statement 101.

```
00300  READ (23,109) Y
```
Reads the value of Y from the file FOR23.DAT, according to FORMAT statement 109.

Restriction:
You cannot READ and WRITE from the same device unit number within the same program. In other words, the sequence:

```
00500  READ (20,141) Y
00600  WRITE (20,153) Y
```
Attempts to write X in the same file.

is strictly illegal and could be replaced by:

```
00700  READ (20,141) Y
00800  WRITE (21,153) Y
```

NOTE
If you omit the filename from an OPEN statement, FORTRAN uses the filename corresponding to the device unit number.

Using Your Own Filename
To use your own filename, place an OPEN statement before the first READ or WRITE statement that accesses the file. The OPEN statement has the format:

```
OPEN (UNIT=n, FILE= 'filename.ext')
```

INTRO 5-3

November 1975
n is the device unit number, and filename.ext is the name of the file you want to use.

Example:

```fortran
00200 OPEN (UNIT=20, FILE='TEST.DAT')
```

Instructs FORTRAN to use the file TEST.DAT when you use the logical unit number 20.

```fortran
00300 READ (20,105) Y
```

Reads Y from the file TEST.DAT

After the last READ or WRITE statement that accesses a file, it is recommended (though not required) that you include a CLOSE statement. The CLOSE statement has the format:

```fortran
CLOSE (UNIT=n, FILE='filename.ext ')
```

n is the device unit number, and filename.ext is the name of the file you are closing.

Example:

```fortran
00500 CLOSE (UNIT=20, FILE='TEST.DAT')
```

Closes the file TEST.DAT.
CHAPTER 6

SOME HELPFUL COMMANDS

TYPE COMMAND (PRINTING OUT YOUR PROGRAM)

It is likely that you have made numerous changes in your program. If you would like the monitor to TYPE out your program on your terminal as it now stands, command it to:

```
.TYPE filename.extension
```

Example:

```
.TYPE ASPEN.FOR
00100 C THIS IS AN EXAMPLE.
00200 TYPE 10
00300 10 FORMAT (* ASPEN IS A NICE PLACE TO SKI!*)
00400 END
```

DIRECT COMMAND (LISTING ALL THE STORED PROGRAMS AND FILES)

The DIRECT command causes the monitor to list all the programs and files stored in disk files under your account number. It also lists the length of each program or file in terms of DECsystem-10 disk blocks (a disk block is 640 characters) and the date on which each was created. This command may be abbreviated to DIR.

Example:

```
.DIR
ASPEN REL  1 <055>  30-JAN-75  DSKC: [27,240]
ASPEN QOR  1 <055>  30-JAN-75
ASPEN FOR  1 <055>  30-JAN-75
NEW QOR   2 <055>  30-JAN-75
QUAD REL  3 <055>  30-JAN-75
NEW FOR   2 <055>  30-JAN-75
QUAD QOR  2 <055>  30-JAN-75
SNOW FOR  1 <055>  30-JAN-75
QUAD FOR  2 <055>  30-JAN-75

TOTAL OF 15 BLOCKS IN 9 FILES ON DSKC: [27,240]
```

These files belong to the programmer(s) with account number 27,240.
You may find that files which were not created by you are also listed. These may be programs and files created by the computer in editing and compiling your program. The compiled program is contained in a file named filename.REL where the filename is the same one that you used. If you have edited your program there will be a program whose name is identical to yours except that it has a Q as the first letter of the extension. This is a backup file containing your program as it existed prior to your most recent editing of it. Each time your program is edited, the program immediately before editing becomes the backup, and the previous backup - if it existed - is lost. In the foregoing example the only files explicitly created were ASPEN.FOR, NEW.FOR, SNOW.FOR, and QUAD.FOR.

DELETE COMMAND (ERASING A PROGRAM OR FILE)
To erase a file from the disk, command the monitor to

```
DELETE filename.extension
```

Example:

```
DELETE ASPEN.FOR
FILES DELETED:
ASPEN.FOR
01 BLOCKS FREED
```

RENAME COMMAND (GIVING A PROGRAM OR FILE A NEW NAME)
To rename a file use the command

```
RENAME newfilename.extension = oldfilename.extension
```

Example:

```
RENAME EXAMP.FOR = SNOW.FOR
FILES RENAMED:
NEW.FOR
```

This will cause the name of SNOW.FOR to be changed to EXAMP.FOR

CTRL/O (SUPPRESSING PRINTED OUTPUT)
CTRL/O (^O) stops printed output on the terminal. The program sending the output continues to run. Use CTRL/O for example, to stop the message of the day during LOGIN, or if you have asked the monitor to TYPE a program and then want to stop it. CTRL/O is typed by holding the Control (CTRL) key down while typing the letter O.
Example:

```
*TYPE ASPEN*FOR /
00100 C THIS IS AN EXAMPLE.
00200 TYPE 10
00300 !0
```

Although CTRL/C also stops output on the terminal, it does so by terminating programs execution.

**GRIPES**

When all else fails and you must gripe to someone, gripe to the computer by commanding the monitor to:

```
*R GRIPE*
```

The computer will respond with

**YES? (DEPRESS ESCAPE KEY WHEN THROUGH)**

Now enter your gripe and press the ESCape key when you have finished. Remember that typing ESCape causes a $ to be printed on the keyboard.

Example:

```
*R GRIPE*
```

**YES? (DEPRESS ESCAPE KEY WHEN THROUGH)**
**THIS CONSOLE IS ALMOST OUT OF PAPER.**$  
**THANK YOU**
CHAPTER 7

SAYING GOODBYE TO THE COMPUTER

KJOB COMMAND (LOGGING-OUT)
To say goodbye to the computer, command the monitor to KJOB (KillJOB):

\[ \text{KJOB} \]

The monitor will respond with

\[ \text{CONFIRM:} \]

Should you now decide to abort the logout, type CTRL/C (^C). If you still wish to
logout, you must instruct the monitor whether to Kill, Preserve, or Save each of
your disk files. If a file is Killed, it is erased from the computer's memory; Saved
and Preserved files, on the other hand, are retained in the computer's memory.
Preserve and Save are essentially alike except in the matter of protection against
inadvertent loss or destruction. Preserve, unlike Save, affords protection against
accidental destruction of your files by another user who shares your account num-
ber. This may occur if the other user fails to recognize the name of your program
during his logout and, failing to see any need for its preservation, Kills it. To take
advantage of the protection afforded by the Preserve file status, it is best to re-
spond to the CONFIRM with the letter U:

\[ \text{CONFIRM: U} \]

This will automatically Preserve any files which have already been Preserved dur-
ing a previous logout. After you have typed in the letter U and pressed the RE-
TURN key, the monitor will list the name and storage information of each of the
unpreserved files stored in your disk area, pausing after each name for your re-
sponse. Following the name of each file you must respond by typing one of the
three commands:

(a) K if you wish to Kill the file,
(b) P to Preserve it, and
(c) S to Save it.

Please remember that Saved or Preserved files occupy valuable space on the disk.
In general, the only files you need Preserved have the extension FOR. If you have
no further changes to make in your program, you may Preserve the compiled ver-
sion – this will have the extension REL.
NOTE

The DECsystem-10 offers the option of detaching the terminal from your job, thereby freeing the terminal and the telephone line for another task while your program is executing. (This option is, of course, only used for programs with long execution times; for details see the DECsystem-10 Operating System Commands Manual.) Therefore, TURNING OFF THE TERMINAL OR BREAKING THE TELEPHONE CONNECTION TO THE COMPUTER DOES NOT END YOUR JOB, NOR DOES IT STOP THE COMPUTER'S CLOCK; ONLY THE COMMAND KJOB WILL DO THIS. If you should inadvertently hang up without using KJOB, the computer clock, unaware that you have not yet completed your job, will keep ticking and CHARGING YOU FOR TERMINAL TIME. So, please remember to USE KJOB BEFORE LEAVING THE TERMINAL. If you should be accidentally disconnected, always call again and end your job properly. (See page INTRO 7-3.)

Example:

```
KJOB
CONFIRM: U
DSKA: $
DSKC:
ASPEN .REL <055> 5. BLKS : S$
ASPEN .QOR <055> 5. BLKS : K$
NEW .QOR <055> 5. BLKS : K$
QUAD .REL <055> 5. BLKS : S$
EXAMP .FOR <055> 5. BLKS : S$
QUAD .QOR <055> 5. BLKS : K$
SNOW .FOR <055> 5. BLKS : P$
QUAD .FOR <055> 5. BLKS : P$
NEW .FOR <055> 5. BLKS : K$
ASPEN .FOR <055> 5. BLKS : P$
DSKB:
JOB 25, USER [27,240] LOGGED OFF TTY 106 1430 30-JAN-75
DELETED 4 FILES (20 BLOCKS)
SAVED FILES (30 BLOCKS)
RUNTIME 32.34 SEC
```

**K/F command (Fast Logout)**

For a fast logout in which *all* programs and files are Saved use **K/F**

```
K/F
```

INTRO 7-2  May 1975
Although this form of the KJOB command has the advantage of being fast, it allows you neither to Preserve those programs you wish to keep nor to Kill those programs you no longer need.

Example:

```
K/F
JOB 25, USER (27,240) LOGGED OFF TTY106 1432 30-JAN-75
SAVED ALL FILES (30 BLOCKS)
RUNTIME 1.56 SEC
```

HELP Command (Getting Assistance)

To get assistance during logout, type H (for Help) and the monitor will respond.

**WHAT TO DO IF YOU ARE DISCONNECTED FROM YOUR JOB (ATTACH)**

Although this can happen to anyone it will most often happen when the telephone lines connecting you and the computer break that connection. If necessary, redial the telephone number to the computer. Unless the system has crashed (in which case you should give up and go to the movies), the computer will print

```
PLEASE LOGIN OR ATTACH
```

You wish to attach yourself to the job on which you had been working. To do this you must know its job number. This is given after your LOGIN command. For example, in the LOGIN example on page INTRO 1-2, the job number is 25.

You may attach to a job using the commar

```
*ATTACH job_number [account_number]
```

The programmer with account number 27,240 may attach to job 25 by typing:

```
*ATTACH 25 (27,240)
```

Provided that the programmer with this account number is the owner or originator of job 25, the monitor asks for his password. As during LOGIN, the password is not printed. If the password is accepted, the monitor prints a period and you are now attached to your job.

Example:

```
*ATTACH 25 (27,240)
PASSWORD:  
```

INTRO 7-3
November 1975
NOTE
Account numbers are often called Project-Programmer Numbers (PPNs). In the ATTACH command, the account number must be enclosed in square brackets [ ]. If your terminal does not have keys labeled [ and ], use SHIFT/K for the left square bracket, [, and SHIFT/M for the right square bracket, ].

FORGOT YOUR JOB NUMBER? (SYS)
Suppose you have forgotten your job number. You have thrown away your LOGIN, or perhaps you are using a Visual Display (CRT) terminal and the LOGIN has long since disappeared from the screen. What now? You may find out which jobs are being run under your account number by typing:

SYS [account number]

Example:

```
SYS (27,240)
25 DET 3 tC SW 1
```

KJOB
ATTACH 25 (27,240)
PASSWORD:

The programmer with account number 27,240 wishes to find out which jobs are logged-in under his account number. The monitor answers that job 25 is logged-in under account number 27,240 and that this job is DETached from a terminal. Then the user ATTACHes to job 25.

The SYS command may be given whether or not the user is logged-in. If the user is not logged-in, the SYS command automatically ends with the KJOB command.
EXAMPLES

Example 1:
This program computes the roots of the quadratic equation \( ax^2 + bx + c = 0 \). Note that FORTRAN statement labels may be in any order and also that carriage control characters are necessary for each of the printing FORMAT statements.

```fortran
00100 C           THIS PROGRAM COMPUTES THE ROOTS OF A
00200 C           QUADRATIC EQUATION OF THE FORM:
00300 C           \[ ax^2 + bx + c = 0 \]
00400 C
00500 C           WRITE (5, 80)
00600 C           FORMAT ('GIVE COEFFICIENTS')
00700 80           READ (5, 10) A, B, C
00800 10           FORMAT (F10.2)
00900 C
01000 C           CALCULATE THE DISCRIMINANT
01100 C           DISC = B*B - 4*A*C
01200 C
01300 C           DO THE RIGHT THING ACCORDING TO THE SIGN OF DISC
01400 C           IF (DISC) 20, 30, 40
01500 C
01600 C
01700 C           POSITIVE DISCRIMINANT
01800 40           SROOT = SQRT (DISC)
01900 C
02000 C           DENOM = 2*A
02100 C
02200 C           ROOT1 = (-B + SROOT) / DENOM
02300 C           ROOT2 = (-B - SROOT) / DENOM
02400 C
02500 C
02600 C           ZERO DISCRIMINANT
02700 30           ROOT = -B / (2*A)
02800 C
02900 60           FORMAT ('ROOT IS', F10.2)
03000 C
03100 C
03200 C           NEGATIVE DISCRIMINANT
03300 20           WRITE (5, 70)
03400 70           FORMAT ('ROOTS ARE COMPLEX')
03500 100          STOP
03600 END
```

INTRO 8-1  May 1975
Below, this program is EXECUTEd twice. In the second execution the words
FORTRAN: QUAD are missing because the program has already been compiled,
making it unnecessary for the compile step to be repeated. The program is simply
loaded into core and executed (see page INTRO 3-1).

```plaintext
.EXECUTE QUAD*FOR -/
FORTRAN: QUAD
MAIN.
LINK: LOADING
[LNKXCT QUAD EXECUTION]

GIVE COEFFICIENTS
2.0
-10.0
12.0
ROOTS ARE 3.00 AND 2.00
STOP

END OF EXECUTION
CPU TIME: 0.13 ELAPSED TIME: 18.95
EXIT

.EXECUTE QUAD*FOR -/
LINK: LOADING
[LNKXCT QUAD EXECUTION]

GIVE COEFFICIENTS
5.0
-2.0
10.0
ROOTS ARE COMPLEX
STOP

END OF EXECUTION
CPU TIME: 0.12 ELAPSED TIME: 18.30
EXIT

.

INTRO 8-2
May 1975
Example 2 (Reading A Disk File):

Student grades are recorded on a disk file named STDGRA.DES. Each record has a student name (40 characters) and his numerical grade (a 3 digit integer). The following program will read the grades and compute the mean and standard deviation.

```
*TYPE GRADE.FOR*
00100 C THIS PROGRAM COMPUTES THE MEAN AND
00200 C STANDARD DEVIATION OF STUDENT GRADES
00300 C
00400 OPEN (UNIT=1, FILE='STDGRA.DES')
00500 NUMBER = 0
00600 SUM = 0
00700 SUMSQ = 0
00800 20 READ (1, 10, END=100) IGRADE
00900 10 FORMAT (40X, 13)
01000 NUMBER = NUMBER + 1
01100 SUM = SUM + IGRADE
01200 SUMSQ = SUMSQ + IGRADE*IGRADE
01300 GO TO 20
01400 100 VARIAN = (SUMSQ - (SUM*SUM)/NUMBER)/NUMBER
01500 STDEV = SQRT (VARIAN)
01600 TYPE 30, NUMBER, AMEAN, STDEV
01700 30 FORMAT ('NUMBER OF STUDENTS = ', I3 /
01800 1' MEAN GRADE = ', F6.2 /
01900 1' STANDARD DEVIATION = ', F6.2)
02000 2100 CLOSE (UNIT =1, FILE='STDGRA.DES')
02100 END
```

*.EX GRADE.FOR*
FORTRAN: GRADE
MAIN.
LINK: LOADING
[LNKXCT GRADE EXECUTION]

NUMBER OF STUDENTS = 17
MEAN GRADE = 80.29
STANDARD DEVIATION = 10.45

END OF EXECUTION
CPU TIME: 0.23 ELAPSED TIME: 1.00
EXIT

INTRO 8-3 May 1975
We are reading the grades from a disk file which is OPENed before any reading logical takes place and CLOSED afterwards (see lines 400 and 2100). Note that the logical unit number given in the OPEN and CLOSE statements (UNIT = 1) is the same as that given in the READ statement (line 800) and refers to the device disk.

CONTINUATION LINES

Lines 1800, 1900, and 2000 are one FORTRAN statement – lines 1900 and 2000 being continuations of line 1800. Since TABs have been used at the beginning of each line to skip over all or part of the label field, a way must be provided to inform the computer that the line is a continuation line. The rule is: if the first character (after the TAB) is any number between 1 and 9, then the line is a continuation line.

Example 3 (Writing A Disk File):

The following is the program which created the data file STDGRA.DES. Notice that in the OPEN, WRITE, and CLOSE statements (lines 400, 1000, and 1300) the device unit number is an integer variable, IUNIT. IUNIT has been given the value 1 (line 350) before it is used.

FILE: WOE, FOR
EDIT: WOE, FOR

00100 C THIS PROGRAM ENTERS STUDENT GRADES
00200 C ENTER GRADE OF -1 AFTER LAST STUDENT GRADE TO END
00300 DIMENSION STUDNT(8)
00400 IUNIT=1
00500 OPEN (UNIT=IUNIT, FILE='STDGRA.DES')
00600 40 ACCEPT 10, (STUDNT(I),I=1,8)
00700 10 FORMAT (8A5)
00800 20 ACCEPT 20, IGRADE
00900 20 FORMAT (I3)
01000 IF (IGRADE .EQ. -1) GO TO 100
01100 WRITE (IUNIT, 30) (STUDNT(I),I=1,8), IGRADE
01200 30 FORMAT (8A5, I3)
01300 GO TO 40
01400 100 CLOSE (UNIT=IUNIT, FILE='STDGRA.DES')
01500 100 STOP 'THIS IS THE END'
01600 END

*After this program has been executed the file STDGRA.DES will be listed by the DIRECT command (see page INTRO 6-1), and during the KJOB command (see page INTRO 7-1).

INTRO 8-4

November 1975
EX WOE FOR
FORTRAN: WOE
MAIN.
LINK: LOADING
[LNKXCT WOE EXECUTION]
GEORGE CLINTON
83
ELBRIDGE GERRY
73
DANIEL D. TOMPKINS
58
JOHN CALHOUN
80
RICHARD M. JOHNSON
79
GEORGE DALLAS
95
WILLIAM R. KING
69
JOHN BRECKINRIDGE
77
HANNIBAL HAMLIN
65
SCHUYLER COLFAX
77
HENRY WILSON
77
WILLIAM WHEELER
96
CHESTER ARTHUR
88
LEVI P. MORTON
91
GARRET HOBART
89
CHARLES DAWES
93
CHARLES CURTIS
75

-1

THIS IS THE END

END OF EXECUTION
CPU TIME: 0.92 ELAPSED TIME: 4:21.33
EXIT
Example 4:

This program prepares grade reports for the students whose grades are recorded on the disk file STDGRA.DES.

```fortran
*TYPE REPORT, FOR /
00100 C PROGRAM TO PREPARE GRADE REPORT
00200 DIMENSION ANAME(8)
00300 OPEN (UNIT=1, FILE='STDGRA.DES')
00400 C PRINT HEADINGS
00500 WRITE (5, 10)
00600 10 FORMAT (1$, 14X, 'STUDENT NAME', 12X, 'GRADE')
00700 30 READ (1, 20, END=50) (ANAME(I), I=1, 8), IGRADE
00800 20 FORMAT (8A5, I3)
00900 WRITE (5, 40) (ANAME(I), I=1, 8), IGRADE
01000 40 FORMAT (1$, 8A5, I3)
01100 GO TO 30
01200 50 CLOSE (UNIT=1, FILE='STDGRA.DES')
01300 STOP ' END OF GRADE REPORT'
01400 END
```

*EX REPORT, FOR /
FORTRAN: REPORT
MAIN.
LINK: LOADING
{LNKXCT REPORT EXECUTION}

<table>
<thead>
<tr>
<th>STUDENT NAME</th>
<th>GRADE</th>
</tr>
</thead>
<tbody>
<tr>
<td>GEORGE CLINTON</td>
<td>83</td>
</tr>
<tr>
<td>ELBRIDGE GERRY</td>
<td>73</td>
</tr>
<tr>
<td>DANIEL D. TOMPKINS</td>
<td>58</td>
</tr>
<tr>
<td>JOHN CALHOUN</td>
<td>80</td>
</tr>
<tr>
<td>RICHARD M. JOHNSON</td>
<td>79</td>
</tr>
<tr>
<td>GEORGE DALLAS</td>
<td>95</td>
</tr>
<tr>
<td>WILLIAM R. KING</td>
<td>69</td>
</tr>
<tr>
<td>JOHN BREEKINRIDGE</td>
<td>77</td>
</tr>
<tr>
<td>HANNIBAL HAMLIN</td>
<td>65</td>
</tr>
<tr>
<td>SCHUYLER COLFA</td>
<td>77</td>
</tr>
<tr>
<td>HENRY WILSON</td>
<td>77</td>
</tr>
<tr>
<td>WILLIAM WHEELER</td>
<td>96</td>
</tr>
<tr>
<td>CHESTER ARTHUR</td>
<td>88</td>
</tr>
<tr>
<td>LEVI P. MORTON</td>
<td>91</td>
</tr>
<tr>
<td>GARRET HOBART</td>
<td>89</td>
</tr>
<tr>
<td>CHARLES DAVES</td>
<td>93</td>
</tr>
<tr>
<td>CHARLES CURTIS</td>
<td>75</td>
</tr>
</tbody>
</table>

END OF GRADE REPORT

END OF EXECUTION
CPU TIME: 0.68 ELAPSED TIME: 1:34.77
EXIT

INTRO 8-6 May 1975
If you didn’t recognize the “students” as Vice Presidents of the United States, you fail American History. The grades, by the way, are fictitious.

Example 5 (Trying To Read A Non-Existent File):

Now DELETE the data file containing the students' grades, STDGRA.DES, and then EXECUTE REPORT.FOR (the program in Example 4). The READ statement in line 700 cannot be executed since the file to which it refers does not exist. The execution is thus ‘aborted’.

```
*DELETE STDGRA.DES *
FILES DELETED:
STDGRA.DES
01 BLOCKS FREED

*EXIT REPORT.FOR *
LINK: LOADING
[LINK XCT REPORT EXECUTION]
```

```
STUDENT NAME                   GRADE
ZFRS DAT ATTEMPT TO READ BEYOND VALID INPUT
UNIT=1 DSK:STDGRA.DES[27,240]<055>/ACCESS=SEQ NOUN/MODE=ASCII

NAME   (LOC)  <<---- CALLER (LOC)  <<#ARGS> [ARG TYPES]
IN.    (402703) <<---- MAIN.+11(220) <<#5> [UIUIU]

? JOB ABORTED

END OF EXECUTION
CPU TIME: 0.35  ELAPSED TIME: 1.22
EXIT
```

INTRO 8-7

May 1975
PART II

FORTRAN-10 Language Manual

The FORTRAN-10 Language Manual reflects the software as of Version 4 of the FORTRAN-10 Compiler, Version 4 of the FORTRAN-10 Object Time System (FOROTS), and Version 4 of the FORTRAN-10 Debugging Program (FORDDT).

November 1975
# CONTENTS

## CHAPTER 1
**INTRODUCTION**
1.1 INTRODUCTION .............................................. 1-1

## CHAPTER 2
**CHARACTERS AND LINES**
2.1 CHARACTER SET .......................................... 2-1
2.2 STATEMENT, DEFINITION, AND FORMAT ............... 2-3
2.2.1 Statement Label Field and Statement Numbers .... 2-3
2.2.2 Line Continuation Field ............................... 2-3
2.2.3 Statement Field ......................................... 2-3
2.2.4 Remarks ................................................. 2-3
2.3 LINE TYPES ............................................. 2-4
2.3.1 Initial and Continuation Line Types ............... 2-4
2.3.2 Multi-Statement Lines ................................. 2-5
2.3.3 Comment Lines and Remarks ......................... 2-5
2.3.4 Debug Lines ........................................... 2-6
2.3.5 Blank Lines ........................................... 2-6
2.3.6 Line-Sequenced Input ................................ 2-6
2.4 ORDERING OF FORTRAN-10 STATEMENTS ............ 2-6

## CHAPTER 3
**DATA TYPES, CONSTANTS, SYMBOLIC NAMES, VARIABLES, AND ARRAYS**
3.1 DATA TYPES ............................................. 3-1
3.2 CONSTANTS ............................................. 3-1
3.2.1 Integer Constants .................................... 3-2
3.2.2 Real Constants ...................................... 3-2
3.2.3 Double Precision Constants ......................... 3-3
3.2.4 Complex Constants ................................... 3-4
3.2.5 Octal Constants ..................................... 3-4
3.2.6 Logical Constants .................................... 3-5
3.2.7 Literal Constants .................................... 3-5
3.2.8 Statement Label Constants ......................... 3-6
3.3 SYMBOLIC NAMES ....................................... 3-6
3.4 VARIABLES ............................................. 3-6
3.5 ARRAYS ................................................. 3-7
3.5.1 Array Element Subscripts ......................... 3-7
3.5.2 Dimensioning Arrays ................................ 3-8
3.5.3 Order of Stored Array Elements .................... 3-9

## CHAPTER 4
**EXPRESSIONS**
4.1 ARITHMETIC EXPRESSIONS ................................. 4-1
4.1.1 Rules for Writing Arithmetic Expressions ....... 4-2
4.2 LOGICAL EXPRESSIONS ................................... 4-2
4.2.1 Relational Expressions ............................... 4-6
4.3 EVALUATION OF EXPRESSIONS ......................... 4-8
4.3.1 Parenthesized Subexpressions ....................... 4-8
4.3.2 Hierarchy of Operators ............................... 4-8
4.3.3 Mixed Mode Expressions ............................. 4-9
4.3.4 Use of Logical Operands in Mixed Mode Expressions 4-10
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5.1</td>
<td>Introduction</td>
<td>5-1</td>
</tr>
<tr>
<td></td>
<td>5.2</td>
<td>Program Statement</td>
<td>5-1</td>
</tr>
<tr>
<td></td>
<td>5.3</td>
<td>Include Statement</td>
<td>5-1</td>
</tr>
<tr>
<td></td>
<td>5.4</td>
<td>End Statement</td>
<td>5-2</td>
</tr>
<tr>
<td>6</td>
<td>6.1</td>
<td>Introduction</td>
<td>6-1</td>
</tr>
<tr>
<td></td>
<td>6.2</td>
<td>Dimension Statement</td>
<td>6-1</td>
</tr>
<tr>
<td></td>
<td>6.2.1</td>
<td>Adjustable Dimensions</td>
<td>6-2</td>
</tr>
<tr>
<td></td>
<td>6.3</td>
<td>Type Specification Statements</td>
<td>6-3</td>
</tr>
<tr>
<td></td>
<td>6.4</td>
<td>Implicit Statements</td>
<td>6-4</td>
</tr>
<tr>
<td></td>
<td>6.5</td>
<td>Common Statement</td>
<td>6-5</td>
</tr>
<tr>
<td></td>
<td>6.5.1</td>
<td>Dimensioning Arrays in COMMON Statements</td>
<td>6-6</td>
</tr>
<tr>
<td></td>
<td>6.6</td>
<td>Equivalence Statement</td>
<td>6-6</td>
</tr>
<tr>
<td></td>
<td>6.7</td>
<td>External Statement</td>
<td>6-7</td>
</tr>
<tr>
<td></td>
<td>6.8</td>
<td>Parameter Statement</td>
<td>6-8</td>
</tr>
<tr>
<td>7</td>
<td>7.1</td>
<td>Introduction</td>
<td>7-1</td>
</tr>
<tr>
<td>8</td>
<td>8.1</td>
<td>Introduction</td>
<td>8-1</td>
</tr>
<tr>
<td></td>
<td>8.2</td>
<td>Arithmetic Assignment Statement</td>
<td>8-1</td>
</tr>
<tr>
<td></td>
<td>8.3</td>
<td>Logical Assignment Statements</td>
<td>8-3</td>
</tr>
<tr>
<td></td>
<td>8.4</td>
<td>Assign (STATEMENT LABEL) Assignment Statement</td>
<td>8-3</td>
</tr>
<tr>
<td>9</td>
<td>9.1</td>
<td>Introduction</td>
<td>9-1</td>
</tr>
<tr>
<td></td>
<td>9.2</td>
<td>Go To Control Statements</td>
<td>9-1</td>
</tr>
<tr>
<td></td>
<td>9.2.1</td>
<td>Unconditional GO TO Statements</td>
<td>9-2</td>
</tr>
<tr>
<td></td>
<td>9.2.2</td>
<td>Computed GO TO Statements</td>
<td>9-2</td>
</tr>
<tr>
<td></td>
<td>9.2.3</td>
<td>Assigned GO TO Statements</td>
<td>9-2</td>
</tr>
<tr>
<td></td>
<td>9.3</td>
<td>If Statements</td>
<td>9-3</td>
</tr>
<tr>
<td></td>
<td>9.3.1</td>
<td>Arithmetic IF Statements</td>
<td>9-3</td>
</tr>
<tr>
<td></td>
<td>9.3.2</td>
<td>Logical IF Statements</td>
<td>9-4</td>
</tr>
<tr>
<td></td>
<td>9.3.3</td>
<td>Logical Two-Branch IF Statements</td>
<td>9-4</td>
</tr>
<tr>
<td></td>
<td>9.4</td>
<td>Do Statement</td>
<td>9-5</td>
</tr>
<tr>
<td></td>
<td>9.4.1</td>
<td>Nested DO Statements</td>
<td>9-6</td>
</tr>
<tr>
<td></td>
<td>9.4.2</td>
<td>Extend Range</td>
<td>9-7</td>
</tr>
<tr>
<td></td>
<td>9.4.3</td>
<td>Permitted Transfer Operations</td>
<td>9-8</td>
</tr>
<tr>
<td></td>
<td>9.5</td>
<td>Continue Statement</td>
<td>9-9</td>
</tr>
<tr>
<td></td>
<td>9.6</td>
<td>Stop Statement</td>
<td>9-9</td>
</tr>
<tr>
<td></td>
<td>9.7</td>
<td>Pause Statement</td>
<td>9-10</td>
</tr>
<tr>
<td></td>
<td>9.7.1</td>
<td>T (TRACE) Option</td>
<td>9-10</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10.1</td>
<td>DATA TRANSFER OPERATIONS</td>
<td>10-1</td>
<td></td>
</tr>
<tr>
<td>10.2</td>
<td>TRANSFER MODES</td>
<td>10-1</td>
<td></td>
</tr>
<tr>
<td>10.2.1</td>
<td>Sequential Mode</td>
<td>10-1</td>
<td></td>
</tr>
<tr>
<td>10.2.2</td>
<td>Random Access Mode</td>
<td>10-1</td>
<td></td>
</tr>
<tr>
<td>10.2.3</td>
<td>Append Mode</td>
<td>10-2</td>
<td></td>
</tr>
<tr>
<td>10.3</td>
<td>I/O STATEMENTS, BASIC FORMATS AND COMPONENTS</td>
<td>10-2</td>
<td></td>
</tr>
<tr>
<td>10.3.1</td>
<td>I/O Statement Keywords</td>
<td>10-3</td>
<td></td>
</tr>
<tr>
<td>10.3.2</td>
<td>FORTRAN-10 Logical Unit Numbers</td>
<td>10-3</td>
<td></td>
</tr>
<tr>
<td>10.3.3</td>
<td>FORMAT Statement References</td>
<td>10-5</td>
<td></td>
</tr>
<tr>
<td>10.3.4</td>
<td>I/O List</td>
<td>10-5</td>
<td></td>
</tr>
<tr>
<td>10.3.4.1</td>
<td>Implied DO Constructs</td>
<td>10-5</td>
<td></td>
</tr>
<tr>
<td>10.3.5</td>
<td>The Specification of Records for Random Access</td>
<td>10-6</td>
<td></td>
</tr>
<tr>
<td>10.3.6</td>
<td>List-Directed I/O</td>
<td>10-6</td>
<td></td>
</tr>
<tr>
<td>10.3.7</td>
<td>NAMELIST I/O Lists</td>
<td>10-8</td>
<td></td>
</tr>
<tr>
<td>10.4</td>
<td>OPTIONAL READ/WRITE ERROR EXIT AND END-OF-FILE ARGUMENTS</td>
<td>10-8</td>
<td></td>
</tr>
<tr>
<td>10.5</td>
<td>READ STATEMENTS</td>
<td>10-9</td>
<td></td>
</tr>
<tr>
<td>10.5.1</td>
<td>Sequential Formatted READ Transfers</td>
<td>10-10</td>
<td></td>
</tr>
<tr>
<td>10.5.2</td>
<td>Sequential Unformatted Binary READ Transfers</td>
<td>10-10</td>
<td></td>
</tr>
<tr>
<td>10.5.3</td>
<td>Sequential List-Directed READ Transfers</td>
<td>10-10</td>
<td></td>
</tr>
<tr>
<td>10.5.4</td>
<td>Sequential NAMELIST-Controlled READ Transfers</td>
<td>10-11</td>
<td></td>
</tr>
<tr>
<td>10.5.5</td>
<td>Random Access Formatted READ Transfers</td>
<td>10-11</td>
<td></td>
</tr>
<tr>
<td>10.5.6</td>
<td>Random Access Unformatted READ Transfers</td>
<td>10-11</td>
<td></td>
</tr>
<tr>
<td>10.6</td>
<td>SUMMARY OF READ STATEMENTS</td>
<td>10-11</td>
<td></td>
</tr>
<tr>
<td>10.7</td>
<td>REREAD STATEMENT</td>
<td>10-12</td>
<td></td>
</tr>
<tr>
<td>10.8</td>
<td>WRITE STATEMENTS</td>
<td>10-13</td>
<td></td>
</tr>
<tr>
<td>10.8.1</td>
<td>Sequential Formatted WRITE Transfers</td>
<td>10-13</td>
<td></td>
</tr>
<tr>
<td>10.8.2</td>
<td>Sequential Unformatted WRITE Transfer</td>
<td>10-14</td>
<td></td>
</tr>
<tr>
<td>10.8.3</td>
<td>Sequential List-Directed WRITE Transfers</td>
<td>10-14</td>
<td></td>
</tr>
<tr>
<td>10.8.4</td>
<td>Sequential NAMELIST-Controlled WRITE Transfers</td>
<td>10-14</td>
<td></td>
</tr>
<tr>
<td>10.8.5</td>
<td>Random Access Formatted WRITE Transfers</td>
<td>10-15</td>
<td></td>
</tr>
<tr>
<td>10.8.6</td>
<td>Random Access Unformatted WRITE Transfers</td>
<td>10-15</td>
<td></td>
</tr>
<tr>
<td>10.9</td>
<td>SUMMARY OF WRITE STATEMENTS</td>
<td>10-15</td>
<td></td>
</tr>
<tr>
<td>10.10</td>
<td>ACCEPT STATEMENT</td>
<td>10-15</td>
<td></td>
</tr>
<tr>
<td>10.10.1</td>
<td>Formatted ACCEPT Transfers</td>
<td>10-15</td>
<td></td>
</tr>
<tr>
<td>10.10.2</td>
<td>ACCEPT Transfers Into FORMAT Statement</td>
<td>10-16</td>
<td></td>
</tr>
<tr>
<td>10.11</td>
<td>PRINT STATEMENT</td>
<td>10-17</td>
<td></td>
</tr>
<tr>
<td>10.12</td>
<td>PUNCH STATEMENT</td>
<td>10-17</td>
<td></td>
</tr>
<tr>
<td>10.13</td>
<td>TYPE STATEMENT</td>
<td>10-18</td>
<td></td>
</tr>
<tr>
<td>10.14</td>
<td>FIND STATEMENT</td>
<td>10-18</td>
<td></td>
</tr>
<tr>
<td>10.15</td>
<td>ENCODE AND DECODE STATEMENTS</td>
<td>10-19</td>
<td></td>
</tr>
<tr>
<td>10.15.1</td>
<td>ENCODE Statement</td>
<td>10-19</td>
<td></td>
</tr>
<tr>
<td>10.15.2</td>
<td>DECODE Statement</td>
<td>10-20</td>
<td></td>
</tr>
<tr>
<td>10.15.3</td>
<td>Example of ENCODE/DECODE Operations</td>
<td>10-20</td>
<td></td>
</tr>
<tr>
<td>10.16</td>
<td>SUMMARY OF I/O STATEMENTS</td>
<td>10-21</td>
<td></td>
</tr>
<tr>
<td>CONTENTS (Cont)</td>
<td>Page</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------------------------------------------------------------------</td>
<td>------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHAPTER 11 NAMELIST STATEMENTS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11.1 INTRODUCTION</td>
<td>11-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11.2 NAMELIST STATEMENT</td>
<td>11-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11.2.1 NAMELIST-Controlled Input Transfers</td>
<td>11-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11.2.2 NAMELIST-Controlled Output Transfers</td>
<td>11-3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHAPTER 12 FILE CONTROL STATEMENTS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12.1 INTRODUCTION</td>
<td>12-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12.2 OPEN AND CLOSE STATEMENTS</td>
<td>12-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12.2.1 Options for OPEN and CLOSE Statements</td>
<td>12-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12.2.2 Summary of OPEN/CLOSE Statement Options</td>
<td>12-9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHAPTER 13 FORMAT STATEMENT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13.1 INTRODUCTION</td>
<td>13-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13.1.1 FORMAT Statement, General Form</td>
<td>13-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13.2 FORMAT DESCRIPTORS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13.2.1 Numeric Field Descriptors</td>
<td>13-4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13.2.2 Interaction of Field Descriptors With I/O List Variables During Transfer</td>
<td>13-6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13.2.3 G, General Numeric Conversion Code</td>
<td>13-7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13.2.4 Numeric Fields with Scale Factors</td>
<td>13-7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13.2.5 Logical Field Descriptors</td>
<td>13-9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13.2.6 Variable Numeric Field Widths</td>
<td>13-9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13.2.7 Alphanumeric Field Descriptors</td>
<td>13-10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13.2.8 Transferring Alphanumeric Data Directly Into or From FORMAT Statements</td>
<td>13-10a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13.2.9 Mixed Numeric and Alphanumeric Fields</td>
<td>13-11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13.2.10 Multiple Record Specifications</td>
<td>13-12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13.2.11 Record Formatting Field Descriptors</td>
<td>13-13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13.3 CARRIAGE CONTROL CHARACTERS FOR PRINTING ASCII RECORDS</td>
<td>13-14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHAPTER 14 DEVICE CONTROL STATEMENTS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14.1 INTRODUCTION</td>
<td>14-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14.2 REWIND STATEMENT</td>
<td>14-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14.3 UNLOAD STATEMENT</td>
<td>14-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14.4 BACKSPACE STATEMENT</td>
<td>14-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14.5 END FILE STATEMENT</td>
<td>14-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14.6 SKIP RECORD STATEMENT</td>
<td>14-2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14.7 SKIP FILE STATEMENT</td>
<td>14-3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14.8 BACKFILE STATEMENT</td>
<td>14-3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14.9 SUMMARY OF DEVICE CONTROL STATEMENTS</td>
<td>14-3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHAPTER 15 SUBPROGRAM STATEMENTS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15.1 INTRODUCTION</td>
<td>15-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15.1.1 Dummy and Actual Arguments</td>
<td>15-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15.2 STATEMENT FUNCTIONS</td>
<td>15-3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15.3 INTRINSIC FUNCTIONS (FORTRAN-10 DEFINED FUNCTIONS)</td>
<td>15-3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15.4 EXTERNAL FUNCTIONS</td>
<td>15-5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15.4.1 Basic External Functions (FORTRAN-10 Defined Functions)</td>
<td>15-6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15.4.2 Generic Function Names</td>
<td>15-6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
15.5 SUBROUTINE SUBPROGRAMS
15.5.1 Referencing Subroutines (CALL Statement) 
15.5.2 FORTRAN-10 Supplied Subroutines 
15.6 RETURN STATEMENT AND MULTIPLE RETURNS 
15.6.1 Referencing External FUNCTION Subprograms 
15.7 MULTIPLE SUBPROGRAM ENTRY POINTS (ENTRY STATEMENT) 

CHAPTER 16 BLOCK DATA SUBPROGRAMS
16.1 INTRODUCTION 
16.2 BLOCK DATA STATEMENT 

APPENDIX A ASCII-1968 CHARACTER CODE SET 

APPENDIX B SYMBOLIC NAMES 

APPENDIX C USING THE COMPILER
C.1 RUNNING THE COMPILER 
C.1.1 Switches Available with FORTRAN-10 
C.1.1.1 The /DEBUG Switch 
C.1.2 COMPILED-Class Commands 
C.2 READING A FORTRAN-10 LISTING 
C.2.1 Compiler Generated Variables 
C.3 ERROR REPORTING 
C.3.1 Fatal Errors and Warning Messages 
C.3.2 Message Summary 

APPENDIX D WRITING USER PROGRAMS
D.1 GENERAL PROGRAMMING CONSIDERATIONS 
D.1.1 Accuracy and Range of Double Precision Numbers 
D.1.2 Writing FORTRAN-10 Programs for Execution on Non-DEC Machines 
D.1.3 Using Floating Point DO Loops 
D.1.4 Computation of DO Loop Iterations 
D.1.5 Subroutines - Programming Considerations 
D.1.6 Reordering of Computations 
D.1.7 Dimensioning of Formal Arrays 
D.2 FORTRAN-10 GLOBAL OPTIMIZATION 
D.2.1 Optimization Techniques 
D.2.1.1 Elimination of Redundant Computations 
D.2.1.2 Reduction of Operator Strength 
D.2.1.3 Removal of Constant Computation From Loops 
D.2.1.4 Constant Folding and Propagation 
D.2.1.5 Removal of Inaccessible Code 
D.2.1.6 Global Register Allocation 
D.2.1.7 I/O Optimization 
D.2.1.8 Uninitialized Variable Detection 
D.2.1.9 Test Replacement 
D.2.2 Improper Function References 
D.2.3 Programming Techniques for Effective Optimization
## CONTENTS (Cont)

<table>
<thead>
<tr>
<th>D.3</th>
<th>INTERACTING WITH NON-FORTRAN-10 PROGRAMS AND FILES</th>
<th>D.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.3.1</td>
<td>Calling Sequences</td>
<td>D-8</td>
</tr>
<tr>
<td>D.3.2</td>
<td>Accumulator Usage</td>
<td>D-9</td>
</tr>
<tr>
<td>D.3.3</td>
<td>Argument Lists</td>
<td>D-10</td>
</tr>
<tr>
<td>D.3.4</td>
<td>Argument Types</td>
<td>D-11</td>
</tr>
<tr>
<td>D.3.5</td>
<td>Description of Arguments</td>
<td>D-12</td>
</tr>
<tr>
<td>D.3.6</td>
<td>Converting Existing MACRO-10 Libraries for use with FORTRAN-10</td>
<td>D-13</td>
</tr>
<tr>
<td>D.3.7</td>
<td>Mixing FORTRAN-10 and F40 Compiled Programs</td>
<td>D-18</td>
</tr>
<tr>
<td>D.3.8</td>
<td>Interaction with COBOL-10</td>
<td>D-18</td>
</tr>
<tr>
<td>D.3.8.1</td>
<td>Calling FORTRAN-10 Subprograms from COBOL-10 Programs</td>
<td>D-19</td>
</tr>
<tr>
<td>D.3.8.2</td>
<td>Calling COBOL-10 Subroutines from FORTRAN-10 Programs</td>
<td>D-20</td>
</tr>
<tr>
<td>D.3.9</td>
<td>LINK-10 Overlay Facilities</td>
<td>D-20</td>
</tr>
<tr>
<td>D.3.9.1</td>
<td>Conventions</td>
<td>D-20</td>
</tr>
<tr>
<td>D.3.10</td>
<td>FOROTS/FORESE Compatibility</td>
<td>D-21</td>
</tr>
<tr>
<td>D.3.10.1</td>
<td>FORTRAN-10/F40 Data File Compatibility</td>
<td>D-21</td>
</tr>
<tr>
<td>D.3.10.2</td>
<td>Conversion of FOROTS-Developed Data Files Into a Form Acceptable to FORSE</td>
<td>D-21</td>
</tr>
<tr>
<td>D.3.10.2.1</td>
<td>Conversion of FOROTS Sequential Mixed Files</td>
<td>D-21</td>
</tr>
<tr>
<td>D.3.10.2.2</td>
<td>Conversion of FOROTS Random Access ASCII Files</td>
<td>D-22</td>
</tr>
<tr>
<td>D.3.10.2.3</td>
<td>Conversion of FOROTS Random Access Binary Files</td>
<td>D-23</td>
</tr>
<tr>
<td>D.3.10.3</td>
<td>General Restrictions</td>
<td>D-23</td>
</tr>
</tbody>
</table>

## APPENDIX E  FOROTS

| E.1  | HARDWARE AND SOFTWARE REQUIREMENTS                | E-1  |
| E.2  | FEATURES OF FOROTS                                | E-1  |
| E.3  | ERROR PROCESSING                                  | E-2  |
| E.4  | INPUT/OUTPUT FACILITIES                           | E-2  |
| E.4.1 | Input/Output Channels Used Internally by FOROTS    | E-3  |
| E.4.2 | File Access Modes                                 | E-3  |
| E.4.2.1 | Sequential Transfer Mode                          | E-3  |
| E.4.2.2 | Random Access Mode                                | E-3  |
| E.5  | ACCEPTABLE TYPES OF DATA FILES AND THEIR FORMATS  | E-3  |
| E.5.1 | ASCII Data Files                                  | E-4  |
| E.5.2 | FORTRAN Binary Data Files                         | E-4  |
| E.5.2.1 | Format of Binary Files                            | E-4  |
| E.5.3 | Mixed Mode Data Files                             | E-11 |
| E.5.4 | Image Files                                       | E-11 |
| E.6  | USING FOROTS                                      | E-12 |
| E.6.1 | FOROTS Entry Points                               | E-12 |
| E.6.2 | Calling Sequences                                 | E-13 |
| E.6.3 | MACRO Calls for FOROTS Functions                  | E-14 |
| E.6.3.1 | Formatted/Unformatted Transfer Statements, Sequential Access Calling Sequences | E-15 |
| E.6.3.2 | NAMELIST Data Transfer Statements, Sequential Access Calling Sequences | E-16 |
| E.6.3.3 | Array Offsets and Factoring                       | E-16 |
| E.6.3.4 | Formatted/Unformatted Data Transfer Statements, Random Access Calling Sequences | E-18 |
| E.6.3.5 | Calling Sequences for Statements Which Use Default Devices | E-19 |
CONTENTS (Cont)

E.6.3.6 Calling Sequences for Statements which Position Magnetic
     Tape Units .............................................. E-20
E.6.3.7 List Directed Input/Output Statements ....................... E-20
E.6.3.8 Input/Output Data Lists ................................ E-21
E.6.3.9 OPEN and CLOSE Statements, Calling Sequences .......... E-24
E.6.3.10 Memory Allocation Routines ................................ E-25
E.6.3.11 Software Channel Allocation and De-allocation Routines E-25
E.7 LOGICAL/PHYSICAL DEVICE ASSIGNMENTS ....................... E-26

APPENDIX F FORDDT
F.1 INPUT FORMAT .......................................... F-2
F.1.1 Variables and Arrays .................................. F-2
F.1.2 Numeric Conventions .................................. F-3
F.1.3 Statement Labels and Source Line Numbers ................. F-3
F.2 NEW USER TUTORIAL ..................................... F-3
F.2.1 Basic Commands ....................................... F-3
F.3 FORDDT AND THE FORTRAN-10/DEBUG SWITCH .......... F-6
F.4 LOADING AND STARTING FORDDT .......................... F-7
F.5 SCOPE OF NAME AND LABEL REFERENCES .................. F-8
F.6 FORDDT COMMANDS ...................................... F-8
F.7 ENVIRONMENT CONTROL ................................ F-16
F.8 FORTRAN-10/IMIZE SWITCH ................................ F-16
F.9 FORDDT MESSAGES ........................................ F-16

APPENDIX G COMPILER MESSAGES

APPENDIX H FORTRAN-10 REAL TIME SOFTWARE

H.1 INTRODUCTION ........................................ H-1
H.2 USING FORRTF ........................................ H-1
H.2.1 Core .................................................. H-2
H.2.2 Modes .................................................. H-2
H.2.3 Priority Interrupt Level ................................ H-2
H.2.4 Masks ................................................. H-2
H.3 SUBROUTINES ........................................ H-2
H.3.1 LOCK .................................................. H-2
H.3.2 RTINIT .................................................. H-2
H.3.3 CONECT ................................................ H-3
H.3.4 RTSTRT ................................................ H-3
H.3.5 BLKRW .................................................. H-3
H.3.6 RTREAD ................................................ H-4
H.3.7 RTWRIT ................................................ H-4
H.3.8 STATO ................................................ H-4
H.3.9 STATI ................................................ H-4
H.3.10 RTSLP ................................................ H-5
H.3.11 RTWAKE .............................................. H-5
H.3.12 DDISMS ............................................... H-5
H.3.13 DISCON .............................................. H-5
H.3.14 UNLOCK .............................................. H-5
H.3.15 GETCOR, A Temporary Subroutine ....................... H-5
<table>
<thead>
<tr>
<th>Table No.</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>FORTRAN-10 Statement Categories</td>
<td>1-2</td>
</tr>
<tr>
<td>2-1</td>
<td>FORTRAN-10 Character Set</td>
<td>2-1</td>
</tr>
<tr>
<td>3-1</td>
<td>Constants</td>
<td>3-2</td>
</tr>
<tr>
<td>3-2</td>
<td>Use of Symbolic Names</td>
<td>3-6</td>
</tr>
<tr>
<td>4-1</td>
<td>Arithmetic Operations and Operators</td>
<td>4-1</td>
</tr>
<tr>
<td>4-2</td>
<td>Type of the Resultant Obtained From Mixed Mode Operations</td>
<td>4-3</td>
</tr>
<tr>
<td>4-3</td>
<td>Permitted Base/Exponent Type Combinations</td>
<td>4-4</td>
</tr>
<tr>
<td>4-4</td>
<td>Logical Operators</td>
<td>4-4</td>
</tr>
<tr>
<td>4-5</td>
<td>Logical Operations, Truth Table</td>
<td>4-5</td>
</tr>
<tr>
<td>4-6</td>
<td>Binary Logical Operations, Truth Table</td>
<td>4-6</td>
</tr>
<tr>
<td>4-7</td>
<td>Relational Operators and Operations</td>
<td>4-7</td>
</tr>
<tr>
<td>4-8</td>
<td>Hierarchy of FORTRAN-10 Operators</td>
<td>4-9</td>
</tr>
<tr>
<td>8-1</td>
<td>Rules for Conversion in Mixed Mode Assignments</td>
<td>8-2</td>
</tr>
<tr>
<td>10-1</td>
<td>FORTRAN-10 Logical Device Assignments</td>
<td>10-4</td>
</tr>
<tr>
<td>10-2</td>
<td>Summary of Read Statements</td>
<td>10-12</td>
</tr>
<tr>
<td>10-3</td>
<td>Summary of WRITE Statements</td>
<td>10-15</td>
</tr>
<tr>
<td>10-4</td>
<td>Summary of FORTRAN-10 I/O Statements</td>
<td>10-22</td>
</tr>
<tr>
<td>12-1</td>
<td>OPEN/CLOSE Statement Arguments</td>
<td>12-9</td>
</tr>
<tr>
<td>13-1</td>
<td>FORTRAN-10 Conversion Codes</td>
<td>13-3</td>
</tr>
<tr>
<td>13-2</td>
<td>Action of Field Descriptors On Sample Data</td>
<td>13-5</td>
</tr>
<tr>
<td>13-3</td>
<td>Numeric Field Codes</td>
<td>13-6</td>
</tr>
<tr>
<td>13-4</td>
<td>Descriptor Conversion of Real and Double Precision Data</td>
<td>13-8</td>
</tr>
<tr>
<td>13-5</td>
<td>FORTRAN-10 Print Control Characters</td>
<td>13-14</td>
</tr>
<tr>
<td>14-1</td>
<td>Summary of FORTRAN-10 Device Control Statements</td>
<td>14-3</td>
</tr>
<tr>
<td>15-1</td>
<td>Intrinsic Functions (FORTRAN-10 Defined Functions)</td>
<td>15-4</td>
</tr>
<tr>
<td>15-2</td>
<td>Basic External Functions (FORTRAN-10 Defined Functions)</td>
<td>15-8</td>
</tr>
<tr>
<td>15-3</td>
<td>FORTRAN-10 Library Subroutines</td>
<td>15-15</td>
</tr>
<tr>
<td>C-1</td>
<td>FORTRAN-10 Compiler Switches</td>
<td>C-2</td>
</tr>
<tr>
<td>C-2</td>
<td>Modifiers to /DEBUG Switch</td>
<td>C-3</td>
</tr>
<tr>
<td>D-1</td>
<td>Argument Types and Type Codes</td>
<td>D-11</td>
</tr>
<tr>
<td>D-2</td>
<td>Upward Compatibility (FORSE TO FOROTS)</td>
<td>D-21</td>
</tr>
<tr>
<td>D-3</td>
<td>Downward Compatibility (FOROTS TO FORSE)</td>
<td>D-22</td>
</tr>
<tr>
<td>E-2</td>
<td>FORTRAN Device Table</td>
<td>E-27</td>
</tr>
<tr>
<td>F-1</td>
<td>Table of Commands</td>
<td>F-1</td>
</tr>
<tr>
<td>H-1</td>
<td>Error Messages, Code Format and Full Message Format</td>
<td>H-6</td>
</tr>
</tbody>
</table>

November 1975
CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION

The FORTRAN-10 language set is compatible with and encompasses the standard set described in "American National Standard FORTRAN, X3.9-1966" (referred to as the 1966 ANSI standard set). FORTRAN-10 also provides many extensions and additions to the standard set which greatly enhance the usefulness of FORTRAN-10 and increases its compatibility with FORTRAN language sets implemented by other major computer manufacturers. In this manual the FORTRAN-10 extensions and additions to the 1966 ANSI standard set are printed in boldface italic type.

A FORTRAN-10 source program consists of a set of statements constructed using the language elements and the syntax described in this manual. A given FORTRAN-10 statement will perform any one of the following functions:

a. It will cause operations such as multiplication, division, and branching to be carried out.

b. It will specify the type and format of the data being processed.

c. It will specify the characteristics of the source program.

FORTRAN-10 statements are comprised of key words (i.e., words which are recognized by the compiler) used with elements of the language set: constants, variables, and expressions. There are two basic types of FORTRAN-10 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 the kind of subprograms that may be included in the program. The compilation of executable statements results in the creation of executable code in the object program. Nonexecutable statements provide information only to the compiler, they do not create executable code.

In this manual the FORTRAN-10 statements are grouped into twelve categories, each of which is described in a separate chapter. The name, definition, and chapter reference for each statement category are given in Table 1-1.

The basic FORTRAN-10 language elements (i.e., constants, variables, and expressions), the character set from which they may be formed, and the rules which govern their construction and use are described in Chapters 2 through 4.
<table>
<thead>
<tr>
<th>Category Name</th>
<th>Description</th>
<th>Chapter Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compilation Control</td>
<td>Statements in this category identify programs and indicate their end.</td>
<td>5</td>
</tr>
<tr>
<td>Statements</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specification Statements</td>
<td>Statements in this category declare the properties of variables, arrays, and functions.</td>
<td>6</td>
</tr>
<tr>
<td>DATA Statement</td>
<td>This statement assigns initial values to variables and array elements.</td>
<td>7</td>
</tr>
<tr>
<td>Assignment Statements</td>
<td>Statements in this category cause named variables and/or array elements to be replaced by specified (assigned) values.</td>
<td>8</td>
</tr>
<tr>
<td>Control Statements</td>
<td>Statements in this category determine the order of execution of the object program and terminate its execution.</td>
<td>9</td>
</tr>
<tr>
<td>Input/Output Statements</td>
<td>Statements in this category transfer data between internal storage and a specified input or output medium.</td>
<td>10</td>
</tr>
<tr>
<td>NAMELIST Statement</td>
<td><em>This statement establishes lists that are used with certain input/output statements to transfer data which appears in a special type of record.</em></td>
<td>11</td>
</tr>
<tr>
<td>File Control Statements</td>
<td><em>Statements in this category identify, open and close files and establish parameters for input and output operations between files and the processor.</em></td>
<td>12</td>
</tr>
<tr>
<td>FORMAT Statement</td>
<td>This statement is used with certain input/output statements to specify the form in which data appears in a FORTRAN record on a specified input/output medium.</td>
<td>13</td>
</tr>
<tr>
<td>Device Control Statements</td>
<td>Statements in this category enable the programmer to control the positioning of records or files on certain peripheral devices.</td>
<td>14</td>
</tr>
<tr>
<td>SUBPROGRAM Statements</td>
<td>Statements in this category enable the programmer to define functions and subroutines and their entry points.</td>
<td>15</td>
</tr>
<tr>
<td>BLOCK DATA Statements</td>
<td>Statements in this category are used to declare data specification subprograms which may initialize common storage areas.</td>
<td>16</td>
</tr>
</tbody>
</table>
CHAPTER 2
CHARACTERS AND LINES

2.1 CHARACTER SET

The digits, letters, and symbols recognized by FORTRAN-10 are listed in Table 2-1. The remainder of the ASCII-1968 character set\(^1\), although acceptable within literal constants or comment text, causes a fatal error in other contexts. An exception is CONTROL-Z, which, when used in Teletype input, means end-of-file.

NOTE

Lower case alphabetic characters are treated as upper case outside the context of Hollerith constants, literal strings, and comments.

Table 2-1
FORTRAN-10 Character Set

<table>
<thead>
<tr>
<th>Letters</th>
<th>Letters</th>
<th>Letters</th>
</tr>
</thead>
<tbody>
<tr>
<td>A,a</td>
<td>J,j</td>
<td>S,s</td>
</tr>
<tr>
<td>B,b</td>
<td>K,k</td>
<td>T,t</td>
</tr>
<tr>
<td>C,c</td>
<td>L,l</td>
<td>U,u</td>
</tr>
<tr>
<td>D,d</td>
<td>M,m</td>
<td>V,v</td>
</tr>
<tr>
<td>E,e</td>
<td>N,n</td>
<td>W,w</td>
</tr>
<tr>
<td>F,f</td>
<td>O,o</td>
<td>X,x</td>
</tr>
<tr>
<td>G,g</td>
<td>P,p</td>
<td>Y,y</td>
</tr>
<tr>
<td>H,h</td>
<td>Q,q</td>
<td>Z,z</td>
</tr>
<tr>
<td>I,i</td>
<td>R,r</td>
<td></td>
</tr>
</tbody>
</table>

(continued)

\(^1\) The complete ASCII-1968 character set is defined in the X3.4-1968 version of the "American National Standard for Information Interchange," and is given in Appendix A.
Table 2-1 (Cont)
FORTRAN-10 Character Set

<table>
<thead>
<tr>
<th>Digits</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbols</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>Exclamation Point</td>
</tr>
<tr>
<td>&quot;</td>
<td>Quotation Marks</td>
</tr>
<tr>
<td>#</td>
<td>Number Sign</td>
</tr>
<tr>
<td>$</td>
<td>Dollar Sign</td>
</tr>
<tr>
<td>&amp;</td>
<td>Ampersand</td>
</tr>
<tr>
<td>'</td>
<td>Apostrophe</td>
</tr>
<tr>
<td>(</td>
<td>Opening Parenthesis</td>
</tr>
<tr>
<td>)</td>
<td>Closing Parenthesis</td>
</tr>
<tr>
<td>*</td>
<td>Asterisk</td>
</tr>
<tr>
<td>+</td>
<td>Plus</td>
</tr>
<tr>
<td>,</td>
<td>Comma</td>
</tr>
<tr>
<td>-</td>
<td>Hyphen (Minus)</td>
</tr>
<tr>
<td>.</td>
<td>Period (Decimal Point)</td>
</tr>
<tr>
<td>/</td>
<td>Slant (slash)</td>
</tr>
<tr>
<td>:</td>
<td>Colon</td>
</tr>
<tr>
<td>;</td>
<td>Semicolon</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less Than</td>
</tr>
<tr>
<td>=</td>
<td>Equals</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater Than</td>
</tr>
<tr>
<td>^</td>
<td>Circumflex</td>
</tr>
</tbody>
</table>

Line Termination Characters

- Line Feed
- Form Feed
- Vertical Tab

Line Formatting Characters

- Carriage Return
- Horizontal Tab
- Blank

Note that horizontal tabs normally advance the character position pointer to the next position that is an even multiple of 8. An exception to this is the initial tab which is defined as a tab that includes or starts in character position 6. (Refer to Section 2.3.1 for a description of initial and continuation line types.) Tabs within literal specifications count as one character even though they may advance the character position pointer as many as eight places.
2.2 STATEMENT, DEFINITION, AND FORMAT

Source program statements are divided into physical lines. A line is defined as a string of adjacent character positions, terminated by the first occurrence of a line termination character regardless of context. Each line is divided into four fields:

<table>
<thead>
<tr>
<th>Line Character Positions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6 7 8 . . . . 70 71 72 73 .</td>
</tr>
<tr>
<td>Statement Label Field</td>
</tr>
<tr>
<td>Continuation Field</td>
</tr>
<tr>
<td>Statement Field</td>
</tr>
<tr>
<td>Remarks</td>
</tr>
</tbody>
</table>

2.2.1 Statement Label Field and Statement Numbers

A one to five digit number may be placed in the statement label field of an initial line to identify the statement. Any source program statement that is referenced by another statement must have a statement number. Statement numbers may be any number from 1 to 99999; leading zeroes and all blanks in the label field are ignored (e.g., the numbers 00105 and 105 are both accepted as statement number 105). The statement numbers given in a source program may be assigned in any order; however, each statement number must be unique with respect to all other statements in the program. Non executable statements, with the exception of FORMAT statements, cannot be labeled.

_When source programs are entered into a DECsystem-10 system via a standard user terminal, an initial tab may be used to skip all or part of the label field._

_If an initial tab is encountered during compilation, FORTRAN-10 examines the character immediately following the tab to determine the type of line being entered. If the character following the tab is one of the digits 1 through 9, FORTRAN-10 considers the line as a continuation line and the second character after the tab as the first character of the statement field. If the character following the tab is other than one of the digits 1 through 9, FORTRAN-10 considers the line to be an initial line and the character following the tab is considered to be the first character of the statement field. The character following the initial tab is considered to be in character position 6 in a continuation line, and in character position 7 in an initial line._

2.2.2 Line Continuation Field

Any alphanumeric character (except a blank or a zero) placed in this field (position 6) identifies the line as a continuation line (see Paragraph 2.3.1 for description).

_Whenever a tab is used to skip all or part of the label field of a continuation line, the next character entered must be one of the digits 1 through 9 to identify the line as a continuation line._

2.2.3 Statement Field

Any FORTRAN-10 statement may appear in this field. Blanks (spaces) and tabs do not affect compilation of the statement and may be used freely in this field for appearance purposes, with the exception of textual data given within either a literal or Hollerith specification where blanks and tabs are significant characters.

2.2.4 Remarks

In lines comprised of 73 or more character positions, only the first 72 characters are interpreted by FORTRAN-10. (Note that tabs generally occupy more than one character position, advancing the counter to the next character position that is an even multiple of eight.) All other characters in the line (character positions 73, 74 . . . etc.) are treated as remarks and do not affect compilation.
LINE TYPES

Initial and Continuation Lines

Note that remarks may also be added to a line in character positions 7 through 72 provided the text of the remark is preceded by the symbol ! (refer to Paragraph 2.3.3).

2.3 LINE TYPES
A line in a FORTRAN-10 source program can be

a. an initial line
b. a continuation line
c. a multi-statement line
d. a comment line
e. a debug line
f. a blank line.

Each of the foregoing line types is described in the following paragraphs.

2.3.1 Initial and Continuation Line Types
A FORTRAN-10 statement may occupy the statement fields of up to 20 consecutive lines. The first line in a multi-line statement group is referred to as the "initial" line; the succeeding lines are referred to as continuation lines.

Initial lines may be assigned a statement number and must have either a blank or a zero in their continuation line field (i.e., character position 6).

If an initial line is entered via a keyboard input device, an initial tab may be used to skip all or part of the label field. An initial tab used for this purpose must be followed immediately by a nonnumeric character (i.e., the first character of the statement field must be nonnumeric).

Continuation lines cannot be assigned statement numbers: they are identified by any alphanumeric character (except for a blank or zero) placed in character position 6 of the line (i.e., continuation line field). The label field of a continuation line is treated as remark text.

If a continuation line is being entered via a keyboard, an initial tab may be used to skip all or part of the label field; however, the tab must be followed immediately by a numeric character other than zero. The tab-numeric combination identifies the line as a continuation line.

Note that blank lines, comments, and debug lines that are treated like comments, i.e., debug lines that are not compiled with the rest of the program (refer to section 2.3.4), terminate a continuation sequence.

Following is an example of a four line FORTRAN-10 FORMAT statement using initial tabs:

```
105 FORMAT (1HI,17HI: INITIAL CHARGE = F10.6,10H COULOME,6X,
213H: RESISTANCE = F9.5,6H OHM/15H CAPACITANCE = F10.6,
38H FARAD,11X,13HI NDUCTANCE = F7.3,8H HENERY///
421H TI ME CURRENT/7H ME,10X,2HMA///)
```

Continuation Line Characters (i.e., 2, 3, and 4)

Version 4 FORTRAN

November 1975
2.3.2 Multi-Statement Lines

More than one FORTRAN-10 statement may be written in the statement field of one line. The rules for structuring a multi-statement line are:

a. successive statements must be separated by a semicolon (;)

b. only the first statement in the series can have a statement number

c. statements following the first statement cannot be a continuation of the preceding statement

d. the last statement in a line may be continued to the next line if the line is made a continuation line.

An example of a multi-statement line is:

450 DIST=RATE * TIME ; TIME=TIME+0.05 ; CALL PRIME(TIME,DIST)

2.3.3 Comment Lines and Remarks

Lines that contain descriptive text only are referred to as comment lines. Comment lines are commonly used to identify and introduce a source program, to describe the purpose of a particular set of statements, and to introduce subprograms.

The rules for structuring a comment line are:

a. One of the characters C (or c), $/,*!, or ! must be in character position 1 of the line to identify it as a comment line.

b. The text may be written into character positions 2 through the end of the line.

c. Comment lines may appear anywhere in the source program, but may not precede a continuation line because comments terminate a continuation sequence.

d. A large comment may be written as a sequence of any number of lines. However, each line must carry the identifying character (C,$/,*!, or !) in its first character position.

The following is an example of a comment that occupies more than one line.

CSUBROUTINE = A12
THE PURPOSE OF THIS SUBROUTINE IS
CTO FORMAT AND STORE THE RESULTS OF
CTEST PROGRAM HEAT TEST-1101

Comment lines are printed on all listings but are otherwise ignored by the compiler.
A remark may be added to any statement field, in character positions 7 through 72, provided the symbol ! precedes the text. For example, in the line

```fortran
IF(N.EQ.0)STOP! STOP IF CARD IS BLANK
```

The character group "Stop if card is blank" is identified as a remark by the preceding ! symbol. Remarks do not result in the generation of object program code, but they will appear on listings. The symbol !, indicating a remark, must appear outside the context of a literal specification.

Note that characters appearing in character positions 73 and beyond are automatically treated as remarks, so that the symbol ! need not be used (refer to Paragraph 2.2.4).

2.3.4 Debug Lines

As an aid in program debugging a D (or d) in character position 1 of any line causes the line to be interpreted as a comment line, i.e., not compiled with the rest of the program unless the / Include switch appears in the command string. (Refer to Appendix C for a description of the compile switch options.) When the / Include switch is present in the command string the D (or d) in character position 1 is treated as a blank so that the remainder of the line is compiled as an ordinary (noncomment) line. Note that the initial and all continuation lines of a debug statement must contain a D (or d) in character position 1.

2.3.5 Blank Lines

Lines consisting of only blanks, tabs, or no characters may be inserted anywhere in a FORTRAN-10 source program except immediately preceding a continuation line, because blank lines are by definition initial lines and as such terminate a continuation sequence. Blank lines are used for formatting purposes only; they cause blank lines to appear in their corresponding positions in object program listings; otherwise, they are ignored by the compiler.

2.3.6 Line-Sequenced Input

FORTRAN-10 optionally accepts DECsystem-10 line-sequenced files as produced by LINED or BASIC. These sequence numbers are used in place of the listing line numbers normally generated by FORTRAN-10.

2.4 ORDERING OF FORTRAN-10 STATEMENTS

The order in which FORTRAN-10 Statements appear in a program unit is important. That is, certain types of statements have to be processed before others in order to guarantee that compilation takes place as expected. The proper sequence for FORTRAN-10 statements is summarized by the following diagram.
### Ordering of Statements

<table>
<thead>
<tr>
<th>PROGRAM, FUNCTION, Subprogram, or BLOCK DATA Statements</th>
<th>IMPLICIT Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comment Lines</td>
<td>PARAMETER Statements</td>
</tr>
<tr>
<td>FORMAT Statements</td>
<td>DIMENSION, COMMON, EQUIVALENCE, EXTERNAL, NAMELIST, or Type Specification Statements</td>
</tr>
<tr>
<td>DATA Statements</td>
<td>Statement Function Definitions</td>
</tr>
<tr>
<td></td>
<td>Executable Statements</td>
</tr>
</tbody>
</table>

**END Statement**

Horizontal lines indicate the order in which FORTRAN-10 statements must appear. That is, the statements in the horizontal sections cannot be interspersed. For example, all PARAMETER statements must appear after all IMPLICIT statements and before any DATA statements, i.e., PARAMETER, IMPLICIT, and DATA statements cannot be interspersed. Statement function definitions must appear after IMPLICIT statements and before executable statements.

Vertical lines indicate the way in which certain types of statements may be interspersed. For example, DATA statements may be interspersed with statement function definitions and executable statements. FORMAT statements may be interspersed with IMPLICIT statements, parameter statements, other specification statements, DATA statements, statement function definitions, and executable statements. The only restrictions on the placement of FORMAT statements are that they must appear after any PROGRAM, FUNCTION, SUBPROGRAM, and BLOCK DATA statements, and before the END statement.

**Special Cases:**

a. The placement of an INCLUDE statement is dictated by the types of statements to be INCLUDED.

b. The ENTRY statement is allowed only in functions or subroutines. All executable references to any of the dummy parameters must physically follow the ENTRY statement unless the references appear in the function definition statement, the subroutine, or in a preceding ENTRY statement.

c. BLOCK DATA subprograms cannot contain any executable statements, statement functions, FORMAT statements, EXTERNAL statements, or NAMELIST statements. (Refer to section 16.1.)

FORTRAN-10 expects users to adhere to the foregoing ordering guidelines and issues warning messages when statements are out of place.
CHAPTER 3

DATA TYPES, CONSTANTS, SYMBOLIC NAMES, VARIABLES, AND ARRAYS

3.1 DATA TYPES

The data types permitted in FORTRAN-10 source programs are

a. integer
b. real
c. double precision
d. complex
e. octal
f. double octal
g. literal
h. statement label, and
i. logical.

The use and format of each of the foregoing data types are discussed in the descriptions of the constant having the same data type (Paragraphs 3.2.1 through 3.2.8).

3.2 CONSTANTS

Constants are quantities that do not change value during the execution of the object program.

The constants permitted in FORTRAN-10 are listed in Table 3-1.
Table 3-1
Constants

<table>
<thead>
<tr>
<th>Category</th>
<th>Constant(s) Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric</td>
<td>Integer, real, double precision, complex, and <em>octal</em></td>
</tr>
<tr>
<td>Truth Values</td>
<td>Logical</td>
</tr>
<tr>
<td>Literal Data</td>
<td>Literal</td>
</tr>
<tr>
<td>Statement Label</td>
<td>Address of FORTRAN statement label</td>
</tr>
</tbody>
</table>

### 3.2.1 Integer Constants

An integer constant is a string of from one to eleven digits which represents a whole decimal number (i.e., a number without a fractional part). Integer constants must be within the range of $-2^{31} - 1$ to $2^{31} - 1$ (i.e., $-34359738367$ to $34359738367$). Positive integer constants may optionally be signed; negative integer constants must be signed. Decimal points, commas, or other symbols are not permitted on integer constants (except for a preceding sign, + or -). Examples of *valid* integer constants are:

- 345
- +345
- -345

Examples of *invalid* integer constants are:

- +345.  (use of decimal point)
- 3,450  (use of comma)
- 34.5   (use of decimal point; not a whole number)

### 3.2.2 Real Constants

A real constant may have any of the following forms:

a. A basic real constant: a string of decimal digits followed immediately by a decimal point which may optionally be followed by a fraction (e.g., 1557.00).

b. A basic real constant followed immediately by a decimal integer exponent written in E notation (i.e., exponential notation) form (e.g., 1559.E2).

c. An integer constant (no decimal point) followed by a decimal integer exponent written in E notation (e.g., 1559E2).

Real constants may be of any size; however, each will be rounded to fit the precision of 27 bits (i.e., 7 to 9 decimal digits).

Precision for real constants is maintained (approximately) to eight digits.¹

---

¹ This is an approximation, the exact precision obtained will depend on the numbers involved.
The exponent field of a real constant written in E notation form cannot be empty (i.e., blank), it must be either a zero or an integer constant. The magnitude of the exponent must be greater than \(-38\) and equal to or less than \(+38\) (i.e., \(-38 \leq n \leq 38\)). The following are examples of valid real constants.

\[
\begin{align*}
-98.765 & \quad \text{(i.e., 7.)} \\
7.0E+0 & \quad \text{(i.e., .0007)} \\
5E+5 & \quad \text{(i.e., 500000.)} \\
50115. & \\
50.E1 & \quad \text{(i.e., 500.)}
\end{align*}
\]

The following are examples of invalid real constants.

\[
\begin{align*}
72.6E75 & \quad \text{(exponent is too large)} \\
.375E & \quad \text{(exponent incorrectly written)} \\
500 & \quad \text{(no decimal point given)}
\end{align*}
\]

### 3.2.3 Double Precision Constants

Constants of this type are similar to real constants written in E notation form; the direct differences between these two constants are:

- Each double precision constant occupies two storage locations.
- The letter D, instead of E, is used in double precision constants to identify a decimal exponent.

Both the letter D and an exponent (even of zero) are required in writing a double precision constant. The exponent given need only be signed if it is negative; its magnitude must be greater than \(-38\) and equal to or less than \(+38\) (i.e., \(-38 < n \leq 38\)). The range of magnitude permitted a double precision constant depends on the type of processor present in the system on which the source program is to be compiled and run. The permitted ranges are:

<table>
<thead>
<tr>
<th>Processor</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>KA10</td>
<td>(1.97 \times 10^{-31}) to (3.4 \times 10^{+38})</td>
</tr>
<tr>
<td>KI10</td>
<td>(0.14 \times 10^{-38}) to (3.4 \times 10^{+38})</td>
</tr>
</tbody>
</table>

The following are valid examples of double precision constants.

- 7.9D03 (i.e., 7900)
- 7.9D+03 (i.e., 7900)
- 7.9D-3 (i.e., .0079)
- 79D03 (i.e., 79000)
- 79D0 (i.e., 79)

The following are invalid examples of double precision constants.

- 7.9D99 (exponent is too large)
- 7.9E5 (denotes a single precision constant)
CHAPTER 3

COMPLEX and OCTAL Constants

3.2.4 Complex Constants
A complex constant can be represented by an ordered pair of integer, real or octal constants written within parentheses and separated by a comma. For example, (.70712, -.70712) and (8.763E3, 2.297) are complex constants.

In a complex constant the first (leftmost) real constant of the pair represents the real part of the number, the second real constant represents the imaginary part of the number. Both the real and imaginary parts of a complex constant can be signed.

The real constants that represent the real and imaginary parts of a complex constant occupy two consecutive storage locations in the object program.

3.2.5 Octal Constants
Octal numbers (radix 8) may be used as constants in arithmetic expressions, logical expressions, and data statements. Octal numbers up to 12 digits in length are considered standard octal constants; they are stored right-justified in one processor storage location. When necessary, standard octal constants are padded with leading zeroes to fill their storage location.

If more than 12 digits are specified in an octal number, it is considered a double octal constant. Double octal constants occupy two storage locations and may contain up to 24 right-justified octal digits; zeroes are added to fill any unused digits.

If a single octal constant is to be assigned to a double precision or complex variable, it is stored, right-justified, in the high order word of the variable. The low order portion of the variable is set to zero.

If a double octal constant is to be assigned to a double precision or complex variable, it is stored right-justified starting in the low order (rightmost) word and precedes leftwards into the high order word.

All octal constants must be

a. preceded by a double quote (") to identify the digits as octal (e.g., "777"), and

b. signed if negative but optionally signed if positive.

The following are examples of valid octal constants:

"123456700007
"123456700007
"+12345
"-7777
"-7777

The following are examples of invalid octal constants:

"12368 (contains a radix digit)
7777 (no identifying double quotes)

When an octal constant is used as an operand in an expression, its form (i.e., bit pattern) is not converted to accommodate it to the type of any other operand. For example, the subexpression (A+"202 400 000 000) has as its result the sum of A with the floating point number 2.0; while the subexpression (B+"202 400 000 000) has as its result the sum of I with a large integer.
3.2.6 Logical Constants

The Boolean values of truth and falsehood are represented in FORTRAN-10 source programs as the logical constants .TRUE. and .FALSE.. Logical constants are always written enclosed by periods as in the preceding sentence.

Logical quantities may be operated on in arithmetic and logical statements. Only the sign bit of a numeric used in a logical IF statement is tested to determine if it is true (sign is negative) or false (sign is positive).

3.2.7 Literal Constants

A literal constant may be either of the following:

a. A string of alphanumeric and/or special characters contained within apostrophes (e.g., 'TEST#5').

b. A Hollerith literal, which is written as a string of alphanumeric and/or special characters preceded by nH (e.g., nHstring). In the prefix nH, the letter n represents a number which specifies the exact number of characters (including blanks) that follow the letter H; the letter H identifies the literal as a Hollerith literal. The following are examples of Hollerith literals:

   2HAB,
   14HLOAD TEST #124,
   6H#124-A

   NOTE
   A tab (→) in a Hollerith literal is counted as one character
   (e.g., 3H→LAB).

Literal constants may be entered into DATA statements as a string of

a. up to ten 7-bit ASCII characters for complex or double precision type variables, and

b. up to five 7-bit ASCII characters for all other type variables.

The 7-bit ASCII characters which comprise a literal constant are stored left-justified (starting in the high order word of a 2-word precision or complex literal) with blanks placed in empty character positions. Literal constants that occupy more than one variable are stored in successive variables in the list. The following example illustrates how the string of characters

   A LITERAL OF MANY CHARACTERS

is stored in a six-element array called A.

   DIMENSION A(6)
   DATA A/'A LITERAL OF MANY CHARACTERS'/

   A(1) is set to 'A_LIT'
   A(2) is set to 'ERAL_'
   A(3) is set to 'OF_MA'
   A(4) is set to 'NY_CH'
   A(5) is set to 'ARACT'
   A(6) is set to 'ERS_-'
3.2.8 Statement Label Constants

Statement labels are numeric identifiers that represent program statement numbers.

Statement label constants are written as a string of from one to five decimal digits which are preceded by either a dollar sign ($) or an ampersand (&). For example, either $11992 or &11992 may be used as statement labels.

Statement label constants are used only in the argument list of CALL statements to define the statement to return to in a multiple RETURN statement. (Refer to Chapter 15.)

3.3 SYMBOLIC NAMES

- Symbolic names may consist of any alphanumeric combination of from one to six characters. More than six characters may be given but FORTRAN-10 will ignore all but the first six. The first character of a symbolic name must be an alphabetic character.

The following are examples of legal symbolic names:

A12345
IAMBIC
ABLE

The following are examples of illegal symbolic names:

#AMBIC (symbol used as first character)
1AB (number used as first character)

Symbolic names are used to identify specific items of a FORTRAN-10 source program; these items, together with an example of a symbolic name and text reference for each, are listed in Table 3-2.

Table 3-2
Use of Symbolic Names

<table>
<thead>
<tr>
<th>Symbolic Names Can Identify</th>
<th>For Example</th>
<th>For a detailed description See Paragraph</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. A Variable</td>
<td>PI, CONST, LIMIT</td>
<td>3.4</td>
</tr>
<tr>
<td>2. An Array</td>
<td>TAX</td>
<td>3.5</td>
</tr>
<tr>
<td>3. An Array element</td>
<td>TAX (NAME, INCOME)</td>
<td>3.5.1</td>
</tr>
<tr>
<td>4. Functions</td>
<td>MYFUNC, VALFUN</td>
<td>15.2</td>
</tr>
<tr>
<td>5. Subroutines</td>
<td>CALCSB, SUB2, LOOKUP</td>
<td>15.5</td>
</tr>
<tr>
<td>6. External</td>
<td>SIN, ATAN, COSH</td>
<td>15.4</td>
</tr>
<tr>
<td>7. COMMON Block Names</td>
<td>DATAR, COMDAT</td>
<td>6.5</td>
</tr>
</tbody>
</table>

3.4 VARIABLES

A variable is a datum (i.e., storage location) that is identified by a symbolic name and is not an array or an array element. Variables specify values which are assigned to them by either arithmetic statements (Chapter 8), DATA statements (Chapter 7), or at run time via I/O references (Chapter 10). Before a variable is assigned a value, it is termed an undefined variable and should not be referenced except to assign a value to it.

If an undefined variable is referenced an unknown value (i.e., garbage) will be obtained.
The value assigned to a variable may be either a constant or the result of a calculation which is performed during the execution of the object program. For example, the statement \texttt{IAB=5} assigns the constant 5 to the variable \texttt{IAB}; in the statement \texttt{IAB=5+B}, however, the value of \texttt{IAB} at a given time will depend on the value of variable \texttt{B} at the time the statement was last executed.

The type of a variable is the type of the contents of the datum which it identifies. Variables may be

a. integer
b. real
c. logical
d. double precision, or
e. complex.

The type of a variable may be declared using either implicit or explicit type declaration statements (Chapter 6). However, if type declaration statements are not used, the following convention is assumed by FORTRAN-10:

a. Variable names which begin with the letters I, J, K, L, M, or N are integer variables.
b. Variable names which begin with any letter other than I, J, K, L, M, or N are real variables.

Examples of determining the type of a variable according to the foregoing convention are given in the following table.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Beginning Letter</th>
<th>Assumed Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ITEMP</td>
<td>I</td>
<td>Integer</td>
</tr>
<tr>
<td>OTEMP</td>
<td>O</td>
<td>Real</td>
</tr>
<tr>
<td>KA123</td>
<td>K</td>
<td>Integer</td>
</tr>
<tr>
<td>AABLE</td>
<td>A</td>
<td>Real</td>
</tr>
</tbody>
</table>

3.5 ARRAYS

An array is an ordered set of data identified by an array name. Array names are symbolic names and must conform to the rules given in Paragraph 3.3 for writing symbolic names.

Each datum within an array is called an array element. Like variables, array elements may be assigned values; before an array element is assigned a value it is considered to be undefined and should not be referenced until it has been assigned a value. If a reference is made to an undefined array element the value of the element will be unknown and unpredictable (i.e., garbage).

Each element of an array is named by using the array name together with a subscript that describes the position of the element within the array.

3.5.1 Array Element Subscripts

The subscript of an array element identifier is given, within parentheses, as either one subscript quantity or a set of subscript quantities delimited by commas. The parenthesized subscript is written immediately after the array name. The general form of an array element name is \texttt{AN (S1, S2,...Sn)}, where \texttt{AN} is the array name and \texttt{S1} through \texttt{Sn} represent \texttt{n} number of subscript quantities. Any number of subscript quantities may be used in an element name; however, the number used must always equal the number of dimensions (Paragraph 3.5.2) specified for the array.
CHAPTER 3

Dimensioning Arrays

A subscript can be any compound expression (Chapter 4), for example:

a. Subscript quantities may contain arithmetic expressions that involve addition, subtraction, multiplication, division, and exponentiation. For example, \((A+B, C*5, D/2)\) and \((A**3, (B/4+C)*E, 3)\) are valid subscripts.

b. \textit{Arithmetic expressions used in array subscripts may be of any type but noninteger expressions (including complex) are converted to integer when the subscript is evaluated.}

c. A subscript may contain function references (Chapter 14). For example: \(\text{TABLE}(\sin(A)*B, 2, 3)\) is a valid array element identifier.

d. Subscripts may contain array element identifiers nested to any level as subscripts. For example, in the subscript \(((I(J(K(L)))), A+B, C)\) the first subscript quantity given is a nested 3-level subscript.

The following are examples of \textit{valid} array element subscripts:

a. \(IAB(1,5,3)\)

b. \(ABLE(A)\)

c. \(\text{TABLE}(10/C+K**2, A, B)\)

d. \(\text{MAT}(A, AB(2*L), 3*\text{TAB}(A, M+1, D), 55)\)

3.5.2 Dimensioning Arrays

The size (i.e., number of elements) of an array must be declared in order to enable FORTRAN-10 to reserve the needed amount of locations in which to store the array. Arrays are stored as a series of sequential storage locations. Arrays, however, are visualized and referenced as if they were single or multi-dimensional rectilinear matrices, dimensioned on a row, column, and plane basis. For example, the following figure represents a 3-row, 3-column, 2-plane array.

![3x3x2 Array Diagram](image)

The size (i.e., number of elements) of an array is specified by an array declarator written as a subscripted array name. In an array declarator, however, each subscript quantity is a dimension of the array and must be either an integer, a variable, or an integer constant.

For example, \(\text{TABLE}(I,J,K)\) and \(\text{MATRIX}(10,7,3,4)\) are valid array declarators.

The total number of elements which comprise an array is the product of the dimension quantities given in its array declarator. For example, the array \(IAB\) dimensioned as \(IAB(2,3,4)\) has 24 elements \((2 \times 3 \times 4 = 24)\).
Arrays are dimensioned only in the specification statements DIMENSION, COMMON, and type declaration (Chapter 6). Subscripted array names appearing in any of the foregoing statements are array declarators; subscripted array names appearing in any other statements are always array element identifiers. In array declarators the position of a given subscript quantity determines the particular dimension of the array (e.g., row, column, plane) which it represents. The first three subscript positions specify the number of rows, columns, and planes which comprise the named array; each following subscript given then specifies a set comprised of n-number (value of the subscript) of the previously defined sets. For example:

<table>
<thead>
<tr>
<th>The Dimension Declarator</th>
<th>Specifies the Array(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAB (2)</td>
<td>1 2</td>
</tr>
<tr>
<td>TAB (2,2)</td>
<td>1,1 1,2 2,1 2,2</td>
</tr>
<tr>
<td>TAB (2,2,2)</td>
<td>1,1,1 1,2,1 2,1,1</td>
</tr>
<tr>
<td>TAB (2,2,2,2)</td>
<td>1,1,1,1 1,2,2,1</td>
</tr>
</tbody>
</table>

**NOTE**

FORTRAN-10 permits any number of dimensions in an array declarator.

3.5.3 Order of Stored Array Elements

The elements of an array are arranged in storage in ascending order, with the value of the first subscript quantity varying between its maximum and minimum values most rapidly, and the value of the last given subscript quantity increasing to its maximum value least rapidly. For example, the elements of the array dimensioned as I(2,3) are stored in the following order:

I(1,1) → I(2,1) → I(1,2) → I(2,2) → I(1,3) → I(2,3)

The following list describes the order in which the elements of the three-dimensional array (B(3,3,3)) are stored:

- B (1,1,1) B (2,1,1) B (3,1,1)
- B (1,2,1) B (2,2,1) B (3,2,1)
- B (1,3,1) B (2,3,1) B (3,3,1)
- B (1,1,2) B (2,1,2) B (3,1,2)
- B (1,2,2) B (2,2,2) B (3,2,2)
- B (1,3,2) B (2,3,2) B (3,3,2)
- B (1,1,3) B (2,1,3) B (3,1,3)
- B (1,2,3) B (2,2,3) B (3,2,3)
- B (1,3,3) B (2,3,3) B (3,3,3)
4.1 ARITHMETIC(expressions)

Arithmetic expressions may be either simple or compound. Simple arithmetic expressions consist of an operand which may be

a. a constant
b. a variable
c. an array element
d. a function reference (see Chapter 14 for description), or
e. an arithmetic or logical expression written within parentheses.

Operands may be of type integer, real, double precision, complex, octal, or literal.

The following are valid examples of simple arithmetic expressions:

105       (integer constant)
IAB       (integer variable)
TABLE (3, 4, 5) (array element)
SIN (X)   (function reference)
(A+B)     (a parenthesized expression)

A compound arithmetic expression consists of two or more operands combined by arithmetic operators. The arithmetic operations permitted in FORTRAN-10 and the operator recognized for each are given in Table 4-1.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operator</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Exponentiation</td>
<td>** or ↑</td>
<td>A**B or A↑B</td>
</tr>
<tr>
<td>2. Multiplication</td>
<td>*</td>
<td>A*B</td>
</tr>
<tr>
<td>3. Division</td>
<td>/</td>
<td>A/B</td>
</tr>
<tr>
<td>4. Addition</td>
<td>+</td>
<td>A+B</td>
</tr>
<tr>
<td>5. Subtraction</td>
<td>-</td>
<td>A-B</td>
</tr>
</tbody>
</table>
4.1.1 Rules for Writing Arithmetic Expressions

The following rules must be observed in structuring compound arithmetic expressions:

a. The operands comprising a compound arithmetic expression may be of different types. Table 4-2 illustrates all permitted combinations of data types and the type assigned to the result of each.

NOTE

Only one combination of data types, double precision with complex, is prohibited in FORTRAN-10.

b. An expression cannot contain two adjacent and unseparated operators. For example, the expression $A^*/B$ is not permitted.

c. All operators must be included, no operation is implied. For example, the expression $A(B)$ does not specify multiplication although this is implied in standard algebraic notation. The expression $A^* (B)$ is required to obtain a multiplication of the elements.

d. In using 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 combination and the type of the result of each combination is given in Table 4-3.

4.2 LOGICAL EXPRESSIONS

Logical expressions may be either simple or compound. Simple logical expressions consist of a logical operand which may be a logical type

   a. constant
   b. variable
   c. array element
   d. function reference (see Chapter 15), or
   e. another expression written within parentheses.

Compound logical expressions consist of two or more operands combined by logical operators.

The logical operators permitted by FORTRAN-10 and a description of the operation each provides are given in Table 4-4.
Table 4-3
Permitted Base/Exponent Type Combinations

<table>
<thead>
<tr>
<th>Base Operand</th>
<th>Integer</th>
<th>Real</th>
<th>Double Precision</th>
<th>Complex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Integer</td>
<td>Real</td>
<td>Double Precision</td>
<td>Complex</td>
</tr>
<tr>
<td>Real</td>
<td>Real</td>
<td>Real</td>
<td>Double Precision</td>
<td>Complex</td>
</tr>
<tr>
<td>Double Precision</td>
<td>Double Precision</td>
<td>Double Precision</td>
<td>Double Precision</td>
<td>Complex</td>
</tr>
<tr>
<td>Complex</td>
<td>Complex</td>
<td>Complex</td>
<td>(Undefined)</td>
<td>Complex</td>
</tr>
</tbody>
</table>

Table 4-4
Logical Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.AND.</td>
<td>AND operator. Both of the logical operands combined by this operator must be true to produce a true result.</td>
</tr>
<tr>
<td>.OR.</td>
<td>Inclusive OR operator. If either or both of the logical operands combined by .OR. are true, the result will be true.</td>
</tr>
<tr>
<td>.XOR.</td>
<td>Exclusive OR operator. If either one but not both of the logical operands combined by .XOR. is true, the result will be true.</td>
</tr>
<tr>
<td>.EQV.</td>
<td>Equivalence operator. If the logical operands being combined by .EQV. are both the same (i.e., both are true or both are false) the result will be true.</td>
</tr>
<tr>
<td>.NOT.</td>
<td>Complementation operator. This operator is used as a prefix that specifies complementation (i.e., inversion) of the item (operand or expression) which it modifies.</td>
</tr>
</tbody>
</table>

Logical expressions are written in the general form \( P_1 \ .OP. \ P_2 \) where \( P \) is a logical operand and .OP. is any logical operator but .NOT. The .NOT. operator complements the value of a logical operand and must be written immediately before the operand which it modifies (e.g., .NOT.P). A truth table illustrating all possible logical combinations of two logical operands (P and Q) and the resultant of each combination is given in Table 4-5.

When an operand of a logical expression is double precision or complex, only the high order word of the operand is used in the specified logical operation.

The assignment of a .TRUE. or a .FALSE. value to a given operand is based only on the sign of the numeric representation of the operand.
### Table 4-5
#### Logical Operations, Truth Table

<table>
<thead>
<tr>
<th>The result of the expression:</th>
<th>When P is:</th>
<th>and Q is:</th>
<th>Is:</th>
</tr>
</thead>
<tbody>
<tr>
<td>.NOT.P</td>
<td>True</td>
<td>(Not Applicable)</td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>P.AND.Q</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td></td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>P.OR.Q</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td></td>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td></td>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td></td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>P.XOR.Q</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td></td>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td></td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>P.EQV.Q</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td></td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
</tbody>
</table>

**Examples**

Assume the following variables:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>REAL, RUN</td>
<td>Real</td>
</tr>
<tr>
<td>I, J, K</td>
<td>Integer</td>
</tr>
<tr>
<td>DP, D</td>
<td>Double Precision</td>
</tr>
<tr>
<td>L, A, B</td>
<td>Logical</td>
</tr>
<tr>
<td>CPX, C</td>
<td>Complex</td>
</tr>
</tbody>
</table>

Examples of valid logical expressions comprised of the foregoing variables are:

- L.AND.B
- (REAL*I) .XOR. (DP+K)
- L.AND. A .OR. .NOT. (I-K)
Logical functions are performed bit-wise on the full 36-bit binary processor representation of the operands involved. The result of a logical operation is found by performing the specified function, simultaneously, for each of the corresponding bits in each operand. For example, consider the expression A=C.OR.D, where C= "456 and D= "201. The operation performed by the processor and the result is:

<table>
<thead>
<tr>
<th>Word Bits</th>
<th>0</th>
<th>1</th>
<th>24</th>
<th>25</th>
<th>26</th>
<th>27</th>
<th>28</th>
<th>29</th>
<th>30</th>
<th>31</th>
<th>32</th>
<th>33</th>
<th>34</th>
<th>35</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operand C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operand D</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Result A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4-6 is a truth table that illustrates all possible logical combinations of two one-bit binary operands (P and Q) and gives the result of each combination.

<table>
<thead>
<tr>
<th>The result of the expression:</th>
<th>When P is:</th>
<th>And Q is:</th>
<th>Is:</th>
</tr>
</thead>
<tbody>
<tr>
<td>.NOT.P</td>
<td>1</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>P.AND.Q</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>P.OR.Q</td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>P.XOR.Q</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>P.EQV.Q</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

4.2.1 Relational Expressions

Relational expressions are comprised of two expressions combined by a relational operator. The relational operator permits the programmer to test, quantitatively, the relationship between two arithmetic expressions.

The result of a relational expression is always a logically true or false value.
In FORTRAN-10, relational operators may be written either as a two-letter mnemonic enclosed within periods (e.g., .GT.) or symbolically using the symbols >, <, = and #. Table 4-7 lists both the mnemonic and symbolic forms of the FORTRAN-10 relational operators and specifies the type of quantitative test performed by each operator.

<table>
<thead>
<tr>
<th>Operators</th>
<th>Symbolic</th>
<th>Relation Tested</th>
</tr>
</thead>
<tbody>
<tr>
<td>.GT.</td>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>.GE.</td>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>.LT.</td>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>.LE.</td>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>.EQ.</td>
<td>==</td>
<td>Equal to</td>
</tr>
<tr>
<td>.NE.</td>
<td>#</td>
<td>Not equal to</td>
</tr>
</tbody>
</table>

Relational expressions are written in the general form A1 .OP. A2, where A represents an arithmetic operand and .OP. is a relational operator.

Arithmetic operands of type integer, real, and double precision may be mixed in relational expressions.

Complex operands may be compared using only the operators .EQ.(=) and .NE.(#). Complex quantities are equal if the corresponding parts of both words are equal.

Examples
Assume the following variables:

<table>
<thead>
<tr>
<th>Variables</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>REAL, RON</td>
<td>Real</td>
</tr>
<tr>
<td>I, J, K</td>
<td>Integer</td>
</tr>
<tr>
<td>DP, D</td>
<td>Double Precision</td>
</tr>
<tr>
<td>L, A, B</td>
<td>Logical</td>
</tr>
<tr>
<td>CPX, C</td>
<td>Complex</td>
</tr>
</tbody>
</table>

Examples of valid relational expressions comprised of the foregoing variables are:

\[(\text{REAL}) \ .GT. \ 10\]
\[ I == 5\]
\[ C \ .EQ. \ CPX\]

Examples of invalid relational expressions comprised of the foregoing variables are:

\[(\text{REAL}) \ .GT \ 10\] (closing period missing from operator)
\[ C > CPX\] (complex operands can only be combined by .EQ. and .NE. operators)
Examples of valid expressions in which both logical and relational operators are used to combine the foregoing variables are:

\[
(1 \cdot \text{GT} \cdot 10) \cdot \text{AND} \cdot (J < K) \\
((I \cdot \text{RON}) = (1/I)) \cdot \text{OR} \cdot K \\
(I \cdot \text{AND} \cdot K) \#((\text{REAL}) \cdot \text{OR} \cdot (\text{RON})) \\
C \#CPX \cdot \text{OR} \cdot \text{RON}
\]

4.3 EVALUATION OF EXPRESSIONS

The order of computation of a FORTRAN-10 expression is determined by

a. the use of parentheses

b. an established hierarchy for the execution of arithmetic, relational, and logical operations and

c. the location of operators within an expression.

4.3.1 Parenthesized Subexpressions

In an expression all subexpressions written within parentheses are evaluated first. When parenthesized subexpressions are nested (one contained within another) the most deeply nested subexpression is evaluated first, the next most deeply nested subexpression is evaluated second and so on, until the value of the final parenthesized expression is computed. When more than one operator is contained by a parenthesized subexpression, the required computations are performed according to the hierarchy assigned operators by FORTRAN-10 (Paragraph 4.3.2).

Example:
The separate computations performed in evaluating the expression

\[
A + B \div ((A/B) + C) - C
\]

are:

a. \(A / B = R1\)

b. \(R1 + C = R2\)

c. \(B / R2 = R3\)

d. \(R3 - C = R4\)

e. \(A + R4 = R5\)

NOTE

R1 through R5 represent the interim and final results of the computations performed.

4.3.2 Hierarchy of Operators

The following hierarchy (i.e., order of execution) is assigned to the classes of FORTRAN-10 operators:

- first — arithmetic operators
- second — relational operators
- third — logical operators
The precedence assigned to the individual operators of the foregoing classes is specified (from highest to lowest) in Table 4-8.

With the exception of integer division and exponentiation, all operations on expressions or subexpressions involving operators of equal precedence are computed in any order that is algebraically correct.

A subexpression of a given expression may be computed in any order. For example, in the expression \((F(X) + A^*B)\) the function reference may be computed either before or after \(A^*B\).

<table>
<thead>
<tr>
<th>Class</th>
<th>Level</th>
<th>Symbol or Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARITHMETIC</td>
<td>First</td>
<td>**</td>
</tr>
<tr>
<td></td>
<td>Second</td>
<td>-(unary minus) and +(unary plus)</td>
</tr>
<tr>
<td></td>
<td>Third</td>
<td>* , /</td>
</tr>
<tr>
<td></td>
<td>Fourth</td>
<td>+ , -</td>
</tr>
<tr>
<td>RELATIONAL</td>
<td>Fifth</td>
<td>.GT., .GE., .LT., .LE., .EQ., .NE.</td>
</tr>
<tr>
<td></td>
<td>or</td>
<td>&gt;, &gt;=, &lt;, &lt;=, =, #</td>
</tr>
<tr>
<td>LCGICAL</td>
<td>Sixth</td>
<td>.NOT.</td>
</tr>
<tr>
<td></td>
<td>Seventh</td>
<td>.AND.</td>
</tr>
<tr>
<td></td>
<td>Eighth</td>
<td>.OR.</td>
</tr>
<tr>
<td></td>
<td>Ninth</td>
<td>.EQV., .XOR.</td>
</tr>
</tbody>
</table>

Operations specifying integer division are evaluated from left to right. For example, the expression \(I/J^*K\) is evaluated as if it had been written as \((I/J)^*K\).

*When a series of exponentiation operations occurs in an expression, they are evaluated in order from right to left. For example, the expression \(A**2**B\) is evaluated in the following order:*

1. first \(2**B = R1\) (intermediate result)
2. second \(A**R1 = R2\) (final result).

4.3.3 Mixed Mode Expressions

Mixed mode expressions are evaluated on a subexpression by subexpression basis with the type of the results obtained converted and combined with other results or terms according to the conversion procedures described in Table 4-2.

Example

Assume the following:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>Double Precision</td>
</tr>
<tr>
<td>X</td>
<td>Real</td>
</tr>
<tr>
<td>I, J</td>
<td>Integer</td>
</tr>
</tbody>
</table>
The mixed mode expression $D \times X^* (1/J)$ is evaluated in the following manner:

**NOTE**

R1, R2, and R3 represent the interim and final results of the computations performed.

a. $(1/J) = R1$  R1 is integer  
b. $X^* R1 = R2$  R1 is converted to type real and is multiplied by X to produce R2  
c. $D + R2 = R3$  R2 is converted to type double precision and is added to D to produce R3

4.3.4 Use of Logical Operands in Mixed Mode Expressions

When logical operands are used in mixed mode expressions, the value of the logical operand is *not* converted in any way to accommodate it to the type of the other operands in the expression. For example, in $L^* R$, where $L$ is type logical and $R$ is type real, the expression is evaluated without converting $L$ to type real.
CHAPTER 5

COMPILATION CONTROL STATEMENTS

5.1 INTRODUCTION
Compilation control statements are used to identify FORTRAN-10 programs and to specify their termination. Statements of this type do not affect either the operations performed by the object program or the manner in which the object program is executed. The three compilation control statements described in this chapter are: PROGRAM statement, INCLUDE statement, and END statement.

5.2 PROGRAM STATEMENT
This statement allows the user to give the main program a name other than "MAIN." The general form of a PROGRAM statement is

```
PROGRAM name
```

where

```
name
```

is a user-formulated symbolic name that begins with an alphabetic character and contains a maximum of six characters. (Refer to section 3.3 for a description of symbolic names.)

The following rule governs the use of the PROGRAM statement:

The PROGRAM statement must be the first statement in a program unit. (Refer to section 2.4 for a discussion of the ordering of FORTRAN-10 statements.)

5.3 INCLUDE STATEMENT
This statement allows the user to include code segments or predefined declarations in a program unit without having them reside in the same physical file as the primary program unit. The general form of the INCLUDE statement is

```
INCLUDE dev:file.ext [proj,prog] /NOLIST
```

where

```
device
```

is a device name. When no device is specified, DSK: is assumed.

```
file.ext
```

is the filename and extension of the FORTRAN-10 statements that the user wishes to include. The name of the file is required; the extension is optional. If only the filename (file) is specified, then .FOR (for FORTRAN-10) is the assumed extension. If the filename and dot (file) are specified, then the null extension is assumed. No wild (*) information may be specified.
[proj,prog] is the project-programmer number. The user’s project-programmer number is assumed if none is specified. Subdirectory information cannot be specified.

/NOLIST is an optional switch that indicates that the included statements are not to be included in the compilation listing.

The following rules govern the use of the INCLUDE statement:

a. The INCLUDED file may contain any legal FORTRAN-10 statement: except another INCLUDE statement, or a statement that terminates the current program unit, such as the END, PROGRAM, FUNCTION, SUBROUTINE, or BLOCK DATA statements.

b. The proper placement of the INCLUDE statement within a program unit depends upon the types of statements to be INCLUDED. (Refer to section 2.4 for information on the ordering of FORTRAN-10 statements.)

Note that an asterisk (*) is appended to the line numbers of the INCLUDED statements on the compilation listing.

5.4 END STATEMENT

This statement is used to signal FORTRAN-10 that the physical end of a source program or subprogram has been reached. END is a nonexecutable statement. The general form of an END statement is

```
END
```

The following rules govern the use of the END statement:

a. This statement must be the last physical statement of a source program or subprogram.

b. When used in a main program, the END statement implies a STOP statement operation; in a subprogram, END implies a RETURN statement operation.

c. An END statement may be labeled.

Version 4 FORTRAN 5-2 October 1974
CHAPTER 6

SPECIFICATION STATEMENT

6.1 INTRODUCTION

Specification statements are used to specify the type characteristics, storage allocations, and data arrangement.

There are seven types of specification statements:

- DIMENSION
- Statements which specify, explicitly, type.
- IMPLICIT
- COMMON
- EQUIVALENCE
- EXTERNAL
- PARAMETER

Specification statements are nonexecutable and are expected to conform to the ordering guidelines described in section 2.4.

6.2 DIMENSION STATEMENT

DIMENSION statements provide FORTRAN-10 with information needed to identify and allocate the space required for source program arrays. Any number of subscripted array names may be specified as array declarators in a DIMENSION statement. The general form of a DIMENSION statement is

```
DIMENSION S1, S2, ..., Sn
```

where $S_i$ is an array declarator. Array declarators are names of the following form:

```
name (min:max, min:max, ..., min:max)
```

where $name$ is the symbolic name of the array and each $min$max value represents the lower and upper bounds of an array dimension.

Each $min:max$ value for an array dimension may be either an integer constant or, if the array is a dummy argument to a subprogram, an integer variable. The value given the minimum specification for a dimension must not exceed the value given the maximum specification. Minimum values of 1 with their following colon delimiter may be omitted from a dimension subscript.
CHAPTER 6

DIMENSION Statements,
Specifying Adjustable Dimensions

Examples

DIMENSION EDGE (-1:1,4:8), NET(5,10,4), TABLE(567)
DIMENSION TABLE (IAB:J,K,M,10:20)

(where IAB, J, K, and M are of type integer).

Note that a slash may be used in place of a colon as the delimiter between the upper and lower bounds of an array dimension.

6.2.1 Adjustable Dimensions

When used within a subprogram, an array declarator may use type integer parameters as dimension subscript quantities. The following rules govern the use of adjustable dimensions in a subprogram:

a. For single entry subprograms, the array name and each subscript variable must be given by the calling program when the subprogram is called. The subscript variables may also be in COMMON.

b. For multiple entry subprograms in which the array name is a parameter, any subscript variables may be passed. If all subscript variables are not passed or in COMMON, the value of the subscript as passed for a previous entry will be used.

c. The type of the array dimension variables cannot be altered within the program.

d. If the value of an array dimension variable is altered within the program, the dimensionality of the array will not be affected.

e. The original size of the array cannot exceed the array dimensions assigned within a subprogram (i.e., the size of an array is not dynamically expandable).

Examples

SUBROUTINE SBR (ARRAY,M1,M2,M3,M4)
DIMENSION ARRAY (M1:M2,M3:M4)
DO 27 L=M3,M4
DO 27 K=M1,M2
ARRAY (K,L)=VALUE
27 CONTINUE
END

SUBROUTINE SB1 (ARR1,M,N)
DIMENSION ARR1(M,N)
ARR1(M,N)=VALUE
ENTRY SB2(ARR1,M)
ENTRY SB3(ARR1,N)
ENTRY SB4(ARR1)

In the foregoing example, the first call made to the subroutine must be made to SB1. Assuming that the call is made at SB1 with the values M=11 and N=13, any succeeding call to SB2 should give M a new value. If a succeeding call is made to SB4, the last values passed through entries SUB1, SUB2, or SUB3 will be used for M and N.
Note that for the calling program of the form:

    CALL SB1(A,11,13)
    M=15
    CALL SB3(A,13)

the value of M used in the dimensionality of the array for the execution of SB3 will be 11 (i.e., the last value passed).

6.3 TYPE SPECIFICATION STATEMENTS

Type specification statements declare explicitly the data type of a variable, array, or function symbolic names. An array name may be given in a type statement either alone (un subscripted) to declare the type of all its elements or in a subscripted form to specify both its type and dimensions.

Type specification statements are written in the following form:

    type list

where type may be any one of the following declarators:

a. INTEGER
b. REAL
c. DOUBLE PRECISION
d. COMPLEX
e. LOGICAL

NOTE

In order to be compatible with the type statements used by other manufacturers, the data type size modifier, *n, is accepted by FORTRAN-10. This size modifier may be appended to the declarators, causing some to elicit messages warning users of the form of the variable specified by FORTRAN-10:

<table>
<thead>
<tr>
<th>Declarator</th>
<th>Form of Variable Specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER*2</td>
<td>Full word integer with warning message</td>
</tr>
<tr>
<td>INTEGER*4</td>
<td>Full word integer</td>
</tr>
<tr>
<td>LOGICAL*1</td>
<td>Full word logical with warning message</td>
</tr>
<tr>
<td>LOGICAL*4</td>
<td>Full word logical</td>
</tr>
<tr>
<td>REAL*4</td>
<td>Full word real</td>
</tr>
<tr>
<td>REAL*8</td>
<td>Double precision real</td>
</tr>
<tr>
<td>COMPLEX*8</td>
<td>Complex</td>
</tr>
<tr>
<td>COMPLEX*16</td>
<td>Complex with warning message</td>
</tr>
</tbody>
</table>
NOTE (Cont)
In addition, the data type size modifier may be appended to individual variables, arrays, or function names. Its effect is to override, for the particular element, the size modifier (explicit or implicit) of the primary type. For example,

REAL*A4, B*8, C*8(10), D

A and D are single precision (full word real), and B and C are double precision real.

The list consists of any number of variable, array, or function names which are to be declared the specified type. The names listed must be separated by commas, and can appear in only one type statement within a program unit.

Examples

INTEGER A, B, TABLE, FUNC
REAL R, M, ARRAY (5:10,10:20,5)

NOTE
Variables, arrays, and functions of a source program, which are not typed either implicitly or explicitly by a specification statement, are typed by FORTRAN-10 according to the following conventions:

a. Variable names, array names, and function names which begin with the letters I, J, K, L, M, or N are type integer.

b. Variable names, array names, and function names which begin with any letter other than I, J, K, L, M, or N are type real.

If a name that is the same as a FORTRAN-10 defined function name appears in a conflicting type statement, it is assumed that the name refers to a user-defined routine of the given type. Placing a generic FORTRAN-10 defined function name in an explicit type statement causes it to lose its generic properties.

6.4 IMPLICIT STATEMENTS

IMPLICIT statements declare the data type of variables and functions according to the first letter of each variable name. IMPLICIT statements are written in the following form:

IMPLICIT type(A1,A2,..,An),type(B1,B2,..,Bn),..,type....

As shown in the foregoing form statement, an IMPLICIT statement is comprised of one or more type declarators separated by commas. Each type declarator has the form

type(A1,A2,..,An)

where type represents one of the declarators listed in section 6.3, and the parenthesized list represents a list of different letters.

Each letter in a type declarator list specifies that each source program variable (not declared in an explicit type specification statement) which starts with that letter is assigned the data type named in the declarator. For example, the IMPLICIT type declarator REAL (R,M,N,O) declares that all names which begin with the letters R, M, N, or O are type REAL names, unless declared otherwise in an explicit type statement.
NOTE

Type declarations given in an explicit type specification override those also given in an IMPLICIT statement. IMPLICIT declarations do not affect the FORTRAN-10 supplied functions.

A range of letters within the alphabet may be specified by writing the first and last letters of the desired range separated by a dash (e.g., A-E for A,B,C,D,E). For example, the statement IMPLICIT INTEGER (I,J,...P) declares that all variables which begin with the letters I,J,M,N,O, and P are INTEGER variables.

More than one IMPLICIT statement may be used, but they must appear before any other declaration statement in the program unit. Refer to section 2.4 for a discussion on ordering FORTRAN-10 statements.

6.5 COMMON STATEMENT

The COMMON statement enables the user to establish storage which may be shared by two or more programs and/or subprograms and to name the variables and arrays which are to 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. COMMON statements are written in the following form:

COMMON/A1/V1,V2,..,Vn.../An/V1,V2,..,Vn

where the enclosed letters /A1/, /A2/, and /An/ represent optional name constructs (referred to as common block names when used).

The list (i.e., V1,V2,..,Vn) appearing after each name construct lists the names of the variables and arrays that are to occupy the common area identified by the construct. The items specified for a common area are ordered within the storage area as they are listed in the COMMON statement.

COMMON storage area may be either labeled or blank (unlabeled). If the common area is to be labeled, a symbolic name must be given within slashes immediately before the list of items that are to occupy the names 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). Common block names bear no relation to internal variables or arrays which have the same name.

If a common area is to be declared but is to be unlabeled (i.e., blank) either nothing or two sequential slashes (//) is given immediately before the list of items that are to occupy blank common. For example, the statement

COMMON/AREA1/A,B,C//TAB(3,3,3)

establishes one labeled (AREA1) and one unlabeled (i.e., blank) 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.

Each labeled common area is treated as a separate, specific storage area. The contents of a common area (i.e., variables and array) may be assigned initial values by DATA statements in BLOCK DATA subprograms. Declarations of a given common area in different subprograms must contain the same number, size, and order of variable and array name as the referenced area.
Items to be placed in a blank common area may also be given in COMMON statements throughout the source program.

During compilation of a source program, FORTRAN-10 will string together all items listed for each labeled common area and for blank common in the order in which they appear in the source program statements. For example, the series of source program statements

\[
\text{COMMON/ST1} /A,B,C/ST2/TAB(2,2)///C,D,E
\]

\[
\cdot
\]

\[
\text{COMMON/ST1/TST(3,4)/M,N}
\]

\[
\cdot
\]

\[
\text{COMMON/ST2/X,Y,Z/}/O,P,Q
\]

have the same effect as the single statement

\[
\text{COMMON/ST1/A,B,C,TST(3,4)/ST2/TAB(2,2),X,Y,Z/}/C,D,E,M,N,O,P,Q
\]

All items specified for blank common are placed into one area. Items within blank common are ordered as they are given throughout the source program. Common block names must be unique with respect to all subroutine, function, and entry point names.

The largest definition of a given common area must be loaded first.

6.5.1 Dimensioning Arrays in COMMON Statements

Subscripted array names may be given in COMMON statements as array dimension declarators. However, variables cannot be used as subscript quantities in a declarator appearing in a COMMON statement; variable dimensioning is not permitted in COMMON.

Each array name given in a COMMON statement must be dimensioned either by the COMMON statement or by another dimensioning statement within the program or subprogram which contains the COMMON statement.

Example

\[
\text{COMMON /A/B(100), C(10,10)}
\]
\[
\text{COMMON X(5,15), Y(5)}
\]

6.6 EQUIVALENCE STATEMENT

The EQUIVALENCE statement enables the user to control the allocation of shared storage within a program or subprogram. This statement causes specific storage locations to be shared by two or more variables of either the same or different types. The EQUIVALENCE statement is written in the following form:

\[
\text{EQUIVALENCE(V1,V2, \ldots, Vn),(W1,W2, \ldots, Wn),(X1,X2, \ldots)}
\]

where each parenthesized list contains the names of variables and array elements which are to share the same storage locations. For example, the statements

\[
\text{EQUIVALENCE (A,B,C)}
\]
\[
\text{EQUIVALENCE (LOC,SHARE(1))}
\]

specify that the variables named A, B, and C are to share the same storage location and the the variable LOC and array element SHARE(1) are to share the same location.
The relationship of equivalence is transitive; for example, the two following statements have the same effect:

\[
\text{EQUIVALENCE (A,B), (B,C)} \\
\text{EQUIVALENCE (A,B,C)}
\]

Array elements, when used in EQUIVALENCE statements, must have either as many subscript quantities as dimensions of the array or only one subscript quantity. In either of the foregoing cases, the subscripts must be integer constants. Note that the single case treats the array as a one-dimensional array of the given type.

The items given in an EQUIVALENCE list may appear in both the EQUIVALENCE statement and in a COMMON statement providing the following rules are observed:

a. No two quantities declared in a COMMON statement can be set equivalent to one another.

b. Quantities placed in a common area by means of an EQUIVALENCE statement are permitted to extend the end of the common area forwards. For example, the statements

\[
\text{COMMON/R/X,Y,Z} \\
\text{DIMENSION A(4)} \\
\text{EQUIVALENCE (A,Y)}
\]

cause the common block R to extend from Z to A(4) arranged as follows:

\[
\begin{align*}
\text{X} \\
\text{Y A(1) (shared location)} \\
\text{Z A(2) (shared location)} \\
\text{A(3)} \\
\text{A(4)}
\end{align*}
\]

c. EQUIVALENCE statements that cause the start of a common block to be extended backwards are not allowed. For example, the invalid sequence

\[
\text{COMMON/R/X,Y,Z} \\
\text{DIMENSION A(4)} \\
\text{EQUIVALENCE(X,A(3))}
\]

would require A(1) and A(2) to extend the starting location of block R in a backwards direction as illustrated by the following diagram:

\[
\begin{array}{c}
\uparrow \text{A(1)} \\
\uparrow \text{A(2)} \\
\text{X A(3)} \\
\text{Y A(4)} \\
\text{Z}
\end{array}
\]

6.7 EXTERNAL STATEMENT

Any subprogram name to be used as an argument to another subprogram must appear in an EXTERNAL statement in the calling subprogram. The EXTERNAL statement declares names to be subprogram names to distinguish them from other variable or array names. The EXTERNAL statement is written in the following form:

\[
\text{EXTERNAL name1,name2, \ldots,nameN}
\]
where each name listed is declared to be a subprogram name. If desired, these subprogram names may be FORTRAN-10 defined functions.

It is also possible to utilize FORTRAN-10 defined function names for user subprograms by prefixing the names by an asterisk (*) or an ampersand (&) within an EXTERNAL statement. For example,

```
EXTERNAL *SIN, &COS
```

declares SIN and COS to be user subprograms. (If a prefixed name is not a FORTRAN-10 defined function, then the prefix is ignored.)

Note that specifying a FORTRAN-10 defined function in an EXTERNAL statement without a prefix (i.e., EXTERNAL SIN) has no effect upon the usage of the function name outside of actual argument lists. If the name has generic properties, they are retained outside of the actual argument list. (The name has no generic properties within an argument list.)

The names declared in a program EXTERNAL statement are reserved throughout the compilation of the program and cannot be used in any other declarator statement, with the exception of a type statement.

6.8 PARAMETER STATEMENT

The PARAMETER statement allows users to define constants symbolically during compilation.

The general form of the PARAMETER Statement is as follows:

```
PARAMETER    P1=C1,P2=C2, . . .
```

where

- $P_i$ is a standard user-defined identifier (referred to in this section as a parameter name)
- $C_i$ is any type of constant (including literals) except a label or complex constant. (Refer to Chapter 3 for a description of FORTRAN-10 constants.)

During compilation the parameter names are replaced by their associated constants provided the following rules are observed:

a. Parameter names appear only within the statement field of an initial or continuation line type, i.e., not within a comment line or literal text.

b. Parameter names are placed only where FORTRAN-10 constants are acceptable.

c. Parameter name references appear after the PARAMETER statement definition.

d. Parameter names are unique with respect to all other names in the program unit.

e. Parameter names are not redefined in subsequent PARAMETER statements.

f. Parameter names are not used as part of some larger syntactical construct (such as a Hollerith constant count, or a data type size modifier).
CHAPTER 7

FORTRAN-10 extensions to the 1966 ANSI standard set are printed in **boldface italic type.**

---

CHAPTER 7

DATA STATEMENT

7.1 INTRODUCTION

DATA statements are used to supply the initial values of variables, arrays, array elements, and labeled common.\(^1\) DATA statements are written in the following form:

```
DATA List 1/Data 1/,List 2/Data 2/,...,List n/Data n/
```

where the `List/Data/` pair identifies a set of items to be initialized and the `/Data/` portion contains the list of values to be assigned the items in the List. For example, the statement

```
DATA IA/5/,IB/10/,IC/15/
```

initializes variable `IA` as the value 5, variable `IB` as the value 10 and the variable `IC` as the value 15. The number of storage locations specified in the list of variables must be less than or equal to the number of storage locations specified in its associated list of values. If the list of variables is larger (specifies more storage locations) than its associated value list, a warning message is output. When the value list specifies more storage locations than the variable list the excess values are ignored.

The `List/Data/` set may contain the names of one or more variables, arrays, array elements, or labeled common variables. An entire array (unsubscripted array name) or a portion of an array may be specified in a DATA statement `List` as an implied DO loop construct (see Paragraph 10.3.4.1 for a description of implied DO loops). For example, the statement

```
DATA (NARY (I), I=1,5)/1,2,3,4,5/
```

initializes the first five elements of array `NARY` as `NARY(1)=1, NARY(2)=2, NARY(3)=3, NARY(4)=4, NARY(5)=5.`

When an implied DO loop is used in a DATA statement, the loop index variable must be of type INTEGER and the loop Initial, Terminal, and Increment parameters must also be of type INTEGER. In a DATA statement, references to an array element must be integer expressions in which all terms are either integer constants or indices of embracing implied DO loops. Integer expressions of the foregoing types cannot include the exponentiation operator.

---

\(^1\) Refer to Paragraph 6.5 for a description of labeled common.
The `/Data/` portion of each List/Data/ set may contain one or more numeric, logical, literal, or octal constants and/or alphanumeric strings.

Octal constants must be identified as octal by preceding them with a double quote (" ") symbol (e.g., "777").

Literal data may be specified as either a Hollerith specification (e.g., 5HABCDE), or a string enclosed in single quotes (e.g., ’ABCDE’). Each ASCII datum is stored left-justified and is padded with blanks up to the right boundary of the variable being initialized.

When the same value is to be assigned to more than one item in List, a repeat specification may be used. The repeat specification is written 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. The statement

```
DATA M,N,L/3*20/
```

assigns the value 20 to the variables M, N, L.

In instances where the type of the data specified is not the same as that of the variable to which it is assigned, FORTRAN-10 converts the datum to the type of the variable. The type conversion is performed using the rules given for type conversion in arithmetic assignments (refer to Chapter 8, Table 8-1). Octal, logical, and literal constants are not converted.

### Sample Statement

<table>
<thead>
<tr>
<th>Use</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>The first 30 elements of array <code>TAB</code> are initialized as 5.0.</td>
<td>DATA PRINT,1,0/&quot;TEST&quot;,30,&quot;777&quot;,TAB(J), J=1,30/30*5</td>
</tr>
<tr>
<td>No conversion required.</td>
<td>DATA ((A(I,J),I=1,5),J=1,6)/30*1.0/</td>
</tr>
<tr>
<td>No conversion required.</td>
<td>DATA ((A(I,J),I=5,10),J=6,15)/60*2.0/</td>
</tr>
</tbody>
</table>

When a literal string is specified which is longer than one variable can hold, the string will be stored left-justified across as many variables as are needed to hold it. If necessary, the last variable used will be padded with blanks up to its right boundary.

Example

Assuming that `X`, `Y`, and `Z` are single precision, the statement

```
DATA X,Y,Z/’ABCDEFGHJKL’/
```

will cause

```
X to be initialized to ’ABCDEF’
Y to be initialized to ’FGHJK’
Z to be initialized to ’KL’
```

When a literal string is to be stored in double precision and/or complex variables and the specified string is only one word long, the second word of the variable is padded with blanks.
Example

Assuming that the variable C is complex, the statement

```plaintext
DATA C/'ABCDE', 'FGHIJ'/
```

will cause the first word of C to be initialized to 'ABCDE' and its second word to be initialized to 'FGHIJ'. The string 'FGHIJ' is ignored.
CHAPTER 8

ASSIGNMENT STATEMENTS

8.1 INTRODUCTION
Assignment statements are used to assign a specific value to one or more program variables. There are three kinds of assignment statements:

a. Arithmetic assignment statements
b. Logical assignment statements
c. Statement Label assignment (ASSIGN) statements.

8.2 ARITHMETIC ASSIGNMENT STATEMENT
Statements of this type are used to assign specific numeric values to variables and/or array elements. Arithmetic assignment statements are written in the form

\[ v = e \]

where \( v \) is the name of the variable or array element which is to receive the specified value and \( e \) is a simple or compound arithmetic expression.

In assignment statements the equals symbol (=) does not imply equality as it would in algebraic expressions; it implies replacement. For example, the expression \( v = e \) is correctly interpreted as “the current contents of the location identified as \( v \) are to be replaced by the final value of expression \( e \); the current contents of \( v \) are lost.”

When the type of the specified variable or array element name differs from that of its assigned value, FORTRAN-10 converts the value of the type of its assigned variable or array element. The type conversion operations performed by FORTRAN-10 for each possible combination of variable and value types are described in Table 8-1.
## Table 8-1
Rules for Conversion in Mixed Mode Assignments

<table>
<thead>
<tr>
<th>Expression Type (x)</th>
<th>Variable Type (y)</th>
<th>Real</th>
<th>Integer</th>
<th>Complex</th>
<th>Double Precision</th>
<th>Logical</th>
</tr>
</thead>
<tbody>
<tr>
<td>REAL</td>
<td></td>
<td>D</td>
<td>C</td>
<td>R,I</td>
<td>H,L</td>
<td>D</td>
</tr>
<tr>
<td>INTEGER</td>
<td></td>
<td>C</td>
<td>D</td>
<td>R,C,I</td>
<td>H,C,L</td>
<td>D</td>
</tr>
<tr>
<td>COMPLEX</td>
<td></td>
<td>R</td>
<td>C,R</td>
<td>D</td>
<td></td>
<td>R</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td></td>
<td>H</td>
<td>C,H,L</td>
<td>D</td>
<td></td>
<td>H</td>
</tr>
<tr>
<td>LOGICAL</td>
<td></td>
<td>D</td>
<td>D</td>
<td>R,I</td>
<td>H,L</td>
<td>D</td>
</tr>
<tr>
<td>OCTAL</td>
<td></td>
<td>D</td>
<td>D</td>
<td>R,I</td>
<td>H,C,L</td>
<td>D,H</td>
</tr>
<tr>
<td>LITERAL</td>
<td></td>
<td>D,H***</td>
<td>C,H***</td>
<td>D**</td>
<td>D***</td>
<td>D***</td>
</tr>
<tr>
<td>DOUBLE OCTAL*</td>
<td></td>
<td>H</td>
<td>H</td>
<td>D****</td>
<td>D</td>
<td>H</td>
</tr>
</tbody>
</table>

### Legend

D = Direct replacement  
C = Conversion between integer and floating-point with truncation  
R = Real part only  
I = Set imaginary part to 0  
H = High order only  
L = Set low order part to 0

### Notes

* Octal numbers comprised of from 13 to 24 digits are termed double octal. Double octals require two storage locations. They are stored right-justified and are padded with zeroes to fill the locations.

** Use the first two words of the literal. If the literal is only one word long, the second word is padded with blanks.

*** Use the first word of the literal.

**** To convert double octal numbers to complex, the low order octal digits are assumed to be the imaginary part and the high order digits are assumed to be the real part of the complex value.
8.3 LOGICAL ASSIGNMENT STATEMENTS

This type of assignment statement is used to assign values to variables and array elements of type logical. The logical assignment statement is written in the form

\[ v = e \]

where \( v \) is one or more variables and/or array element names and \( e \) is a logical expression.

Examples

Assuming that the variables \( L, F, M, \) and \( G \) are of type logical, the following statements are valid:

Sample Statement

\[ \begin{align*}
\text{L} &= \text{TRUE}. \\
\text{F} &= \text{NOT.G} \\
\text{M} &= \text{A} > \text{T} \\
\text{L} &= \text{(I.GT.H).AND.(J <=K))}
\end{align*} \]

The contents of \( L \) are replaced by logical truth.

The contents of \( L \) are replaced by the logical complement of the contents of \( G \).

If \( A \) is greater than \( T \), the contents of \( M \) are replaced by logical truth; if \( A \) is less than or equal to \( T \), the contents of \( M \) are replaced by logical false.

The contents of \( L \) is replaced by either the true or false resultant of the expression.

8.4 ASSIGN (STATEMENT LABEL) ASSIGNMENT STATEMENT

The ASSIGN statement is used to assign a statement label constant (i.e., a 1- to 5-digit statement number) to a variable name. The ASSIGN statement is written in the following form

\[ \text{ASSIGN } n \text{ TO } l \]

where \( n \) represents the statement number and \( l \) is a variable name. For example, the statement

\[ \text{ASSIGN 2000 TO LABEL} \]

specifies that the variable \( \text{LABEL} \) represents the statement number 2000.

With the exception of complex and double precision, any type of variable may be used in an ASSIGN statement.

Once a variable has been assigned a statement number, FORTRAN-10 will consider it a label variable. If a label variable is used in an arithmetic statement, the result will be unpredictable.

The ASSIGN statement is used in conjunction with assigned GO TO control statements (Chapter 9); it sets up statement label variables which are then referenced in subsequent GO TO control statements. The following sequence illustrates the use of the ASSIGN statement:
555  TAX=(A+B+C)*.05
     
     ASSIGN 555 TO LABEL
     
     GO TO LABEL
CHAPTER 9

FORTRAN-10 extensions to the 1966 ANSI standard set are printed in *boldface italic type.*

CHAPTER 9

CONTROL STATEMENTS

9.1 INTRODUCTION

FORTRAN-10 object programs are normally executed statement-by-statement in the order in which they were presented to the compiler. The following source program control statements, however, enable the user to alter the normal sequence of statement execution:

- a. GO TO
- b. IF
- c. DO
- d. CONTINUE
- e. STOP
- f. PAUSE

9.2 GO TO CONTROL STATEMENTS

There are three kinds of GO TO statements:

- a. Unconditional
- b. Computed
- c. Assigned.

A GO TO control statement causes the statement which it identifies to be executed next, regardless of its position within the program. Each type of GO TO statement is described in the following paragraphs.
9.2.1 Unconditional GO TO Statements
GO TO statements of this type are written in the form

   GO TO n

where n is the label (i.e., statement number) of an executable statement (e.g., GO TO 555). When executed, an unconditional GO TO statement causes control of the program to be transferred to the statement which it specifies.

An unconditional GO TO statement may be positioned anywhere in the source program except as the terminating statement of a DO loop.

9.2.2 Computed GO TO Statements
GO TO statements of this type are written in the form

   GO TO (N1,N2, ..,Nk)E

where the parenthesized list is a list of statement numbers and E is an arithmetic expression. Any number of statement numbers may be included in the list of this type of GO TO statement; however, each number given must be used as a label within the program or subprogram containing the GO TO statement.

   NOTE
   A comma may optionally follow the parenthesized list.

The value of the expression E must be reducible to an integer value that is greater than 0 and less than or equal to the number of statement numbers given in the statement’s list. If E does not compute within the foregoing range, the next statement is executed.

When a computed GO TO statement is executed, the value of its expression (i.e., E) is computed first. The value of E specifies the position within the given list of statement numbers, of the number which identifies the statement to be executed next. For example, in the statement sequence

   GO TO (20, 10, 5)K
   CALL XRANGE(K)

the variable K acts as a switch causing a transfer to statement 20 if K=1, to statement 10 if K=2, or to statement 5 if K=3. The subprogram XRANGE is called if K is less than 1 or greater than 3.

9.2.3 Assigned GO TO Statements
GO TO statements of this type may be written in either of the following forms:

   GO TO K
   GO TO K, (L1,L2, ..,Ln)

where K is a variable name and the parenthesized list of the second form contains a list of statement labels (i.e., statement numbers). The statement numbers given must be within the program or subprogram containing the GO TO statement.
CHAPTER 9

Arithmetic IF Statements

Assigned GO TO statements of either of the foregoing forms must be logically preceded by an ASSIGN statement that assigns a statement label to the variable name represented by K. The value of the assigned label variable must be in the same program unit as the GO TO statement in which it is used. In statements written in the form

GO TO K, (L1, L2, . . . , Ln)

if K is not assigned one of the statement numbers given in the statement's list, then the next sequential statement is executed.

Examples

GO TO STAT1
GO TO STAT1, (177, 207, 777)

9.3 IF STATEMENTS

There are three kinds of IF statements: arithmetic, logical, and logical two-branch.

9.3.1 Arithmetic IF Statements

IF statements of this type are written in the form

IF (E) L1, L2, L3

where (E) is an expression enclosed within parenthesis and L1, L2, L3 are the labels (i.e., statement numbers) of three executable statements.

This type of IF statement causes control of the program to be transferred to one of the given statements, according to the computed value of the given expression. If the value of the expression is:

a. less than 0, control is transferred to the statement identified by L1;

b. equal to 0, control is transferred to the statement identified by L2;

c. greater than 0, control is transferred to the statement identified by L3.

All three statement numbers must be given in arithmetic IF statements; the expression given may not compute to a complex value.

Examples

Sample Statement

IF (ETA) 4, 7, 12

Transfer control to statement 4 if ETA is negative, to statement 7 if ETA is 0 and to statement 12 if ETA is greater than 0.

IF (KAPPA – L(10)) 20, 14, 14

Transfer control to statement 20 if KAPPA is less than the 10th element of array L and to statement 14 if KAPPA is greater than or equal to the 10th element of array L.

Version 1A FORTRAN

9-3

January 1974
9.3.2 Logical IF Statements

IF statements of this type are written in the form

\[ \text{IF (E) S} \]

where E is any expression enclosed in parentheses and S is a complete executable statement.

Logical IF statements cause control of the program to be transferred either to the next sequential executable statement or the statement given in the IF statement (i.e., S) according to the computed logical value of the given expression. If the value of the given logical expression is true (negative), control is given to the executable statement within the IF statement. If the value of the expression is false (positive or zero), control is transferred to the next sequential executable program statement.

The statement given in a logical IF statement may be any FORTRAN-10 executable statement except a DO statement or another logical IF statement.

Examples

Sample Statement

\[ \text{IF (T.OR.S) X = Y + 1} \]

An arithmetic replacement operation is performed if the result of IF is true.

\[ \text{IF (Z.GT.X(K)) CALL SWITCH (S,Y)} \]

A subprogram transfer is performed if the result of IF is true.

\[ \text{IF (K.EQ.IND) GO TO 15} \]

An unconditional transfer is performed if the result of IF is true.

9.3.3 Logical Two-Branch IF Statements

IF statements of this type are written in the form

\[ \text{IF (E) N1, N2} \]

where E is any expression enclosed in parentheses and N1 and N2 are statement labels defined within the program unit.

Logical two-branch IF statements cause control of the program to be transferred to either statement N1 or N2 depending on the computed value of the given expression. If the value of the given logical expression is true (negative), control is transferred to statement N1. If the value of the expression is false (positive or zero), control is transferred to statement N2.

Note that the statement immediately following the logical two-branch IF must be numbered so that control can later be transferred to the portion of code that was skipped.

Examples

Sample Statement

\[ \text{IF (LOGI) 10,20} \]

Transfer control to statement 10 if LOGI is negative; otherwise transfer control to statement 20.

\[ \text{IF (A.LT.B.AND.A.LT.C) 31, 32} \]

Transfer control to statement 31 if A is less than both B and C; transfer control to statement 32 if A is greater than or equal to either B or C.
9.4 DO STATEMENT

DO statements simplify the coding of iterative procedures; they are written in the following form:

where

a. *Terminal Statement Label N* is the statement number of the last statement of the DO statement range. The range of a DO statement is defined as the series of statements which follows the DO statement up to and including its specified terminal statement.

b. *Index Variable I* is an unsubscripted variable, the value of which is defined at the start of the DO statement operations. The index variable is available for use throughout each execution of the range of the DO statement but its value should not be altered within this range. It is also made available for use in the program when

1. control is transferred outside the range of the DO loop by a GO TO, IF, or RETURN statement located within the DO range,

2. a CALL is executed from within the DO statement range which uses the index variable as an argument, and

3. if an Input—Output statement with either or both the options END= or ERR= (Chapter 10) appear within the DO statement range.

c. *Initial Parameter M1* assigns the index variable, V, its initial value. This parameter may be any variable, array element, or expression.

d. *Terminal Parameter M2* provides the value which determines how many repetitions of the DO statement range are performed.

e. *Increment Parameter M3* specifies the value to be added to the initial parameter (M1) on completion of each cycle of the DO loop.

An indexing parameter may be any arithmetic expression which should result in either a positive or negative value. The values of the indexing parameters are calculated only once, at the start of each DO-loop operation. The number of times that a DO loop will be executed is specified by the formula:

\[(M2-M1)/M3+1\]
Since the count is computed at the start of a DO loop operation, changing the value of the loop index variable within the loop cannot affect the number of times that the loop is executed. At the start of a DO loop operation, the index value is set to the value of the initial parameter (M1) and a count variable (generated by the compiler) is set to the negative of the calculated count. At the end of each DO loop cycle the value of the increment parameter (M3) is added to the index variable and the count variable is incremented. If the number of specified iterations have not been performed, another cycle of the loop is initiated.

One execution of a DO loop range is always performed regardless of the initial values of the index variable and the indexing parameters.

Exit from a DO loop operation on completion of the number of iterations specified by the loop count is referred to as a normal exit. In a normal exit, control is passed to the first executable statement after the DO loop range terminal statement and the value of the DO statement index variable is considered undefined.

Exit from a DO loop may also be accomplished by a transfer of control by a statement within the DO loop range to a statement outside the range of the DO statement (Paragraph 9.4.3).

9.4.1 Nested DO Statements
One or more DO statements may be contained (i.e., nested) within the range of another DO statement. The following rules govern the nesting of DO statements.

a. The range of each nested DO statement must be entirely within the range of the containing DO statement.

Example

<table>
<thead>
<tr>
<th>Valid</th>
<th>Invalid</th>
</tr>
</thead>
<tbody>
<tr>
<td>DO 1</td>
<td>DO 1</td>
</tr>
<tr>
<td>DO 2</td>
<td>DO 2</td>
</tr>
<tr>
<td></td>
<td>The range of DO 2 is outside that of DO 1.</td>
</tr>
</tbody>
</table>

b. The ranges of nested DO statements cannot overlap.

Example

<table>
<thead>
<tr>
<th>Valid</th>
<th>Invalid</th>
</tr>
</thead>
<tbody>
<tr>
<td>DO 1</td>
<td>DO 1</td>
</tr>
<tr>
<td>DO 2</td>
<td>DO 2</td>
</tr>
<tr>
<td>DO 3</td>
<td>DO 3</td>
</tr>
<tr>
<td></td>
<td>The ranges of loop DO 2 and DO 3 overlap.</td>
</tr>
</tbody>
</table>

9-6
CHAPTER 9

DO Statement, Extended Range and Transfer Operations

c. More than one DO loop within a nest of DO loops may end on the same statement. When this occurs, the terminal statement is considered to belong to the *innermost* DO statement that ends on that statement. The statement label 4 of the shared terminal statement cannot be used in any GO TO or arithmetic IF statement that occurs anywhere but within the range of the DO statement to which it belongs.

Example

```
DO 4
  DO 4
  DO 4
    DO 4
```

All the DO statements share the same terminal statement, however, it belongs to DO 4.

9.4.2 Extend Range

The extended range of a DO statement is defined as the set of statements that are executed between the transfers out of the *innermost* DO statement of a set of nested DO’s and the transfer back into the range of this innermost DO statement. The extended range of a nested DO statement is illustrated as follows:

```
DO 1
  DO 2
    DO 3
      .
      .
      .
      (out)
      .
      .
      (in)
```

Extended Range

9-7
The following rules govern the use of a DO statement extended range:

a. 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.

b. 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.

c. The extended range of a DO statement must not contain another DO statement.

d. The extended range of a DO statement cannot change the index variable or indexing parameters of the DO statement.

e. The use of and return from a subprogram from within an extended range is permitted.

9.4.3 Permitted Transfer Operations

The transfer of program control from within a DO statement range or the ranges of nested DO statements is governed by the following rules:

a. A transfer out of the range of any DO loop is permitted at any time. When such a transfer is executed the value of the controlling DO statement’s index variable is defined as the current value.

b. A transfer into the range of a DO statement is permitted if it is made from the extended range of the DO statement.

c. The use of and return from a subprogram from within the range of any DO loop, nested DO loop, or extended range is permitted.

The following examples illustrate the transfer operations permitted from within the ranges of nested DO statements.

Valid Transfers

Invalid Transfers
9.5 CONTINUE STATEMENT

CONTINUE statements may be placed 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 GO TO, PAUSE, STOP, RETURN, arithmetic IF, another DO statement, or a logical IF statement containing any of the foregoing statements. This statement is written as

```
  12 CONTINUE
```

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=NVENTRY (ITEM)
  CALL UPDATE (STOCK,TALLY)
  IF (ITEM.EQ.LAST) GO TO 77
  45 CONTINUE

  77 PRINT 20, HEADNG,PAGE NO
```

9.6 STOP STATEMENT

When executed, the STOP statement causes the execution of the object program to be terminated and control returned to the DECsystem-10 Monitor. A descriptive message may, optionally, be included in the STOP statement to be output to the user's I/O terminal immediately before program execution is terminated. This statement may be written as

```
  STOP
  STOP 'N'
```

or

```
  STOP n
```

where 'N' is a string of ASCII characters enclosed by single quotes and n is an octal string up to 12 digits. The string N or the value n is printed at the user's I/O terminal when the STOP statement is executed; it may be of any length, continuation lines may be used for large messages.

Examples

```
  STOP 'Termination of the Program'
```

or

```
  STOP 7777
```
9.7 PAUSE STATEMENT
When executed, a PAUSE statement causes a suspension of the execution of the object program and gives the user the option to:

a. Continue execution of the program
b. Exit
c. initiate a TRACE operation (Paragraph 9.7.1).

The permitted forms of the PAUSE statement are:

a. PAUSE
b. PAUSE 'literal string'
c. PAUSE n, where n is an octal string up to 12 digits.

The execution of a PAUSE statement of any of the foregoing forms causes the standard instruction:

    TYPE G TO CONTINUE, X TO EXIT, T TO TRACE

to be printed at the user's terminal. If the form of the PAUSE statement contains either a literal string or an integer constant, the string or constant is printed on a line preceding the standard message. For example, the statement

    PAUSE 'TEST POINT A'

causes the following to be printed at the user's terminal:

    TEST POINT A
    TYPE G TO CONTINUE, X TO EXIT, T TO TRACE

The statement

    PAUSE 1

causes the following to be printed at the user’s terminal:

    PAUSE 000001
    TYPE G TO CONTINUE, X TO EXIT, T TO TRACE

9.7.1 T (TRACE) Option
The entry of the character T in response to the message output by the execution of a PAUSE statement starts a TRACE routine. This routine causes the printing, at the user's terminal, of a complete history of all subroutine calls made during the execution of the program, up to the execution of the PAUSE statement. The history printed by the TRACE routine consists of:

a. The names of all subroutines called, arranged in the reverse order of their call;
b. The absolute location (written within parentheses) of the called subroutine;
c. The name of the calling subroutine plus an offset factor and the absolute location (written within parentheses) of the statement within the routine which initiated the call;
d. The number of arguments involved (written within angle brackets);

e. An alphabetic code (written within square brackets) that specifies the type of each argument involved. The alphabetic codes used and the meaning of each are:

<table>
<thead>
<tr>
<th>Code Character</th>
<th>Type Specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>Undefined type; the use of the argument will determine its type.</td>
</tr>
<tr>
<td>L</td>
<td>Logical</td>
</tr>
<tr>
<td>I</td>
<td>INTEGER</td>
</tr>
<tr>
<td>F</td>
<td>Single precision REAL</td>
</tr>
<tr>
<td>O</td>
<td>Octal</td>
</tr>
<tr>
<td>S</td>
<td>Statement Number</td>
</tr>
<tr>
<td>D</td>
<td>Double precision REAL</td>
</tr>
<tr>
<td>C</td>
<td>COMPLEX</td>
</tr>
<tr>
<td>K</td>
<td>A literal or constant</td>
</tr>
</tbody>
</table>

Example

The following printout illustrates the execution of the PAUSE statement “PAUSE ’TEST POINT A’”, the entry of a T character to initiate the TRACE routine, the resulting trace printout, and the entry of the character G to continue the execution of the program.

```
TEST POINT A
TYPE G TO CONTINUE, X TO EXIT, T TO TRACE.
*T
NAME (LOC) <<--- CALLER (LOC) <#ARGS> [ARG TYPES]
TRACE. (411653) <<--- MAIN.+612(1032) <#1> [U]
TYPE G TO CONTINUE, X TO EXIT, T TO TRACE.
*G
```

In addition to its use with the PAUSE statement, the TRACE routine may be called directly, using the form

```
CALL TRACE
```

or as a function, using the form

```
X = TRACE (x)
```

Execution of the foregoing statements starts the TRACE routine which causes the printing of the history of all subprogram calls made during the execution of the program, up to the execution of the CALL statement, or up to the execution of the function, respectively. The history printed by the TRACE routine under these circumstances is exactly the same as described in the preceding paragraph.
CHAPTER 10

I/O STATEMENTS

10.1 DATA TRANSFER OPERATIONS

FORTRAN-10 I/O statements permit data to be transferred between processor storage (core) and peripheral devices and/or between storage locations. Data in the form of logical records may be transferred using an a) sequential, b) random access, or c) append transfer mode. The areas in core from which data is to be taken during output (write) operations and into which data is stored during input (read) operations are specified by

a. a list in the I/O statement which initiated the transfer
b. a list defined by a NAMELIST statement, or
c. between a specified FORMAT statement and the external medium.

The type and arrangement of transferred data may be specified by format specifications located in either a FORMAT statement or an array (formatted I/O) or by the contents of an I/O list (i.e., list-directed I/O).

The transfer modes, I/O lists, type conversion and arrangement of data, and the statements required to initiate I/O transfer operations are described in the following paragraphs.

10.2 TRANSFER MODES

The characteristics and requirements of the a) sequential, b) random access, and c) append data modes are described in the following paragraphs.

10.2.1 Sequential Mode

Records are transferred during a sequential mode of operation in the same order as they appear in the external data file. Each I/O statement executed in a sequential mode transfers the record immediately following the last record transferred from the accessed source file.

10.2.2 Random Access Mode

This mode permits records to be accessed and transferred from a file in any desired order. Random access transfers, however, may be made only to (or from) a device that permits random-type data addressing operations (i.e., disk) and to files that have previously been set up for random access transfer operation. Files for random access must contain a specified number of identically sized records that may be accessed, individually, by a record number.

10-1

January 1974
The FORTRAN-10 OPEN statement or a subroutine call to DEFINE FILE may be used to set up random access files.

The OPEN statement is used to establish a random access mode to permit the execution of random access data transfer operations. The OPEN statement should logically precede the first I/O statement for the specified logical unit in the user source program.

10.2.3 Append Mode

This mode is a special version of the sequential transfer mode: it may be used only for sequential output (write) operations. The append mode permits the user to write a record immediately after the last logical record of the accessed file. During an append transfer, the records already in the accessed file remain unchanged, the only function performed is the appending of the transferred records to the end of the file.

An OPEN statement (Chapter 12) must be used to establish an append mode before append I/O operations can be executed.

10.3 I/O STATEMENTS, BASIC FORMATS AND COMPONENTS

The majority of the I/O statements described in this chapter are written in one of the following basic forms, or in some modification of these forms:

<table>
<thead>
<tr>
<th>Basic Statement Forms</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyword(u,f)list</td>
<td>Formatted I/O Transfer</td>
</tr>
<tr>
<td>Keyword(u#R,f)list</td>
<td>Random Access Formatted I/O Transfer</td>
</tr>
<tr>
<td>Keyword(u,*j)list</td>
<td>List-Directed I/O Transfer</td>
</tr>
<tr>
<td>Keyword(u,N)</td>
<td>NAMELIST-Controlled I/O Transfer</td>
</tr>
<tr>
<td>Keyword(u)list</td>
<td>Binary I/O Transfer</td>
</tr>
<tr>
<td>Keyword(u#R)list</td>
<td>Random Access Binary I/O Transfer</td>
</tr>
</tbody>
</table>

where

Keyword = the statement name (i.e., READ or WRITE)

u = FORTRAN-10 logical unit number

f = FORMAT statement number or the name of an array that contains the desired format specifications

list = I/O list

#R = the delimiter # followed by the number of a record in an established random-access file

* = symbol specifying a list-directed I/O transfer.

N = the name of an I/O list defined by a NAMELIST statement.

Details of the foregoing statement components are given in the following paragraphs.
10.3.1 I/O Statement Keywords

The keywords (i.e., names) of the FORTRAN-10 I/O statements described in this chapter are:

a. READ

b. REREAD

c. WRITE

d. ACCEPT

e. PRINT

f. PUNCH

g. TYPE

h. FIND

i. ENCODE

j. DECODE

10.3.2 FORTRAN-10 Logical Unit Numbers

The physical devices used for most FORTRAN-10 I/O operations are identified by decimal numbers. During compilation, the compiler assigns default logical unit numbers for the REREAD, READ, ACCEPT, PRINT, PUNCH, and TYPE statements. Default unit numbers are negatively signed decimal numbers that are inaccessible to the user.

The logical device assignments may be made by the user at run time (FORTRAN-10 User's Guide, DEC-10-LFUGA-A-B) or the standard assignments contained by the FORTRAN-10 Object Time System (FOROTS) may be used. The standard logical device assignments are listed in Table 10-1. It is recommended that the user specify the device explicitly in the OPEN statement.

10.3.3 FORMAT Statement References

A FORMAT statement contains a set of format specifications which define the structure of a record and the form of the data fields which comprise the record. Format specifications may also be stored in an array rather than in a FORMAT statement. (Refer to Chapter 13 for a complete description of the FORMAT statement.)

The execution of an I/O statement that includes either a FORMAT statement number or the name of an array which contains format specifications causes the structure and data of the transferred record to assume the form specified in the referenced statement or array. Records transferred under the control of a format specification are referred to as "formatted" records. Conversely, records transferred by I/O statements that do not reference a format specification are referred to as "unformatted" records. During unformatted transfers, data is transferred on a one-to-one correspondence between internal (processor) and external (device) locations, with no conversion or formatting operations.

Unformatted files are binary files divided into records by FORTRAN-10 embedded control words; the control words are invisible to the user. Files of this type cannot be prepared by the user without utilizing FOROTS. Unformatted files are intended to be used only within the FORTRAN-10 environment.
Table 10-1
FORTRAN-10 Logical Device Assignments

<table>
<thead>
<tr>
<th>Device/Function</th>
<th>Default Filename</th>
<th>FORTRAN Logical Unit Number</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Devices*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 DSK 00</td>
<td>FORxx.DAT 01</td>
<td></td>
<td>ILLEGAL</td>
</tr>
<tr>
<td>CDR</td>
<td>02</td>
<td></td>
<td>DISK</td>
</tr>
<tr>
<td>LPT</td>
<td>03</td>
<td></td>
<td>Card Reader</td>
</tr>
<tr>
<td>CTY</td>
<td>04</td>
<td></td>
<td>Line Printer</td>
</tr>
<tr>
<td>TTY</td>
<td>05</td>
<td></td>
<td>Console Teletype</td>
</tr>
<tr>
<td>PTR</td>
<td>06</td>
<td></td>
<td>User's Teletype</td>
</tr>
<tr>
<td>PTP</td>
<td>07</td>
<td></td>
<td>Paper Tape Reader</td>
</tr>
<tr>
<td>DIS</td>
<td>08</td>
<td></td>
<td>Display</td>
</tr>
<tr>
<td>DTA1</td>
<td>09</td>
<td></td>
<td>DECtape</td>
</tr>
<tr>
<td>DTA2</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DTA3</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DTA4</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DTA5</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DTA6</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DTA7</td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MTA0</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MTA1</td>
<td>17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MTA2</td>
<td>18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FORTR</td>
<td>19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSK</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSK</td>
<td>21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSK</td>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSK</td>
<td>23</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DSK</td>
<td>24</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEV1</td>
<td>25</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEV2</td>
<td>26</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEV3</td>
<td>27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEV4</td>
<td>28</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEV5</td>
<td>29</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEV63</td>
<td>FOR63.DAT 63</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Default Devices (inaccessible to the user)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REREAD</td>
<td>Current file</td>
<td>-6</td>
<td>REREAD statement</td>
</tr>
<tr>
<td>CDR</td>
<td>FORCDR.DAT</td>
<td>-5</td>
<td>READ statement</td>
</tr>
<tr>
<td>TTY</td>
<td>FORTTY.DAT</td>
<td>-4</td>
<td>ACCEPT statement</td>
</tr>
<tr>
<td>LPT</td>
<td>FORLPT.DAT</td>
<td>-3</td>
<td>PRINT statement</td>
</tr>
<tr>
<td>PTP</td>
<td>FORFTP.DAT</td>
<td>-2</td>
<td>PUNCH statement</td>
</tr>
<tr>
<td>TTY</td>
<td>FORTTY.DAT</td>
<td>-1</td>
<td>TYPE statement</td>
</tr>
</tbody>
</table>

*The total number of standard devices permitted is on installation parameter.
CHAPTER 10

10.3.4 I/O List

An I/O list specifies the names of variables, arrays, and array elements to which input data is to be assigned or from which data is to be output. Implied DO constructs (Paragraph 10.3.4.1), which specify specific sets of array elements, may also be included in I/O lists. The number of items in a statement's list determines the amount of data to be transferred during each execution of the statement.

10.3.4.1 Implied DO Constructs – When an array name is given in an I/O list all elements of the array are transferred in the order described in Chapter 3 (Paragraph 3.5.3). If only a specific set of array elements is involved, they may be specified in the I/O list either individually or in the form of an implied DO construct.

Implied DO's are written within parentheses in a format similar to that of DO statements. They may contain one or more variable, array, and/or array element names, delimited by commas and followed by indexing parameters that are defined as for DO statements.

The general form of an implied DO is

\[(\text{name}(\text{SL}), \text{I}=\text{M1}, \text{M2}, \text{M3})\]

where

\[\begin{align*}
\text{name} & = \text{an array name} \\
\text{SL} & = \text{the subscript list of an array name or an array element identifier} \\
\text{I} & = \text{the index control variable that represents a subscript appearing in a preceding subscript list} \\
\text{M1,M2,M3} & = \text{the indexing parameters that specify, respectively, the initial, terminal, and increment values that control the range of I. If M3 is omitted (with its preceding comma), a value of 1 is assumed.}
\end{align*}\]

Examples

\[(A(S), S=1,5)\] Specifies the first five elements of the one-dimension array A (i.e., A(1), A(2), A(3), A(4), A(5)).

\[(A(2,5), S=1,10,2)\] Specifies the elements A(2,1), A(2,3), A(2,5), A(2,7), A(2,9) of array A.

As stated previously, implied DO constructs may also contain one or more variable names.

Example

I, J, B, and C must be integer variables.

\[((A(B,C), B=1,10), C=1,10), I, J\] Specifies a 10 X 10 set of elements of array A, the location identified by I and the location identified by J.

Implied DO constructs may also be nested. Nested implied DO's may share one or more sets of indexing parameters.

Example

\[((A(J,K), J=1,5), D(K), K=1,10)\] Specifies a 5 X 10 set of elements of array A and the first 10 elements of array D.
CHAPTER 10

When an array or set of array elements are specified as either a storage or transmitting area for I/O purposes, the array elements involved are accessed in ascending order with the value of the first subscript quantity varying most rapidly and the value of the last given subscript increasing to its maximum value least rapidly. For example, the elements of an array dimensional as TAB(2,3) are accessed in the order:

TAB(1,1)
TAB(2,1)
TAB(1,2)
TAB(2,2)
TAB(1,3)
TAB(2,3)

10.3.5 The Specification of Records for Random Access

Records to be transferred in a random access mode must be identified in an I/O statement by an integer expression or variable preceded by a 'delimiter (e.g., '101).

NOTE
A number sign ( # ) may be used in place of the 'delimiter (e.g., both #101 and '101 are accepted by FORTRAN-10).

10.3.6 List-Directed I/O

The use of an asterisk in an I/O statement in place of a FORMAT statement number causes the specified transfer operation to be "list-directed." In a list-directed transfer, the data to be transferred and the type of each transferred datum are specified by the contents of an I/O list included in the I/O command used. The transfer of data in this mode is performed without regard for column, card, or line boundaries. The list-directed mode is specified by the substitution of an asterisk (*) for the FORMAT statement reference (i.e., f) of an I/O statement. The general form of a list-directed I/O statement is

keyword (u,*)list

Example

READ (5,*)[I,J,AB,M,L]

List-directed transfers may be used to input data from any acceptable input device including an input keyboard terminal.

NOTE
Device positioning commands, such as BACKSPACE, SKIP RECORD, etc., should not be used in conjunction with list-directed I/O operations. If such a combination is used, the results will be unpredictable.

Data for list-directed transfers should consist of alternate constants and delimiters. The constants used should have the following characteristics:

a. Input constants must be of a type acceptable to FORTRAN-10. Octal constants, although acceptable, are not permitted in list-directed I/O operations.

b. Literal constants must be enclosed within single quotes (e.g., 'ABLE').

10-6 November 1975
c. Blanks serve as delimiters; therefore, they are not permitted in any but literal constants.

d. Decimal points may be omitted from real constants which do not have a fractional part. FORTRAN-10 assumes that the decimal point follows the right-most digit of a real constant.

Delimiters in data for list-directed input must comply with the following:

a. Delimiters may be either commas or blanks.

b. Delimiters may be either preceded by or followed by any number of blanks, carriage return/line feed characters, tabs, or line terminators; any such combination is considered by FORTRAN-10 as being only a single delimiter.

c. A null, the complete absence of a datum, is represented by two consecutive commas which have no intervening constant(s). Any number of blanks, tabs, carriage return/line feed characters, or end-of-input conditions may be placed between the commas of a null. Each time a null item is specified in the input data, its corresponding list element is skipped (i.e., unchanged). The following illustrates the effect of a null input:

```
INPUT Items                  101, 'A', tab, 'NO1',
                          ↓↓↓↓
Corresponding I/O List Items A , L.T,I.AB,NUMBER
                          ↓↓↓↓
Resulting Contents of List Items 101. A unchanged
                              ↓
                              NO1
                              ↓
                              I.AB
```

d. Slashes (/) cause the current input operation to be terminated even if all the items of the directing list are not filled. The contents of items of the directing I/O list which either are skipped (by null inputs) or have not received an input datum before the transfer is terminated remain unchanged. Once the I/O list of the controlling I/O statement is satisfied, the use of the / delimiter is optional.

e. Once the I/O list has been satisfied (transfers have been made to each item of the list) any items remaining in the input record are skipped.

Constants or nulls in data for list-directed input may be assigned a repetition factor to cause an item to be repeated.

The repetition of a constant is written as

```
r*K
```

where r is an integer constant that specifies the number of times the constant represented by K is to be repeated.

The repetition of a null is written as an integer followed by an asterisk.

Examples

- `10K5` represents `5,5,5,5,5,5,5,5,5,5`
- `3NABLE` represents `NABLE','NABLE','NABLE`'
- `3*` represents null,null,null
10.3.7 NAMELIST I/O Lists

One or more lists may be defined by a NAMELIST statement (Chapter 11). Each I/O list defined in a NAMELIST statement is identified by a unique (within the routine) 1 to 6 character name that may be referenced by one or more READ or WRITE statements. The first character of each I/O list name must be alphabetic. Referencing a NAMELIST-defined I/O list enables any of the foregoing statements to be written without an I/O list and permits the same list to be used by more than one statement.

I/O statements which reference a NAMELIST-defined I/O list cannot contain either a FORMAT statement reference or an I/O list. NAMELIST-controlled I/O operation cannot be used to transfer octal numbers or literal strings.

Records for NAMELIST-controlled input operations must be formatted in the following manner:

$NAME D1, D2, D3, ... Dn$

where

a. $ symbols delimit the beginning and end of the record. The first $ must be in column 2 of the input record; column 1 must be blank.

b. NAME is the name of a NAMELIST-defined input list. The named list identifies the processor storage locations that are to receive the data items read from the accessed record.

c. D1 through Dn are values of the items of data contained by the record; these items cannot be octal numbers or literal strings.

Only NAMELIST-controlled READ statements may be used to input records formatted in the foregoing manner.

NAMELIST-controlled WRITE statements will output records in the foregoing format.

NOTE

Device positioning commands such as BACKSPACE, SKIP RECORD, etc., should not be used in conjunction with NAMELIST-controlled I/O operations. If such a combination is used, the results will be unpredictable.

10.4 OPTIONAL READ/ WRITE ERROR EXIT AND END-OF-FILE ARGUMENTS

Either or both an error exit or an end-of-file argument may, optionally, be added to the parenthesized portion of most forms of the READ and WRITE I/O statements.

The error exit argument is written as ERR=c where c is a statement number. The use of this argument causes the current I/O operation to be terminated and program control transferred to the statement identified by the argument if a device error is detected. For example, the detection of an error during the execution of

READ(10,77,ERR=101) TABLE, I, M, J

terminates the input operation and transfers program control to statement 101.
The end-of-file argument is written as END=d where d is a statement number. The use of this argument causes the current I/O operation to be terminated and program control to be transferred to the statement identified by the argument when an end-of-file condition is detected. For example, the detection of an end-of-file condition during the execution of

\[ \text{READ}(10,77,\text{END}=50)\text{TABLE},1,M,J \]

transfers program control to statement 50.

If the END= argument is not present and an end of file (EOF) condition is detected, the file is closed, program execution is terminated, and control is returned to the monitor.

10.5 READ STATEMENTS

READ statements transfer data from peripheral devices into specified processor storage locations. The permitted forms of this type of input statement permit READ statements to be used on both sequential and random access transfer modes for formatted, unformatted, list-directed, and NAMELIST-controlled data transfers.

10.5.1 Sequential Formatted READ Transfers

Descriptions of the READ statements that may be used for the sequential transfer of formatted data follow:

a. Form: \( \text{READ}(u,\text{list}) \)

Use: Input data from logical unit \( u \), formatted according to the specifications given in \( \text{list} \), into the processor storage locations identified in input list.

Example: \( \text{READ}(10,555)\text{TABLE}(10,20),\text{ABLE},\text{BAKER},\text{CHARL} \)

b. Form: \( \text{READ}(u,\text{f}) \)

Use: Input the data from logical unit \( u \) directly into either a Hollerith (H) field descriptor or a literal field descriptor given within the format specifications of the referenced FORMAT statement. If the referenced FORMAT statement does not contain either of the foregoing types of format field descriptors, the input record is skipped. If a required field descriptor is present, its contents are replaced by the input data.

Example: \( \text{READ}(15,101) \)

c. Form: \( \text{READ}(\text{f}) \)

Use: Input the data from the READ default device (card reader) directly into either a Hollerith (H) field descriptor or a literal field descriptor given within the format specifications of the referenced FORMAT statement. If the referenced FORMAT statement does not contain either of the foregoing types of format field descriptors, the input record is skipped. If a required field descriptor is present, its contents are replaced by the input data.

Example: \( \text{READ} \ 66 \)
CHAPTER 10

Sequential Binary and
List-Directed READ Statements

d. Form: READ f, list

Use: Input the data from the READ default device (card reader) into the processor storage locations identified in the input list. The input data is formatted according to the specifications given in f.

Example: READ 15, ARRAY (20,30)

10.5.2 Sequential Unformatted Binary READ Transfers

Only the following form of the READ statement may be used for the sequential transfer of unformatted input FORTRAN binary data:

Form: READ (u)list

Use: Input one logical record of data from logical unit u into processor storage as the value of the location identified in list. Only binary files that have been output by a FORTRAN-10 unformatted WRITE statement may be read by this type of READ statement.

NOTE
If the form READ (u) is used, it will cause one unformatted input record to be skipped.

Example: READ (10) BINFIL (10,20,30)

10.5.3 Sequential List-Directed READ Transfers

The following forms of the READ statements may be used to control a sequential, list-directed input transfer:

a. Form: READ (u,*)list

Use: Input data from logical device u into processor storage or the value of the locations identified in list. Each input datum is converted, if necessary, to the type of its assigned list variable.

Example: READ (10,*) IARY (20,20), A,B,M

b. Form: READ *, list

Use: Input the data from the READ default device (card reader, CDR) into the processor storage locations identified in the input list. Each input datum is converted, if necessary, to the type of its assigned list variable.

Example: READ *, ABEL(10,20), J, K

October 1974
10.5.4 Sequential NAMELIST-Controlled READ Transfers

Only the following form of the READ statement may be used to initiate a sequential NAMELIST-controlled input transfer:

Form: \[ \text{READ (u,n)} \]

Use: Input data from logical unit u into processor storage as the value of the location identified by the NAMELIST input list specified by the name n. The input data is converted to the type of assigned variable if type conflicts occur. Only input files that contain records formatted and identified for NAMELIST operations (Paragraph 10.3.7) may be read by READ statements of this form.

10.5.5 Random Access Formatted READ Transfers

Only the following form of the READ statement may be used to initiate a random access formatted input transfer:

Form: \[ \text{READ (u\#R,f\#list)} \]

Use: Input data from record R of logical unit u. Format each input datum according to the format specifications of f and place into processor storage as values of the locations identified in list. Only disk files that have been set up by either an OPEN or DEFINE FILE statement may be accessed by a READ statement of this form. (If record R has not been written, a fatal error results.)

10.5.6 Random Access Unformatted READ Transfers

Only the following form of the READ statement may be used to initiate a random-access unformatted input transfer:

Form: \[ \text{READ (u\#R\#list)} \]

Use: Input data from record R of logical unit u. Place the input data into processor storage as the value of the locations identified in list. Only binary files that have been output by an unformatted random-access WRITE statement may be accessed by a READ statement of this form. (If record R has not been written, a fatal error results.)

Example: \[ \text{READ (1\#20) BINFIL} \]

Read record number 20 into array BINFIL.

NOTE

If the form READ (u\#R) is used, it will cause one logical input record to be skipped.

10.6 SUMMARY OF READ STATEMENTS

The various forms of the READ statements are summarized in Table 10-2.
### Table 10-2
Summary of Read Statements

<table>
<thead>
<tr>
<th>Type of Transfer</th>
<th>Transfer Mode</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sequential</td>
<td>Random Access</td>
</tr>
<tr>
<td>Formatted</td>
<td>READ (u,f)list</td>
<td>READ (u#R,f)list</td>
</tr>
<tr>
<td></td>
<td>READ (u,f)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>READ f_list</td>
<td></td>
</tr>
<tr>
<td></td>
<td>READ f</td>
<td></td>
</tr>
<tr>
<td>Unformatted</td>
<td>READ (u)list</td>
<td>READ (u#R)list</td>
</tr>
<tr>
<td></td>
<td>READ (u)</td>
<td>READ (u#R)</td>
</tr>
<tr>
<td></td>
<td>READ * list</td>
<td></td>
</tr>
<tr>
<td>List-Directed</td>
<td>READ (u,*)list</td>
<td></td>
</tr>
<tr>
<td>NAMELIST</td>
<td>READ (u,N)</td>
<td></td>
</tr>
</tbody>
</table>

Note: The ERR=c and END=d arguments may be included in any of the above READ statements. When included, the foregoing arguments must be last, e.g., READ (10,20,END=101,ERR=500) ARRAY (50,100).

### 10.7 REREAD STATEMENT

The REREAD statement causes the last record read from the last active input device to again be accessed and processed.

The REREAD feature of FORTRAN-10 cannot be used until an input (READ) transfer from a file has been accomplished. If REREAD is used prematurely, an error message will be output by FORTRAN-10 at execution time.

Once a record has been accessed by a formatted READ statement the record transferred may be reread as many times as desired. In a formatted transfer, the same or new format specification may be used by each successive REREAD statement.

The REREAD statement may be used for sequential formatted data transfers only. The form of the REREAD statement is:

**Form:**

```
REREAD f,list
```

**Use:**

Reread the last record read during the last initiated READ operation and input the data contained by the record into the processor storage locations specified in the input list. Format the data read according to the format specifications given in statement f.
Example:  
\[ \text{DIMENSION ARRAY}(10,10),\text{FORMA}(10,10),\text{FORMB}(10,10),\text{FORMC}(10,10) \]

90 READ(16,100)ARRAY 
   
   100 FORMAT(_______)
   
   110 RREAD 100,FOMA
115 RREAD 150,FOMB
120 RREAD 160,FOMC

150 FORMAT(_______)
160 FORMAT(_______)

In the above sequence, statement 90 inputs data formatted according to statement 100 into the array ARRAY. Statement 110 reads the record read by statement 90 and inputs the data formatted as in the initial READ operation into the array FORMA.

Statement 115 reads the record read by statements 90 and inputs the data formatted according to statement 150 into the array FORMB.

Statement 120 reads the record read by statement 90 and inputs the data formatted according to statement 160 into the array FORMC.

10.8 WRITE STATEMENTS

WRITE statements transfer data from specified processor storage locations to peripheral devices. The various forms of the WRITE statement enable it to be used in sequential, append and random access transfer modes for formatted, unformatted, list-directed and NAMELIST-controlled data transfers.

10.8.1 Sequential Formatted WRITE Transfers

The following forms of the WRITE statement may be used for the sequential transfer of formatted data:

a. Form: WRITE (u,f) list

   Use: Output the values of the processor storage locations identified in list, into the file associated with logical unit \( u \). Convert and arrange the output data according to the specifications given in statement or array \( f \).

   Example: WRITE(06,500)OUT(10,20),A,B

b. Form: WRITE f,list

   Use: Output the values of the processor storage locations identified in list to the default device (i.e., line printer, LPT). Convert and arrange the output data according to the specifications given in \( f \).

   Example: WRITE 10, SEND(5,10),A,B,C
CHAPTER 10

Sequential, NAMELIST—Controlled and Random Access WRITE Statements

c. Form: WRITE f

Use: Output the contents of any Hollerith (H) or literal (") field descriptor(s) contained by f to the default device (i.e., line printer, LPT). If neither of the foregoing types of field specifications are found in f, no output transfer is performed.

Example: WRITE 10

10.8.2 Sequential Unformatted WRITE Transfer

The following form of the WRITE statements may be used for the sequential transfer of unformatted data:

Form: WRITE (u) list

Use: Output the values of the processor storage locations identified in list into the file associated with logical unit u. No conversion or arrangement of output data is performed.

Example: WRITE(12)ITAB(20,20),SUMS(10,5,2)

10.8.3 Sequential List-Directed WRITE Transfers

The following form of the WRITE statement may be used to initiate a sequential list-directed output transfer.

Form: WRITE(u,*list

Use: Output the values of the processor storage locations identified in list into the file associated with logical unit u. The conversion of each datum from internal to external form is performed according to the type of the list variable from which the datum is read.

Example: WRITE(12,*C,X,Y,ITAB(10,10))

10.8.4 Sequential NAMELIST-Controlled WRITE Transfers

Only the following form of the WRITE statement may be used to initiate a sequential NAMELIST output transfer.

Form: WRITE(u,N)

Use: Output the values of the processor storage locations identified by the contents of the NAMELIST-defined list specified by name N.

Example: WRITE(12,NMLST)

10.8.5 Random Access Formatted WRITE Transfers

Only the following form of the WRITE statement may be used to initiate a random access type formatted output transfer:

Form: WRITE(u#R,f)list

Use: Output the values of the processor storage locations identified by the contents of list to record R of logical device u. Only disk files which have been set up by either an OPEN or a DEFINE FILE statement may be accessed by a WRITE transfer of this form. The data transferred will be formatted according to the specifications given in statement or array f. Only those records which have been specifically written are available to be read.

Version 4 FORTRAN 10-14

November 1975
10.8.6 Random Access Unformatted WRITE Transfers

Only the following form of the WRITE statement may be used to initiate a random access unformatted output transfer:

Form: \[ \text{WRITE}(u\#R)\text{list} \]

Use: Output the values of the processor storage locations identified by the contents of list to record \( R \) of the logical device unit \( u \). Only disk files which have been set up by either an OPEN or a call to the DEFINE FILE subroutine may be accessed by a WRITE transfer of this form. Only those records which have been specifically written are available to be read.

10.9 SUMMARY OF WRITE STATEMENTS

The various forms of the WRITE statements are summarized in Table 10-3.

<table>
<thead>
<tr>
<th>Type of Transfer</th>
<th>Transfer Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sequential</td>
</tr>
<tr>
<td>Formatted</td>
<td>WRITE((u,f))list</td>
</tr>
<tr>
<td></td>
<td>WRITE (f)list</td>
</tr>
<tr>
<td></td>
<td>WRITE (f)</td>
</tr>
<tr>
<td>Unformatted</td>
<td>WRITE((u))list</td>
</tr>
<tr>
<td>List-Directed</td>
<td>WRITE((u,*))list</td>
</tr>
<tr>
<td>NAMELIST-controlled</td>
<td>WRITE((u,N))</td>
</tr>
</tbody>
</table>

Note: The \( \text{ERR=}c \) and \( \text{END=}d \) arguments may be included in any WRITE statement; however, they must be last.

10.10 ACCEPT STATEMENT

The ACCEPT statement enables the user to input data via either a terminal keyboard or a Batch control file directly into specified processor storage locations. This statement is used only in the sequential transfer mode for the formatted transfer of inputs from the user's terminal keyboard during program execution. The permitted forms of the ACCEPT statement are described in the following paragraphs.

10.10.1 Formatted ACCEPT Transfers

The following forms of the ACCEPT statement are used for the sequential transfer of formatted data.

a. Form: \[ \text{ACCEPT } f,\text{list} \]

Use: Input data character-by-character into the processor storage locations identified by the contents of list. Format the input data according to the format specifications given in \( f \).

Example: \[ \text{ACCEPT 101,LINE(73)} \]
b.  Form:    ACCEPT *,list
Usage:     Input data character-by-character into the processor storage locations identified by the contents of list. Convert the input characters, where necessary, to the type of its assigned list variable.

Example:   ACCEPT *, IAB, ABE, KAB, MAR

10.10.2 ACCEPT Transfers Into FORMAT Statement
The following form of the ACCEPT statement may be used to input data from the user's terminal keyboard directly into a specified FORMAT statement if the FORMAT statement has either or both a Hollerith (H) or literal ('s') field descriptor. If the referenced statement has neither of the foregoing field descriptors, the input record is skipped.

Form:     ACCEPT f
Usage:    Replace the contents of the appropriate fields of statement f with the data entered at the user's terminal keyboard.

Example:  ACCEPT 101

10.11 PRINT STATEMENT
The PRINT statement causes data from specified processor storage locations to be output on the standard output device (i.e., line printer, LPT, Table 10-1). This statement may be used only for sequential formatted data transfer operation and may be written in either of the three following forms:

a.  Form:    PRINT f,list
Usage:     Output the values of the processor storage locations identified by the contents of list to the line printer. The values output are to be formatted and arranged according to the format specifications given in statement f.

Example:   PRINT 55,TABLE(10,20),I,J,K

b.  Form:    PRINT *,list
Usage:     Output the values of the processor storage locations identified by the contents of list to the line printer. The conversion of each datum from internal to external form is performed according to the type of the list variable from which the datum is read.

Example:   PRINT *,C,X,Y,ITAB(10,10)

c.  Form:    PRINT f
Usage:     Output the contents of the FORMAT statement Hollerith (H) or literal field descriptors to the line printer. If neither an H nor a literal field descriptor is present in the referenced FORMAT statement, no operation is performed.

Example:   PRINT 55
The second form of the PRINT statement is particularly useful when employed with ACCEPT f statements to cause desired data (i.e., comments or headings) to be inserted into reports at program execution time.

Example
The sequence

```
55 FORMAT ('$END$OF$ROUTINE')
```

results in the printing of the phrase END OF ROUTINE on the line printer.

10.12 PUNCH STATEMENT

The PUNCH statement causes data from specified processor storage locations to be output to the system's standard paper tape punch (PTP). (See Table 10-1 for device assignments.) This statement may be used only for sequential formatted data transfers and may be written in one of the three following forms:

a. Form: PUNCH f,list

Use: Output the values of the processor storage locations identified by the contents of list to the standard PTP unit. The values output are to be formatted and arranged according to the format specifications given in statement f.

Example: PUNCH 10,TABLE(10,20),I,J,K

b. Form: PUNCH *,list

Use: Output the values of the processor storage locations identified by the contents of list to the paper tape punch unit. The conversion of each datum from internal to external form is performed according to the type of the list variable from which the datum is read.

Example: PUNCH *,I,A,B,M,TAB(5,10)

c. Form: PUNCH f

Use: Output the contents of the referenced FORMAT statement Hollerith (H) or literal field descriptors to the standard PTP unit. If neither an H nor a literal field descriptor is present in the referenced FORMAT statement, no operation is performed.

The latter form of the PUNCH statement is particularly useful when employed in conjunction with an ACCEPT f statement to cause user-entered data (i.e., comments or headings) to be added to an output file at program execution time.
CHAPTER 10

10.13  TYPE STATEMENT

The TYPE statement causes data from specified processor storage locations to be output to the user's (control) terminal printing or display device (see Table 10-1 for device assignment for TYPE). This statement may be used only for sequential formatted data transfers and may be written in one of the following forms:

a. Form: TYPE f,list

   Use: Output the values of the processor storage locations identified by the contents of list to the user's terminal printing or display device. The values output are to be formatted according to the format specifications given in statement f.

   Example: TYPE 101,TABLE(10,20)I,J,K

b. Form: TYPE f

   Use: Output the contents of the referenced FORMAT statement Hollerith (H) or literal field descriptors to the user's terminal printing or display device. If the referenced FORMAT statement does not contain either an H or a literal field descriptor, no operation is performed.

   Example: TYPE 101

c. Form: TYPE *list

   Use: Output the values of the processor storage locations identified by the contents of list to the printing or display device of the user's terminal. The conversion of each datum from internal to external form is performed according to the type of the list variable from which the datum is read.

   Example: TYPE *I,AB(1,5),A,B

10.14  FIND STATEMENT

The FIND statement does not initiate a data transfer operation; it is used during random access read operations to locate the next record to be read while the current record is being input. The main program does not have access to the "found" record until the next READ statement is executed.

The form of the FIND statement is

    FIND (u#R)

Example

In the sequence

    READ (01#90)
    FIND (01#101)
    .
    .
    .
    READ (01#101)

the FIND statement will locate record #101 on device 01 after record 90 has been retrieved. Record #101 is not actually processed until the second READ statement in the sequence is executed.
10.15 ENCODE AND DECODE STATEMENTS

The ENCODE and DECODE statements are used to perform sequential formatted data transfer between two defined areas of processor storage (i.e., an I/O list and a user-defined buffer); no peripheral I/O device is involved in the operations performed by these statements.

The ENCODE statement transfers data from the variables of a specified I/O list into a specified user storage area. ENCODE operations are similar to those performed by a WRITE statement.

The DECODE statement transfers data from a specified user storage area into the processor storage locations identified by the variables of an I/O list. DECODE operations are similar to those performed by a READ statement.

The ENCODE and DECODE statements are written in the following forms:

```
ENCODE(c,f,s)list
DECODE(c,f,s)list
```

where

`c` specifies the number of characters to be in each internal storage area. This argument may be an integer, an integer expression, or either a real or double precision expression that is converted to an integer form.

**NOTE**

Characters are stored in the buffer five characters per storage location without regard to the type of variable given as the starting location.

`f` specifies either a FORMAT statement or an array that contains format specifications.

`s` specifies the address of the first storage location that is to be used in the transfer operations. When multiple records are specified by the format being used, the succeeding records follow each other in order of increasing storage addresses.

`list` specifies an I/O list of the standard form (Paragraph 10.3.4).

When multiple records are stored by ENCODE, each new record is started on a new boundary rather than there being a CRLF inserted between records.

10.15.1 ENCODE Statement

A description of the form and use of the ENCODE statement follows:

**Form:**

```
ENCODE(c,f,s)list
```

**Use:**

The values of the processor storage locations identified by the contents of list are converted to ASCII character strings according to the format specifications contained by f. The converted characters are then written into the destination area starting at location s. If more characters are to be transferred than the specified area can contain, the excess characters are ignored; they are not written into any following records.

If fewer characters are to be transferred than specified for the record size, the empty character locations are filled with blanks.

**Example:**

```
ENCODE(500,101,START)TABLE
```
10.15.2 DECODE Statement
A description of the form and use of the DECODE statement follows:

**Form:** \( \text{DECODE}(c,f,s)\mid \text{list} \)

**Use:** The character strings stored in the internal reference and are read starting at location \( s \), converted (decoded) according to the format specifications contained by \( f \), and stored as the values of the locations identified in \( \text{list} \).

If the format specification requires more characters from a record than are specified by \( c \), the extra characters are assumed to be blanks. If fewer characters are required from a record than are specified by \( c \), the extra characters are ignored.

**Example:** \( \text{DECODE}(50,50,\text{START})\text{GET}(5,10) \)

10.15.3 Example of ENCODE/DECODE Operations
The following program illustrates the use of both the ENCODE and DECODE statements:

**Example**

Assume the contents of the variables to be as follows:

- \( A(1) \) contains the floating point binary number 300.45
- \( A(2) \) contains the floating point binary number 3.0
- \( J \) is an integer variable
- \( B \) is a four-word array of indeterminate contents
- \( C \) contains the ASCII string 12345

```
DO 2 J = 1,2
ENCODE(16,10,B) J, A(J)
10 FORMAT (1X,2HA(J1,4H)\$=\$,F8.2)
   TYPE 11,B
11 FORMAT (4A5)
2 CONTINUE
   DECODE (4, 12, C) B
12 FORMAT (3F1.0,1X,F1.0)
   TYPE 13,B
13 FORMAT (4F5.2)
END
```

Array \( B \) can contain twenty ASCII characters. The result of the ENCODE statement after the first iteration of the DO loop is:

\[
\begin{align*}
B(1) & \quad A(1) & \text{Typed as} \\
B(2) & = & \\
B(3) & \quad 300.4 & \quad A(1)=300.45 \\
B(4) & \quad 5 & \\
\end{align*}
\]

The result after the second iteration is:

\[
\begin{align*}
B(1) & \quad A(2) & \text{Typed as} \\
B(2) & = & \\
B(3) & \quad 3.0 & \quad A(2)=3.0 \\
B(4) & & \\
\end{align*}
\]
CHAPTER 10

The DECODE statement

a. extracts the digits 1, 2, and 3 from C
b. converts them to floating point binary value
c. stores them in B(1), B(2), and B(3)
d. skips the next character
e. extracts the digit 5 from C
f. converts it to a floating point binary value, and,
g. stores it in B(4).

10.16 SUMMARY OF I/O STATEMENTS
A summary of all permitted forms of the FORTRAN-10 I/O statement is given in Table 10-4.
### Table 10-4
Summary of FORTRAN-10 I/O Statements

<table>
<thead>
<tr>
<th>I/O Statements</th>
<th>Formatted</th>
<th>Transfer Format Control</th>
<th>Namelist</th>
<th>List-Directed</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ Sequential</td>
<td>READ(u,f)list</td>
<td>READ(u)list</td>
<td>READ(u,n)</td>
<td>READ(u,*f)list</td>
</tr>
<tr>
<td>Random</td>
<td>READ(u#R,f)list</td>
<td>READ(u#R)list</td>
<td></td>
<td>READ(*,list)</td>
</tr>
<tr>
<td>WRITE Sequential or Append</td>
<td>WRITE(u,f)list</td>
<td>WRITE(u)list</td>
<td>WRITE(u,n)</td>
<td>WRITE(u,*f)list</td>
</tr>
<tr>
<td>Random</td>
<td>WRITE(u#R,f)list</td>
<td>WRITE(u#R)list</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REREAD Sequential</td>
<td>REREAD f.list</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIND Random-only</td>
<td>FIND(u#R)</td>
<td>FIND(u#R)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACCEPT Sequential only</td>
<td>ACCEPT f.list or ACCEPT f</td>
<td></td>
<td>ACCEPT *f.list</td>
<td></td>
</tr>
<tr>
<td>PRINT Sequential only</td>
<td>PRINT f.list or PRINT f</td>
<td></td>
<td>PRINT *f.list</td>
<td></td>
</tr>
<tr>
<td>PUNCH Sequential only</td>
<td>PUNCH f.list or PUNCH f</td>
<td></td>
<td>PUNCH *f.list</td>
<td></td>
</tr>
<tr>
<td>TYPE Sequential only</td>
<td>TYPE f.list or TYPE f</td>
<td></td>
<td>TYPE *f.list</td>
<td></td>
</tr>
<tr>
<td>ENCODE Sequential only</td>
<td>ENCODE(c,f,s)list</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DECODE Sequential only</td>
<td>DECODE(c,f,s)list</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Legend:**
- **u**: logical unit number
- **f**: statement number of FORMAT statement or name of array containing format information
- **list**: I/O list
- **n**: name of specific NAMELIST I/O list
- ***: symbol used to specify list-directed I/O operator
- **#R**: variable which specifies logical record position
- **c**: number of characters per internal record
- **s**: address of the first storage location to be used

---

1. An OPEN statement must be used to set up an append mode.
2. Either the OPEN statement or a call to the DEFINE FILE subroutine must be used to set up a random access mode.
CHAPTER 11

NAMELIST STATEMENTS

11.1 INTRODUCTION

The NAMELIST statement is used to define I/O lists similar to those described in Chapter 10 (Paragraph 10.3.4). Defined NAMELIST I/O lists are referenced in special forms of the READ and WRITE statements to provide a method of transferring and converting data without referencing format specifications or specifying an I/O list in the I/O statement.

11.2 NAMELIST STATEMENT

NAMELIST statements are written in the following form:

NAMELIST/N1/A1,A2,..,An/N2/B1,B2,..,Bn/Nn/...

where

/N/ through /Nn/  represents names of individual lists, the names are always written enclosed by slashes (/N/)

A1 through An

and

B1 through Bn

are the items of the lists identified, respectively, by names N1 and N2. A list may contain one or more variable, array, or array element names. The items of a list are delimited by commas. Each list of a NAMELIST statement is identified (and referenced to) by the name immediately preceding the list.

Example

NAMELIST/TABLE/A,B,C/SUMS/TOTAL

In the foregoing example, the name TABLE identifies the list A,B,C(2,4) and the name SUMS identifies the list comprised of the array TOTAL.

Once a list has been defined in a NAMELIST statement, its name may be referenced by one or more I/O statements.
The rules for structuring a NAMELIST statement are:

- A NAMELIST name may not be longer than six characters; it must start with an alphabetic character; it must be enclosed in slashes; it must precede the list of entries to which it refers; and it must be unique within the program.

- A NAMELIST name may be defined only once and must be defined by a NAMELIST statement. Once defined, a name may appear only in READ or WRITE statements. The NAMELIST name must be defined in advance of the I/O statement in which it is used.

- A variable used in a NAMELIST statement cannot be used as a dummy argument in a SUBROUTINE definition.

- Any dimensioned variable contained in a NAMELIST statement must have been defined in a preceding array declaration statement.

### 11.2.1 NAMELIST-Controlled Input Transfers

During input (read) transfer operations in which a NAMELIST-defined name is referenced, the record accessed is scanned until the symbol $ followed by the referenced name is found. Once the proper symbol-name combination is found, the data items following it are transferred on a one-to-one basis to the processor storage locations identified by the contents of the referenced list. The input data is always converted to the type of the list variable when there is a conflict of types. The input operation continues until another $ symbol is detected. If variables appear in the NAMELIST record that do not appear in the NAMELIST list, an error condition will occur. Data items of records to be input (read) using NAMELIST-defined lists must be separated by commas and may be of the following form:

\[ V=K_1, K_2, \ldots, K_n \]

where

- \( V \) may be a variable, array, or array element name.

- \( K_1 \) through \( K_n \) are constants of type integer, real, double precision, complex (written as \((A,B)\) where \(A\) and \(B\) are real), or logical (written as \(T\) for true or \(F\) for false). A series of identical constants may be represented as a single constant preceded by a repetition factor (e.g., \(5*5\) represents \(5,5,5,5,5\)).

In transfers of this type, logical and complex constants must be equated to variables of their own type. Other type constants (real, double precision, and integer) may be equated to any other type of variable (except logical or complex), and will be converted to the variable type. For example, assume \(A\) is a 2-dimensional real array, \(B\) is a 1-dimensional integer array, \(C\) is an integer variable, and that the input data is as follows:

\[ $FRED A(7,2)=4, B=3.6 * 2.8, C=3.32$ \]

A READ statement referring to the NAMELIST defined name FRED will result in the following: the integer 4 will be converted to floating point and placed in \(A(7,2)\). The integer 3 will be placed in \(B(1)\) and the integer 2 (converted) will be placed in \(B(2), B(3), \ldots, B(7)\). The floating point number 3.32 will be converted to the integer 3 and placed in \(C\).
11.2.2 NAMELIST-Controlled Output Transfers

When a WRITE statement refers to a NAMELIST-defined name, all variables and arrays and their values belonging to the named list are written out, each according to its type. Arrays are written out by columns. Output data is written so that:

a. The fields for the data will be large enough to contain all the significant digits.

b. The output can be read by an input statement referencing a NAMELIST-defined list.

For example, if JOE is a $2 \times 3$ array, the statement

```
NAMELIST/NAM1/JOE,K1,ALPHA
WRITE (u,NAM1)
```

generates the following form of output:

```
Column
```

```
+  
SNAM1

JOE= -6.75  .234E-04,  680,
     -17.8,  0.0  -.197E+07,
K1  =73.1,  ALPHA=3.3
```
CHAPTER 12

FILE CONTROL STATEMENTS

12.1 INTRODUCTION

File control statements are used to set up files and establish parameters for I/O operations and to terminate I/O operations.

- The OPEN and CLOSE statements are described in this chapter.

12.2 OPEN AND CLOSE STATEMENTS

Both the OPEN and CLOSE statements are unique to FORTRAN-10; they both use the same format and have the same options and arguments.

The OPEN statement enables the user to define, explicitly, all of the important aspects of each desired data transfer operation; they provide an extensive list of required and optional arguments which define in detail:

a. the name and location of the data file
b. the type of access required
c. the data format within the file
d. the protection code\(^1\) to be assigned an output data file
e. the disposition of the data file
f. data file record, block and file sizes
g. a data file version identifier

In addition, a DIALOG argument is provided which permits the user to establish a dialogue mode of operation when the OPEN statement containing it is executed. In a dialogue mode, interactive user terminal/program communication is established. This enables the user, during program execution, to define, redefine, or defer the values of the optional arguments contained by the current OPEN statement.

\(^1\)Refer to Chapter 6 of the DECsystem-10 Monitor Calls Manual, DEC-10-MRRC-D, for a description of file access protection codes.
The general form of the OPEN statement is:

\[ \text{OPEN}(\text{Arg1,Arg2,...,Argn}) \]

The CLOSE statement is used in the termination of an I/O operation to dissociate the I/O device being used from the active file and file-related information, and to restore the core occupied by I/O buffers and other transfer-related operations. All required device dependent termination functions are also performed on the execution of a CLOSE statement, including reloading the unit. Note that the CLOSE statement can change the name, protection, directory, and disposition of the file being closed.

Once a CLOSE statement has been executed, another OPEN statement is required to regain access to the closed file.

The general form of the CLOSE statement is:

\[ \text{CLOSE}(\text{Arg1,Arg2,...,Argn}) \]

12.2.1 Options for OPEN and CLOSE Statements

The options and their arguments, which may be used in both the OPEN and CLOSE statements, are:

a. UNIT

This option is required; it defines the FORTRAN I/O unit number to be used. FORTRAN devices are identified by assigned decimal numbers within the range 1–63; however, UNIT may be assigned an integer variable or constant. The general form of this argument is:

\[ \text{UNIT} = \text{An integer variable or constant} \]

NOTE

FORTRAN-10 standard logical unit assignments are described in Chapter 10 (Table 10-1). The range (i.e., 1–63) for the possible UNIT numbers is an installation defined parameter.

b. DEVICE

This option may specify either the physical or the logical name of the I/O device involved. (A logical name always takes precedence over a physical name.) The DEVICE arguments may specify I/O devices located at remote stations, as well as logical devices. The general form of the DEVICE argument is:

\[ \text{DEVICE} = \text{A literal constant or variable} \]

If this option is omitted, the first logical name \( u \) (where \( u \) is the decimal unit number) is tried; if this is not successful, the standard (default) device is attempted.

c. ACCESS

A required option, ACCESS describes the type of input and/or output statements and the file access mode to be used in a specified data transfer operation. ACCESS may be assigned any one of six possible names, each of which specifies a specific type of I/O operation. The assignable names and the operations specified are:
1. **SEQIN**  
The specified data file is to be read in sequential access mode.

2. **SEQOUT**  
The specified data file is to be written in a sequential access mode.

3. **SEQINOUT**  
The specified data file may be first read then written (READ/WRITE sequence) record-by-record in a sequential access mode. When SEQINOUT is specified, a WRITE/READ sequence is illegal unless the file has been removed.

4. **RANDOM**  
The specified data file may be either read or written into, one record at a time. In a random access mode of operation, the relative position of each record is independent of the previous READ or WRITE statement; all records accessed must have a fixed logical record length. This argument is required for random access operations. A disk device must be specified when the random argument is used.

5. **RANDIN**  
This argument enables the user to establish a special, read-only random access mode with a named file. During a RANDIN mode, the user may read the named file simultaneously with other users who have also established a RANDIN mode and with the owner of the file. The use of RANDIN enables a data base to be shared by more than one user at the same time.

6. **APPEND**  
The record specified by a corresponding WRITE statement is to be added to the logical end of a named file. The modified file must be closed then reopened in order to permit it to be read.

The general form of the ACCESS argument is:

```
'SEQIN'
'SEQOUT'
'SEQINOUT'
'RANDOM'
'RANDIN'
'APPEND'

variable (set to literal)
```

d. **MODE**

This option defines the character set of an external file or record. The use of this argument is optional; if it is not given, one of the following is assumed:

- **ASCII** for a formatted I/O file transfer
- **Binary** for an unformatted I/O file transfer
One of the following character set specifications must be used with the MODE argument:

**NOTE**
Refer to the DECsystem-10 Monitor Calls Manual for a detailed description of the data modes given in the following list.

<table>
<thead>
<tr>
<th>Literal</th>
<th>Action Indicated</th>
</tr>
</thead>
<tbody>
<tr>
<td>'ASCII'</td>
<td>Specifies an ASCII character set.</td>
</tr>
<tr>
<td>'BINARY'</td>
<td>Specifies data formatted as a FORTRAN binary data file.</td>
</tr>
<tr>
<td>'IMAGE'</td>
<td>Specifies an image (I) mode data transfer for the associated READ or WRITE statements. IMAGE is an unformatted binary mode.</td>
</tr>
<tr>
<td>'DUMP'</td>
<td>The data file to be transferred is to be handled in a DUMP mode of operation.</td>
</tr>
</tbody>
</table>

The general form of the MODE argument is:

```
MODE =
    'ASCII'
    'BINARY'
    'IMAGE'
    'DUMP'
    variable (set to literal)
```

e. **DISPOSE**

This option specifies an action to be taken regarding a file at close time. When used, DISPOSE must be either an ASCII variable or one of the following literals:

<table>
<thead>
<tr>
<th>Literal</th>
<th>Action Indicated</th>
</tr>
</thead>
<tbody>
<tr>
<td>'SAVE'</td>
<td>Leave the file on the device.</td>
</tr>
<tr>
<td>'DELETE'</td>
<td>If the device involved is either a DECtape or disk, remove the file; otherwise, take no action.</td>
</tr>
<tr>
<td>'PRINT'</td>
<td>If the file is on disk, queue it for printing; otherwise, take no action.</td>
</tr>
<tr>
<td>'LIST'</td>
<td>If the file is on disk, queue it for printing and delete the file; otherwise take no action.</td>
</tr>
<tr>
<td>'PUNCH'</td>
<td>Paper tape punch output.</td>
</tr>
<tr>
<td>'RENAME'</td>
<td>Change filename. (This is redundant if a new filename is given.)</td>
</tr>
</tbody>
</table>

If the DISPOSE argument is not given, the argument DISPOSE = SAVE is assumed. The general form of the DISPOSE argument is:

```
DISPOSE =
    'SAVE'
    'DELETE'
    'PRINT'
    'PUNCH'
    'RENAME'
    variable (set to literal)
```
This option specifies the name of the file involved in the data transfer operation. FILE must be either an ASCII literal, double precision, complex, or single precision variable. Single precision variables are assumed to contain a 1 to 5 character file specification; double precision variables, permit 10-character file specification. The format is a 1 to 6 character filename optionally followed by a period and a 0 to 3 character extension. Any excess characters in either the name or extension are ignored. If the period and extension are omitted, the extension .DAT is assumed; if just the extension is omitted, a "." is assumed.

If a file name is not specified or is zero, a default name is generated which has the form

FORxx:DAT

where xx is the FORTRAN logical unit number (decimal) or is the logical unit name for the default statements ACCEPT, PRINT, PUNCH, READ, or TYPE. The general form of a FILE argument is:

FILE = An ASCII literal or variable (set to literal)

g. PROTECTION

This option specifies a protection code to be assigned the data file being transferred. The protection code determines the level of access to the file that three possible classes of users (i.e., owner, member, or other) will have. PROTECTION may be a 3-digit octal literal or a variable; if the argument is assigned a zero value or is not given, the default protection code established for the DECSYSTEM-10 installation is used. The general form of the PROTECTION argument is:

PROTECTION = 3-digit octal or integer variable

h. DIRECTORY

This option is used for disk files only. It specifies the location of the user file directory (UFD) or the sub-file directory (SFD) which contains the file specified in the OPEN statement. A directory identifier may consist of either:

1. the user's project programmer number which identifies the UFD, for example, 10,7, or

2. A UFD/SFD directory path specification. A path specification lists the UFD and the names of its SFD's which form a path to the desired SFD. For example, the following path specification identifies the path leading to SFD 1234:

10,7,SFDA,SFDB,1234

NOTE
Refer to the DECSYSTEM-10 Monitor Calls manual for a complete description of directories and multilevel directory structures.
The general form of a DIRECTORY argument is:

\[ \text{DIRECTORY} = \text{Literal or variable containing UFD name or directory path specification} \]

The user may also establish an array containing the directory specification as its elements and reference the array in the DIRECTORY argument. Single precision arrays permit 5-character directory names to be used; double precision arrays permit 6-character names to be used. A zero (0) entry must be used to terminate a directory path specification given in an array.

Examples of the use of single and double precision arrays in an OPEN statement DIRECTORY specification follow:

1. Single Precision Array

\[ \text{OPEN (UNIT = 5, DIRECTORY = PATH, \ldots)} \]

where PATH and its elements are:

\[
\begin{align*}
\text{DIMENSION PATH (5)} \\
\text{PATH (1)} &= "10 -- (PROJECT NUMBER) \\
\text{PATH (2)} &= "7 -- (PROGRAMMER NUMBER) \\
\text{PATH (3)} &= 'SFDA' \quad \text{Names of sub-file directories (SFD's)} \\
\text{PATH (4)} &= 'SFDB' \\
\text{PATH (5)} &= 0
\end{align*}
\]

2. Double Precision Array

\[ \text{OPEN (UNIT=5, DIRECTORY = PATH, \ldots)} \]

where PATH and its elements are:

\[
\begin{align*}
\text{DOUBLE PRECISION PATH (5)} \\
\text{PATH (1)} &= "000010000007 -- (PROJ, PROG. NUMBERS = UFD) \\
\text{PATH (2)} &= 'SFDABC' \quad \text{names of sub-file directories (SFD's)} \\
\text{PATH (3)} &= 'MYAREA' \\
\text{PATH (4)} &= 'YOURIT' \\
\text{PATH (5)} &= 0
\end{align*}
\]

The elements of a directory specification may then be either a literal or a single or double precision array.

The following is an example of a literal specification:

\[ \text{(DIRECTORY} = '10,7,SFD1,SFD2,SFD3') \]

*Project*  *Sub-File*

*Programmer*  *Directory*

*Number*  *Path*
Whenever the specification is an array, the required project and programmer numbers may be specified either as one word with the project number in the left half and the programmer number in the right half, or as the right halves of separate sequential word locations.

**i. BUFFER COUNT**

This option enables the user to specify the number of I/O buffers to be assigned to a particular device. If this argument is not given or is assigned a value of zero, the Monitor default is assumed. The general form of this argument is:

\[
\text{BUFFER COUNT} = \text{An integer constant or variable}
\]

**j. FILE SIZE**

This option is used for disk operations only; it enables the user to estimate the number of words that an output file is going to contain. The use of FILE SIZE enables the user to ensure at the start of a program that enough space is available for its execution. If the size specified is found to be too small during program executions, the Monitor allocates additional space according to the normal Monitor algorithms. The value assigned to the FILE SIZE arguments may be an integer constant or variable. The general form of this argument is:

\[
\text{FILE SIZE} = \text{An integer constant or variable}
\]

**k. VERSION**

This option is used for disk operations only; it enables the user to assign a 12-digit octal version number to a file when it is output. The quantity assigned to the VERSION argument may be either an octal constant or variable. The general form of the argument is:

\[
\text{VERSION} = \text{An octal constant or integer variable}
\]

**l. BLOCK SIZE**

This option can be used for all storage media except disk and DECtape. It enables the user to specify a physical storage block size for devices other than disk or DECtape. The value assigned the BLOCK SIZE arguments may be an integer constant or variable. The size specified must be greater than or equal to 3 and less than or equal to 4095. The general form of this argument is:

\[
\text{BLOCK SIZE} = \text{An integer constant or variable}
\]

**m. RECORD SIZE**

This option enables the user to force all logical records to be a specified length. If a logical record exceeds the specified length, it is truncated; if a logical record is less than the specified size, nulls are added to pad the record to its full size. The RECORD SIZE argument is required whenever a random access mode is specified. The value assigned to this argument may be either an integer constant or variable, and may be expressed as the numbers of words or characters depending on the mode of the file being described. The general form of this argument is:

\[
\text{RECORD SIZE} = \text{An integer constant or variable}
\]
n. **ASSOCIATE VARIABLE**

This option is for disk random access operations only. It provides storage for the number of the record to be accessed next if the program being executed were to continue to access records one after another from the specified random access file. The general form of this argument is:

\[
\text{ASSOCIATE VARIABLE} = \text{Integer variable}
\]

o. **PARITY**

This option is for magnetic tape operations only; it permits the user to specify the type of parity to be observed (odd or even) during the transfer of data. The general form of this option is:

\[
\text{PARITY} = \begin{cases} 
\text{'ODD'} \\
\text{'EVEN'} \\
\text{variable (set to literal)} 
\end{cases}
\]

p. **DENSITY**

This option is for magnetic tape operations only; it permits the user to specify any of three possible bit-per-inch (bpi) tape density parameters for magnetic tape transfer operations. The general form of this option is:

\[
\text{DENSITY} = \begin{cases} 
\text{'200'} \\
\text{'356'} \\
\text{'800'} \\
\text{variable (set to literal)} 
\end{cases}
\]

q. **DIALOG**

The use of this option in an OPEN statement enables the user to supersede or defer, at execution time, the values previously assigned to the arguments of the statement. There are two forms of this argument. The first is:

\[
\text{DIALOG}
\]

This form establishes a dialogue with the user's terminal when the OPEN statement is executed. FORTS outputs the following messages at the user's terminal.

**ENTER FILE SPECIFICATIONS FOR LOGICAL UNIT XX**

(FORTS then types the existing file specifications defined by the current OPEN statement.)

Once the message and defined file specification are output the user may enter any desired changes. Only the arguments that are to be changed need to be entered.

The second form of the argument is:

\[
\text{DIALOG} = \text{Literal or array}
\]

The value assigned to DIALOG may be a literal or array containing a file specification with the desired information.
12.2.2 Summary of OPEN/CLOSE Statement Options

The options permitted and required in the OPEN and CLOSE statements and the type of value required by each are summarized in Table 12-1.

Table 12-1
OPEN/CLOSE Statement Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Values Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIT =</td>
<td>Integer variable or constant</td>
</tr>
<tr>
<td>MODE =</td>
<td>Literal constant or variable</td>
</tr>
<tr>
<td>DIRECTORY =</td>
<td>Literal or variable</td>
</tr>
<tr>
<td>FILE SIZE =</td>
<td>Integer constant or variable</td>
</tr>
<tr>
<td>BUFFER COUNT =</td>
<td>Integer constant or variable</td>
</tr>
<tr>
<td>ASSOCIATE VARIABLE =</td>
<td>Integer variable</td>
</tr>
<tr>
<td>ACCESS =</td>
<td>'SEQIN', 'SEQOUT', 'SEQINOUT', 'RANDIN', 'RANDOM', 'APPEND', or variable</td>
</tr>
<tr>
<td>FILE =</td>
<td>Literal constant or variable</td>
</tr>
<tr>
<td>DIALOG =</td>
<td>Literal or array</td>
</tr>
<tr>
<td>BLOCK SIZE =</td>
<td>Integer constant or variable</td>
</tr>
<tr>
<td>VERSION =</td>
<td>Octal constant or variable</td>
</tr>
<tr>
<td>DEVICE =</td>
<td>Literal constant or variable</td>
</tr>
<tr>
<td>PROTECTION =</td>
<td>An octal constant or integer variable</td>
</tr>
<tr>
<td>DISPOSE =</td>
<td>Literal constant or variable</td>
</tr>
<tr>
<td>RECORD SIZE =</td>
<td>Integer constant or integer variable</td>
</tr>
<tr>
<td>PARITY =</td>
<td>Literal constant or variable</td>
</tr>
<tr>
<td>DENSITY =</td>
<td>Literal constant or variable</td>
</tr>
</tbody>
</table>
CHAPTER 13
FORMAT STATEMENT

13.1 INTRODUCTION
FORMAT statements are used in conjunction with the I/O list of I/O statements during formatted data transfer operations. The FORMAT statements contain field descriptors which, together with the list items of associated I/O statements, specify the forms of the data and data fields which comprise each record.

FORMAT statements may appear almost anywhere in a FORTRAN-10 source program. The only placement restrictions are that they follow PROGRAM, FUNCTION, SUBPROGRAM, or BLOCK DATA statements, and that they precede the END statement. (Refer to Section 2.4.)

FORMAT statements must be labeled so that they can be referenced by I/O statements.

13.1.1 FORMAT Statement, General Form
The general form of a FORMAT statement follows:

\[ k \text{ FORMAT}(S{A}_1, S{A}_2, \ldots, S{A}_n, S{B}_1, S{B}_2, \ldots, S{B}_n) \]

where

- \( k \) = the required statement label (which can only be referenced by I/O statements).
- \( S{A}_1 \) through \( S{A}_n \) = individual field descriptor sets
- \( S{B}_1 \) through \( S{B}_n \) = the required statement label (which can only be referenced by I/O statements).

In the foregoing statement form the individual field descriptors are delimited by commas (,) field descriptor sets and records are delimited by slashes (/). For example, a FORMAT statement of the form:

\[ \text{FORMAT}(S{A}_1, S{A}_2/SC_1, S{B}_1, S{B}_2) \]

contains format specifications for three records with each record comprised of two field descriptor sets.
Adjacent slashes (//) in a FORMAT statement specify that a record is to be skipped during input or is to consist of an empty record on output. For example, a FORMAT statement of the form:

```
FORMAT(SA1,SA2//[SB1,SB2]
```

specifies four records are to be processed; however, the second and third records are to be skipped.

Repeated field descriptors or groups of field descriptors may be represented using a repeat form. The repetition of a single field descriptor is written by preceding the descriptor with an integer constant which specifies how many times the descriptor is to be repeated. For example, a FORMAT statement of the form

```
FORMAT(SA1,SA2,SA3,SA1,SA2,SA3,SA1,SA2,SA3)
```

may be written as

```
FORMAT(3(SA1,SA2,SA3))
```

The repeat forms of field descriptor may be nested to any depth. For example, a FORMAT statement of the form

```
FORMAT(SA1,SA2,SA3,SA1,SA2,SA3,SA1,SA2,SA3)
```

may also be written in the form

```
FORMAT(2(SA1,2SA2,SA3))
```

The manner in which the foregoing statement forms may be used and the effect each has on the data involved are discussed in the following paragraphs.

### 13.2 FORMAT DESCRIPTORS

FORMAT statement descriptors describe the record structure of the data, the format of the fields within the record, and the conversion, scaling, and editing of data within specific fields. The following descriptors can be used with FORTRAN-10:

<table>
<thead>
<tr>
<th>Descriptors</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>rFw.d</td>
<td>Floating point numeric field descriptors</td>
</tr>
<tr>
<td>rEw.d</td>
<td></td>
</tr>
<tr>
<td>rDw.d</td>
<td></td>
</tr>
<tr>
<td>rGw.d</td>
<td></td>
</tr>
<tr>
<td>rIw</td>
<td>Integer field descriptor</td>
</tr>
<tr>
<td>rLw</td>
<td>Logical field descriptor</td>
</tr>
<tr>
<td>rAw</td>
<td>Alphanumeric data field descriptor</td>
</tr>
<tr>
<td>rRw</td>
<td></td>
</tr>
<tr>
<td>kHs 'text'</td>
<td>Alphanumeric data in a FORMAT statement field descriptor</td>
</tr>
<tr>
<td>rX</td>
<td>Field formatting descriptors</td>
</tr>
<tr>
<td>Tw</td>
<td></td>
</tr>
</tbody>
</table>
chapter 13

FORTRAN Conversion Codes

<table>
<thead>
<tr>
<th>Descriptors</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>nP</td>
<td>Numerical scale factor descriptor</td>
</tr>
<tr>
<td>/</td>
<td>Record delimiter</td>
</tr>
<tr>
<td>$</td>
<td>Carriage return suppression for terminal</td>
</tr>
<tr>
<td>rOw</td>
<td>Octal field descriptor</td>
</tr>
</tbody>
</table>

where

\[ r = \text{an optional unsigned integer that represents a repeat count. This option enables a field descriptor to be repeated } r \text{ times.} \]

\[ w = \text{an optional integer constant which represents the width (total number of characters contained) of the external form of the field being described. All characters including digits, decimal points, signs, and blanks that are to comprise the external form of the field must be included in the value of } w. \]

\[ .d = \text{an optional unsigned integer that specifies the number of fractional digits which are to appear in the external representation of the field being described. Note that } w \text{ must be specified if } .d \text{ is included in the descriptor.} \]

\[ k = \text{An unsigned integer that specifies the number of characters to be processed during the transfer of alphanumeric data.} \]

\[ s = \text{represents a string of ASCII (alphanumeric) characters.} \]

\[ n = \text{a signed integer constant (plus signs are optional).} \]

The characters A, D, E, F, G, H, I, L, O, P, and R indicate the manner of conversion and editing to be performed between the internal (processor) and external representations of the data within a specific field; these characters are referred to as conversion codes. The FORTRAN-10 conversion codes and a brief description of the function of each are given in Table 13-1.

<table>
<thead>
<tr>
<th>Table 13-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORTRAN-10 Conversion Codes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Transfer alphanumeric data</td>
</tr>
<tr>
<td>D</td>
<td>Transfer real data with a D exponent (^1)</td>
</tr>
<tr>
<td>E</td>
<td>Transfer real data with an E exponent (^1)</td>
</tr>
<tr>
<td>F</td>
<td>Transfer real data without an exponent</td>
</tr>
<tr>
<td>G</td>
<td>Transfer integer, real, complex, or logical data</td>
</tr>
<tr>
<td>H</td>
<td>Transfer literal data</td>
</tr>
<tr>
<td>I</td>
<td>Transfer integer data</td>
</tr>
<tr>
<td>L</td>
<td>Transfer logical data</td>
</tr>
<tr>
<td>O</td>
<td>Transfer octal data</td>
</tr>
<tr>
<td>P</td>
<td>Numerical scaling Factor</td>
</tr>
<tr>
<td>R</td>
<td>Transfer alphanumeric data</td>
</tr>
</tbody>
</table>

\(^1\) An exponent of 0 is assumed if none is given.

Version 4 FORTRAN

13-3

October 1974
The use of commas to delineate format descriptors within a format specification is optional as long as no ambiguity exists. For example,

```markdown
FORMAT (3X,A2)
```
can be written as

```markdown
FORMAT (3XA2)
```

Since interpretation of a format specification is left associative, the specification

```markdown
FORMAT (12,15)
```
can be written as

```markdown
FORMAT (12,15)
```

However, a comma is required when the user wishes to specify

```markdown
FORMAT (12,215)
```

Detailed descriptions of the various types of format descriptors, the manner in which they are written and employed and their use in FORMAT statements are given in the following paragraphs.

### 13.2.1 Numeric Field Descriptors

The forms of the field descriptors used to specify the format and conversion of numeric data follow.

<table>
<thead>
<tr>
<th>Description</th>
<th>Type of Data Used For</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dw.d</td>
<td>Double precision real data with a D exponent</td>
</tr>
<tr>
<td>Ew.d</td>
<td>Real data with an E exponent</td>
</tr>
<tr>
<td>Ew.d,Ew.d</td>
<td>For the real and imaginary parts of a complex datum</td>
</tr>
<tr>
<td>Fw.d</td>
<td>Real data without an exponent</td>
</tr>
<tr>
<td>Fw.d,Fw.d</td>
<td>For the real and imaginary parts of a complex datum</td>
</tr>
<tr>
<td>lw</td>
<td>Integer data</td>
</tr>
<tr>
<td>Ow</td>
<td>Octal data</td>
</tr>
<tr>
<td>Gw.d</td>
<td>Real or double precision data</td>
</tr>
<tr>
<td>Gw</td>
<td>For integer (or logical) data</td>
</tr>
<tr>
<td>Gw.d,Gw.d</td>
<td>For the real and imaginary parts of a complex datum</td>
</tr>
</tbody>
</table>

**NOTE**

The G conversion code may be used for all but octal numeric data types.

#### Examples

Consider the following program segment:

```fortran
INTEGER I1, I2
REAL R1, R2, R3
DOUBLE PRECISION D1, D2
I1 = 506
I2 = 8
R1 = 506.0
R2 = 18.1
R3 = 506001.0
D1 = 18.0
D2 = -504.0
```

...
CHAPTER 13

Field Descriptors, Action of

The actions performed by several types of formatted WRITE statements on the data given in the foregoing program segment are described in Table 13-2.

<table>
<thead>
<tr>
<th>Item</th>
<th>Descriptor Form</th>
<th>Sample Descriptor</th>
<th>WRITE Statement Using the Sample Descriptor</th>
<th>External Form of Sample Field Described</th>
<th>External Appearance of Sample Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dw.d</td>
<td>D8.2</td>
<td>WRITE (\cdot) D1</td>
<td>Z.nnD±nn</td>
<td>0.18D+02</td>
</tr>
<tr>
<td>2</td>
<td>Ew.d</td>
<td>E8.2</td>
<td>WRITE (\cdot) R1</td>
<td>Z.nnE±nn</td>
<td>0.51E+03</td>
</tr>
<tr>
<td>3</td>
<td>Fw.d</td>
<td>F5.2</td>
<td>WRITE (\cdot) R2</td>
<td>aa.nnE±nn</td>
<td>18.10</td>
</tr>
<tr>
<td>4</td>
<td>Iw</td>
<td>I5</td>
<td>WRITE (\cdot) I1</td>
<td>aaaaan</td>
<td>$&amp;$506</td>
</tr>
<tr>
<td>5</td>
<td>Iw</td>
<td>I2</td>
<td>WRITE (\cdot) I1</td>
<td>an</td>
<td>**</td>
</tr>
<tr>
<td>6</td>
<td>Ow</td>
<td>O5</td>
<td>WRITE (\cdot) I2</td>
<td>nnnnn</td>
<td>00010</td>
</tr>
<tr>
<td>7</td>
<td>Gw.d</td>
<td>G8.2</td>
<td>WRITE (\cdot) D2</td>
<td>Z.nnD±nn</td>
<td>-.50D+02</td>
</tr>
<tr>
<td>8</td>
<td>Gw.d</td>
<td>G8.2</td>
<td>WRITE (\cdot) R3</td>
<td>Z.nnE±nn</td>
<td>0.51E+06</td>
</tr>
<tr>
<td>9</td>
<td>Gw.d</td>
<td>G8.2</td>
<td>WRITE (\cdot) R2</td>
<td>aaaaan</td>
<td>18.10</td>
</tr>
<tr>
<td>10</td>
<td>Gw</td>
<td>G5</td>
<td>WRITE (\cdot) I1</td>
<td>$&amp;$506</td>
<td></td>
</tr>
</tbody>
</table>

where:  
a. $n$ represents a numeric character  
b. $Z$ represents either a - or 0 (Note that if n-d > 6, a negative number cannot be output.)  
c. $a$ represents a digit, leading blank ($\&$) or a minus sign depending on the numeric output.

Notes:

1. In Item 1, the value D1 has only 2 significant digits and d=2, so no rounding will occur on input.

2. In Item 2, since R1 has 3 significant digits, it is rounded to fit into the specified field.

3. In Item 5, the width (w) part of a format descriptor specifies an exact field which permits no rounding of its contents. If the w specification is too small for the datum to be transferred, asterisks are output to indicate that the transfer was not made.

4. In Item 6, Integer 8 = Octal 10.

5. In Items 8 and 9, the relationship between G and fixed and floating real data is discussed in Paragraph 13.2.3.

6. In items 1, 2, 3, 7, and 8 the D and E exponent prefixes are optional in the external form of the floating point constants. For example, 1.1E+3 may be written as 1.1+3.

The internal and external forms of the data specified by the numeric format conversion code are summarized in Table 13-3.

Version 4 FORTRAN 13-5 November 1975
### Table 13-3
*Field Descriptors With List Variables*

<table>
<thead>
<tr>
<th>Internal Form</th>
<th>Conversion Code</th>
<th>External Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary floating point double precision</td>
<td>D</td>
<td>Decimal floating point with D exponent</td>
</tr>
<tr>
<td>Binary floating point</td>
<td>E</td>
<td>Decimal floating point with E exponent</td>
</tr>
<tr>
<td>Binary floating point</td>
<td>F</td>
<td>Decimal fixed point</td>
</tr>
<tr>
<td>Binary integer</td>
<td>I</td>
<td>Decimal integer</td>
</tr>
<tr>
<td><strong>Binary word</strong></td>
<td><strong>O</strong></td>
<td><strong>Octal value</strong></td>
</tr>
<tr>
<td>One of the following: single precision, binary floating point, binary integer, binary logical, or binary complex</td>
<td>G</td>
<td>Single precision decimal floating point integer, logical (T or F), or complex (two decimal floating point numbers), depending upon the internal form</td>
</tr>
</tbody>
</table>

Complex quantities are transferred as two independent real quantities. The format specification for complex quantities consists of either two successive real field descriptors or one repeated real field descriptor. For example, the statement

```
FORMAT(2E15.4,2(F8.3,F8.5))
```

may transfer up to three complex quantities.

The equivalent of the foregoing statement is

```
FORMAT(E15.4,E15.4,F8.3,F8.5,F8.3,F8.5)
```

#### 13.2.2 Interaction of Field Descriptors With I/O List Variables During Transfer

The execution of an I/O statement that specifies a formatted data transfer operation initiates format control. The actions performed by format control depend on information provided by the elements of the I/O statement’s list of variables and the field descriptors which comprise the referenced FORMAT statement’s format specifications.

In processing each FORMAT controlled I/O statement which has an I/O list, FORTRAN-10 scans the contents of the list and the format specifications in step. Each time another variable or array element name is obtained from the list, the next field specification is obtained from the format specification. If the end of the format specification is reached and more items remain in the list, a new line or record is established and the scan process is restarted, either at the first item in the format specification or, if parenthesized sets of format specifications exist within the format specification, with the last set within the format specification.

When the I/O list is exhausted, control proceeds to the next statement in the program, but not before the FORMAT statement is scanned either to its end or to the next variable transfer format descriptor. (That is, the FORMAT statement is scanned past slashes, literal constants, and spacing descriptors, but not past data field descriptors.)
A record is terminated by one of the following:

a. a slash in the FORMAT specification
b. the delimiting right parentheses, ), of the FORMAT statement
c. a lack of items in the I/O list
d. a lack of Hollerith field descriptors in the FORMAT statement

On input, an additional record is read only when a single slash, /, is encountered in the FORMAT statement. A record is skipped when two slashes, //, are encountered or a slash is followed by the end of the FORMAT statement. If the FORMAT statement finishes a record by a slash or the end of the FORMAT statement, then any data left in the input record is ignored. If the input record is exhausted before the data transfers are completed, the remainder of the transfer is completed as if the record were extended with blanks.

On output, an additional record is written only when a slash, /, is encountered in the FORMAT statement. If two consecutive slashes, //, or a single slash followed by the end of the FORMAT statement, is encountered, then an empty record is written.

13.2.3 G, General Numeric Conversion Code

The G conversion code may be used in field descriptors for the format control of real, double precision, integer, logical, or complex data.

With the exception of real and double precision data, the type of conversion performed by a G type field descriptor depends on the type of its corresponding I/O list variable. In the case of real and double precision data, the kind of conversion performed is a function of the external magnitude of the datum being transferred. Table 13-4 illustrates the conversions performed for various ranges of magnitude (external form) of real and double-precision data.

13.2.4 Numeric Fields with Scale Factors

Scale factors may be added to D,E,F, and G conversion codes in field descriptors. The scale factor has the form $nP$

where $n$ is a signed integer (+ is optional) and $P$ identifies the operation. When used, a scale factor is added as a prefix to field descriptors.

Examples

-2PF10.5
1PE8.2

When added to an F type field descriptor (or G type if the external field is a fixed point decimal) a scale factor specifies a power of 10 so that

$$\text{External Form of Number} = (\text{Internal Form}) \times 10^{\text{scale factor}}$$

For example, assuming the data involved to be the real number 26.451, the field descriptor

F8.3

produces the external field

$$\text{26.451}$$

Version 4 FORTRAN 13-7 November 1975
Table 13-4
Descriptor Conversion of Real and Double Precision Data
According to Magnitude

<table>
<thead>
<tr>
<th>Magnitude of Data in its External Form (M)</th>
<th>Equivalent Method of Conversion Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0.1 \leq M &lt; 1$</td>
<td>$F(w-4),d,4X$</td>
</tr>
<tr>
<td>$1 \leq M &lt; 10$</td>
<td>$F(w-4),(d-1),4X$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>$10^{d-2} \leq M &lt; 10^{d-1}$</td>
<td>$F(w-4),1,4X$</td>
</tr>
<tr>
<td>$10^{d-1} \leq M &lt; 10^d$</td>
<td>$F(w-4),0,4X$</td>
</tr>
<tr>
<td>ALL OTHERS</td>
<td>$Ew,d$</td>
</tr>
</tbody>
</table>

Note: In all numeric field conversions the field width (w) specified should be large enough to include the decimal point, sign, and exponent character in addition to the number of digits. If the specified width is too small to accommodate the converted number, the field will be filled with asterisks (*). If the number converted occupies fewer character positions than specified by w, it will be right-justified in the field and leading blanks will be used to fill the field.

The addition of the scale factor of $-1P$

$-1PF8.3$

produces the external field

$E02645$

When added to D, E, and G (external field not a decimal fixed point) type field descriptors, the scale factor multiplies the number by the specified power of ten and the exponent is changed accordingly.

In input operations, F type (and G type, if the external field is decimal fixed point) conversions are the only ones affected by scale factors.

When no scale factor is specified, it is understood to be zero. Once a scale factor is specified, however, it holds for all subsequent D, E, F, and G type field descriptors within the same format specification unless another scale factor is specified. A scale factor is reset to zero by specifying a scale factor of zero. Scale factors have no effect on I and O type field descriptors.
13.2.5 Logical Field Descriptors

Logical data may be transferred under format control in a manner similar to numeric data transfer by use of the field descriptor

\[ L \text{w} \]

where \( L \) is the control character and \( w \) is an integer specifying the field width. The data is transmitted as the value of a corresponding logical variable in the associated input/output list.

On input, the first non-blank character in the logical data field must be \( T \) or \( F \), the value of the logical variable is stored in the list variable as true or false, respectively. If the entire input data field is blank or empty, a value of false is stored.

On output, \( w \) minus 1 blanks followed by \( T \) or \( F \) will be output if the value of the logical variable is true or false, respectively.

13.2.6 Variable Numeric Field Widths

Several of the conversion codes are acceptable in FORMAT statements without field width specifications (i.e., the \( \text{w.d} \) portion of the specification is omitted\(^1\)).

On input, the conversion codes \( D, \text{E, F, G, I, L, O} \) are acceptable without field width specifications. The field begins with the first non-blank character encountered and ends with the first illegal character in the given field. (Blanks and tabs also terminate a field.) Note that for conversion code \( L \) (logical data) all consecutive alphabets following a \( T \) (true) or an \( F \) (false) are considered part of the field and are ignored. In succeeding fields the input stream is scanned until a non-blank character is encountered. If the character is a comma (,) the next field is skipped and the following input field begins with the character following the comma. Any character other than a comma is assumed to be the first character in the next input field. Null fields are denoted by successive commas, optionally separated by blanks or tabs. A null field is equivalent to a fixed-field input of blanks. For example, the source code

```fortran
READ 1, X, Y, Z, L, I, J
1 FORMAT (3F, L, I, A3)
```

with data as follows

\[ .10E+5,,TRUE XXX1,ABC \]

results in

- \( X = 0.0 \)
- \( Y = 1.0E+5 \)
- \( Z = 0.0 \)
- \( L = \text{TRUE} \)
- \( I = 1 \)
- \( J = 'ABC' \)

Note that if a comma is included in the input data after the \( XXX1 \) and before the blanks, i.e., the data is

\[ .10E+5,, TRUE XXX1,ABC \]

then \( J = 'ABC' \)

\(^1\)If \( d \) is given, then \( w \) must also be specified.
On output, the format codes A, D, E, F, G, I, L, O, and R are acceptable without field width specifications. The following defaults are assumed:

<table>
<thead>
<tr>
<th>Format Code</th>
<th>Assumed Default for KA10</th>
<th>Assumed Default for K110</th>
</tr>
</thead>
<tbody>
<tr>
<td>A single precision</td>
<td>A5</td>
<td>A5</td>
</tr>
<tr>
<td>A double precision</td>
<td>A10</td>
<td>A10</td>
</tr>
<tr>
<td>D</td>
<td>D25.16</td>
<td>D25.18</td>
</tr>
<tr>
<td>E</td>
<td>E15.7</td>
<td>E15.7</td>
</tr>
<tr>
<td>F</td>
<td>F15.7</td>
<td>F15.7</td>
</tr>
<tr>
<td>G single precision</td>
<td>G15.7</td>
<td>G15.7</td>
</tr>
<tr>
<td>G double precision</td>
<td>G25.16</td>
<td>G25.18</td>
</tr>
<tr>
<td>I</td>
<td>I15</td>
<td>I15</td>
</tr>
<tr>
<td>L</td>
<td>L15</td>
<td>L15</td>
</tr>
<tr>
<td>O</td>
<td>O15</td>
<td>O15</td>
</tr>
<tr>
<td>R single precision</td>
<td>R5</td>
<td>R5</td>
</tr>
<tr>
<td>R double precision</td>
<td>R10</td>
<td>R10</td>
</tr>
</tbody>
</table>

13.2.7 Alphanumeric Field Descriptors

The formatted transfer of alphanumeric data may be accomplished in a manner similar to the formatted transfer of numeric data by use of the field descriptors Aw and Rw, where A and R are the control characters and w is the number of characters in the field.

The A and R descriptors both transfer alphanumeric data into or from a variable in an input/output list depending on the I/O operation. A list variable may be of any type. For example,

```
READ (6,5) V
5 FORMAT (A4)
```

causes four alphanumeric characters to be read from the card reader and stored in the variable V.

The A descriptor deals with variables containing left-justified, blank-filled characters, and the R descriptor deals with variables containing right-justified, zero-filled characters. The following paragraphs summarize the result of alphanumeric data transfer (both internal and external representations) using the A and R descriptors. These paragraphs assume that w represents the field width and m represents the total number of characters possible in the variable. Double precision variables contain 10 characters (i.e., m=10); and all other variables contain 5 (i.e., m=5).

A Descriptor

a. INPUT, where \( w \geq m \) – The rightmost \( m \) characters of the field are read in and stored left-justified and blank-filled in the associated variable.

b. INPUT, where \( w < m \) – All \( w \) characters are read in and stored left-justified and blank-filled in the associated variable.

c. OUTPUT, where \( w \geq m \) – \( m \) characters are output and right-justified in the field. The remainder of the field is blank-filled.

d. OUTPUT, where \( w < m \) – The left-most \( w \) characters of the associated variable are output.
R Descriptor

a. INPUT, where $w \geq m$ — The right-most $m$ characters of the field are read in and stored right-justified, zero-filled in the associated variable.

b. INPUT, where $w < m$ — All $w$ characters are read in and stored right-justified, zero-filled in the associated variable.

c. OUTPUT, where $w \geq m$ — $m$ characters are output and right-justified in the field. The remainder of the field is blank filled.

d. OUTPUT, where $w < m$ — The right-most $w$ characters of the associated variable are output.

13.2.8 Transferring Alphanumeric Data Directly Into or From FORMAT Statements

Alphanumeric data may be transmitted directly into or from the FORMAT statement by two different methods: H-conversion, or the use of single quotes (i.e., a literal field descriptor).

In H-conversion, the alphanumeric string is specified in the form nH, where $H$ is the control character and $n$ is the total number of characters (including blanks) in the string. For example, the following statement sequence may be used to print the words PROGRAM COMPLETE on the device LPT:

```
PRINT 101
101 FORMAT (17H"PROGRAM\$COMPLETE")
```

Read and write operations of this type are initiated by I/O statements which reference a format statement and a logical device but do not contain an I/O list (see preceding example).

Write transfers from a FORMAT statement cause the contents of the statement field descriptor to be output to a specified logical device. The contents of the field descriptor, however, remain unchanged.

Read transfers with a FORMAT statement cause the contents of the field descriptors involved to be replaced by the characters input from the specified logical device.

Alphanumeric data is stored in a field descriptor left justified. If the data input into a field has fewer characters than the field, trailing blanks are added to fill the field. If the data input is larger than the field of the descriptor, the excess right-most characters are lost.

Examples

```
WRITE(1,101)
101 FORMAT (17H"PROGRAM\$COMPLETE")
```

cause the string PROGRAM COMPLETE to be output to the file on device 1.
Assuming the string START on device 1, the sequence

```
READ (1,101)
101  FORMAT (17H$PROGRAM$COMPLETE)
```

would change the contents of statement 101 to

```
101  FORMAT (17H$START$)
```

The foregoing functions may also be accomplished by a literal field descriptor consisting of the desired character string enclosed within apostrophes (i.e., 'string'). For example, the descriptors

```
101  FORMAT (17H$PROGRAM$COMPLETE)
```

and

```
101  FORMAT ('$PROGRAM$COMPLETE')
```

may be used in the same manner.

The result of literal conversion is the same as H-conversion; on input, the characters between the apostrophes are replaced by input characters and, on output, the characters between the apostrophes (including blanks) are written as part of the output data.

An apostrophe character within a literal field should be represented by two successive apostrophe marks; otherwise, the statement containing the field will not compile. For example, the statement sequence

```
50  FORMAT ('DON''T')
PRINT 50
```

will compile and will cause the word DON'T to be output on the line printer. The statement

```
50  FORMAT ('DON'T')
```

however, will cause a compile error.

13.2.9 Mixed Numeric and Alphanumeric Fields
An alphanumeric field descriptor may be placed among other fields of the format. For example, the statement:

```
FORMAT (14,7H$FORCE=F10.5)
```

may be used to output the line:

```
$22$FORCE=$17.68901
```

The separating comma may be omitted after an alphanumeric format field, as shown in the foregoing statement.

When a comma delimiter is omitted from a format specification, format control associates as much information as possible with the leftmost of the two field descriptors.
13.2.10 Multiple Record Specifications

To handle a group of input/output records where different records have different field descriptors, a slash is used to indicate a new record. For example, the statement

```
FORMAT (308/I5,2F8.4)
```

is equivalent to

```
FORMAT (308)
```

for the first record, and

```
FORMAT (I5,2F8.4)
```

for the second record.

Separating commas may be omitted when a slash is used. When \( n \) slashes appear at the end or beginning of a format, \( n \) blank records will be written on output or skipped on input. When \( n \) slashes appear in the middle of a format, \( n-1 \) blank records are written on output or \( n-1 \) records skipped on input.

Both the slash and the closing parenthesis at the end of the format indicate the termination of a record. If the list of an input/output statement dictates that the transmission of data is to continue after the closing parenthesis of the format is reached, the format is repeated starting with that group repeat specification terminated by the last right parenthesis of level one or level zero if no level one group exists.

Thus, the statement

```
FORMAT (F7.2/2(E15.5E15.4),I7))
```

causes the format

```
2(E15.5,E15.4),I7
```

to be used on the first record.

As a further example, consider the statement

```
FORMAT (F7.2)/(2(E15.5,E15.4),I7))
```

The first record has the format

```
F7.2
```

and successive records have the format

```
2(E15.5E15.4),I7
```
13.2.11 Record Formatting Field Descriptors

Two field descriptors, Tw and nX, may be used to position data within a record.

The field descriptor Tw may be used to specify the character position (external form) in which a record begins. In the Tw field descriptor the letter T is the control character and w is an unsigned integer constant which specifies the character position, in a FORTRAN-10 record, where the transfer of data is to begin. When the output is printed, w corresponds to the (w-1)th print position since the first character of the output buffer is a forms control character and is not printed. It is recommended that the first field specification of the output format be IX, except where a forms control character is used.

NOTE
Two successive T field specifications will result in the second field overwriting the first field.

Examples

The statement sequence

    PRINT 2
    2 FORMAT (T50,'BLACK'T30,'WHITE')

causes the following line to be printed

    WHITE   BLACK
    ⬆         ⬆
(print position 29)   (print position 49)

The statement sequence

    1 FORMAT (T35,'MONTH')
    READ (2,1)

causes the first 34 characters of the input data associated with logical unit 2 to be skipped, and the next five characters to replace the characters M,O,N,T, and H in storage. If an input record containing

    ABCbbbXYZ

is read with the format specification

    10 FORMAT (T7,A3,T1,A3)

then the characters XYZ and ABC are read, in that order.

The field descriptor nX may be used to introduce blanks into output records or to skip characters of input records. The letter X specifies the operation and n is a positive integer that specifies the number of character positions to be either made blanks (output) or skipped (input).

Example

The statement

    FORMAT (5H*STEP15,10X2HY=F7.3)

may be used to print the line

    STEP15 28 3.872
13.3 CARRIAGE CONTROL CHARACTERS FOR PRINTING ASCII RECORDS

The first character of an ASCII record may be used to control the spacing operations of the line printer or Teletype terminal printer unit on which the record is being printed. The control character desired is specified by beginning the FORMAT field specification for the ASCII record to be output with 1Ha... where a is the desired control character. The control characters permitted in FORTRAN-10 and the effect each has on the printing device are described in Table 13-5.

<table>
<thead>
<tr>
<th>FORTRAN Character</th>
<th>Printer Character</th>
<th>Octal Value</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>space</td>
<td>LF</td>
<td>012</td>
<td>Skip to next line with form feed after 60 lines</td>
</tr>
<tr>
<td>0 zero</td>
<td>LF,LF</td>
<td>012</td>
<td>Skip a line</td>
</tr>
<tr>
<td>1 one</td>
<td>FF</td>
<td>014</td>
<td>Form feed – go to top of next page</td>
</tr>
<tr>
<td>+ plus</td>
<td></td>
<td></td>
<td>Suppress skipping – overprint the line</td>
</tr>
<tr>
<td>* asterisk</td>
<td>DC3</td>
<td>023</td>
<td>Skip to next line with no form feed</td>
</tr>
<tr>
<td>- minus</td>
<td>LF,LF,LF</td>
<td>012</td>
<td>Skip two lines</td>
</tr>
<tr>
<td>2 two</td>
<td>DLE</td>
<td>020</td>
<td>Space 1/2 of a page</td>
</tr>
<tr>
<td>3 three</td>
<td>VT</td>
<td>013</td>
<td>Space 1/3 of a page</td>
</tr>
<tr>
<td>/ slash</td>
<td>DC4</td>
<td>024</td>
<td>Space 1/6 of a page</td>
</tr>
<tr>
<td>. period</td>
<td>DC2</td>
<td>022</td>
<td>Triple space with a form feed after every 20 lines printed</td>
</tr>
<tr>
<td>, comma</td>
<td>DC1</td>
<td>021</td>
<td>Double space with a form feed after every 30 lines printed</td>
</tr>
</tbody>
</table>

Note: Printer control characters DLE, DC1, DC2, DC3, and DC4 affect only the line printer.
CHAPTER 14

DEVICE CONTROL STATEMENTS

14.1 INTRODUCTION

The following device control statements may be used in FORTRAN-10 source programs:

1. REWIND
2. UNLOAD
3. BACKSPACE
4. ENDFILE
5. SKIPRECORD
6. SKIPFILE, and
7. BACKFILE

The general form of the foregoing device control statements is

keyword u
keyword (u)

where

keyword is the statement name
u is the FORTRAN-10 logical device number (Chapter 10, Table 10-1)

The operations performed by the device control statement are normally used only for magnetic tape device (MTA). In FORTRAN-10, however, the device control operations are simulated for disk devices.

---

1. The results of these commands are unpredictable when used on list-directed and NAMELIST-controlled data.
14.2 REWIND STATEMENT

Descriptions of the form and use of the REWIND statement follow:

Form: REWIND u

Use: Move the file contained by device u to its initial (load) point. If the medium is already at its load point, this statement has no effect. Subsequent READ or WRITE statements that reference device u will transfer data to or from the first record located on the medium mounted on device u.

Example: REWIND 16

14.3 UNLOAD STATEMENT

Descriptions of the form and use of the UNLOAD statement follow:

Form: UNLOAD u

Use: Move the medium contained on device u past its load point until it has been completely rewound onto the source reel.

Example: UNLOAD 16

14.4 BACKSPACE STATEMENT

Descriptions of the form and use of the BACKSPACE statement follow:

Form: BACKSPACE u

Use: Move the medium contained on device u to the start of the record that precedes the current record. If the preceding record prior to execution of this statement was an endfile record, the endfile record becomes the next record after execution. If the current record is the first record of the file, this statement has no effect.

NOTE
This statement cannot be used for files set up for random access or NAMELIST-controlled I/O operations.

Example: BACKSPACE 16

14.5 END FILE STATEMENT

Descriptions of the form and use of the END FILE statement follow:

Form: END FILE u

Use: Write an endfile record in the file located on device u. The endfile record defines the end of the file which contains it. If an endfile record is reached during an I/O operation initiated by a statement that does not contain an END= option, the operation of the current program is terminated.

Example: END FILE 16
14.6 SKIP RECORD STATEMENT

Descriptions of the form and use of the SKIP RECORD statement follow:

Form: \text{SKIP RECORD } u

Use: In accessing the file located on device \( u \), skip the record immediately following the current (last accessed) record. The repeat option may be used to cause any desired number of records to be skipped.

Example: \text{SKIP RECORD 16}

14.7 SKIP FILE STATEMENT

Descriptions of the form and use of the SKIP FILE statement follow:

Form: \text{SKIP FILE } u

Use: In accessing the medium located on unit \( u \), skip the file immediately following the current (last accessed) file. If the number of SKIP FILE operations specified exceeds the number of following files available, an error will occur.

Example: \text{SKIP FILE 01}

14.8 BACKFILE STATEMENT

Descriptions of the form and use of the BACKFILE statement follow:

Form: \text{BACKFILE } u

Use: Move the medium mounted on device \( u \) to the start of the file which precedes the current (last accessed) file.

If the number of BACKFILE operations performed exceeds the number of preceding files, completion of the last operation will move the medium to the start of the first file on the medium.

Example: \text{BACKFILE 20}

14.9 SUMMARY OF DEVICE CONTROL STATEMENTS

The form and use of the FORTRAN-10 device control statements are summarized in Table 14-1.

Table 14-1

<table>
<thead>
<tr>
<th>Statement Form</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>REWIND ( u )</td>
<td>Rewind medium to its load point</td>
</tr>
<tr>
<td>UNLOAD ( u )</td>
<td>Rewind medium onto its source reel</td>
</tr>
<tr>
<td>END FILE ( u )</td>
<td>Write an endfile record in to the current file</td>
</tr>
<tr>
<td>SKIP RECORD ( u )</td>
<td>Skip the next record</td>
</tr>
<tr>
<td>SKIP FILE ( u )</td>
<td>Skip the next file</td>
</tr>
<tr>
<td>BACKFILE ( u )</td>
<td>Move medium backwards 1 file</td>
</tr>
<tr>
<td>BACKSPACE ( u )</td>
<td>Move medium back one record</td>
</tr>
</tbody>
</table>
CHAPTER 15

SUBPROGRAM STATEMENTS

15.1 INTRODUCTION

Procedures that are used repeatedly by a program may be written once and then referenced each time the procedure is required. Procedures that may be referenced are either internal (written and contained within the program in which they are referenced) or external (self-contained executable procedures that may be compiled separately). The kinds of FORTRAN-10 procedures that may be referenced are:

a. statement functions

b. intrinsic functions (FORTRAN-10 defined functions)

c. external functions, and

d. subroutines

The first three of the foregoing categories are referred to, collectively, as either functions or function procedures; procedures of the last category are referred to as either subroutines or subroutine procedures.

15.1.1 Dummy and Actual Arguments

Since subprograms may be referenced at more than one point throughout a program, many of the values used by the subprogram may be changed each time it is used. Dummy arguments in subprograms represent the actual values to be used which are passed to the subprogram when it is called.

Functions and subroutines use dummy arguments to indicate the type of the actual arguments which they represent and whether the actual arguments are variables, array elements, arrays, subroutine names or the names of external functions. Each dummy argument must be used within a function or subroutine as if it were a variable, array, array element, subroutine, or external function identifier. Dummy arguments are given in an argument list associated with the identifier assigned to the subprogram; actual arguments are normally given in an argument list associated with a call made to the desired subprogram. (Examples of argument lists are given in the following paragraphs.)

The position, number, and type of each dummy argument in a subprogram list must agree with the position, number, and type of each actual argument given in the argument list of the subprogram reference.
Dummy arguments may be

a. variables

b. array names

c. subroutine identifiers

d. function identifiers, or

e. *statement label identifiers which are denoted by the symbol *, $, or &.*

When a subprogram is referenced, its dummy arguments are replaced by the corresponding actual arguments supplied 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 results of the subprogram computations will be unpredictable. Argument association may be carried through more than one level of subprogram reference if a valid association is maintained through each level. The dummy/actual argument associations established when a subprogram is referenced are terminated when the desired subprogram operations are completed.

The following rules govern the use and form of dummy arguments:

a. 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 the procedure is referenced.

b. Dummy argument names may not appear in EQUIVALENCE, DATA, or COMMON statements.

c. A variable dummy argument should have a variable, an array element identifier, an expression, or a constant as its corresponding actual argument.

d. 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. Each element of a dummy array is associated directly with the corresponding elements of the actual array.

e. A dummy argument representing a subroutine identifier should have a subroutine name as its actual argument.

f. A dummy argument representing an external function must have an external function as its actual argument.

g. A dummy argument may be defined or redefined in a referenced subprogram only if its corresponding actual argument is a variable. If dummy arguments are array names, then elements of the array may be redefined.

Additional information regarding the use of dummy and actual arguments is given in the description of how subprograms are defined and referenced.
15.2 STATEMENT FUNCTIONS

Statement functions define an internal subprogram in a single statement. The general form of a statement function is:

\[ \text{name}(\text{arg1, arg2, ..., argn}) = E \]

where

- \text{name} is a user-formulated name comprised of from 1 to 6 characters. The name used must conform to the rules for symbolic names given in Paragraph 3.3.
- The type of a statement function is determined either by the first character of its name or by being declared in an explicit or implicit type statement.
- \text{(arg1, ..., argn)} represents a list of dummy arguments.
- \( E \) is an arbitrary expression.

The expression \( E \) of a statement function may be any legitimate arithmetic expression which uses the given dummy arguments and indicates how they are combined to obtain the desired value. The dummy arguments may be used as variables or indirect function references; but they cannot be used as arrays. The dummy argument names bear no relation to their use outside the context of the statement function except for their data type. The expression may reference FORTRAN-10 defined functions (Paragraph 15.3) or any other defined statement function, or call an external function. It may not reference any function that directly or indirectly references the given statement function or any subprogram in the chain of references. That is, recursive references are not allowed. Statement functions produce only one value, the result of the expression which it contains. A statement function cannot reference itself.

All statement functions within a program unit must be defined before the first executable statement of the program unit. When used, the statement function name must be followed by an actual argument list enclosed within parentheses and may appear in any arithmetic or logical expression.

Examples

\[ \text{SSQR}(K) = (K^2 + K + 1)/6 \]
\[ \text{ACOSH}(X) = (\exp(X/A) + \exp(-X/A))/2.0 \]

15.3 INTRINSIC FUNCTIONS (FORTRAN-10 DEFINED FUNCTIONS)

Intrinsic functions are subprograms that are defined and supplied by FORTRAN-10. An intrinsic function is referenced by using its assigned name as an operand in an arithmetic or logical expression. The names of the FORTRAN-10 intrinsic functions, the type of the arguments which each accepts, and the function it performs are described in Table 15-1. These names always refer to the intrinsic function unless they are preceded by an asterisk (*) or ampersand (&) in an EXTERNAL statement, declared in a conflicting explicit type statement, or are specified as a routine dummy parameter.
### Table 15-1

<table>
<thead>
<tr>
<th>Function</th>
<th>Mnemonic</th>
<th>Definition</th>
<th>Number of Arguments</th>
<th>Type of Argument</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Absolute value:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Real</td>
<td>ABS*</td>
<td>arg</td>
<td>1</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>Integer</td>
<td>IABS*</td>
<td>arg</td>
<td>1</td>
<td>Integer</td>
<td>Integer</td>
</tr>
<tr>
<td>Double precision</td>
<td>DABS*</td>
<td>arg</td>
<td>1</td>
<td>Double</td>
<td>Double</td>
</tr>
<tr>
<td>Complex to real</td>
<td>CABS</td>
<td>$c=(x^2+y^2)^{1/2}$</td>
<td>1</td>
<td>Complex</td>
<td>Real</td>
</tr>
<tr>
<td><strong>Conversion:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Integer to real</td>
<td>FLOAT**</td>
<td></td>
<td>1</td>
<td>Integer</td>
<td>Real</td>
</tr>
<tr>
<td>Real to integer</td>
<td>IFIX**</td>
<td>Sign of arg *</td>
<td>1</td>
<td>Real</td>
<td>Integer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>largest integer $&lt;</td>
<td>arg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double to real</td>
<td>SNGL</td>
<td></td>
<td>1</td>
<td>Double</td>
<td>Real</td>
</tr>
<tr>
<td>Real to double</td>
<td>DBLE*</td>
<td></td>
<td>1</td>
<td>Real</td>
<td>Double</td>
</tr>
<tr>
<td>Integer to double</td>
<td>D.FLOAT</td>
<td></td>
<td>1</td>
<td>Integer</td>
<td>Double</td>
</tr>
<tr>
<td>Complex to real (obtain real part)</td>
<td>REAL*</td>
<td></td>
<td>1</td>
<td>Complex</td>
<td>Real</td>
</tr>
<tr>
<td>Complex to real (obtain imaginary part)</td>
<td>AIMAG</td>
<td></td>
<td>1</td>
<td>Complex</td>
<td>Real</td>
</tr>
<tr>
<td>Real to complex</td>
<td>CMPLX*</td>
<td>$c=\text{Arg}_1+i\text{Arg}_2$</td>
<td>2</td>
<td>Real</td>
<td>Complex</td>
</tr>
<tr>
<td><strong>Truncation:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Real to real</td>
<td>AINT</td>
<td>Sign of arg *</td>
<td>1</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td></td>
<td></td>
<td>largest integer $\leq</td>
<td>arg</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Real to integer</td>
<td>INT*</td>
<td></td>
<td>1</td>
<td>Real</td>
<td>Integer</td>
</tr>
<tr>
<td>Double to integer</td>
<td>IDINT</td>
<td></td>
<td>1</td>
<td>Double</td>
<td>Integer</td>
</tr>
<tr>
<td><strong>Remaindering:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Real</td>
<td>AMOD</td>
<td>The remainder when Arg 1 is divided by Arg 2</td>
<td>2</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>Integer</td>
<td>MOD*</td>
<td></td>
<td>2</td>
<td>Integer</td>
<td>Integer</td>
</tr>
<tr>
<td>Double precision</td>
<td>DMOD</td>
<td></td>
<td>2</td>
<td>Double</td>
<td>Double</td>
</tr>
<tr>
<td><strong>Maximum value:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AMAX0</td>
<td></td>
<td>$\geq$ 2</td>
<td>Integer</td>
<td>Real</td>
</tr>
<tr>
<td></td>
<td>AMAX1*</td>
<td></td>
<td>$\geq$ 2</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td></td>
<td>MAX0*</td>
<td>Max(Arg$_1$,Arg$_2$,...)</td>
<td>$\geq$ 2</td>
<td>Integer</td>
<td>Integer</td>
</tr>
<tr>
<td></td>
<td>MAX1</td>
<td></td>
<td>$\geq$ 2</td>
<td>Real</td>
<td>Integer</td>
</tr>
<tr>
<td></td>
<td>DMA1</td>
<td></td>
<td>$\geq$ 2</td>
<td>Double</td>
<td>Double</td>
</tr>
<tr>
<td><strong>Minimum Value:</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>AMIN0</td>
<td></td>
<td>$\geq$ 2</td>
<td>Integer</td>
<td>Real</td>
</tr>
<tr>
<td></td>
<td>AMIN1*</td>
<td></td>
<td>$\geq$ 2</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td></td>
<td>MIN0*</td>
<td>Min(Arg$_1$,Arg$_2$,...)</td>
<td>$\geq$ 2</td>
<td>Integer</td>
<td>Integer</td>
</tr>
<tr>
<td></td>
<td>MIN1</td>
<td></td>
<td>$\geq$ 2</td>
<td>Real</td>
<td>Integer</td>
</tr>
<tr>
<td></td>
<td>DMIN1</td>
<td></td>
<td>$\geq$ 2</td>
<td>Double</td>
<td>Double</td>
</tr>
</tbody>
</table>

*In line functions.

**In line functions on K110 only.
15.4 EXTERNAL FUNCTIONS

External functions are function subprograms that consist of a FUNCTION statement followed by a sequence of FORTRAN-10 statements that define one or more desired operations; subprograms of this type may contain one or more RETURN statements and must be terminated by an END statement. Function subprograms are independent programs that may be referenced by other programs.

The FUNCTION statement that identifies an external function has the form

    type FUNCTION name (arg1, arg2, ..., argn)

where

- **type** is an optional type specification as described in section 6.3. These include INTEGER, REAL, DOUBLE PRECISION, COMPLEX or LOGICAL (plus the optional size modifier, *n, for compatibility with other manufacturers.)

- **name** is the name assigned to the function. The name may consist of from 1 to 6 characters, the first of which must be alphabetic. The optional size modifier (*n) may be included with the name if the type is specified. (Refer to section 6.3.)

- **(arg1, arg2, ..., argn)** is a list of dummy arguments.

If type is not given in the FUNCTION statement, the type of the function may be assigned, by default, according to the first character of its name, or may be specified by an IMPLICIT statement or by an explicit statement given within the subprogram itself.

Note that if a user wants to use the same name for a user-defined function as the name of a FORTRAN-10 defined function (library basic external function), the desired name must be declared in an EXTERNAL statement and prefixed by an asterisk (*) or ampersand (&) in the referencing routine. (Refer to section 6.7 for a description of the EXTERNAL statement.)
The following rules govern the structuring of a FUNCTION subprogram:

a. The symbolic name assigned a FUNCTION subprogram must also be used as a variable name in the subprogram. During each execution of the subprogram this variable must be defined and, once defined, may be referenced as redefined. The value of the variable at the time of execution on any RETURN statement is the value of the subprogram.

NOTE
A RETURN statement returns control to the calling statement that initiated the execution of the subprogram. See Paragraph 15.4.1 for a description of this statement.

b. The symbolic name of a FUNCTION subprogram must not be used in any nonexecutable statement in the subprogram except in the initial FUNCTION statement or a type statement.

c. Dummy argument names may not appear in any EQUIVALENCE, COMMON, or DATA statement used within the subprogram.

d. The function subprogram may define or redefine one or more of its arguments so as to effectively return results in addition to the value of the function.

e. The function subprogram may contain any FORTRAN-10 statement except BLOCK DATA, SUBROUTINE PROGRAM, another FUNCTION statement or any statement that directly or indirectly references the function being defined or any subprogram in the chain of subprograms leading to this function.

f. The function subprogram should contain at least one RETURN statement and must be terminated by an END statement. The RETURN statement signifies a logical conclusion of the computation made by the subprogram and returns the computed function value and control to the calling program. A subprogram may have more than one RETURN statement.

The END statement specifies the physical end of the subprogram and implies a return.

15.4.1 Basic External Functions (FORTRAN-10 Defined Functions)
FORTRAN-10 contains a group of predefined external functions which are referred to as basic functions. Table 15-2 describes each basic function, its name, and its use. These names always refer to the basic external functions unless declared in an EXTERNAL or conflicting explicit type statement.

15.4.2 Generic Function Names
The compiler generates a call to the proper FORTRAN-10 defined function, depending on the type of the arguments, for the following generic function names:
CHAPTER 15

ABS
AMAX1
AMIN1
ATAN
ATAN2
COS
INT
MOD
SIGN
SIN
SORT
EXP
ALOG
ALOG10

In the following example

\[ K = \text{ABS}(I) \]

the type of I determines which function is called. If I is an integer, the compiler generates a call to the function IABS. If I is real, the compiler generates a call to the function ABS. If I is double precision, the compiler generates a call to the function DABS.

The function name loses its generic properties if it appears in an explicit type statement, if it is specified as a dummy routine parameter, or if it is prefixed by "*" or "&" in an EXTERNAL statement. When a generic function name, which was specified unprefixed in an EXTERNAL statement, is used as a routine parameter, it is assumed to reference a FORTRAN-10 defined function of the same name, or if none exist, a user-defined function. Note that IMPLICIT statements have no effect upon the data type of generic function names unless the name has been removed from its class using an EXTERNAL statement.

15.5 SUBROUTINE SUBPROGRAMS

A subroutine is an external computational procedure which is identified by a SUBROUTINE statement and may or may not return values to the calling program. The SUBROUTINE statement used to identify a subprogram of this type has the form:

\[
\text{SUBROUTINE name}(\text{arg1, arg2, \ldots, argn})
\]

where

\[ \text{name} \]

is the symbolic name of the subroutine to be defined.

\[ (\text{arg1, \ldots, argn}) \]

is an optional list of dummy arguments.
### Table 15-2
Basic External Functions (FORTRAN-10 Defined Functions)

<table>
<thead>
<tr>
<th>Function</th>
<th>Mnemonic</th>
<th>Definition</th>
<th>Number of Arguments</th>
<th>Type of Argument</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponential:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Real</td>
<td>EXP</td>
<td>$e^{\text{Arg}}$</td>
<td>1</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>Double</td>
<td>DEXP</td>
<td></td>
<td>1</td>
<td>Double</td>
<td>Double</td>
</tr>
<tr>
<td>Complex</td>
<td>CEXP</td>
<td></td>
<td>1</td>
<td>Complex</td>
<td>Complex</td>
</tr>
<tr>
<td>Logarithm:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Real</td>
<td>ALOG</td>
<td>$\log_{e}(\text{Arg})$</td>
<td>1</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td></td>
<td>ALOG10</td>
<td>$\log_{10}(\text{Arg})$</td>
<td>1</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>Double</td>
<td>DLOG</td>
<td>$\log_{e}(\text{Arg})$</td>
<td>1</td>
<td>Double</td>
<td>Double</td>
</tr>
<tr>
<td></td>
<td>DLOG10</td>
<td>$\log_{10}(\text{Arg})$</td>
<td>1</td>
<td>Double</td>
<td>Double</td>
</tr>
<tr>
<td>Complex</td>
<td>CLOG</td>
<td>$\log_{e}(\text{Arg})$</td>
<td>1</td>
<td>Complex</td>
<td>Complex</td>
</tr>
<tr>
<td>Square Root:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Real</td>
<td>SQRT*</td>
<td>$(\text{Arg})^{1/2}$</td>
<td>1</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>Double</td>
<td>DSQRT</td>
<td>$(\text{Arg})^{1/2}$</td>
<td>1</td>
<td>Double</td>
<td>Double</td>
</tr>
<tr>
<td>Complex</td>
<td>CSQRT</td>
<td>$(\text{Arg})^{1/2}$</td>
<td>1</td>
<td>Complex</td>
<td>Complex</td>
</tr>
<tr>
<td>Sine:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Real (radians)</td>
<td>SIN*</td>
<td>$\sin(\text{Arg})$</td>
<td>1</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>Real (degrees)</td>
<td>SIND</td>
<td></td>
<td>1</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>Double (radians)</td>
<td>DSN</td>
<td>$\sin(\text{Arg})$</td>
<td>1</td>
<td>Double</td>
<td>Double</td>
</tr>
<tr>
<td>Complex</td>
<td>CSN</td>
<td></td>
<td>1</td>
<td>Complex</td>
<td>Complex</td>
</tr>
<tr>
<td>Cosine:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Real (radians)</td>
<td>COS*</td>
<td>$\cos(\text{Arg})$</td>
<td>1</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>Real (degrees)</td>
<td>COSD</td>
<td></td>
<td>1</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>Double (radians)</td>
<td>DCOS</td>
<td>$\cos(\text{Arg})$</td>
<td>1</td>
<td>Double</td>
<td>Double</td>
</tr>
<tr>
<td>Complex</td>
<td>CCOS</td>
<td></td>
<td>1</td>
<td>Complex</td>
<td>Complex</td>
</tr>
<tr>
<td>Hyperbolic:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sine</td>
<td>SINH</td>
<td>$\sinh(\text{Arg})$</td>
<td>1</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>Cosine</td>
<td>COSH</td>
<td>$\cosh(\text{Arg})$</td>
<td>1</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>Tangent</td>
<td>TANH</td>
<td>$\tanh(\text{Arg})$</td>
<td>1</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>Arc sine</td>
<td>ASIN</td>
<td>$\text{asin}(\text{Arg})$</td>
<td>1</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>Arc cosine</td>
<td>ACOS</td>
<td>$\text{acos}(\text{Arg})$</td>
<td>1</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>Arc tangent</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Real</td>
<td>ATAN*</td>
<td>atan(\text{Arg})</td>
<td>1</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>Double</td>
<td>DATAN</td>
<td>atan(\text{Arg})</td>
<td>1</td>
<td>Double</td>
<td>Double</td>
</tr>
<tr>
<td>Two REAL arguments</td>
<td>ATAN2*</td>
<td>atan(\text{Arg1}/\text{Arg2})</td>
<td>2</td>
<td>Real</td>
<td>Real</td>
</tr>
<tr>
<td>Two DOUBLE arguments</td>
<td>DATAN2</td>
<td>atan(\text{Arg1}/\text{Arg2})</td>
<td>2</td>
<td>Double</td>
<td>Double</td>
</tr>
</tbody>
</table>

*Generic Functions*
Table 15-2 (Cont)
Basic External Functions (FORTRAN-10 Defined Functions)

<table>
<thead>
<tr>
<th>Function</th>
<th>Mnemonic</th>
<th>Definition</th>
<th>Number of Arguments</th>
<th>Type of</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complex Conjugate</td>
<td>CONJG</td>
<td>Arg=X+iY, CONJG=X-iY</td>
<td>1</td>
<td>Complex</td>
</tr>
<tr>
<td>Random Number</td>
<td>RAN</td>
<td>Result is a random number in the range of 0 to 1.0.</td>
<td>1 Dummy Argument</td>
<td>Real</td>
</tr>
</tbody>
</table>

The following rules control the structuring of a subroutine subprogram:

a. The symbolic name of the subprogram must not appear in any statement within the defined subprogram except the SUBROUTINE statement itself.

b. The given dummy arguments may not appear in an EQUIVALENCE, COMMON, or DATA statement within the subprogram.

c. The subroutine subprogram may define or redefine one or more of its arguments so as to effectively return results.

d. The subroutine subprogram may contain any FORTRAN-10 statement except BLOCK DATA, FUNCTION, another SUBROUTINE statement, or any statement that either directly or indirectly references the subroutine being defined or any of the subprograms in the chain of subprogram references leading to this subroutine.

e. Dummy arguments that represent statement labels may be either an *, $, or &.

f. The subprogram should contain at least one RETURN statement and must be terminated by an END statement. The RETURN statements indicate the logical end of a computational routine; the END statement signifies the physical end of the subroutine.

g. Subroutine subprograms may have as many entry points as desired (see description of ENTRY statement given in Paragraph 15.4.1).

15.5.1 Referencing Subroutines (CALL Statement)
Subroutine subprograms must be referenced using a CALL statement of the following form:

```
CALL name(arg1, arg2, ..., argn)
```

where

*name* is the symbolic name of the desired subroutine subprogram.

*(arg1, ..., argn)* is an optional list of actual arguments. If the list is included, the given actual arguments must agree in order, number, and type with the corresponding dummy arguments given in the defining SUBROUTINE statement.
The use of literal constants is an exception to the rule requiring agreement of type between dummy and actual arguments. An actual argument in a CALL statement may be:

a. a constant
b. a variable name
c. an array element identifier
d. an array name
e. an expression
f. the name of an external subroutine, or
g. a statement label.

Example
The subroutine

    SUBROUTINE MATRIX(I,J,K,M,*)
    ...
    END

may be referenced by

    CALL MATRIX(10,20,30,40,$101)

15.5.2 FORTRAN-10 Supplied Subroutines
FORTRAN-10 provides the user with an extensive group of predefined subroutines. The descriptions and names of these predefined subroutines are given in Table 15-3.

15.6 RETURN STATEMENT AND MULTIPLE RETURNS
The RETURN statement causes control to be returned from a subprogram to the calling program unit. This statement has the form

    RETURN (standard return)

or

    RETURN e    (multiple returns)

where e represents an integer constant, variable, or expression. The execution of this statement in the first of the foregoing forms (i.e., standard return) causes control to be returned to the statement of the calling program which follows the statement that called the subprogram.
The multiple returns form of this statement (i.e., RETURN e) enables the user to select any labeled statement of the calling program as a return point. When the multiple returns form of this statement is executed, the assigned or calculated value of e specifies that the return is to be made to the eth statement label in the argument list of the calling statement. The value of e should be a positive integer which is equal to or less than the number of statement labels given in the argument list of the calling statement. If e is less than 1 or is larger than the number of available statement labels, a standard return operation is performed.

NOTE
A dummy argument for a statement label must be either a *, $, or & symbol.

Any number of RETURN (standard return) statements may be used in any subprogram. The use of the multiple returns form of the RETURN statement, however, is restricted to SUBROUTINE subprograms. The execution of a RETURN statement in a main program will terminate the program.

Example
Assume the following statement sequence in a main program:

CALL EXAMP(1,$10,K,$15,M,$20)
GO TO 101

10 ..............

15 ..............

20 ..............

Assume the following statement sequence in the called SUBROUTINE subprogram:

SUBROUTINE EXAMP (L,*,M,*,N,*)

RETURN

RETURN

RETURN(C/D)

END

15-11
Each occurrence of RETURN returns control to the statement GO TO 101 in the calling program.

If, on the execution of the \texttt{RETURN(C/D)} statement, the value of \((C/D)\) is:

<table>
<thead>
<tr>
<th>Less than or equal to:</th>
<th>The following is performed:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\textit{a standard return to the GO TO 101 statement is made}</td>
</tr>
<tr>
<td>1</td>
<td>\textit{the return is made to statement 10}</td>
</tr>
<tr>
<td>2</td>
<td>\textit{the return is made to statement 15}</td>
</tr>
<tr>
<td>3</td>
<td>\textit{the return is made to statement 20}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Greater than or equal to:</th>
<th>The following is performed:</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>\textit{a standard return to the GO TO 101 statement is made}.</td>
</tr>
</tbody>
</table>

15.6.1 Referencing External \texttt{FUNCTION} Subprograms

An external function subprogram is referenced by using its assigned name as an operand in an arithmetic or logical expression in the calling program unit. The name must be followed by an actual argument list. The actual arguments in an external function reference may be:

a. a variable name

b. an array element identifier

c. an array name

d. an expression

e. a statement number

f. the name of another external procedure (\texttt{FUNCTION} or \texttt{SUBROUTINE}).

\textbf{NOTE}

Any subprogram name to be used as an argument to another subprogram must first appear in an \texttt{EXTERNAL} statement (Chapter 6) in the calling program unit.

Example

The subprogram defined as:

\begin{verbatim}
INTEGER FUNCTION ICALC(X,Y,Z)
  .
  .
  RETURN
END
\end{verbatim}

may be referenced in the following manner:

\begin{verbatim}
  .
  .
  TOTAL = ICALC(IAA,IB,1AC)+500
\end{verbatim}
15.7 MULTIPLE SUBPROGRAM ENTRY POINTS (ENTRY STATEMENT)

FORTRAN-10 provides an ENTRY statement which enables the user to specify additional entry points into an external subprogram. This statement used in conjunction with a RETURN statement enables the user to employ only one computational routine of a subprogram which contains several such routines. The form of the ENTRY statement is:

\[ \text{ENTRY name}(\text{arg}1, \text{arg}2, \ldots, \text{arg}n) \]

where

- \text{name} is the symbolic name to be assigned the desired entry point
- \( (\text{arg}1, \ldots, \text{arg}n) \) is an optional list of dummy arguments. This list may contain
  a. variable names
  b. array declarators
  c. the name of an external procedure (SUBROUTINE or FUNCTION), or
  d. an address constant denoted by either a ",", ",", or "symbol"

The rules for the use of an ENTRY statement follow.

a. The ENTRY statement allows entry into a subprogram at a place other than that defined by the subroutine or function statement. Any number of ENTRY statements may be included in an external subprogram.

b. Execution is begun at the first executable statement following the ENTRY statement.

b. Appearance of an ENTRY statement in a subprogram does not preclude the rule that statement functions in subprograms must precede the first executable statement.

d. Entry statements are nonexecutable and do not affect the execution flow of a subprogram.

e. An ENTRY statement may not appear in a main program, nor may a subprogram reference itself through its entry points.

f. An ENTRY statement may not appear in the range of a DO or an extended DO statement construction.

g. The dummy arguments in the ENTRY statement need not agree in order, number, or type with the dummy arguments in SUBROUTINE or FUNCTION statements or any other ENTRY statement in the subprogram. However, the arguments for each call or function reference must agree with the dummy arguments in the SUBROUTINE, FUNCTION, or ENTRY statement that is referenced.

h. Entry into a subprogram initializes the dummy arguments of the referenced ENTRY statement, all appearances of these arguments in the entire subprogram are initialized.

i. A dummy argument may not be referenced unless it appears in the dummy list of an ENTRY, SUBROUTINE, or FUNCTION statement by which the subprogram is entered.
j. The source subprogram must be ordered such that references to dummy arguments in executable statements must follow the appearance of the dummy argument in the dummy list of a SUBROUTINE, FUNCTION, or ENTRY statement.

k. Dummy arguments that were defined for a subprogram by some previous reference to the subprogram are undefined for subsequent entry into the subprogram.

l. The value of the function must be returned by using the current entry name.
## Table 15-3
### FORTRAN-10 Library Subroutines

<table>
<thead>
<tr>
<th>Subroutine Name</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXIS</td>
<td>CALL AXIS(X,Y,ASC,NASC,S,THETA,XMIN,DX)</td>
</tr>
</tbody>
</table>

Causes an axis with tic marks and scale values at 1-inch increments to be drawn. An identifying label may also be plotted along the axis. Parameters X and Y specify the start of the axis. The axis is plotted, starting at X, Y, at an angle of THETA degrees for a distance of 8 inches. The angle THETA is usually either 0 (X axis) or 90.0 (Y axis). Characters NASC of array ASC are plotted as a label for the axis drawn. If NASC is positive, the tic marks, label, and scale values are placed to the counterclockwise side of the axis; if NASC is negative, the foregoing items are placed to the clockwise side of the axis.

Parameter XMIN is the value of the scale at the beginning of the axis; parameter DX is the change in scale for a 1-inch increment. The values of XMIN and DX may be determined by subroutine SCALE.

### DATE

Places today’s date as left-justified ASCII characters into a dimensioned 2-word array.

CALL DATE (array)

where array is the 2-word array. The date is in the form

```
dd-mmm-yy
```

where dd is a 2-digit day (if the first digit is 0, it is converted to a blank), mmm is a 3-digit month (e.g., Mar), and yy is a 2-digit year. The data is stored in ASCII code, left-justified, in the two words.

A DEFINE FILE call can be used to establish and define the structure of each file to be used for random access I/O operations.

### NOTE

The OPEN statement may be used to perform the same functions as DEFINE FILE.

The format of a DEFINE FILE call may be

CALL DEFINE FILE (u,s,v,f,pj,pg)

where
Table 15-3 (Cont)
FORTRAN-10 Library Subroutines

<table>
<thead>
<tr>
<th>Subroutine Name</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFINE FILE</td>
<td>( u = ) logical FORTRAN-10 device numbers.</td>
</tr>
<tr>
<td>(cont)</td>
<td>( s = ) the size of the records which comprise the file being defined. The argument ( s ) may be an integer constant or variable.</td>
</tr>
<tr>
<td></td>
<td>( v = ) an associated variable. The associated variable is an integer variable that is set to a value that points to the record that immediately follows the last record transferred. This variable is used by the FIND statement (Chapter 10). At the end of each FIND operation the variable is set to a value that points to the record found. The variable ( v ) cannot appear in the I/O list of any I/O statement that accesses the file set up by the DEFINE FILE statement.</td>
</tr>
<tr>
<td></td>
<td>( f = ) filename to be given the file being defined.</td>
</tr>
<tr>
<td></td>
<td>( pj = ) user’s project number.</td>
</tr>
<tr>
<td></td>
<td>( pg = ) user’s programmer’s number.</td>
</tr>
<tr>
<td></td>
<td>NOTE</td>
</tr>
<tr>
<td></td>
<td>Numbers ( pj ) and ( pg ) identify the User’s File Directory.</td>
</tr>
</tbody>
</table>

Example

The statement

```
CALL DEFINE FILE (1,10,ASCVAR, ‘FORTFL.DAT’,0,0)
```

establishes a file named FORTFL.DAT on device 01 (i.e., disk) which contains word records. The associated variable is ASCVAR, and the file is in the user’s area.

Causes particular portions of core to be dumped and is referred to in the following form:

```
CALL DUMP (L_1, U_1, F_1, \ldots, L_n, U_n, F_n)
```

where \( L_1 \) and \( U_1 \) are the variable names which give the limits of core memory to be dumped. Either \( L_1 \) or \( U_1 \) may be upper or lower limits. \( F_1 \) is a number indicating the format in which the dump is to be performed: \( 0 = \) octal, \( 1 = \) real, \( 2 = \) integer, and \( 3 = \) ASCII.

If \( F_1 \) is not 0, 1, 2, 3, the dump is in octal. If \( F_n \) is missing, the last section is dumped in octal. If \( U_n \) and \( F_n \) are missing, an octal dump is made from \( L \) to the end of the job area. If \( L_n \), \( U_n \), and \( F_n \) are missing, the entire job area is dumped in octal.

The dump is terminated by a call to EXIT.

15-16

Version 4 FORTRAN

November 1975
### Table 15-3 (Cont)

#### FORTRAN-10 Library Subroutines

<table>
<thead>
<tr>
<th>Subroutine Name</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERRSET</td>
<td>Allows the user to control the timeout of execution-time arithmetic error messages. ERRSET is called with one argument in integer mode.</td>
</tr>
<tr>
<td></td>
<td><strong>CALL ERRSET(N)</strong></td>
</tr>
<tr>
<td></td>
<td>Typeout of each type of error message is suppressed after N occurrences of that error message. If ERRSET is not called, the default value of N is 2.</td>
</tr>
<tr>
<td>EXIT</td>
<td>Returns control to the Monitor and, therefore, terminates the execution of the program.</td>
</tr>
<tr>
<td>ILL</td>
<td>Sets the ILLEG flag. If the flag is set and an illegal character is encountered in floating point/double precision input, the corresponding word is set to zero.</td>
</tr>
<tr>
<td></td>
<td><strong>CALL ILL</strong></td>
</tr>
<tr>
<td>LEGAL</td>
<td>Clears the ILLEG flag. If the flag is set and an illegal character is encountered in the floating point/double precision input, the corresponding word is set to zero.</td>
</tr>
<tr>
<td></td>
<td><strong>CALL LEGAL</strong></td>
</tr>
<tr>
<td>LINE</td>
<td>Causes a line to be drawn through the N points specified by (X(1),Y(1)),(X(2),Y(2)),...,(X(N),Y(N)) where the elements of X and Y are spaced K words apart in storage.</td>
</tr>
<tr>
<td></td>
<td><strong>CALL LINE (X,Y,N,K)</strong></td>
</tr>
<tr>
<td>MKTBL</td>
<td>Specifies a special character set where I is the number to be assigned the set and J contains the starting address of a character table of 200 size consecutive words. In each character table word the left half contains the number of strokes in the character (0 if nothing is to be plotted for the word) and the right half contains the address of the table of strokes for the character.</td>
</tr>
<tr>
<td></td>
<td><strong>CALL MKTBL(I,J)</strong></td>
</tr>
<tr>
<td>NUMBER</td>
<td>Causes a floating point number to be plotted as text. Parameters X, Y, SIZE and THETA have the same meanings as for the SYMBOL call. Parameter NDIGIT is the number of digits plotted to the right of the decimal point. If NDIGIT is a negative value, only the integer part of the number is plotted. FNUM specifies the number to be plotted.</td>
</tr>
<tr>
<td></td>
<td><strong>CALL NUMBER(X,Y,SIZE,FNUM,THETA,NDIGIT)</strong></td>
</tr>
<tr>
<td>PDUMP</td>
<td>The arguments are the same as those for DUMP. PDUMP is the same as DUMP except that control returns to the calling program after the dump has been executed.</td>
</tr>
<tr>
<td></td>
<td><strong>CALL PDUMP(L_1,U_1,F_1,...,L_n,U_n,F_n)</strong></td>
</tr>
<tr>
<td>Subroutine Name</td>
<td>Effect</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------</td>
</tr>
</tbody>
</table>
| PLOT            | CALL PLOT(X,Y,IPEN)  
Move the pen in a straight line from its current position to the position specified by X,Y. If IPEN=3, the pen is raised before the movement; if IPEN=2 the pen is lowered before movement; if IPEN=1 the pen is left unchanged from its previous state. If the value of IPEN is negative (−1, −2 or −3) the pen action is the same as for the corresponding positive values except that on completion of the indicated motion the new pen position is taken as a new origin and the output buffer is sent to the plotter. The plotter is not released on completion of the specified movement. |
| PLOTS           | CALL PLOTS (I)  
The plotter setup routine is called. If the plotter is not available, I is set to −1; if it is available, I is set to 0. This call must be the first plotter routine called. |
| RELEAS          | CALL RELEAS(unit*)  
Closes out I/O on a device initialized by the FORTRAN Operating System and returns it to the uninitialized state. |
| SAVRAN          | SAVRAN is called with one argument in integer mode. SAVRAN sets its argument to the last random number (interpreted as an integer) that has been generated by the function RAN. |
| SCALE           | CALL SCALE(X,N,S,XMIN,DX)  
Selects scale values for an AXIS call where X and N specify a 1-dimensional array X with the length N. Parameter S specifies the length of the desired axis, SCALE determines a value of DX which allows X to be plotted in S inches. XMIN is selected as the smallest element of the array X, and is truncated to be a multiple of DX. |
| SETABL          | CALL SETABL(I,J)  
Specifies a character set where I is an integer which gives the number of the desired character set. If a character set has been defined by I, the value of J is set to 0; if not, J is set to −1. The standard ASCII character set is defined as 1. |
<p>| SETRAN          | SETRAN has one argument which must be a non-negative integer &lt; 2^{31}. The starting value of the function RAN is set to one value of this argument, unless the argument is zero. In this case, RAN uses its normal starting value. |
| SLITE(i)        | Turn sense lights on or off. i is an integer expression. For 1 ≤ i ≤ 36 sense light i will be turned on. If i=0, all sense lights will be turned off. |</p>
<table>
<thead>
<tr>
<th>Subroutine Name</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLITET(i,j)</td>
<td>Checks the status of sense light i and sets the variable j accordingly and turns off sense light i. If i is on, j is set to 1; and if i is off, j is set to 2.</td>
</tr>
<tr>
<td>SSWTCH(i,j)</td>
<td>Checks the status of data switch i (0 ≤ i ≤ 35) and sets the variable j accordingly. If i is set OFF, j is set to 1; and if i is ON, j is set to 2.</td>
</tr>
<tr>
<td>SYMBOL</td>
<td>CALL SYMBOL(X,Y,SIZE,ASC,THETA,NASC) Raise pen and move it to position specified by X and Y. Lower pen and plot characters found in array ASC. Parameter SIZE specifies the height of the characters plotted in inches (floating point values); THETA specifies the direction of the base line on which the text of array ASC is to be plotted, and NASC specifies the number of characters in array ASC.</td>
</tr>
<tr>
<td>TIME</td>
<td>Returns the current time in its argument(s) in left-justified ASCII characters. If TIME is called with one argument, CALL TIME(X) the time is in the form hh:mm where hh is the hours (24-hour time) and mm is the minutes. If a second argument is requested, CALL TIME(X,Y) the first argument is returned as before and the second has the form bss.t where ss is the seconds, t is the tenths of a second, and b is a blank.</td>
</tr>
<tr>
<td>WHERE</td>
<td>CALL WHERE(X,Y) Variables X and Y are set to the values which identify the current position of the pen.</td>
</tr>
</tbody>
</table>
16.1 INTRODUCTION

Block data subprograms are used to initialize data to be stored in any common areas. Only specification and DATA statements are permitted (i.e., DATA, COMMON, DIMENSION, EQUIVALENCE, and TYPE) in block subprograms. A subprogram of this type must start with a BLOCK DATA statement.

If any entry of a labeled common block is initialized by a BLOCK DATA subprogram, the entire block must be included even though some of the elements of the block do not appear in DATA statements.

Initial values may be entered into more than one labeled common block in a single subprogram of this type.

An executable program may contain more than one block data subprogram.

16.2 BLOCK DATA STATEMENT

The form of the BLOCK DATA statement is

    BLOCK DATA name

where

    name    is a symbolic name given to identify the subprogram.
APPENDIX A
ASCII-1968 CHARACTER CODE SET

The character code set defined in the X3.4-1968 Version of the American National Standard for Information Interchange (ASCII) is given in the following matrix.

<table>
<thead>
<tr>
<th>1st 2 octal digits</th>
<th>Last octal digit</th>
</tr>
</thead>
<tbody>
<tr>
<td>00x</td>
<td>NUL</td>
</tr>
<tr>
<td>01x</td>
<td>BS</td>
</tr>
<tr>
<td>02x</td>
<td>DLE</td>
</tr>
<tr>
<td>03x</td>
<td>CAN</td>
</tr>
<tr>
<td>04x</td>
<td>@</td>
</tr>
<tr>
<td>05x</td>
<td>(</td>
</tr>
<tr>
<td>06x</td>
<td>0</td>
</tr>
<tr>
<td>07x</td>
<td>8</td>
</tr>
<tr>
<td>10x</td>
<td>A</td>
</tr>
<tr>
<td>11x</td>
<td>H</td>
</tr>
<tr>
<td>12x</td>
<td>P</td>
</tr>
<tr>
<td>13x</td>
<td>X</td>
</tr>
<tr>
<td>14x</td>
<td>grave</td>
</tr>
<tr>
<td>15x</td>
<td>h</td>
</tr>
<tr>
<td>16x</td>
<td>p</td>
</tr>
<tr>
<td>17x</td>
<td>x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1st 2 octal digits</th>
<th>Last octal digit</th>
</tr>
</thead>
<tbody>
<tr>
<td>00x</td>
<td>SOH</td>
</tr>
<tr>
<td>01x</td>
<td>HT</td>
</tr>
<tr>
<td>02x</td>
<td>DC1</td>
</tr>
<tr>
<td>03x</td>
<td>EM</td>
</tr>
<tr>
<td>04x</td>
<td>`</td>
</tr>
<tr>
<td>05x</td>
<td>*</td>
</tr>
<tr>
<td>06x</td>
<td>1</td>
</tr>
<tr>
<td>07x</td>
<td>9</td>
</tr>
<tr>
<td>10x</td>
<td>B</td>
</tr>
<tr>
<td>11x</td>
<td>J</td>
</tr>
<tr>
<td>12x</td>
<td>Q</td>
</tr>
<tr>
<td>13x</td>
<td>Y</td>
</tr>
<tr>
<td>14x</td>
<td>a</td>
</tr>
<tr>
<td>15x</td>
<td>i</td>
</tr>
<tr>
<td>16x</td>
<td>q</td>
</tr>
<tr>
<td>17x</td>
<td>y</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1st 2 octal digits</th>
<th>Last octal digit</th>
</tr>
</thead>
<tbody>
<tr>
<td>00x</td>
<td>STX</td>
</tr>
<tr>
<td>01x</td>
<td>LF</td>
</tr>
<tr>
<td>02x</td>
<td>DC2</td>
</tr>
<tr>
<td>03x</td>
<td>SUB</td>
</tr>
<tr>
<td>04x</td>
<td>#</td>
</tr>
<tr>
<td>05x</td>
<td>+</td>
</tr>
<tr>
<td>06x</td>
<td>2</td>
</tr>
<tr>
<td>07x</td>
<td>;</td>
</tr>
<tr>
<td>10x</td>
<td>C</td>
</tr>
<tr>
<td>11x</td>
<td>K</td>
</tr>
<tr>
<td>12x</td>
<td>S</td>
</tr>
<tr>
<td>13x</td>
<td>[</td>
</tr>
<tr>
<td>14x</td>
<td>b</td>
</tr>
<tr>
<td>15x</td>
<td>j</td>
</tr>
<tr>
<td>16x</td>
<td>r</td>
</tr>
<tr>
<td>17x</td>
<td>z</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1st 2 octal digits</th>
<th>Last octal digit</th>
</tr>
</thead>
<tbody>
<tr>
<td>00x</td>
<td>ETX</td>
</tr>
<tr>
<td>01x</td>
<td>VT</td>
</tr>
<tr>
<td>02x</td>
<td>DC3</td>
</tr>
<tr>
<td>03x</td>
<td>ESC</td>
</tr>
<tr>
<td>04x</td>
<td>$</td>
</tr>
<tr>
<td>05x</td>
<td>.</td>
</tr>
<tr>
<td>06x</td>
<td>3</td>
</tr>
<tr>
<td>07x</td>
<td>&lt;</td>
</tr>
<tr>
<td>10x</td>
<td>D</td>
</tr>
<tr>
<td>11x</td>
<td>L</td>
</tr>
<tr>
<td>12x</td>
<td>T</td>
</tr>
<tr>
<td>13x</td>
<td>\</td>
</tr>
<tr>
<td>14x</td>
<td>c</td>
</tr>
<tr>
<td>15x</td>
<td>k</td>
</tr>
<tr>
<td>16x</td>
<td>s</td>
</tr>
<tr>
<td>17x</td>
<td>{</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1st 2 octal digits</th>
<th>Last octal digit</th>
</tr>
</thead>
<tbody>
<tr>
<td>00x</td>
<td>EOT</td>
</tr>
<tr>
<td>01x</td>
<td>FF</td>
</tr>
<tr>
<td>02x</td>
<td>DC4</td>
</tr>
<tr>
<td>03x</td>
<td>FS</td>
</tr>
<tr>
<td>04x</td>
<td>$</td>
</tr>
<tr>
<td>05x</td>
<td>.</td>
</tr>
<tr>
<td>06x</td>
<td>4</td>
</tr>
<tr>
<td>07x</td>
<td>=</td>
</tr>
<tr>
<td>10x</td>
<td>E</td>
</tr>
<tr>
<td>11x</td>
<td>M</td>
</tr>
<tr>
<td>12x</td>
<td>U</td>
</tr>
<tr>
<td>13x</td>
<td>^</td>
</tr>
<tr>
<td>14x</td>
<td>f</td>
</tr>
<tr>
<td>15x</td>
<td>m</td>
</tr>
<tr>
<td>16x</td>
<td>t</td>
</tr>
<tr>
<td>17x</td>
<td>l</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1st 2 octal digits</th>
<th>Last octal digit</th>
</tr>
</thead>
<tbody>
<tr>
<td>00x</td>
<td>ENQ</td>
</tr>
<tr>
<td>01x</td>
<td>CR</td>
</tr>
<tr>
<td>02x</td>
<td>NAK</td>
</tr>
<tr>
<td>03x</td>
<td>SYN</td>
</tr>
<tr>
<td>04x</td>
<td>%</td>
</tr>
<tr>
<td>05x</td>
<td>&gt;</td>
</tr>
<tr>
<td>06x</td>
<td>6</td>
</tr>
<tr>
<td>07x</td>
<td>?</td>
</tr>
<tr>
<td>10x</td>
<td>F</td>
</tr>
<tr>
<td>11x</td>
<td>N</td>
</tr>
<tr>
<td>12x</td>
<td>V</td>
</tr>
<tr>
<td>13x</td>
<td>^</td>
</tr>
<tr>
<td>14x</td>
<td>g</td>
</tr>
<tr>
<td>15x</td>
<td>o</td>
</tr>
<tr>
<td>16x</td>
<td>v</td>
</tr>
<tr>
<td>17x</td>
<td>~</td>
</tr>
</tbody>
</table>

Characters inside parentheses are ASCII-1963 Standard.

APPENDIX B
SYMBOLIC NAMES

To be supplied
APPENDIX C
USING THE COMPILER

This appendix explains how to access FORTRAN-10 and how to make use of the information it provides. The reader should be familiar with the FORTRAN-10 language and the DECSYSTEM-10 TOPS-10 monitor.

C.1 RUNNING THE COMPILER
The command to run FORTRAN-10 is

```
.R FORTRA
```

The compiler responds with an asterisk (*) and is then ready to accept a command string. A command is of the general form

```
object filename, listing filename=source filename(s)
```

The following options are given to the user:

1. The filenames can be fully specified SFD paths.
2. The user may specify more than one input file in the compilation command string. These files will be logically concatenated by the compiler and treated as one source file.
3. Program units need not be terminated at file boundaries and may consist of more than one file.
4. If no object filename is specified, no relocatable binary file is generated.
5. If no listing filename is specified, no listing is generated.
6. If no extension is given, the defaults are .LST (listing), .REL (relocatable binary), and .FOR (source) for their respective files.

C.1.1 Switches Available with FORTRAN-10
Switches to FORTRAN-10 are accepted anywhere in the command string. They are totally position and file independent. The switches are shown in Table C-1.
Table C-1
FORTRAN-10 Compiler Switches

<table>
<thead>
<tr>
<th>Switch</th>
<th>Meaning</th>
<th>Defaults</th>
</tr>
</thead>
<tbody>
<tr>
<td>CROSSREF</td>
<td>Generate a file that can be input to the CREF program.</td>
<td>OFF</td>
</tr>
<tr>
<td>DEBUG</td>
<td>See Section C.1.1.1.</td>
<td>OFF</td>
</tr>
<tr>
<td>EXPAND</td>
<td>Include the octal-formatted version of the object file in the listing.</td>
<td>OFF</td>
</tr>
<tr>
<td>INCLUDE</td>
<td>Compile a D in card column 1 as a space.</td>
<td>OFF</td>
</tr>
<tr>
<td>KA10</td>
<td>Compile code to run on a KA10 processor.</td>
<td>Compilation processor.</td>
</tr>
<tr>
<td>KI10</td>
<td>Compile code to run on a KI10 processor.</td>
<td>Compilation processor.</td>
</tr>
<tr>
<td>MACROCODE</td>
<td>Add the mnemonics translation of the object code to the listing file.</td>
<td>OFF</td>
</tr>
<tr>
<td>NOERRORS</td>
<td>Do not print error messages on the terminal.</td>
<td>OFF</td>
</tr>
<tr>
<td>NOWARNINGS</td>
<td>Do not output warning messages.</td>
<td>OFF</td>
</tr>
<tr>
<td>OPTIMIZE</td>
<td>Perform global optimization.</td>
<td>OFF</td>
</tr>
<tr>
<td>SYNTAX</td>
<td>Perform syntax check only.</td>
<td>OFF</td>
</tr>
</tbody>
</table>

Each switch must be preceded by a slash (/). Switches need consist of only those letters that are required to make the switch unique. But users are encouraged to use at least three letters to prevent conflict with switches in future implementations.

Example

```
.R FORTRA
*OFILE,LFILE=SFILE/MAC,S2FILE
```

The /MAC switch will cause the MACRO code equivalent of SFILE and S2FILE to appear in LFILE.LST.

If the user does not specify a processor (KA10 or KI10 switch), the code will be compiled for the processor type on which the compilation occurs. The processor type of the code in the object file and all switches, used or implied, are printed at the top of each listing page.

C.1.1.1 The /DEBUG Switch – Using the /DEBUG switch tells FORTRAN-10 to compile a series of debugging features into the user program. Several of these features are specifically designed to be used with FORDDT. Refer to Appendix F for more information. By adding the modifiers listed in Table C-2, the user is able to include specific debugging features.
<table>
<thead>
<tr>
<th>Modifiers</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>:DIMENSIONS</td>
<td>Generates dimension information in .REL file for FORDDT.</td>
</tr>
<tr>
<td>:TRACE</td>
<td>Generates references to FORDDT required for its trace features (automatically activates :LABELS).</td>
</tr>
<tr>
<td>:LABELS</td>
<td>Generates a label for each statement of the form “line-number L.”(This option may be used without FORDDT.)</td>
</tr>
<tr>
<td>:INDEX</td>
<td>Forces DO LOOP indices to be stored at the beginning of each iteration rather than held in a register for the duration of the loop.</td>
</tr>
<tr>
<td>:BOUNDS</td>
<td>Generates the bounds checking code for all array references. Bounds violations will produce run-time error messages. Note that the technique of specifying dimensions of 1 for subroutine arrays will cause bounds check errors. (This option may be used without FORDDT.)</td>
</tr>
</tbody>
</table>

The format of the /DEBUG switch and its modifiers is as follows:

/DEBUG:modifier

or

/DEBUG:(modifier list)

Options available with the /DEBUG modifiers are:

1. No debug features — Either do not specify the /DEBUG switch or include /DEBUG:NONE.
2. All debug features — Either /DEBUG or /DEBUG:ALL.
3. Selected features — Either /DEBUG or /DEBUG:ALL.

   /DEBUG:BOU /DEBUG:LAB

   or a list of modifiers

   /DEBUG:(BOU,LAB, . . .)

4. Exclusion of features (if the user wishes all but one or two modifiers and does not wish to list them all, he may use the prefix "NO" before the switch he wishes to exclude). The exclusion of one or more features implicitly includes all the others, i.e., /DEBUG:NOBOU is the same as /DEBUG:(DIM,TRA,LAB,IND).
If more than one statement is included on a single line, only the first statement will receive a label (/DEBUG:LABELS) or FORDDT reference (/DEBUG:TRACE). (The /DEBUG option and the /OPTIMIZE option cannot be used at the same time.)

NOTE

If a source file contains line sequence numbers that occur more than once in the same subprogram, the /DEBUG option cannot be used.

The following formulas may be used to determine the increases in program size that will occur due to the addition of various /DEBUG options.

DIMENSIONS: For each array, have $3+3^*N$ words where $N$ is the number of dimensions, and up to 3 constants for each dimension.

TRACE: One instruction per executable statement.

LABELS: No increase.

INDEX: One instruction per inner loop plus one instruction for some of the references to the index of the loop.

BOUNDS: For each array, the formula is the same as DIMENSIONS:

For each reference to an array element, use $5+N$ words where $N$ is the number of dimensions in the array. If BOUNDS: was not specified, approximately $1+3*(N-1)$ words would be used.

C.1.2 COMPIL-Class Commands

FORTRAN-10 can also be invoked by using COMPIL-class commands. These commands cause the monitor to run the COMPIL program which interprets the command and constructs new command strings for the system program actually processing the command. When both FORTRAN-10 and F40 are present in a DECSYSTEM-10 system, users can specify which compiler is to be used by adding the switches /F10 or /F40 to the following commands:

COMPIL
LOAD
EXECUTE
DEBUG

Example

.EXEC Rotor /F10

The compiler switches KA, K1, OPT, CREF, and DEBUG may be specified directly in COMPIL-class commands and may be used globally or locally.

Example

.EXECUTE/CREF/KA/F10P1.FOR,P2.FOR/DEBUG:NOBOU

The other compiler switches must be passed in parentheses for each specific source file.
Example

EXECUTE PI.FOR(M.I)

Refer to the DECsystem-10 Operating System Commands Manual for further information.

C.2 READING A FORTRAN-10 LISTING

When a user requests a listing from the FORTRAN-10 compiler, it contains the following information:

1. A printout of the source program plus an internal sequence number assigned to each line by the compiler. This internal sequence number is referenced in any error or warning messages generated during the compilation. If the input file is line sequenced, the number from the file is used. If code is added via the INCLUDE statement, all INCLUDED lines will have an asterisk (*) appended to their line sequence number.

2. A summary of the names and relative program locations (in octal) of scalars and arrays in the source program plus compiler generated variables.

3. All COMMON blocks and the relative locations (in octal) of the variables in each COMMON block.

4. A listing of all equivalenced variables or arrays and their relative locations.

5. A listing of the subprograms referenced (both user defined and FORTRAN-10 defined library functions).

6. A summary of temporary locations generated by the compiler.

7. A heading on each page of the listing containing the program unit name (MAIN., program, subroutine or function, principal entity), the input filename, the list of compiler switches, and the date and time of compilation. Whether or not a specific processor switch (/K10, /K110) was used, the processor for which the code was generated is also at the top of the listing page.

8. If the /MACRO switch was used, a mnemonic printout of the generated code (in a format similar to MACRO-10) is appended to the listing. This section has four fields:

   LINE: This column contains the internal sequence number of the line corresponding to the mnemonic code. It appears on the first of the code sequence associated with that internal sequence number. An asterisk indicates a compiler inserted line.

   LOC: The relative location in the object program of the instruction.

   LABEL: Any program or compiler generated label. Program labels have the letter “P” appended. Labels generated by the compiler are followed by the letter “M”. Labels generated by the compiler and associated with the /DEBUG:LABELS switch consist of the internal sequence number followed by an “L”.

   GENERATED CODE: The MACRO-10 mnemonic code.

9. A list of all argument blocks generated by the compiler. A zero argument appears first followed by argument blocks for subroutine calls and function references (in order of their appearance in the program). Argument blocks for all I/O operations follow this.
10. Format statement listings.

11. A summary of errors detected or warning messages issued during compilation.

C.2.1 Compiler Generated Variables

In certain situations the compiler will generate internal variables. Knowing what these variables represent can help in reading the macro expansion. The variables are of the form:

.letter digit digit digit digit

i.e., .S0001

where:

<table>
<thead>
<tr>
<th>Letter</th>
<th>Function of Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Result of the DO LOOP step size expression of computed iteration count for a loop.</td>
</tr>
<tr>
<td>O</td>
<td>Result of a common subexpression or constant computation.</td>
</tr>
<tr>
<td>I</td>
<td>Result of a DO LOOP initial value expression or parameters of an adjustably dimensioned array.</td>
</tr>
<tr>
<td>F</td>
<td>Arithmetic statement function formal parameters.</td>
</tr>
<tr>
<td>R</td>
<td>Result of reduced operator strength expression (D.2.1.2).</td>
</tr>
</tbody>
</table>

The user may find these variables on the listing under SCALARS and ARRAYS.

The following example shows a listing where all these features are pointed out.
Name of Program: TIM1
Name of Source File: FORTRAN V.2B(135) /K1/H 6-JUN-74

Version 4 FORTRAN-10

00100 IMPLICIT INTEGER (A-Z)
00200 DIMENSION A(100,200),B(100,200)
00300 SUM1=0
00400 SUM2=0
00500 DO 100 J=1,200
00600 DO 100 I=1,100
00700 K1=I+J
00800 IF (K1 .LT. 500 .OR. K1 .GT. 1500) K1=0
00900 A(I,J)=K1
01000 K2=I+J
01100 IF (K2 .EQ. 100 .OR. K2 .EQ. 200 .OR. K2 .EQ. 100) K2=K2+1
01200 B(I,J)=K2
01300 SUM1=SUM1+K1
01400 SUM2=SUM2+K2
01500 100 CONTINUE
01600 10 TYPE 10,SUM1,SUM2
01700 10 FORMAT(7H SUM1=,I9,10H SUM2=,I9)
01900 END

SUBPROGRAMS CALLED

The relative address of all variables is given.

SCALARS AND ARRAYS

*K1 1 B
*SUM2 116105 #1

*Temporaries

Compiler generated variables
<table>
<thead>
<tr>
<th>LINE</th>
<th>LOC</th>
<th>LABEL</th>
<th>GENERATED CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>JFCL</td>
<td>0,0</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>JSP</td>
<td>16,RESET</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>0,0</td>
<td></td>
</tr>
<tr>
<td>300</td>
<td>3</td>
<td>SETZB</td>
<td>2, SUM1</td>
</tr>
<tr>
<td>400</td>
<td>4</td>
<td>MOVEM</td>
<td>2, SUM2</td>
</tr>
<tr>
<td>500</td>
<td>5</td>
<td>MOVE</td>
<td>2, [777670000001]</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>HLREM</td>
<td>2, 80000</td>
</tr>
<tr>
<td>7</td>
<td>2 M1</td>
<td>HRRZM</td>
<td>2, J</td>
</tr>
<tr>
<td>600</td>
<td>10</td>
<td>MOVE</td>
<td>2, [777634000001]</td>
</tr>
<tr>
<td>700</td>
<td>11</td>
<td>MOVE</td>
<td>3, J</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>IMULI</td>
<td>3, 0(2)</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>MOVEM</td>
<td>3, J1</td>
</tr>
<tr>
<td>800</td>
<td>14</td>
<td>CAIL</td>
<td>3, 764</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td>CAILE</td>
<td>3, 2734</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>JRST</td>
<td>0, 6M</td>
</tr>
<tr>
<td>17</td>
<td>17</td>
<td>JRST</td>
<td>0, 5M</td>
</tr>
<tr>
<td>800</td>
<td>20</td>
<td>SETZB</td>
<td>4, K1</td>
</tr>
<tr>
<td>900</td>
<td>21</td>
<td>MOVE</td>
<td>3, 144</td>
</tr>
<tr>
<td>22</td>
<td>5 M1</td>
<td>IMUL</td>
<td>3, J</td>
</tr>
<tr>
<td>23</td>
<td></td>
<td>ADDI</td>
<td>3, 0(2)</td>
</tr>
<tr>
<td>24</td>
<td></td>
<td>MOVE</td>
<td>4, K1</td>
</tr>
<tr>
<td>25</td>
<td></td>
<td>MOVE</td>
<td>4, A-145(3)</td>
</tr>
<tr>
<td>1000</td>
<td>26</td>
<td>MOVE</td>
<td>3, J</td>
</tr>
<tr>
<td>27</td>
<td>27</td>
<td>ADDI</td>
<td>3, 0(2)</td>
</tr>
<tr>
<td>1100</td>
<td>28</td>
<td>MOVEM</td>
<td>3, K2</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>CAIE</td>
<td>3, 144</td>
</tr>
<tr>
<td>33</td>
<td>33</td>
<td>CAIN</td>
<td>3, 310</td>
</tr>
<tr>
<td>34</td>
<td>34</td>
<td>JRST</td>
<td>0, 8M</td>
</tr>
<tr>
<td>35</td>
<td>35</td>
<td>CAIN</td>
<td>5, 454</td>
</tr>
<tr>
<td>1100</td>
<td>36</td>
<td>AQS</td>
<td>3, K2</td>
</tr>
<tr>
<td>1200</td>
<td>37</td>
<td>MOVEI</td>
<td>3, 144</td>
</tr>
<tr>
<td>38</td>
<td></td>
<td>IMUL</td>
<td>3, J</td>
</tr>
</tbody>
</table>

*Internal sequence number on first instruction that goes with that line.

*Octal displacement of instruction.

*Compiler generated label.
C.3 ERROR REPORTING

If an error occurs during the initial pass of the compiler (while the actual source code is being read and processed), an error message is printed on the listing immediately following the line in which the error occurred. Each error references the internal sequence number of the incorrect line. The error messages along with the statement in error are output to the user terminal. For example:

```
.EXECUTE DAY.FOR
FORTREN: DAY
01300
?FTNNRNC LINE:01300
01500 100
?FTNMSF LINE:01500
01600 ?
?FTNCL LINE:01600

K1
STATEMENT NOT RECOGNIZED
CONTINE
STATEMENT NAME MISSPELLED
ILLEGAL CHARACTER C IN LABEL FIELD
3 FATAL ERRORS AND NO WARNINGS
```

EXIT

If errors are detected after the initial pass of the compiler, they appear in the list file after the end of the source listing. They are output to the user terminal without the statement in error but they do reference its internal sequence number.

C.3.1 Fatal Errors and Warning Messages

There are two levels of messages, warning and fatal error. Warning messages are preceded by "%" and indicate a possible problem. The compilation will continue, and the object program will probably be correct. Fatal errors are preceded by a "?". If a fatal error is encountered in any pass of the compiler, the remaining passes will not be called. Additional errors that would be detected in later compiler passes may not become apparent until the first errors are corrected. As the word fatal denotes, it is not possible to generate a correct object program for a source program containing a fatal error.

The format of messages is

```
?FTNXXX LINE:n text
```

or

```
%FTNXXX LINE:n text
```

where:

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>fatal</td>
</tr>
<tr>
<td>%</td>
<td>warning</td>
</tr>
<tr>
<td>FTN</td>
<td>FORTRAN mnemonic</td>
</tr>
<tr>
<td>XXX</td>
<td>3-letter mnemonic for the error message</td>
</tr>
<tr>
<td>LINE:n</td>
<td>line number where error occurred</td>
</tr>
<tr>
<td>text</td>
<td>explanation of error</td>
</tr>
</tbody>
</table>

Version 4 FORTRAN-10 C-10 October 1974
The printing of fatal errors and warning messages on the user's terminal can be suppressed by the use of the /NOERRORS switch; however, messages will still appear on the listing. The /NOWARNINGS switch will suppress warning messages on both user terminal and listing.

C.3.2 Message Summary
At the end of the listing file and on the terminal, a message summary is printed after each program unit is compiled. This message has two forms:

1. when one or more messages were issued

   \[ ?FTNFTL \]
   \[ %FTNWRN \]
   \[ name \ NO/number \ FATAL \ ERRORS \ AND \ NO/number \ WARNINGS \]
   \[ or \]

2. when no messages were issued

   \[ name \ [NO ERRORS DETECTED] \]

where name is the program or subprogram name. ([NO ERRORS DETECTED] appears on the listing only.) For a complete list of fatal errors and warning messages, see Appendix G.

C.4 CREATING A REENTRANT FORTRAN PROGRAM WITH LINK-10
To produce a sharable program from the .REL file (e.g., MAIN.REL), give either one of the following commands to LINK-10:

1. /SEG:DEFAULT MAIN/G
2. /OTS:NONSHAR MAIN/G

The resulting core image can be SSAVE'd or the /SSAVE switch can be used to produce a .SHR file.
APPENDIX D
WRITING USER PROGRAMS

This appendix is a guide for writing effective user programs with FORTRAN-10. It contains techniques for optimization, interaction with non-FORTRAN programs, mixing of FORTRAN-10 and F40 compiler programs, and other useful programming hints.

D.1 GENERAL PROGRAMMING CONSIDERATIONS
Programming considerations that should be observed when preparing a FORTRAN program to be compiled by FORTRAN-10 are described in the following paragraphs.

D.1.1 Accuracy and Range of Double Precision Numbers
Floating point and real numbers may consist of up to 16 digits in a double precision mode. Their range is specified in Chapter 3, Section 3.2 of this manual. Care must be taken when testing the value of a number within the specified range since, although numbers up to \(10^{98}\) may be represented, FORTRAN-10 can only test numbers of up to eight significant digits (REAL precision) and 16 significant digits (DOUBLE precision).

Care must also be taken when testing the floating point computation for a result of 0. In most cases the anticipated result (i.e., 0) will be obtained; however, in some cases the result may be a very small number that approximates 0. Such an approximation of 0 would cause tests within statements (i.e., an arithmetic IF) to fail.

D.1.2 Writing FORTRAN-10 Programs for Execution on Non-DEC Machines
If a program is to be prepared to run on both a DECsystem-10 computer and a non-DEC machine, the user should

1. avoid using the non-ANSI Standard features of FORTRAN-10, and
2. consider the accuracy and size of the numbers that the non-DEC machine is capable of handling.

D.1.3 Using Floating Point DO Loops
FORTRAN-10 permits the user to employ non-integer single or double precision numbers as the parameter variables in a DO statement. The primary advantage of the foregoing is to enable the user to generate a wider range of values for the DO loop index variables which may, in turn, be used inside the loop for computations. Care should be taken in considering the loss of precision that may occur in this context.

D.1.4 Computation of DO Loop Iterations
The number of times through a DO loop is computed outside the loop and is not affected by any changes to the DO index parameters within the loop. The formula for the number of times a DO loop is executed is

\[
\text{MAX} \left( 1, \frac{m_2 - m_1 + m_3}{m_3} \right) = \text{Number of cycles}
\]

Version 4 FORTRAN-10

D-1

October 1974
The values of the parameters (i.e., \(m_1, m_2, m_3\)) may be of any type; however, proper consideration must be given to the foregoing formula, particularly when using logicals. One iteration of each DO loop is always performed even if the result of the foregoing calculation is less than or equal to zero.

D.1.5 Subroutines – Programming Considerations
The following items must be considered when preparing and executing subroutines:

1. During execution, no check is made to see if the proper number of parameters were passed.

2. If the number of actual arguments passed to a subroutine are less than the number of dummy arguments specified, the values of the unspecified arguments are undefined.

3. If the number of actual arguments passed to a subroutine is greater than the number of dummy arguments given, the excess arguments are ignored.

4. If an actual parameter is a constant and its corresponding dummy argument is set to another value, all references made to the constant in the calling program may be changed to the value of the dummy argument.

5. No check is made to see if the parameters passed are of the same type as the dummy parameters. If an actual parameter is a constant and the corresponding dummy is of type real, be sure to include the decimal point with the constant. If the dummy is double precision, be sure to specify the constant with a “D”.

Examples
If the function \(F(A)\) is called by inputting \(F(2)\) and \(A\) is type real, \(F\) interprets the integer 2 as an unnormalized floating point number. In this instance, \(F(A)\) should be called with \(F(2.0)\).

Similarly, if the function \(F1(D)\) is called by inputting \(F1(2.5)\) and \(D\) is double precision, \(F1\) assumes that its parameters have been specified with two words of precision and picks up whatever follows the constant 2.5 in core. The proper method is to use \(F1(2.5D00)\).

NOTE
No notice is given to the user if any of the situations described in items 1, 2, 3, 4, and 5 occur.

D.1.6 Reordering of Computations
Computations that are not enclosed within parentheses may be reordered by the compiler. Sometimes it is necessary to use parentheses to prevent improper results from being obtained from a specific computation.

For example, assuming that

1. \(RL1\) represents a large number such that \(RL1*RL2\) will cause an overflow condition, and
2. RS1 is a very small number (i.e., less than 1) the program sequence

\[ A = RS1 \times RL1 \times RL2 \]
\[ B = RS2 \times RL2 \times RL1 \]

will not produce an overflow when evaluated left to right since the first computation in each expression (i.e., RS1*RL1 and RS2*RL2) will produce an interim result that is smaller than either large number (RL1 or RL2).

However, the compiler will recognize RL1*RL2 as a common subexpression and generate the following sequence:

\[
\begin{align*}
\text{temp} &= RL1 \times RL2 \\
A &= RS1 \times \text{temp} \\
B &= RS2 \times \text{temp}
\end{align*}
\]

The computation of temp will cause an overflow.

The program sequence should be written in the following manner to ensure that the desired results are obtained:

\[ A = (RS1 \times RL1) \times RL2 \]
\[ B = (RS2 \times RL2) \times RL1 \]

Computations may be reordered even when global optimization is not selected.

D.1.7 Dimensioning of Formal Arrays

When an array is specified as a formal parameter to a subprogram unit, it is necessary to indicate to the compiler that the parameter is an array. The user must dimension the array in a specification statement. This is the only way the compiler is able to distinguish a reference to such an array from a function reference. Designating the array with a dimension of 1 has become a common practice among users.

Example

\[
\begin{align*}
\text{SUBROUTINE } & \text{SUB1(A,B)} \\
\text{DIMENSION } & \text{A(1)}
\end{align*}
\]

There are disadvantages to using the above technique because the dimension information provided is not adequate in some cases, specifically.
1. Reading or writing the array by name.

    DIMENSION ARRAY (10)
    READ (1) ARRAY

    This is a binary read that will read 10 words into ARRAY.

    SUBROUTINE SUB1(A)
    DIMENSION A(1)
    READ(1,A)

    This binary read will cause 1 word to be read into A.

2. Reading the array as a format

    SUBROUTINE SUB2 (FMT)
    DIMENSION FMT(1)
    READ (1,FMT)

    This will cause 1 word of the array FMT to be written over with the characters read from the record on unit 1.

When using the /DEBUG:BOUNDS compilation switch, the dimension information used is that which is specified in the array declaration.

    SUBROUTINE DO IT(A)
    DIMENSION A(1)
    A(2) = 0

The reference to A(2) will cause the out-of-bounds warning message to be generated.

D.2 FORTRAN-10 GLOBAL OPTIMIZATION

The user has the option of invoking a global optimizer during compilation. This optimizer treats groups of statements in the source program as a single entity. The purpose of the global optimizer is to prepare a more efficient object program that produces the same results as the original unoptimized program but takes significantly less execution time. The output of the lexical and syntax analysis phase of the compiler is developed into an optimized source program equivalent (in results) to the original. The optimized program is then processed by the standard compiler code generation phase.

D.2.1 Optimization Techniques

D.2.1.1 Elimination of Redundant Computations – Often the same subexpression will appear in more than one computation throughout a program. If the values of the operands of such a common expression are not changed between computations, the subexpression may be written as a separate arithmetic expression, and the variable
representing its resultant may then be substituted where the subexpression appears. This eliminates unnecessary recomputation of the subexpression. For example, the instruction sequence

\[ A = B^*C + E^*F \]
\[ \vdots \]
\[ H = A + G - B^*C \]
\[ \vdots \]
\[ IF((B^*C)-H) 10,20,30 \]

contains the subexpression \( B^*C \) three times when it really needs to be computed only once. Rewriting the foregoing sequence as

\[ T = B^*C \]
\[ A = T + E^*F \]
\[ \vdots \]
\[ H = A + G - T \]
\[ \vdots \]
\[ IF((T)-H) 10,20,30 \]

eliminates two computations of the subexpression \( B^*C \) from the overall sequence.

Decreasing the number of arithmetic operations performed in a source program by the elimination of common subexpressions shortens the execution time of the resulting object program.

D.2.1.2 Reduction of Operator Strength – The time required to execute arithmetic operations will vary according to the operator(s) involved. The hierarchy of arithmetic operations according to the amount of execution time required is

\[ \begin{array}{c|c}
\text{MOST TIME} & \text{OPERATOR} \\
\hline
& ** \\
& / \\
& * \\
& +,- \\
\text{LEAST TIME} & \\
\end{array} \]

During program optimization, the global optimizer replaces, where possible,\(^1\) some arithmetic operations that require the most time with operations that require less time. For example, consider the following DO loop that is used to create a table for the conversion of from 1 to 20 miles to their equivalents in feet.

\[
\begin{array}{c}
\text{DO 10 MILES = 1,20} \\
10 \quad \text{IFEET(MILES) = 5280*MILES}
\end{array}
\]

\(^1\)Numerical analysis considerations severely limit the number of cases where it is possible.
The execution time of the foregoing loop would be shorter if the time consuming multiply operation (i.e., 5280*MILES) could be replaced by a faster operation. Since the variable MILES is incremented by 1 on each iteration of the loop, the multiply operation may be replaced by an add and total operation.

In its optimized form, the foregoing loop would be replaced by a sequence equivalent to

```
K=5280
DO 10 MILES = 1,20
  FF=8(MILES) = K
10   K=K+5280
```

In the optimized form of the loop, the value of K is set to 5280 for the first iteration of the loop and is increased by 5280 for each succeeding iteration of the loop.

The foregoing situation occurs frequently in subscript calculations which implicitly contain multiplications whenever the dimensionality is two or greater.

D.2.1.3 Removal of Constant Computation From Loops – The speed with which a given algorithm may be executed can be increased if instructions and/or computations are moved out of frequently traversed program sequences into less frequently traversed program sequences. Movement of code is possible only if none of the arguments in the items to be moved are redefined within the code sequences from which they are to be taken. Computations within a loop comprised of variables or constants that are not changed in value within the loop may be moved outside the loop. Decreasing the number of computations made within a loop will greatly decrease the execution time required by the loop.

For example, in the sequence

```
DO 10 I=1,100
10   F=2.0*Q*A(I)+F
```

the value of the computation 2.0*Q, once calculated on the first iterations, will remain unchanged during the remaining 99 iterations of the loop. Reforming the foregoing sequence to:

```
QQ = 2.0*Q
DO 10 I=1,100
10   F=QQ*A(I)+F
```

moves the calculation 2.0*Q outside of the scope of the loop. This movement of code eliminates 99 multiply operations.

In addition it is possible to remove entire assignment statements from loops. This action can be easily detected from the macro expanded listings. The internal sequence number remains with the statement and appears out of order in the leftmost column of the macro expanded listing (LINE).
D.2.1.4 Constant Folding and Propagation — In this method of optimization, expressions containing determinate constant values are detected and the constants are replaced, at compile time, by their defined or calculated value. For example, assume that the constant PI is defined and used in the following manner:

\[
\pi = 3.14159
\]

\[
X = 2\pi Y
\]

At compile time, the optimizer will have used the defined value of PI to calculate the value of the subexpression \(2\pi Y\). The optimized sequence would then be

\[
\pi = 3.14159
\]

\[
X = 6.28318Y
\]

thereby eliminating a multiply operation from the object code program.

The computation of determinate constant values at compile time is termed "folding"; the use of the defined value of a constant for replacement purposes throughout a program sequence is termed "propagation of the constants". The execution time saved by the foregoing type of compile time optimization is particularly important when the modified instruction occurs in a loop.

D.2.1.5 Removal of Inaccessible Code — The optimizer detects and eliminates any code within the source program that cannot be accessed. In general, the foregoing condition will not exist since programmers will not normally include such code in their programs; however, inaccessible code may appear in a program during the debugging process. The removal of inaccessible code by the optimizer will reduce the size of the optimized object program. A warning message is generated for each inaccessible line removed.

D.2.1.6 Global Register Allocation — During the compilation of a source program the optimizer controls the allocation of registers to minimize computation time in the optimized object program. The intent of the allocation process is to minimize the number of MOVE and MOVEM machine instructions that will appear in the most frequently executed portions of the code.

D.2.1.7 I/O Optimization — Every effort is made to minimize the number of calls required into the FOROTS system. This is done primarily through extensive analysis of implied DO loop constructs on READ, WRITE, ENCODE, DECODE and REREAD statements. The formats of these special blocks are described in Appendix E. These optimizations reduce the size of the program (argument code plus argument block size is reduced) and greatly improve the performance of programs that use implied DO loop I/O statements.
D.2.1.8 Uninitialized Variable Detection – A warning message is generated when a scalar variable is referenced before it could possibly have received a value.

D.2.1.9 Test Replacement – If the only use of a DO loop index is to reduce operator strength (D.2.1.2) and the loop does not contain exits (GO TO's out of the loop), the DO loop index is not needed and can be replaced by the reduced variable. This actually occurs quite often in double precision array subscript computations.

For example:

\[
\begin{align*}
\text{DO } 10 & \ i=1,10 \\
& \ K=K+7*i \\
10 & \text{ CONTINUE}
\end{align*}
\]

Reduction of operator strength and test replacement together transform this loop into

\[
\begin{align*}
\text{DO } 10 & \ i=7,70,70 \\
& \ K=K+i \\
10 & \text{ CONTINUE}
\end{align*}
\]

Although this particular example is trivial, the actual situation occurs frequently in subscript computation.

D.2.2 Improper Function References

The ANSI FORTRAN standard prohibits the use of a function's reference that has side effects that will influence the statement in which the function is referenced (such as defining or redefining other elements in the statement). The compiler depends on strict adherence to the foregoing rule. The generated object program may not yield the desired results if this rule is violated.

D.2.3 Programming Techniques for Effective Optimization

The following recommendations, when observed during the coding of a FORTRAN source program, will improve the effectiveness of the optimizer:

1. DO loops with an extended range should not be used.
2. Specify label lists when using assigned GO TO's.
3. Nest loops so that the innermost index is the one with the largest range of values.
4. Avoid the use of associated input/output variables.
5. Avoid unnecessary use of COMMON and EQUIVALENCE.

D.3 INTERACTING WITH NON-FORTRAN-10 PROGRAMS AND FILES

D.3.1 Calling Sequences

The standard procedures for writing DECsystem-10 subroutine calls are described in the following paragraphs.

1. Procedure

   a. The calling program must load the right half of accumulator (AC) 16 with the address of the first argument in the argument list.

   b. The left half of AC 16 must be set to zero.
c. The subroutine is then called by a PUSHJ instruction using AC 17.

d. The returns will be made to the instruction immediately after the PUSHJ 17 instruction.

e. If the /DEBUG:BOUNDS option of the FOROTS trace facility is being used, the calling sequence must be

   MOVEI 16.AP
   PUSHJ 17,F

   where AP is the pointer to the argument list. If the trace facility is to be used, the word preceding the first word of an entry point should have its name in sixbit.

2. Restrictions

   a. Skip returns are not permitted.

   b. The contents of the pushdown stack located before the address specified by AC 17 belongs to the calling program; it cannot be read by the called subprogram.

   c. FOROTS assumes that it has control of the stack; therefore, the user must not create his own stack. The FOROTS stack is initialized by

   JSP 16.RESET.

   or the library routine

   CALL RESET.

D.3.2 Accumulator Usage

The specific functions performed by accumulators (AC) 17,16,0 and 1 are as follows:

1. Pushdown Pointer — AC 17 is always maintained as a pushdown pointer. Its right half points to the last location in use on the stack and its left half contains the negative of the number of (words–1) allocated to the unused remainder of the stack (a trap occurs when something is pushed into the next location. The trap instruction may itself be a PUSHJ on the K110 processor which uses the last location). A positive left half is not permitted.

2. Argument List Pointer — AC 16 is used as the argument pointer. The called subprogram does not need to preserve its contents. The calling program cannot depend on getting back the address of the argument list passed to the callee. AC 16 cannot point to the AC’s or to the stack.

3. Temporary and Value Return Registers — AC 0 and 1 are used as temporary registers and for returning values. The called subprogram does not need to preserve the contents of AC 0 or 1 (even if not returning a value). The calling program must never depend on getting back the original contents of the data passed to the called subprogram.

4. Returning Values — At the option of the designer of a called subprogram, a subroutine may pass back results by modifying the arguments, returning a single precision value in AC 0 or a double precision or complex value in AC 0 and 1. A combination of the above may be used. However, two single precision values cannot be returned in AC 0 and 1 since FORTRAN would not be able to handle it.
5. Preserved AC's – FORTRAN-10 FUNCTION subprograms preserve AC's 2–15; subroutine subprograms do not.

The design of the called subprogram cannot depend on the contents of any of the AC's being set up by the calling subprogram, except for AC's 16 and 17. Passing information must be done explicitly by the argument list mechanism. Otherwise, the called subprograms cannot be written in either FORTRAN-10 or COBOL.

D.3.3 Argument Lists

The format of the argument list is as follows:

```
arg count word
arg list addr. — — first arg entry
       second arg entry

...

last arg entry
```

The format of the arg count word is:

```
bits 0–17  These contain \(-n\), where \(n\) is the number of arg entries.
bits 18–35 These are reserved and must be 0.
```

The format of an arg entry is as follows (each entry is a single word):

```
bits 0–8  Reserved for future DEC development (set to 0 for now).
bits 9–12  Arg type code.
bits 13    Indirect bit if desired.
bits 14–17 Index field, must be 0 for present.
bits 18–35 Address of the argument.
```

The following restrictions should be observed:

1. Neither the argument lists nor the arguments themselves can be on the stack. This restriction is imposed so that the stack can be moved. The same restriction applies to any indirect argument pointers.

2. The called program may not modify the argument list itself. The argument list may be in a write-protected segment.

Note that the arg count word is at position \(-1\) with respect to the contents of AC16. This word is always required even if the subroutine does not handle a variable number of arguments. A subroutine that has no arguments must still provide an argument list consisting of two words (i.e., the argument count word with a 0 in it and a zero argument word).
Example

MOVEI 16, AP
PUSH 17,SUB

;SET UP ARG POINTER
;CALL SUBROUTINE
;RETURN HERE

;ARGUMENT LIST
-3,0
AP: A
B
C

;SUBROUTINE TO SET THIRD ARG TO SUM OF FIRST TWO ARGS

SUB: MOVE T,0(16) ;GET FIRST ARG
ADD T,1(16) ;ADD SECOND ARG
MOVEM T,2(16) ;SET THIRD ARG
POJ 17, ;RETURN TO CALLER

D.3.4 Argument Types

<table>
<thead>
<tr>
<th>Type Code</th>
<th>Description</th>
<th>FORTRAN Use</th>
<th>COBOL Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Unspecified</td>
<td></td>
<td>Unspecified</td>
</tr>
<tr>
<td>1</td>
<td>FORTRAN Logical</td>
<td></td>
<td>Not applicable</td>
</tr>
<tr>
<td>2</td>
<td>Integer</td>
<td></td>
<td>1-word COMP</td>
</tr>
<tr>
<td>3</td>
<td>Reserved</td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>4</td>
<td>Real</td>
<td></td>
<td>COMP-1</td>
</tr>
<tr>
<td>5</td>
<td>Reserved</td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>6</td>
<td>Reserved</td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>7</td>
<td>Label</td>
<td></td>
<td>Procedure address</td>
</tr>
<tr>
<td>10</td>
<td>Double real</td>
<td></td>
<td>Not applicable</td>
</tr>
<tr>
<td>11</td>
<td>Not applicable</td>
<td></td>
<td>2-word COMP</td>
</tr>
<tr>
<td>12</td>
<td>Reserved</td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>13</td>
<td>Reserved</td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>14</td>
<td>Complex</td>
<td></td>
<td>Not applicable</td>
</tr>
<tr>
<td>15</td>
<td>Not applicable</td>
<td></td>
<td>Byte string descriptor</td>
</tr>
<tr>
<td>16</td>
<td>Reserved</td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>17</td>
<td>ASCIZ string</td>
<td></td>
<td>Not applicable</td>
</tr>
</tbody>
</table>

Literal arguments are permitted, but they must reside in a writable segment. This is because the FORTRAN-10 compiler makes a local of all non-array elements and copies all formals back to the caller's arguments. All unused type codes are reserved for future DEC development.
D.3.5 Description of Arguments

The types of the arguments that may be passed are:

1. Type 0 – Unspecified
   The calling program has not specified the type. The called subprogram should assume that the argument is of the correct type if it is checking types. If several types are possible, the called subprogram should assume a default as part of its specification. If none of the above conditions are true, the called subprogram should handle the argument as an integer (type 2).

2. Type 1 – FORTRAN logical
   A 36-bit binary value containing 0 or positive to specify .FALSE. and negative to specify .TRUE..

3. Type 2 – Integer and 1-word-COMP
   A 36-bit 2’s complement signed binary integer.

4. Type 4 – Real and COMP-1
   A 36-bit DECsystem-10 format floating point number.

   bit 0     sign
   bits 1–8  excess 128 exponent
   bits 9–35  mantissa

5. Type 6 – Label and procedure address
   A 36-bit unsigned binary value.

6. Type 7 Label and procedure address
   A 23-bit memory address.

   bits 0–12     always 0
   bit 13         indirect flag
   bits 14–17     0
   bits 18–35     the address

7. Type 10 – Double real
   A double precision floating point number for the CPU on which code is being executed (i.e., KA format on a KA10 processor and K1 format on a K110 processor).

8. Type 11 – 2-word COMP
   A 2-word (72-bit) 2’s complement signed binary integer.

   word 1, bit 0       sign
   word 1, bits 1–35   high order
   word 2, bit 0       same as word 1, bit 0
   word 2, bits 1–35   low order

9. Type 12 – Double octal
   A 72-bit unsigned binary value.
10. Type 14 – Complex

A complex number represented as an ordered pair of 36-bit floating point numbers. The first of which represents the real part and the second of which represents the imaginary part.

11. Type 15 – Byte String Descriptor

The format of the byte string descriptor is

\[
\begin{align*}
\text{word 1:} & \quad \text{ILDB-type pointer (i.e., aimed at the byte preceding the first byte of the string)} \\
\text{word 2:} & \quad \text{EXP byte count}
\end{align*}
\]

The byte descriptor may not be modified by the called program. The byte string itself must consist of a string of contiguous bytes of a uniform size. The byte size may be any number of bits from 1 to 36. The byte count must be large enough to encompass 256K words of storage, i.e., 24 bits for 1-bit bytes. (See COBOL Program Reference Manual.)

12. Type 17 – ASCIZ string

A string of contiguous 7-bit ASCII bytes left justified on the word boundary of the first word and terminated by a null byte in the last word. The length of the string may be from 1 to 256K words.

D.3.6 Converting Existing MACRO-10 Libraries for use with FORTRAN-10

The following simple example illustrates the FORTRAN-10 calling sequence.
AN EXAMPLE OF A CALL TO A SUBROUTINE WITH A VARIETY OF ARGUMENTS

DOUBLE PRECISION DP
DIMENSION B(10)

THE ARGUMENTS ARE:
1. A REAL VARIABLE
2. AN ARRAY NAME
3. AN ARRAY ELEMENT REFERENCE
4. AN INTEGER ELEMENT
5. A DOUBLE PRECISION VARIABLE
6. AN OCTAL CONSTANT
7. A LITERAL

CALL SUB1(A,B,B(1),KDP,"777","ABC")

END

SUBPROGRAMS CALLED
SUB1

SCALARS AND ARRAYS
KDP 1
B 2
A 14
I 15

TEMPORARIES
Q0000 16
<table>
<thead>
<tr>
<th>LINE</th>
<th>LOC</th>
<th>LABEL</th>
<th>GENERATED CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>JFCL</td>
<td>0,0</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>JSP</td>
<td>16,RESET.</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>0,0</td>
</tr>
<tr>
<td>1400</td>
<td>3</td>
<td>MOVE</td>
<td>2,I</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>MOVEI</td>
<td>2,B=1(2)</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>MOVEM</td>
<td>2,,Q0000</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>MOVEI</td>
<td>16,,2M</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>PUSHJ</td>
<td>17,,SUB1</td>
</tr>
<tr>
<td>1600</td>
<td>10</td>
<td>MOVEI</td>
<td>16,,1M</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>PUSHJ</td>
<td>17,,EXIT.</td>
</tr>
</tbody>
</table>

**ARGUMENT BLOCKS**

| 12   | 1M  | 0,,0 |
| 13   |     | 0,,0 |
| 14   |     | 777772,,0 |
| 15   | 2M  | 200,,A |
| 16   |     | 200,,B |
| 17   |     | 220,,Q0000 |
| 20   |     | 100,,KDP |
| 21   |     | [0000000000777] |
| 22   |     | [406050320100] |

**MAIN**: NO ERRORS DETECTED
SUBROUTINE SUB1(REAL1, ARYNAM, ARYELM, INT1, DBLPRC, OCT, LIT)

DOUBLE PRECISION DBLPRC

DIMENSION ARYNAM(10)

THIS SUBPROGRAM ILLUSTRATES THE USE AND MODIFICATION OF
SOME OF THE ARGUMENT TYPES

REAL1=ARYELM
O=ARYNAM(INT1)
OCT="777"
RETURN
END

SUBPROGRAMS CALLED

SCALARS AND ARRAYS

OCT 1 O 2 ARYELM 3 REAL1 4 INT1 5 ARYNAM 6

TEMPORARIES

October 1974
<table>
<thead>
<tr>
<th>LINE</th>
<th>LOC</th>
<th>LABEL</th>
<th>GENERATED CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td></td>
<td>SUB1:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td>636542,,210000</td>
</tr>
<tr>
<td>101</td>
<td></td>
<td></td>
<td>MOVE 0,0(16)</td>
</tr>
<tr>
<td>102</td>
<td></td>
<td></td>
<td>MOVE 7,REAL1</td>
</tr>
<tr>
<td>103</td>
<td></td>
<td></td>
<td>MOVE 2,0(16)</td>
</tr>
<tr>
<td>104</td>
<td></td>
<td></td>
<td>MOVE 7,ARYNAME</td>
</tr>
<tr>
<td>105</td>
<td></td>
<td></td>
<td>MOVE 2,ARYELM</td>
</tr>
<tr>
<td>106</td>
<td></td>
<td></td>
<td>MOVE 1,0(16)</td>
</tr>
<tr>
<td>107</td>
<td></td>
<td></td>
<td>MOVE 1,INT1</td>
</tr>
<tr>
<td>108</td>
<td></td>
<td></td>
<td>MOVE 1,0(16)</td>
</tr>
<tr>
<td>109</td>
<td></td>
<td></td>
<td>MOVE 1,0(16)</td>
</tr>
<tr>
<td>1200</td>
<td>12</td>
<td>3MI:</td>
<td></td>
</tr>
<tr>
<td>900</td>
<td>13</td>
<td></td>
<td>MOVEM 0,REAL1</td>
</tr>
<tr>
<td>901</td>
<td>14</td>
<td></td>
<td>MOVE 2,INT1</td>
</tr>
<tr>
<td>902</td>
<td>15</td>
<td></td>
<td>ADD 2,ARYNAME</td>
</tr>
<tr>
<td>903</td>
<td>16</td>
<td></td>
<td>MOVE 2,7777777(2)</td>
</tr>
<tr>
<td>1000</td>
<td>17</td>
<td></td>
<td>MOVEM 2,0</td>
</tr>
<tr>
<td>1201</td>
<td>21</td>
<td>2MI:</td>
<td></td>
</tr>
<tr>
<td>1202</td>
<td>22</td>
<td></td>
<td>MOVE 2,REAL1</td>
</tr>
<tr>
<td>1203</td>
<td>23</td>
<td></td>
<td>MOVEM 2,0(16)</td>
</tr>
<tr>
<td>1204</td>
<td>24</td>
<td></td>
<td>MOVE 0,0(16)</td>
</tr>
<tr>
<td>1205</td>
<td>25</td>
<td></td>
<td>MOVEM 2,0(16)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>POPJ 17,0</td>
</tr>
</tbody>
</table>

**ARGUMENT BLOCKS**

<table>
<thead>
<tr>
<th>LINE</th>
<th>LOC</th>
<th>LABEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td></td>
<td>2,3</td>
</tr>
<tr>
<td>27</td>
<td></td>
<td>1MI:</td>
</tr>
</tbody>
</table>

**SUB1** [ NO ERRORS DETECTED ]
To conveniently convert existing MACRO-10 programs so that they will still load and execute correctly when called from F40 or FORTRAN-10, the user can

1. transfer the initial entry sequence for a routine to

entry: CAIA
PUSH 17, CEXIT. ##

2. change all returns to POPJ 17,0

These are the functions performed by the HELLO and GOODBY macros. These macros (available in the file FORPRM.MAC, part of the FOROTS release) were successfully used to convert the library routines to run with both F40 and FORTRAN-10.

In addition, since the FORTRAN-10 compiler uses the indirect bits on argument lists (note that this permits shared, pure code argument lists), it is essential that code which accesses parameters takes this into account. Specifically, sequences that obtained the values of parameters through use of operations such as

HRRZ R,1(16)

to pick up the address of the second argument should be changed to

MOVEI R,@1(16)

This latter operation will work when interfacing to either F40 or FORTRAN-10.

Refer to the previous example which illustrates the code generated by the FORTRAN-10 compiler for specific details of how each argument is accessed. Note specifically that in the case of the formal array, it is the address of the array that is accessed.

D.3.7 Mixing FORTRAN-10 and F40 Compiled Programs
Starting with Version 1A of LINK-10, use of the switch /MIXFOR will permit loading FORTRAN-10 and F40 programs. This is achieved by modifying the code while it is loaded.

This introduces extra code that results in a degradation of the execution of programs so loaded. This feature is provided as a convenience for conversion. It is not intended to be used for other than conversion assistance.

D.3.8 Interaction with COBOL-10
The FORTRAN-10 programmer may call COBOL-10 programs as subprograms and, conversely, the COBOL programmers may call FORTRAN-10 programs as subprograms.

In either of the foregoing cases, I/O operation must not be performed in the called subprogram.
D.3.8.1 Calling FORTRAN-10 Subprograms from COBOL-10 Programs - COBOL programmers may write subprograms in FORTRAN-10 to utilize the conveniences and facilities provided by this language. The COBOL verb ENTER is used to call FORTRAN-10 subroutines. The form of ENTER is as follows:

\[
\text{ENTER FORTRAN program name [USING } \begin{array}{l}
\text{identifier-1} \\
\text{literal-1} \\
\text{procedure name-1}
\end{array} \begin{array}{l}
\text{identifier-2} \\
\text{literal-2} \\
\text{procedure-2}
\end{array} \ldots \]
\]

The USING clause of the foregoing forms moves the data within the COBOL program which is to be passed to the called FORTRAN-10 subprogram. The passed data must be in a form acceptable to FORTRAN-10.

The calling sequence used by COBOL in calling a FORTRAN-10 subprogram is:

MOVEI 16, address of first entry in argument list
PUSHJ 17, subprogram address

If the USING clause appears in the ENTER statement, an argument list is created which contains an entry for each identifier or literal in the order of appearance in the USING clause. It is preceded by a word containing, in its left half, the negative number of the number of entries in the list. If no USING clause is present, the argument list contains an empty word and the preceding word is set to 0. Each entry in the list is one 36-bit word at the form:

<table>
<thead>
<tr>
<th>0 — — — — — — 8</th>
<th>9 — — — — — — 12</th>
<th>13 — — — — — — 35</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>address</td>
<td></td>
</tr>
</tbody>
</table>

Bits 0–8 are always 0.

Bits 9–12 contain a type code that indicates the USAGE of the argument.

Bits 13–35 contain the address of the argument or the first word of the argument; the address can be indexed or indirect.

The types, their codes, how the codes appear in the argument list, and the locations specified by the addresses are described as follows:

a. For 1-word COMPUTATIONAL items
   CODE: 2
   IN ARGUMENT LIST: XWD 100, address
   ADDRESS: that of the argument itself

b. For 2-word COMPUTATIONAL items
   CODE: 11
   IN ARGUMENT LIST: XWD 440, address
   ADDRESS: that of the high-order word of the argument

c. For COMPUTATIONAL-1 items
   CODE: 4
   IN ARGUMENT LIST: XWD 200, address
   ADDRESS: that of the argument itself
d. For DISPLAY-6 and DISPLAY-7 items

| CODE:          | 15 |
| IN ARGUMENT LIST: | XWD 640, address |
| ADDRESS:      | that of a 2-word descriptor for the argument |
| WORD1:        | a byte pointer to the identifier or literal |
| WORD2:        | bit 0 is 1 if the item is numeric |
|               | bit 1 is 1 if the item is signed |
|               | bit 2 is 1 if the item is a figurative constant (including ALL) |
|               | bit 3 is 1 if the item is a literal |
|               | bits 4 through 11 are reserved for expansion |
|               | bit 12 is 1 if the item has a PICTURE with one or more Ps just before |
|               | the decimal point (e.g., 99PPV) |
|               | bits 13 through 17 are the number of decimal places. If bit 12 is 1, this |
|               | is the number of Ps. |
|               | bits 18 through 35 contain the size of the item in bytes. |

e. For procedure names (which cannot be used for calls to COBOL subprograms)

| CODE:          | 7 |
| IN ARGUMENT LIST: | XWD 340, address |
| ADDRESS:      | that of the procedure |

The return from a subprogram is POPJ 17 statement after call.

D.3.8.2 Calling COBOL-10 Subroutines from FORTRAN-10 Programs – To call COBOL subroutines use the standard subroutine calling Mechanism:

```
CALL COBOLS (args...) subroutine call
X=COBOLS (args...) function call
```

The COBOL subroutine must have been compiled using the COBOL compiler described in the DECsystem-10 COBOL Programmer's Reference Manual.

D.3.9 LINK-10 Overlay Facilities

LINK-10 provides several routines that are accessible directly from a FORTRAN-10 program. These routines are presented here briefly together with the FORTRAN-10 specifications of their parameters. In general the user will rely on the LINK-10 commands rather than these routines. They are available only for user convenience. Full details of the use of the overlay facilities can be found in the LINK-10 Reference Manual.

D.3.9.1 Conventions -- The following terms are used to describe the parameter to LINK-10 overlay routines.

- **File spec**: An ASCII string consisting of device: filename.ext [directory]
- **Name**: A LINK name or number which is a literal constant or variable.
- **List of link names**: A sequence of name items separated by commas.

The routines available are:

- **INIOVL**: (File spec) Used to specify the overlay file to be found if the load time specification is to be overridden.
- **GETOVL**: (list of link names) Used to change the overlay structure in core
- **RUNOVL**: (name) Loads the specified LINK and transfers to that LINK.

Version 4 FORTRAN-10 D-20

October 1974
REMOVVL  (List of link names) Remove the specified LINKs from core.

1. LOGOVVL  (File spec) Used to specify where the log file is to be written. If no arguments are given, the log file is closed.

For a full description of these routines, refer to the LINK-10 Reference Manual.

D.3.10 FOROTS/FORSE Compatibility

The information presented in Paragraphs D.3.10.1 and D.3.10.2 is intended only for those users who have programs and data files that were developed using the F40 FORTRAN compiler and the FORSE object time system. The manner in which both upward and downward compatibility between the FORTRAN-10, FOROTS and F40, FORSE FORTRAN systems may be achieved is described in the following sections.

D.3.10.1 FORTRAN-10/F40 Data File Compatibility – Upward compatibility of data files (FORSE TO FOROTS) is described in Table D-2. Downward compatibility of data files (FOROTS TO FORSE) is described in Table D-3.

Table D-2
Upward Compatibility (FORSE TO FOROTS)

<table>
<thead>
<tr>
<th>FORSE File Type</th>
<th>May be Read by FOROTS</th>
<th>In the Following Manner:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Sequential ASCII</td>
<td>Yes</td>
<td>May be read directly; record positioning operations (e.g., BACKSPACE, SKIP RECORD) may be used.</td>
</tr>
<tr>
<td>2. Sequential Binary</td>
<td>Yes</td>
<td>May be read directly in a forward fashion only, record positioning operations are not permitted.</td>
</tr>
<tr>
<td>3. Sequential Mixed Files</td>
<td>Yes</td>
<td>May be read directly in a forward fashion only, record positioning operations not permitted.</td>
</tr>
<tr>
<td>4. Random Access ASCII Files</td>
<td>No</td>
<td>NOTE: It is suggested that the random access file be read (using FORSE) and be rewritten as a sequential file that can be accepted by FOROTS.</td>
</tr>
<tr>
<td>5. Random Access Binary Files</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

D.3.10.2 Conversion of FOROTS-Developed Data Files Into a Form Acceptable to FORSE – The following paragraphs describe procedures that may be used to convert FOROTS sequential mixed, random access ASCII and random access binary data files into a form that can be read by FORSE.

D.3.10.2.1 Conversion of FOROTS Sequential Mixed Files – The following steps are suggested as a method of converting a FOROTS sequential mixed file into either a sequential ASCII or sequential binary file acceptable to FORSE.

1. Prepare and run a FORTRAN-10 I/O program that will produce either a sequential ASCII or a sequential binary output file.

2. If a sequential ASCII file is produced, it must be line-blocked before it can be read by FORSE. Line-blocking is accomplished by copying the file using either the system COPY command (with an A switch) or PIP. The copy will be line blocked and will be acceptable to FORSE. The following is an example of the command sequence needed to line-block the data file FOROT.DAT:

   .COPY FOROT.DAT = FOROT.DATA/A
3. If a sequential binary file is produced it must be record-blocked before it can be read by FORSE. Record-blocking is accomplished using the /K feature of the program BAKWDS. The following is an example of the command sequence needed to record-block the data file FOROT.DAT:

```
.R BAKWDS
*FOROT.DAT=FOROT.DAT/K
```

<table>
<thead>
<tr>
<th>FOROTS File Type</th>
<th>May be Read by FORSE</th>
<th>In the Following Manner:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Sequential ASCII File</td>
<td>Yes</td>
<td>This operation is permitted if the file is line-blocked. This may be accomplished by making a copy of the file using either the system copy command (with an A switch) or the PIP program. The resulting copy will be line-blocked. An example of the command sequence needed to line block a FOROTS file, using PIP, follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.R PIP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>*FOROTS.DAT=FOROTS.DAT/A</td>
</tr>
<tr>
<td>2. Sequential Binary File</td>
<td>Yes</td>
<td>This operation is permitted if the file is record-blocked. This type of blocking is accomplished by using the /K option of the program BAKWDS. The following is an example of a command sequence which record-blocks a file.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.R BAKWDS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>*FORSE.DAT=FOROTS.DAT/K</td>
</tr>
<tr>
<td>3. Sequential Mixed File</td>
<td>No</td>
<td>(See Paragraph D.3.10.2.1 for suggested conversion procedure.)</td>
</tr>
<tr>
<td>4. Random Access ASCII File</td>
<td>No</td>
<td>(See Paragraph D.3.10.2.2 for suggested conversion procedure.)</td>
</tr>
<tr>
<td>5. Random Access Binary File</td>
<td>No</td>
<td>(See Paragraph D.3.10.2.3 for suggested conversion procedure.)</td>
</tr>
</tbody>
</table>

D.3.10.2.2 Conversion of FOROTS Random Access ASCII Files – The following procedure is suggested as a method of converting a FOROTS random access ASCII file into a form acceptable to FORSE.

1. Prepare and run a FORTRAN-10 I/O program that will create a sequential ASCII file comprised of the records of the random access file.
2. Line-block the sequential ASCII file using either the system COPY command (with an A switch) or the PIP program. The following is an example of the COPY command:

```
.COPY LNBK.DAT=SEQFL.DAT/A
```

The foregoing command would produce a line-blocked copy (LNBK.DAT) of the sequential file SEQFL.DAT.

3. Prepare and run an F40 I/O program that will read the file produced in step 2 and will rewrite the file as a FORSE generated random access file.

D.3.10.2.3 Conversion of FOROTS Random Access Binary Files – The following procedure is suggested as a method of converting a FOROTS random access binary file into a form acceptable to FORSE.

1. Prepare and run a FORTRAN-10 I/O program that will create a sequential binary file comprised of the records of the random access file.

2. Record-block the sequential file. This is accomplished by using the /K feature of the program BAKWDS. The following example illustrates the command sequence required to convert the file FOROTS.DAT into the record-blocked file FORBLK.DAT.

```
.R BAKWDS
*FORBLK.DAT=FOROTS.DAT/K
```

3. An F40 I/O program may then be written to convert the sequential record-blocked file into a FORSE generated random access file.

D.3.10.3 General Restrictions – The following restriction must be observed during the preparation of FORTRAN-10 programs and data files:

CHAIN functions (as implemented for the F40 compiler) are not implemented in FORTRAN-10. An overlay capability which is greatly superior to CHAIN is available with LINK-10 version 2.
APPENDIX E
FOROTS

This appendix describes the facilities FOROTS provides for the FORTRAN user. FOROTS implements all standard FORTRAN I/O operations as set forth in the "American National Standard FORTRAN, ANSI X3.9-1966." In addition it provides the user with capabilities and programming features beyond those defined in the ANSI standard.

The primary function of FOROTS is to act as a direct interface between user object programs and the DECSYSTEM-10 monitor during input and output operations. Other capabilities include

1. Job initialization
2. Channel and core management
3. Error handling and reporting
4. File management
5. Formatting of data
6. Mathematical library
7. User library (non-mathematical)
8. Specialized applications packages
9. Overlay facilities
10. F40 compatibility

E.1 HARDWARE AND SOFTWARE REQUIREMENTS

FOROTS can be run using either a DECSYSTEM-10 KA10 or K110 Processor, and it may interface with all DECSYSTEM-10 peripheral devices. In addition to monitor or user program requirements, a minimum of 13 pages of user core is needed to run FOROTS.

The software required with FOROTS is the 5.06 monitor or a later version. Other software items that can be associated with FOROTS include

1. The MACRO-10 assembler (version 47 or later)
2. The LINK10 loader (version 1A or later)
3. The system program COMPIL (version 22 or later) and
4. The FORTRAN-10 compiler (version 1 or later)

E.2 FEATURES OF FOROTS

Many specific user features are described briefly in the following list; more detailed information concerning the implementation of these features is given later in this appendix.

1. A user program may run in either batch or timesharing mode without changing the program. All differences between batch mode and timesharing mode operations are resolved by FOROTS.

2. User programs may access both directory and non-directory devices in the same manner.
3. FOROTS helps provide complete data file compatibility between all DECsystem-10 devices.

4. FOROTS does not require line-blocking (a requirement that each output buffer must contain only an integral number of lines).

5. Up to 15 data files may be accessed simultaneously. Any number or all of the open data files may be accessed randomly.

6. FOROTS treats devices located at remote stations similarly to local devices.

7. Programs written for magnetic tape operations will run correctly on disk under FOROTS supervision. FOROTS simulates the commands needed for magnetic tape operations.

8. Object program device and file specifications may be changed or specified via a FOROTS interactive dialogue mode.

9. Non-FORTRAN binary data files may be read in image mode by FOROTS.

10. FOROTS provides interactive program/operating system error processing routines. These routines permit the user to route the execution of the program to specific error processing routines whenever designated types of errors are detected.

11. An error traceback facility for fatal errors provides a history of all subprogram calls made back to the main program together with the address of the point at which the error occurred.

12. FOROTS provides a trap handling system for arithmetic functions, including default values and error reports.

13. ASCII and binary records may be mixed in the same file and both may be accessed in either sequential or random access mode.

14. FOROTS permits the user program to switch from READ to WRITE on the same I/O device without loss of data or buffering.

15. Although primarily designed for use with the FORTRAN-10 compiler, FOROTS may also be used as an independent I/O system. FOROTS may be used as an I/O system for MACRO-10 object programs as well as for FORTRAN-10 and F40 object programs.

E.3 ERROR PROCESSING
Whenever a run-time error is detected, the FOROTS error processing system takes control of program execution. This system determines the class of the error and either outputs an appropriate message to the controlling user terminal or branches the program to a predesignated processing routine.

E.4 INPUT/OUTPUT FACILITIES
FOROTS utilizes monitor-buffered I/O during all modes of access, except DUMP mode. Display devices are supported in dump mode; formatted text is handled in ASCII line mode; unformatted files are accessed as FORTRAN binary files. (Refer to DECsystem-10 Monitor Calls Manual.)

I/O data channel and access modes are individually described in the following paragraphs.
E.4.1 Input/Output Channels Used Internally by FOROTS

Fifteen software channels (1–15) are available in I/O operations. Software channel 0 is reserved for the following system functions:

1. The printing of error messages, and
2. The loading and initialization of FOROTS (GETSEG UUO operations)

Software channels 1 through 15 are available for user program data transfer operations. When a request is made for a data channel, a table is scanned until a free channel is found. The first free channel is assigned to the requesting program; on completion of the assigned transfer, control of the software channel is returned to FOROTS.

E.4.2 File Access Modes

Data may be transferred between processor storage and peripheral devices in two major modes — sequential, and random.

E.4.2.1 Sequential Transfer Mode — In sequential data transfer operations, the records involved are transferred in the same order as they appear in the source file. Each I/O statement executed in this mode transfers the record immediately following the last record transferred from the accessed source file. A special version of the sequential mode (referred to as Append) is available for output (write) operations. The special Append mode permits the user to write a record immediately after the last logical record of the accessed file. During the Append operation, the records already in the accessed file remain unchanged; the only function performed is the appending of the transferred records to the end of the file.

Transfer modes (other than SEQINOUT) must be specified by setting the ACCESS option of a FORTRAN-10 OPEN statement to one of several possible arguments. For the sequential mode, the arguments are

ACCESS = 'SEQIN' (sequential read-only mode)
ACCESS = 'SEQOUT' (sequential write-only mode)
ACCESS = 'SEQINOUT' (sequential read followed by a sequential write)
ACCESS = 'APPEND' (sequential Append mode)

E.4.2.2 Random Access Mode — This transfer mode permits records to be accessed and transferred from a source file in any desired order. Random access transfers must be made between processor core and a device that permits random addressing operations (i.e., disk) to files that have been set up for random access. Files for random access must contain a specified number of identically-sized records which may be individually accessed by a record number.

Random access transfers may be carried out in either a read/write mode or a special read-only mode. Random transfer modes must be specified by setting the ACCESS option of an OPEN statement to one of several possible arguments.

ACCESS = 'RANDOM' (random read/write mode)
ACCESS = 'RANDIN' (random special read-only mode)

E.5 ACCEPTABLE TYPES OF DATA FILES AND THEIR FORMATS

The types of data files that are acceptable to FOROTS are individually described in the following paragraphs.
E.5.1 ASCII Data Files

Each record within an ASCII data file consists of a set of contiguous 7-bit characters; each set is terminated by a vertical paper-motion character (i.e., Form Feed, Vertical Tab, or Line Feed). All ASCII records start on a word boundary; the last word in a record is padded with nulls, if necessary, to ensure that the record also ends on a word boundary. Logical records may be split across physical blocks. There is no implied maximum length for logical records.

NOTE
On sequential input, FOROTS does not require conformation to word boundaries; it reads what it sees; therefore, any file that is written by FOROTS will conform to the foregoing format requirements.

E.5.2 FORTRAN Binary Data Files

Each logical record in a FORTRAN binary data file contains data that may be referred to by either a READ or WRITE statement in the program being executed. A logical record is preceded by a control word and may have one or more control words embedded within it. In FORTRAN binary data files, there is no relationship between logical records and physical device block sizes. There is no implied maximum length for logical records.

E.5.2.1 Format of Binary Files – A FOROTS binary file may contain three forms of Logical Segment Control Words (LSCW). These LSCWs give FOROTS the ability to distinguish ASCII files from binary files.

<table>
<thead>
<tr>
<th>LSCW</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>START</td>
<td>001+ the number of words in the segment (exclusive of any &quot;END&quot; LSCWs)</td>
</tr>
<tr>
<td>CONTINUE</td>
<td>002 indicates that the segment of a disk block boundary continues</td>
</tr>
<tr>
<td>END</td>
<td>003+ number of words in the preceding segment including LSCWs.</td>
</tr>
</tbody>
</table>

If the access specified for a file (through the OPEN statement ACCESS = parameter) is 'SEQIN', 'SEQOUT', or 'SEQINOUT', all three LSCWs may appear in a record. If the access specified is 'RANDIN', or 'RANDOM', all records are of the same length, and there are no CONTINUE LSCWs.

The following examples illustrate the LSCW. The random access binary file contains only 001 and 003 LSCW's.

```fortran
C LOOK AT A BINARY FILE AND SEE THE LOGICAL SEGMENT
C CONTROL WORDS.

OPEN(UNIT=1, ACCESS='RANDOM', MODE='BINARY', RECOR=100)
I=5
WRITE(1,1) (I, J=1,100)
J=7
WRITE(1,2) (J, K=1,100)
END
```
Number of words in record counting
END LSCW or the
number of words
following this
word to the
END LSCW.

--END LSCW

Containing the
number of words
in the record
including LSCW's.
In the sequential access binary file the second record crosses the 128-word disk boundary and contains a 002 (CONTINUE) LSCW.

**C**

Look at a binary file and see the logical segment control words.

```fortran
OPEN(UNIT=1,MODE='BINARY')
```

```fortran
I=5
WRITE(1) (I,J=1,100)
```

```fortran
J=7
WRITE(1) (J,K=1,100)
END
```

<table>
<thead>
<tr>
<th>00.000</th>
<th>00100</th>
<th>00145</th>
<th>00.000</th>
<th>00100</th>
<th>00005</th>
</tr>
</thead>
<tbody>
<tr>
<td>00.001</td>
<td>00200</td>
<td>00005</td>
<td>00.004</td>
<td>00200</td>
<td>00005</td>
</tr>
<tr>
<td>00.002</td>
<td>00200</td>
<td>00005</td>
<td>00.005</td>
<td>00200</td>
<td>00005</td>
</tr>
<tr>
<td>00.003</td>
<td>00200</td>
<td>00005</td>
<td>00.006</td>
<td>00200</td>
<td>00005</td>
</tr>
<tr>
<td>00.004</td>
<td>00200</td>
<td>00005</td>
<td>00.007</td>
<td>00200</td>
<td>00005</td>
</tr>
<tr>
<td>00.010</td>
<td>00300</td>
<td>00005</td>
<td>00.011</td>
<td>00300</td>
<td>00005</td>
</tr>
<tr>
<td>00.012</td>
<td>00300</td>
<td>00005</td>
<td>00.013</td>
<td>00300</td>
<td>00005</td>
</tr>
<tr>
<td>00.014</td>
<td>00300</td>
<td>00005</td>
<td>00.015</td>
<td>00300</td>
<td>00005</td>
</tr>
<tr>
<td>00.016</td>
<td>00300</td>
<td>00005</td>
<td>00.017</td>
<td>00300</td>
<td>00005</td>
</tr>
<tr>
<td>00.020</td>
<td>00300</td>
<td>00005</td>
<td>00.021</td>
<td>00300</td>
<td>00005</td>
</tr>
<tr>
<td>00.022</td>
<td>00300</td>
<td>00005</td>
<td>00.023</td>
<td>00300</td>
<td>00005</td>
</tr>
<tr>
<td>00.024</td>
<td>00300</td>
<td>00005</td>
<td>00.025</td>
<td>00300</td>
<td>00005</td>
</tr>
<tr>
<td>00.026</td>
<td>00300</td>
<td>00005</td>
<td>00.027</td>
<td>00300</td>
<td>00005</td>
</tr>
<tr>
<td>00.030</td>
<td>00300</td>
<td>00005</td>
<td>00.031</td>
<td>00300</td>
<td>00005</td>
</tr>
<tr>
<td>00.032</td>
<td>00300</td>
<td>00005</td>
<td>00.033</td>
<td>00300</td>
<td>00005</td>
</tr>
<tr>
<td>00.034</td>
<td>00300</td>
<td>00005</td>
<td>00.036</td>
<td>00300</td>
<td>00005</td>
</tr>
<tr>
<td>00.035</td>
<td>00300</td>
<td>00005</td>
<td>00.037</td>
<td>00300</td>
<td>00005</td>
</tr>
<tr>
<td>00.038</td>
<td>00300</td>
<td>00005</td>
<td>00.041</td>
<td>00300</td>
<td>00005</td>
</tr>
<tr>
<td>00.042</td>
<td>00300</td>
<td>00005</td>
<td>00.043</td>
<td>00300</td>
<td>00005</td>
</tr>
</tbody>
</table>

Version 4 FORTRAN-10
Version 4 FOROTS

October 1974
<table>
<thead>
<tr>
<th>00106</th>
<th>00000</th>
<th>00005</th>
</tr>
</thead>
<tbody>
<tr>
<td>00107</td>
<td>00000</td>
<td>00005</td>
</tr>
<tr>
<td>00110</td>
<td>00000</td>
<td>00005</td>
</tr>
<tr>
<td>00111</td>
<td>00000</td>
<td>00005</td>
</tr>
<tr>
<td>00112</td>
<td>00000</td>
<td>00005</td>
</tr>
<tr>
<td>00113</td>
<td>00000</td>
<td>00005</td>
</tr>
<tr>
<td>00114</td>
<td>00000</td>
<td>00005</td>
</tr>
<tr>
<td>00115</td>
<td>00000</td>
<td>00005</td>
</tr>
<tr>
<td>00116</td>
<td>00000</td>
<td>00005</td>
</tr>
<tr>
<td>00117</td>
<td>00000</td>
<td>00005</td>
</tr>
<tr>
<td>00120</td>
<td>00000</td>
<td>00005</td>
</tr>
<tr>
<td>00121</td>
<td>00000</td>
<td>00005</td>
</tr>
<tr>
<td>00122</td>
<td>00000</td>
<td>00005</td>
</tr>
<tr>
<td>00123</td>
<td>00000</td>
<td>00005</td>
</tr>
<tr>
<td>00124</td>
<td>00000</td>
<td>00005</td>
</tr>
<tr>
<td>00125</td>
<td>00000</td>
<td>00005</td>
</tr>
<tr>
<td>00126</td>
<td>00000</td>
<td>00005</td>
</tr>
<tr>
<td>00127</td>
<td>00000</td>
<td>00005</td>
</tr>
<tr>
<td>00130</td>
<td>00000</td>
<td>00005</td>
</tr>
<tr>
<td>00131</td>
<td>00000</td>
<td>00005</td>
</tr>
<tr>
<td>00132</td>
<td>00000</td>
<td>00005</td>
</tr>
<tr>
<td>00133</td>
<td>00000</td>
<td>00005</td>
</tr>
<tr>
<td>00134</td>
<td>00000</td>
<td>00005</td>
</tr>
<tr>
<td>00135</td>
<td>00000</td>
<td>00005</td>
</tr>
<tr>
<td>00136</td>
<td>00000</td>
<td>00005</td>
</tr>
<tr>
<td>00137</td>
<td>00000</td>
<td>00005</td>
</tr>
<tr>
<td>00140</td>
<td>00000</td>
<td>00005</td>
</tr>
<tr>
<td>00141</td>
<td>00000</td>
<td>00005</td>
</tr>
<tr>
<td>00142</td>
<td>00000</td>
<td>00005</td>
</tr>
<tr>
<td>00143</td>
<td>00000</td>
<td>00005</td>
</tr>
<tr>
<td>00144</td>
<td>00000</td>
<td>00005</td>
</tr>
<tr>
<td>00145</td>
<td>00000</td>
<td>00146</td>
</tr>
<tr>
<td>00146</td>
<td>01000</td>
<td>00032</td>
</tr>
<tr>
<td>00147</td>
<td>00000</td>
<td>00007</td>
</tr>
<tr>
<td>00150</td>
<td>00000</td>
<td>00007</td>
</tr>
<tr>
<td>00151</td>
<td>00000</td>
<td>00007</td>
</tr>
<tr>
<td>00152</td>
<td>00000</td>
<td>00007</td>
</tr>
<tr>
<td>00153</td>
<td>00000</td>
<td>00007</td>
</tr>
<tr>
<td>00154</td>
<td>00000</td>
<td>00007</td>
</tr>
<tr>
<td>00155</td>
<td>00000</td>
<td>00007</td>
</tr>
<tr>
<td>00156</td>
<td>00000</td>
<td>00007</td>
</tr>
<tr>
<td>00157</td>
<td>00000</td>
<td>00007</td>
</tr>
<tr>
<td>00160</td>
<td>00000</td>
<td>00007</td>
</tr>
<tr>
<td>00161</td>
<td>00000</td>
<td>00007</td>
</tr>
<tr>
<td>00162</td>
<td>00000</td>
<td>00007</td>
</tr>
<tr>
<td>00163</td>
<td>00000</td>
<td>00007</td>
</tr>
<tr>
<td>00164</td>
<td>00000</td>
<td>00007</td>
</tr>
<tr>
<td>00165</td>
<td>00000</td>
<td>00007</td>
</tr>
<tr>
<td>00166</td>
<td>00000</td>
<td>00007</td>
</tr>
<tr>
<td>00167</td>
<td>00000</td>
<td>00007</td>
</tr>
<tr>
<td>00170</td>
<td>00000</td>
<td>00007</td>
</tr>
<tr>
<td>00171</td>
<td>00000</td>
<td>00007</td>
</tr>
<tr>
<td>00172</td>
<td>00000</td>
<td>00007</td>
</tr>
</tbody>
</table>

E-8

October 1974
```
C LOOK AT AN IMAGE MODE FILE AND SEE NO LOGICAL SEGMENT
   CONTROL WORDS.
OPEN(UNIT=1,MODE='IMAGE')
I=5
WRITE(1) (I,J=1:100)
J=7
WRITE(1) (J,K=1:100)
END
```

Image mode files contain no LSCW's. This file cannot be backspaced.
E.5.3 Mixed Mode Data Files

FOROTS permits files containing both ASCII and binary data records to be read. Mixed files may be accessed in either sequential or random access mode. Logical ASCII and binary records have the same format as described in the preceding paragraphs. In random access mode, the record size must be large enough to contain the largest record either ASCII or binary.

E.5.4 Image Files

The image data transfer mode is a buffered mode in which data is transferred in a blocked format consisting of a word count located in the right half of the first data word of the buffer followed by the number of 36-bit data words. The devices which permit image data transfers and the form in which the data is read or written are:

<table>
<thead>
<tr>
<th>Device</th>
<th>Data Forms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Card Punch</td>
<td>In image mode, each buffer contains three 12-bit bytes. Each byte corresponds to one card column. Since there is room for 81 columns in the buffer (3 × 27) and there are only 80 columns on a card, the last word contains only 2 bytes of data; the third byte is thrown away. Image mode causes exactly one card to be punched for each output. The CLOSE punches the last partial card and then punches an EOF card.</td>
</tr>
</tbody>
</table>
Device | Data Forms
---|---
Card Reader | All 12 punches in all 80 columns are packed into the buffer as 12-bit bytes. The first 12-bit byte contains column 1. The last word of the buffer contains columns 79 and 80 as the left and middle bytes, respectively. Cards are not split between two buffers.

Disk | Data is written on the disk exactly as it appears in the buffer. Data consists of 36-bit words.

Magnetic Tape | Data appears on magnetic tape exactly as it appears in the buffer. No processing or checksumming of any kind is performed by the service routine. The parity checking of the magnetic tape system is sufficient assurance that the data is correct. Normally, all data, both binary and ASCII, is written with odd parity and at 800 bits per inch unless changed by the installation.

Paper Tape Punch | Binary words taken from the output buffer are split into six 6-bit bytes and punched with the eighth hole punched in each frame. There is no format control or checksumming performed by the I/O routine. Data punched in this mode is read back by the paper-tape reader in the same mode.

Paper Tape Reader | Characters not having the eighth hole punched are ignored. Characters are truncated to six bits and packed six to the word without further processing. This mode is useful for reading binary tapes having arbitrary blocking format.

Plotter | Six 6-bit characters per word are transmitted to the plotter exactly as they appear in the buffer.

E.6 USING FOROTS

FOROTS has been designed to lend itself for use as an I/O system for programs written in languages other than FORTRAN. Currently, MACRO programmers may employ FOROTS as a general I/O system by writing simple MACRO calls which simulate the calls made to FOROTS by a FORTRAN compiler. The calls made to FOROTS are to routines that implement FORTRAN I/O statements such as READ, WRITE, OPEN, CLOSE, RELEASE, etc.

FOROTS will provide automatic memory allocation, data conversion, I/O buffering, and device interface operations to the MACRO user.

E.6.1 FOROTS Entry Points

FOROTS provide the following entry points for calls from either a FORTRAN compiler or a non-FORTRAN program:

<table>
<thead>
<tr>
<th>Entry Point</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALCRN.</td>
<td>Allocate software channels</td>
</tr>
<tr>
<td>ALCOR.</td>
<td>Allocate dynamic core blocks</td>
</tr>
<tr>
<td>CLOSE.</td>
<td>Close a file</td>
</tr>
<tr>
<td>DEC.</td>
<td>DECODE routine</td>
</tr>
<tr>
<td>DECHN.</td>
<td>De-allocate software channels</td>
</tr>
<tr>
<td>DECOR.</td>
<td>De-allocate dynamic core blocks</td>
</tr>
<tr>
<td>ENC.</td>
<td>ENCODE routine</td>
</tr>
<tr>
<td>EXIT.</td>
<td>Terminate program execution</td>
</tr>
<tr>
<td>FIN.</td>
<td>Input/Output list termination routine</td>
</tr>
<tr>
<td>FIND.</td>
<td>Position to the next record (RANDOM ACCESS)</td>
</tr>
<tr>
<td>FORER.</td>
<td>Error processor</td>
</tr>
<tr>
<td>IN.</td>
<td>Formatted input routine</td>
</tr>
</tbody>
</table>

E-12 October 1974
<table>
<thead>
<tr>
<th>Entry Point</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOLST.</td>
<td>Input/Output list routine</td>
</tr>
<tr>
<td>MTOP.</td>
<td>File utility processing routine</td>
</tr>
<tr>
<td>NLI.</td>
<td>NAMELIST input routine</td>
</tr>
<tr>
<td>NLO.</td>
<td>NAMELIST output routine</td>
</tr>
<tr>
<td>OPEN.</td>
<td>Open a file</td>
</tr>
<tr>
<td>OUT.</td>
<td>Formatted output routine</td>
</tr>
<tr>
<td>RELEA.</td>
<td>Release a device (CLOSE implied)</td>
</tr>
<tr>
<td>RESET.</td>
<td>Job initialization entry</td>
</tr>
<tr>
<td>RTB.</td>
<td>Binary input routine</td>
</tr>
<tr>
<td>TRACE.</td>
<td>Trace subroutine calls</td>
</tr>
<tr>
<td>WTB.</td>
<td>Binary output routine</td>
</tr>
</tbody>
</table>

### E.6.2 Calling Sequences

All calls made to FOROTS must be made using the following general form:

```
MOVEJ 16, ARGBLK
PUSHJ 17, Entry Point
        (control is returned here)
```

where:

a. ARGBLK is the address of a specifically formatted argument block which contains information needed by FOROTS to accomplish the desired operation.

b. Entry Point is an entry point identifier (see list given in Paragraph E.6.1) which specifies the entry point of the desired FOROTS routine.

With three exceptions, all returns from FOROTS will be made to the program instruction immediately following the call (PUSHJ 17, entry point instruction). The exceptions are:

a. An error return to a specified statement number (i.e., READ or WRITE statement ERR= option),

b. An end-of-file return to a statement number (i.e., READ or WRITE statement END= option),

c. A fatal error which returns to the monitor or to a debug package.

Paragraphs E.6.3.1 through E.6.3.11 give the MACRO calls and required argument block formats needed to initialize FOROTS and FOROTS I/O operations.

Argument blocks conform to the subprogram calling convention described in Appendix D. However, there is one exception in dealing with the first word of an I/O initialization call (i.e., WTB., ENC., RTW, etc.) for a FORTRAN logical unit number. In previous versions of FOROTS and FORTRAN-10, if the indirect bit was not set, the argument was immediate; if it was set to 1 (one), the argument was the address of the variable. The type field was always 0 (zero).

With Version 4 of FORTRAN-10 and Version 4 of FOROTS this convention has been changed. If the type field of the first word of an I/O initialization call for the FORTRAN logical unit number is 0 (zero), the argument is an immediate mode (18 bit) constant wherever possible. If the type field is integer, the argument is indirect (see Appendix D, Table D-2, Type 2).
This exception should not cause any upward compatibility problems since all previously working programs will still function. An added feature with this convention is that it permits the following construct to be correctly implemented:

```
N = -4
READ (N,100) I,J
100 FORMAT(2I5)
```

### E.6.3 MACRO Calls for FOROTS Functions

The forms of the MACRO calls to FOROTS that are made by the FORTRAN-10 compiler are described in the following paragraphs. The calls described are identified according to the language statement which they implement. The following terms and abbreviations may be used in the description of the argument block (ARGBLK) of each call:

- $\rightarrow$ = pointer to the second word in the argument block (This is the address pointed to by the argument ARGBLK in the calling sequence),
- $n$ = count of ASCII characters,
- $f$ = FORMAT statement address,
- $v$ = the name of an array containing ASCII characters,
- $\text{list}$ = an Input/Output list,
- $c$ = the statement to which control is transferred on an "END OF FILE" condition,
- $d$ = the statement to which control is transferred on an "ERROR" condition,
- $\text{name}$ = a NAMELIST name,
- $R$ = a variable specifying the logical record number for random access mode,
- * = list directed I/O; the FORMAT statement is not used,
- $\text{type}$ = type specification of a variable or constant,

where ARGBLK is

<table>
<thead>
<tr>
<th>0</th>
<th>9</th>
<th>13</th>
<th>14</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>-6</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Reserved</td>
<td>2</td>
<td>I</td>
<td>X</td>
<td>n</td>
</tr>
<tr>
<td>7</td>
<td>I</td>
<td>X</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>I</td>
<td>X</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>I</td>
<td>X</td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>I</td>
<td>X</td>
<td>Format Size (in words)</td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>0</td>
<td>I</td>
<td>X</td>
<td>v</td>
</tr>
</tbody>
</table>

Version 4 FORTRAN-10
Version 4 FOROTS

E-14
October 1974
E.6.3.1 Formatted/Unformatted Transfer Statements, Sequential Access Calling Sequences — The READ and WRITE statements for formatted sequential data transfer operations and their calling sequences are:

```plaintext
READ (u, END=c, ERR=d) list
MOVEI 16, ARGBLK
PUSHJ 17, IN.
```

and

```plaintext
WRITE (u, END=c, ERR=d) list
MOVEI 16, ARGBLK
PUSHJ 17, OUT.
```

where ARGBLK is

<table>
<thead>
<tr>
<th>0</th>
<th>9</th>
<th>13</th>
<th>14</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>-5</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Reserved</td>
<td>2</td>
<td>I</td>
<td>X</td>
<td>u</td>
</tr>
<tr>
<td>7</td>
<td>I</td>
<td>X</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>I</td>
<td>X</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>I</td>
<td>X</td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>2</td>
<td>I</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

The READ and WRITE statements for unformatted sequential data transfer operations and their calling sequences are:

```plaintext
READ (u, END=c, ERR=d) list
MOVEI 16, ARGBLK
PUSHJ 17, RTB.
```

and

```plaintext
WRITE (u, END=c, ERR=d) list
MOVEI 16, ARGBLK
PUSHJ 17, WTB.
```

where ARGBLK is

<table>
<thead>
<tr>
<th>0</th>
<th>9</th>
<th>13</th>
<th>14</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>-3</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Reserved</td>
<td>2</td>
<td>I</td>
<td>X</td>
<td>u</td>
</tr>
<tr>
<td>7</td>
<td>I</td>
<td>X</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>7</td>
<td>I</td>
<td>X</td>
<td>d</td>
</tr>
</tbody>
</table>
E.6.3.2 NAMELIST Data Transfer Statements, Sequential Access Calling Sequences – The READ and WRITE statements for namelist-directed sequential data transfer operations and their calling sequences are:

READ (u, name)
READ (u, name, END=c, ERR=d)

MOVEI 16, ARGBLK
PUSHJ 17, NLI.

and

WRITE (u, name)
WRITE (u, name, END=c, ERR=d)

MOVEI 16, ARGBLK
PUSHJ 17, NLO.

where ARGBLK is

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>9</th>
<th>13</th>
<th>14</th>
<th>18</th>
<th>35</th>
</tr>
</thead>
<tbody>
<tr>
<td>-4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Reserved</td>
<td>2</td>
<td>I</td>
<td>X</td>
<td></td>
<td>u</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>I</td>
<td>X</td>
<td></td>
<td>c</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>I</td>
<td>X</td>
<td></td>
<td>d</td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>0</td>
<td>I</td>
<td>X</td>
<td></td>
<td></td>
<td>Namelist table addr.</td>
</tr>
</tbody>
</table>

The NAMELIST table is generated from the FORTRAN NAMELIST. The first word of the table is the NAMELIST name; following that are a number of 2-word entries for scalar variables, and a number of \((N+3)\) word entries for array variables, where \(N\) is the dimensionality of the array.

The names specified in the NAMELIST statement are stored, in SIXBIT form, first in the table. Each name is followed by a list of arguments associated with the name; this argument list may be of any length and is terminated by a zero entry. The name argument list may be in either a scalar or an array form (refer to the following diagrams).

E.6.3.3 Array Offsets and Factoring – Address calculations used to reference a given array element involve factors and offsets. For example:

Array A is dimensioned

\[ \text{DIMENSION A (L1/U1,L2/U2,L3/U3, \ldots ,Ln/Un)} \]

The size of each dimension is represented by

\[ S1 = U1-L1+1 \]
\[ S2 = U2-L2+1 \]
\[ \text{etc.} \]

Version 4 FORTRAN-10
Version 4 FOROTS

E-16

October 1974
In order to calculate the address of an element referenced by

A (I1, I2, I3, ... In)

the following formula is used:


\[ A + (I1-L1) + (I2-L2)S1 + (I3-L3)S2S1 + \ldots + (In-Ln)S[n-1]S[n-2]S[n-3] \]

The terms are factored out depending on the dimensions of the array and not on the element referenced to arrive at the formula


\[ A + (-L1-L2S1-L3S2S1 \ldots )I1 + I2S1 + I3S2S1 \ldots \]

The parenthesized part of this formula is the offset for a single precision array and is referred to as the Array Offset.

For each dimension of a given array, there is a corresponding factor by which a subscript in that position will be multiplied. From the last expression, one can determine the factor for dimension n to be

\[ S[n-1]S[n-2] \ldots S2S1 \]

For double precision and complex arrays, the expression becomes

\[ A + 2*(I1-L1) + 2*(I2-L2)S1 + 2*(I3-L3)S2S1 + \ldots \]

Therefore, the array offset for a double precision array is

\[ 2*(-L1-L2S1-L3S2S1 \ldots ) \]

and the factor for the nth dimension is

\[ 2*S[n-1]S[n-2] \ldots S2S1 \]

The factor for the first dimension of a double precision array is always 2. The factor for the first dimension of a single precision array is always 1.

**SCALAR ENTRY in a NAMELIST Table**

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9</td>
<td>12</td>
<td>15</td>
<td>18</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>SIXBIT/SCALAR NAME/</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td></td>
<td></td>
<td>Scalar addr</td>
</tr>
</tbody>
</table>

Version 4 FORTRAN-10
Version 4 FOROTS
E-17
October 1974
ARRAY ENTRY in a NAMELIST Table

<table>
<thead>
<tr>
<th>#DIMS</th>
<th>type</th>
<th>I</th>
<th>X</th>
<th>OFFSET</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARRAY SIZE</td>
<td></td>
<td>I</td>
<td>X</td>
<td>Factor 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>I</td>
<td>X</td>
<td>Factor 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>I</td>
<td>X</td>
<td>Factor 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>I</td>
<td>X</td>
<td>Factor n</td>
</tr>
</tbody>
</table>

E.6.3.4 Formatted/Unformatted Data Transfer Statements, Random Access Calling Sequences – The READ and WRITE statements for random access data transfer operations and their calling sequences are:

READ (u#R, f, END=c, ERR=d) list
MOVEI 16, ARGBLK
PUSHJ 17, RTB.

and

WRITE (u#R, f, END=c, ERR=d) list
MOVEI 16, ARGBLK
PUSHJ 17, WTB.

where ARGBLK is

<table>
<thead>
<tr>
<th>0</th>
<th>9</th>
<th>13</th>
<th>14</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-6</td>
<td>u</td>
<td>c</td>
<td>d</td>
</tr>
</tbody>
</table>

Reserved 2 I X address
7 I X Record Number
7 I X
6 0
0 0
Reserved 2 I X
E.6.3.5 Calling Sequences for Statements Which Use Default Devices — The FORTRAN-10 statements that require the use of a reserved system default device and their calling sequences are:

**Default Device**

<table>
<thead>
<tr>
<th>Function</th>
<th>UNIT</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCEPT f, list</td>
<td>-4</td>
<td>(TTY)</td>
</tr>
<tr>
<td>READ f, list</td>
<td>-5</td>
<td>(CDR)</td>
</tr>
<tr>
<td>REREAD f, list</td>
<td>-6</td>
<td>(REREAD)</td>
</tr>
</tbody>
</table>

MOVEI 16, ARGBLK
PUSHJ 17, IN.

where ARGELK is

<table>
<thead>
<tr>
<th>0</th>
<th>9</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>17</th>
<th>18</th>
<th>35</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>-5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>2</td>
<td>I</td>
<td>X</td>
<td>u</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>7</td>
<td>I</td>
<td>X</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>7</td>
<td>I</td>
<td>X</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>0</td>
<td>I</td>
<td>X</td>
<td>f</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Default Device**

<table>
<thead>
<tr>
<th>Function</th>
<th>UNIT</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRINT f, list</td>
<td>-3</td>
<td>(LPT)</td>
</tr>
<tr>
<td>PUNCH f, list</td>
<td>-2</td>
<td>(PTP)</td>
</tr>
<tr>
<td>TYPE f, list</td>
<td>-1</td>
<td>(TTY)</td>
</tr>
</tbody>
</table>

MOVEI 16, ARGBLK
PUSHJ 17, OUT.

where ARGBLK is

<table>
<thead>
<tr>
<th>0</th>
<th>9</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>17</th>
<th>18</th>
<th>35</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>2</td>
<td>I</td>
<td>X</td>
<td>u</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>7</td>
<td>I</td>
<td>X</td>
<td>c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>7</td>
<td>I</td>
<td>X</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
E.6.3.6 Calling Sequences for Statements which Position Magnetic Tape Units – The FORTRAN-10 statements that may be used to control the positioning of a magnetic tape device and their calling sequences are:

<table>
<thead>
<tr>
<th>Function</th>
<th>FOROTS Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>SKIPFILE (u)</td>
<td>7</td>
</tr>
<tr>
<td>BACKFILE (u)</td>
<td>3</td>
</tr>
<tr>
<td>BACKSPACE (u)</td>
<td>2</td>
</tr>
<tr>
<td>ENDFILE (u)</td>
<td>4</td>
</tr>
<tr>
<td>REWIND (u)</td>
<td>0</td>
</tr>
<tr>
<td>SKIPRECORD (u)</td>
<td>5</td>
</tr>
<tr>
<td>UNLOAD (u)</td>
<td>1</td>
</tr>
</tbody>
</table>

CALL:

MOVEI 16, ARGBLK
PUSHJ 17, MTOP.

where ARGBLK is

<table>
<thead>
<tr>
<th>0________8</th>
<th>9________12</th>
<th>13</th>
<th>14________17</th>
<th>18________35</th>
</tr>
</thead>
<tbody>
<tr>
<td>-3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>FOROTS code</td>
<td>I</td>
<td>X</td>
<td>u</td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td>I</td>
<td>X</td>
<td>c</td>
</tr>
<tr>
<td></td>
<td></td>
<td>I</td>
<td>X</td>
<td>d</td>
</tr>
</tbody>
</table>

E.6.3.7 List Directed Input/Output Statements – Any form of a formatted input/output statement may be written as a list-directed statement by replacing the referenced FORMAT statement number with an asterisk (*). The list-directed forms of the READ and WRITE statements and their calling sequences are:

READ (u, *, END=c, ERR=d) list
READ (u#R, *, END=c, ERR=d) list

MOVEI 16, ARGBLK
PUSHJ 17, IN.

and

WRITE (u, *, END=c, ERR=d) list
WRITE (u#R, *, END=c, ERR=d) list

MOVEI 16, ARGBLK
PUSHJ 17, OUT.
where ARGBLK is

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>9</th>
<th>13</th>
<th>14</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reserved</td>
<td>2</td>
<td>1</td>
<td>X</td>
<td>u</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>1</td>
<td>X</td>
<td>c</td>
<td></td>
</tr>
<tr>
<td>Reserved</td>
<td>7</td>
<td>1</td>
<td>X</td>
<td>d</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

E.6.3.8 Input/Output Data Lists – The compiler generates a calling sequence to the runtime system if an I/O list is defined for the READ or WRITE statement. The argument block associated with the calling sequence contains the addresses of the variables and arrays to be transferred to or from an I/O buffer. The general form of an I/O list calling sequence is:

MOVEI 16, ARGBLK
PUSHJ 17, IOLST.

Any number of elements may be included in the ARGBLK. The end of the argument block is specified by a zero entry or a call to the FIN. entry.

<table>
<thead>
<tr>
<th>Mnemonic Name</th>
<th>FOROTS Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA</td>
<td>1</td>
</tr>
<tr>
<td>SLIST</td>
<td>2</td>
</tr>
<tr>
<td>ELIST</td>
<td>3</td>
</tr>
<tr>
<td>FIN</td>
<td>4</td>
</tr>
</tbody>
</table>

The elements of an I/O list are:

1. DATA

The DATA element converts one single, double, or complex precision item from external to internal form for a READ statement and from internal to external form for a WRITE statement. Each DATA element has the following format.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>9</th>
<th>13</th>
<th>14</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATA</td>
<td>type</td>
<td>I</td>
<td>X</td>
<td>SCALAR ADDR</td>
<td></td>
</tr>
</tbody>
</table>

2. SLIST

The SLIST argument converts an entire array from internal to external form or vice versa depending on the type of statement (i.e., READ or WRITE) involved. An SLIST table has the following form:
<table>
<thead>
<tr>
<th>0</th>
<th>9</th>
<th>13</th>
<th>14</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>12</td>
<td>17</td>
<td>35</td>
<td></td>
</tr>
</tbody>
</table>

SLIST.

<table>
<thead>
<tr>
<th>type</th>
<th>1</th>
<th>X</th>
<th>#ELEMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>X</td>
<td>INCREMENTS</td>
</tr>
</tbody>
</table>

| type | 1 | X | BASE ADDR1. |

For example, the sequence:

DIMENSION A(100, B(100)
READ(--) A
or
READ (--) (A(I), I=1,100)

! only when the /OPT switch is used
develops an SLIST argument of the form:

<table>
<thead>
<tr>
<th>0</th>
<th>9</th>
<th>13</th>
<th>14</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>12</td>
<td>17</td>
<td>35</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>1</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>type</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The increment may be zero. This could be produced by the sequence:

DIMENSION A(100)
WRITE(--) (K, I=1,100)

! only when the /OPT switch is used

The zero may not appear as an immediate constant in the argument block. The SLIST for the previous example would be:

<table>
<thead>
<tr>
<th>0</th>
<th>9</th>
<th>13</th>
<th>14</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>12</td>
<td>17</td>
<td>35</td>
<td></td>
</tr>
</tbody>
</table>

SLIST

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pointer to a word containing a zero

K

3. ELIST

The SLIST format permits only a single increment for a number of arrays to be specified while the ELIST permits different increments to be specified for different arrays.

The format of the ELIST is
For example, the FORTRAN sequence

```
DIMENSION IC(6,100), IB(100)
WRITE(,--) (IB(I), IC(1,I), I=1,100)
```

produces the ELIST

```
<table>
<thead>
<tr>
<th>0</th>
<th>8</th>
<th>9</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>17</th>
<th>18</th>
<th>35</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>100</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>IB</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>IC</td>
<td></td>
</tr>
</tbody>
</table>
```

**4. FIN**

The end of an I/O list is indicated by a call to the FIN routine in the object time system. This call must be made after each I/O initialization call, including calls with a null I/O list. The FIN routine may be entered by an explicit call or by an argument in the I/O list argument block. If both calls are used, the explicit call has no meaning. The FIN element has the following format:

**EXPLICIT CALL:**

```
MOVE 16, ZERBLK
PUSH 17, FIN.
```

where ZERBLK is

```
<table>
<thead>
<tr>
<th>0</th>
<th>8</th>
<th>9</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>17</th>
<th>18</th>
<th>35</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
```
E.6.3.9 OPEN and CLOSE Statements, Calling Sequences – The form and calling sequences for the OPEN and CLOSE FORTRAN-10 statements are:

**OPEN STATEMENT CALL**

MOVEI 16, ARGBLK
PUSHJ 17, OPEN.

**CLOSE STATEMENT CALL**

MOVEI 16, ARGBLK
PUSHJ 17, CLOSE.

where ARGBLK is

<table>
<thead>
<tr>
<th>0_____8</th>
<th>9_____12</th>
<th>13</th>
<th>14_____17</th>
<th>18_____35</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

Negative of the # of words in block not including this one.

| → G | 2 | 1 | X | u |
| G   | 7 | 1 | X | c |
| G   | 7 | 1 | X | d |
| G   | type | 1 | X | H |
| G   | type | 1 | X | H |
| G   | type | 1 | X | H |
| .   | . | . | . | . |
| .   | . | . | . | . |
| .   | . | . | . | . |
| G   | type | 1 | X | H |

The G field (bits 0–8) contains a 2-digit numeric which defines the argument name; the H field (bits 18–35) contains an address which points to the value of the argument.

The numeric codes which may appear in the G field and the argument which each identifies are

<table>
<thead>
<tr>
<th>G Field</th>
<th>Open Argument</th>
<th>G Field</th>
<th>Open Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>DIALOG</td>
<td>10</td>
<td>DIRECTORY</td>
</tr>
<tr>
<td>02</td>
<td>ACCESS</td>
<td>11</td>
<td>LIMIT</td>
</tr>
<tr>
<td>03</td>
<td>DEVICE</td>
<td>12</td>
<td>MODE</td>
</tr>
<tr>
<td>04</td>
<td>BUFFER COUNT</td>
<td>13</td>
<td>FILE SIZE</td>
</tr>
<tr>
<td>05</td>
<td>BLOCK SIZE</td>
<td>14</td>
<td>RECORD SIZE</td>
</tr>
<tr>
<td>06</td>
<td>FILENAME</td>
<td>15</td>
<td>DISPOSE</td>
</tr>
<tr>
<td>07</td>
<td>PROTECTION</td>
<td>23</td>
<td>PARITY</td>
</tr>
<tr>
<td></td>
<td></td>
<td>24</td>
<td>DENSITY</td>
</tr>
</tbody>
</table>

Version 4 FORTRAN-10  
Version 4 FOROTS  

E-24  
October 1974
E.6.3.10 Memory Allocation Routines — The memory management module is called to allocate or de-allocate core blocks. There are two entry points, ALCOR and DECOR, that control memory allocation and de-allocation.

The ALCOR entry is used to allocate the number of words specified in the argument block variable. Upon return AC 0 will contain either the address of the allocated core block or a -1 value which indicates that core is not available. The calling sequence for ALCOR call is:

```
MOVEI 16, ARGBLK
PUSHJ 17, ALCOR.
```

where ARGBLK is

<table>
<thead>
<tr>
<th>0</th>
<th>9</th>
<th>13</th>
<th>14</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>type</td>
<td>I</td>
<td>X</td>
<td>Address of Number of Words</td>
</tr>
</tbody>
</table>

The DECOR entry is used to de-allocate a previously allocated block of memory; the argument variable must be loaded with the address of the core block to be returned. Upon return AC 0 is set to 0.

If the number of desired words is N, ALCOR actually removes N+1 words from free storage. The pointer returned points to the 2nd word (word 1 as opposed to word 0) removed from free storage. The 0 word contains the negative value of N in its left half. This word is used by FOROTS to maintain linked lists of allocated (using ALCOR) and free storage.

The calling sequence for a DECOR call is:

```
MOVEI 16, ARGBLK
PUSHJ 17, DECOR.
```

where ARGBLK is

<table>
<thead>
<tr>
<th>0</th>
<th>9</th>
<th>13</th>
<th>14</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>type</td>
<td>I</td>
<td>X</td>
<td>Pointer to word containing address of block to be returned</td>
</tr>
</tbody>
</table>

E.6.3.11 Software Channel Allocation and De-allocation Routines — Software channels may be allocated by MACRO programs via calls to the ALCHN routine and de-allocated by calls to the DECHN routine. Values are returned in AC 0.

The ALCHN entry is used to allocate a particular channel or the next available channel. If bits 18–35 of the ARGBLK are zero, the next available channel will be assigned; if non-zero, they must contain the requested channel number (1–15 octal). If the channel requested is not available or all channels are in use, ALCHN returns with -1 in AC0. Normal returns contain the assigned channel number in AC0.

Version 4 FORTRAN-10
Version 4 FOROTS

E-25

November 1975
The calling sequence of an ALCHN. routine is:

MOVEI 16, ARGBLK
PUSHJ 17, ALCHN.

where ARGBLK is

<table>
<thead>
<tr>
<th>0</th>
<th>9</th>
<th>13</th>
<th>14</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>type</td>
<td>I</td>
<td>X</td>
<td>Pointer to a word containing the channel# or zero</td>
</tr>
</tbody>
</table>

The DECHN. entry is used to de-allocate a previously assigned channel. The channel to be released is passed to DECHN. in the argument block variable. If the channel to be de-allocated was not assigned by ALCHN, and thus cannot be de-assigned, AC 0 is set to -1 on return.

The calling sequence for a DECHN. routine is:

MOVEI 16, ARGBLK
PUSHJ 17, DECHN.

where ARGBLK is

<table>
<thead>
<tr>
<th>0</th>
<th>9</th>
<th>13</th>
<th>14</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>type</td>
<td>I</td>
<td>X</td>
<td>Pointer to a word containing the channel# to be released</td>
</tr>
</tbody>
</table>

E.8 LOGICAL/PHYSICAL DEVICE ASSIGNMENTS

FORTRAN logical and physical device assignments are made by the user at run time or standard system assignments are made according to a FOROTS Device Table (i.e., DEVTB.). The standard assignments contained by the Device Table are shown in Table E-2.
where

- type = the FORTRAN argument type (see Appendix D)
- function number = the number of one of the required functions
- error code = the 3-letter mnemonic output by the object time system after ?., %, or [. (See Table E-1.)
- status = undefined on the call and set on the return with one of the values below.

<table>
<thead>
<tr>
<th>status</th>
<th>Function Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Function not implemented</td>
</tr>
<tr>
<td>0</td>
<td>Successful return</td>
</tr>
<tr>
<td>1...n</td>
<td>Specific error message</td>
</tr>
</tbody>
</table>

Table E-1  
Function Numbers and Function Codes

<table>
<thead>
<tr>
<th>Function Number</th>
<th>Function Mnemonic</th>
<th>Function Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ILL</td>
<td>Illegal function</td>
</tr>
<tr>
<td>1</td>
<td>GAD</td>
<td>Allocates core from a specific address</td>
</tr>
<tr>
<td>2</td>
<td>COR</td>
<td>Allocates core from available core</td>
</tr>
<tr>
<td>3</td>
<td>RAD</td>
<td>De-allocates core</td>
</tr>
<tr>
<td>4</td>
<td>GCH</td>
<td>Gets or assigns an I/O channel</td>
</tr>
<tr>
<td>5</td>
<td>RCH</td>
<td>Releases an I/O channel</td>
</tr>
<tr>
<td>6</td>
<td>GOT</td>
<td>Allocates core from FOROTS</td>
</tr>
<tr>
<td>7</td>
<td>ROT</td>
<td>De-allocates core from FOROTS</td>
</tr>
<tr>
<td>8</td>
<td>RNT</td>
<td>Returns the initial runtime from FOROTS</td>
</tr>
<tr>
<td>9</td>
<td>IFS</td>
<td>Returns initial runtime file spec. from FOROTS</td>
</tr>
<tr>
<td>10</td>
<td>CBC</td>
<td>Cuts back core if possible</td>
</tr>
</tbody>
</table>

FUNCTION 0 (ILL) – This function is illegal. The argument block is ignored, and the function always returns a status of -1.

FUNCTION 1 (GAD) – This function allocates core from a specific address. The arguments are

- arg 1: address at which to begin core allocation
- arg 2: number of words of core to allocate

The return statuses are

0: core allocated (arg 1 and arg 2 unchanged)
1: not enough core available in system (arg 1 and arg 2 unchanged)
2: cannot allocate core at specified address (arg 1 and arg 2 unchanged)
3: illegal arguments (i.e., address + size is greater than 256K) (arg 1 and arg 2 unchanged)

FUNCTION 2 (COR) – This function allocates core from any address. The arguments are

- arg 1: undefined
- arg 2: size of core to allocate

The returned statuses are

0: core allocated (arg 2 unchanged, arg 1 beginning address of the allocated core)
1: not enough core available in system (arg 2 unchanged)
3: illegal argument (i.e., size is greater than 256K)
FUNCTION 3 (RAD) – This function de-allocates core at the specified address. The arguments are

arg 1 address of core to be de-allocated
arg 2 number of words to be de-allocated

The returned statuses are

0 core de-allocated
1 core cannot be de-allocated
3 illegal argument (i.e., both the address and the size are greater than 256K)

FUNCTION 4 (GCH) – This function assigns an I/O channel. The argument is

arg 1 undefined

The returned statuses are

0 I/O channel assigned (arg 1 channel number)
1 no I/O channels available

FUNCTION 5 (RCH) – This function releases an I/O channel. The argument is

arg 1 I/O channel number to be released

The returned statuses are

0 channel released
1 invalid channel number

FUNCTION 6 (GOT) – This function gets core from the object time system list. The arguments are

arg 1 address at which to allocate core
arg 2 number of words of core to allocate

The returned statuses are

0 core allocated (arg 1 and arg 2 unchanged)
1 not enough core available in system (arg 1 and arg 2 unchanged)
2 cannot allocate core at specified address (arg 1 and arg 2 unchanged)
3 illegal argument(s)

This function differs from function 1 in that if the object time system has two free core lists, then function 1 is used to allocate space for links, and this function is used to allocate space for I/O buffers. Function 1 uses the free core list for LINK-10, and function 6 uses the list for the object time system.

FUNCTION 7 (ROT) – This function returns core to the object time system. The arguments are

arg 1 address of core to be de-allocated and returned
arg 2 size of core to be de-allocated and returned
The returned statuses are

- 0 core de-allocated
- 1 core cannot be de-allocated
- 3 illegal argument

**FUNCTION 8 (RNT)** — This function returns the initial runtime from the object time system. The argument is

arg 1 undefined

The returned status is

- 0 always (arg 1 — runtime from the object time system)

This function is used only if the user desires a log file.

**FUNCTION 9 (IFS)** — This function returns the initial runtime file specification from the object time system. The specification is obtained from accumulators 0, 7, and 11 after the initial RUN command. The arguments are

arg 1 undefined
arg 2 undefined
arg 3 undefined

The returned status is

- 0 always (arg 1 — device from accumulator 11, arg 2 — filename from accumulator 0, and arg 3 — directory from accumulator 7)

This function tells the overlay handler which file to read after the initial RUN command.

**FUNCTION 10 (CBC)** — This function cuts back core if possible and is used to reduce the size of the user job. There are no arguments.

The returned status is

- 0 always

### E.8 LOGICAL/PHYSICAL DEVICE ASSIGNMENTS

FORTRAN logical and physical device assignments are made by the user at run time or standard system assignments are made according to a FOROTS Device Table (i.e., DEVTB.). The standard assignments contained by the Device Table are shown in Table E-2.
<table>
<thead>
<tr>
<th>Device/Function</th>
<th>FORTRAN Logical Unit Number</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>REREAD</td>
<td>-6</td>
<td>REREAD statement</td>
</tr>
<tr>
<td>CDR</td>
<td>-5</td>
<td>READ statement</td>
</tr>
<tr>
<td>TTY</td>
<td>-4</td>
<td>ACCEPT statement</td>
</tr>
<tr>
<td>LPT</td>
<td>-3</td>
<td>PRINT statement</td>
</tr>
<tr>
<td>PTP</td>
<td>-2</td>
<td>PUNCH statement</td>
</tr>
<tr>
<td>TTY</td>
<td>-1</td>
<td>TYPE statement</td>
</tr>
<tr>
<td>0</td>
<td>00</td>
<td>ILLEGAL</td>
</tr>
<tr>
<td>DSK</td>
<td>01</td>
<td>DISK</td>
</tr>
<tr>
<td>CDR</td>
<td>02</td>
<td>Card Reader</td>
</tr>
<tr>
<td>LPT</td>
<td>03</td>
<td>Line Printer</td>
</tr>
<tr>
<td>CTY</td>
<td>04</td>
<td>Console Teletype</td>
</tr>
<tr>
<td>TTY</td>
<td>05</td>
<td>User's Teletype</td>
</tr>
<tr>
<td>PTR</td>
<td>06</td>
<td>Paper Tape Reader</td>
</tr>
<tr>
<td>PTP</td>
<td>07</td>
<td>Paper Tape Punch</td>
</tr>
<tr>
<td>DIS</td>
<td>08</td>
<td>Display</td>
</tr>
<tr>
<td>DTA1</td>
<td>09</td>
<td>DECTape</td>
</tr>
<tr>
<td>DTA2</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>DTA3</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>DTA4</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>DTA5</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>DTA6</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>DTA7</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>MTA0</td>
<td>16</td>
<td>Magnetic Tape</td>
</tr>
<tr>
<td>MTA1</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>MTA2</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>FORTR</td>
<td>19</td>
<td>Assignable Device</td>
</tr>
<tr>
<td>DSK</td>
<td>20</td>
<td>DISK</td>
</tr>
<tr>
<td>DSK</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>DSK</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>DSK</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>DSK</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>DEV1</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>DEV2</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>DEV3</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>DEV4</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>DEV5</td>
<td>29</td>
<td></td>
</tr>
</tbody>
</table>
APPENDIX F
FORDDT

FORDDT is an interactive program used to debug FORTRAN programs and control their execution. By using the symbols created by the FORTRAN compiler, FORDDT allows the user to examine and modify the data and FORMAT statements in his program, set breakpoints at any executable statement or routine, trace his program statement by statement, and make use of many other debugging techniques described in this appendix.

Table F-1 provides a brief glance at all the commands available to the user of FORDDT.

<table>
<thead>
<tr>
<th>Command</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Access Commands</td>
<td></td>
</tr>
<tr>
<td>ACCEPT</td>
<td>Modifies data locations.</td>
</tr>
<tr>
<td>TYPE</td>
<td>Displays data locations.</td>
</tr>
<tr>
<td>Declarative Commands</td>
<td></td>
</tr>
<tr>
<td>GROUP</td>
<td>Defines indirect lists for TYPE statements.</td>
</tr>
<tr>
<td>MODE</td>
<td>Specifies format of typeout.</td>
</tr>
<tr>
<td>OPEN</td>
<td>Accesses program unit symbol table.</td>
</tr>
<tr>
<td>PAUSE</td>
<td>Places pause requests.</td>
</tr>
<tr>
<td>REMOVE</td>
<td>Removes pause requests.</td>
</tr>
<tr>
<td>DIMENSION</td>
<td>Defines dimensions of arrays for FORDDT references. (Unnecessary if /DEBUG:DIMENSIONS was used.)</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>Defines dimensions of double precision arrays for FORDDT references. (Unnecessary if /DEBUG:DIMENSIONS was used.)</td>
</tr>
</tbody>
</table>
### Table F-1 (Cont)
#### Table of Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Control Commands</strong></td>
<td></td>
</tr>
<tr>
<td>START</td>
<td>Begins execution of FORTRAN program.</td>
</tr>
<tr>
<td>CONTINUE</td>
<td>Continues execution after a pause.</td>
</tr>
<tr>
<td>GOTO</td>
<td>Transfers control to some program statement within the open program unit.</td>
</tr>
<tr>
<td>NEXT</td>
<td>Traces execution of the program.</td>
</tr>
<tr>
<td>STOP</td>
<td>Terminates program and returns to monitor mode.</td>
</tr>
<tr>
<td>DDT</td>
<td>Enters DDT (if DDT is loaded).</td>
</tr>
<tr>
<td><strong>Other Commands</strong></td>
<td></td>
</tr>
<tr>
<td>LOCATE</td>
<td>Lists program unit names in which a given symbol is defined.</td>
</tr>
<tr>
<td>STRACE</td>
<td>Displays routine backtrace of current program status.</td>
</tr>
<tr>
<td>WHAT</td>
<td>Displays current DIMENSION, GROUP, and PAUSE information.</td>
</tr>
</tbody>
</table>

### F.1 INPUT FORMAT

FORDDT commands are made up of alphabetic FORTRAN-like identifiers and need consist of only those characters that are required to make the command unique. If the user wishes to specify parameters, a space or tab is required following the command name. FORDDT expects a parameter if a delimiter is found.

#### F.1.1 Variables and Arrays

FORDDT allows the user to access and modify the data locations in his program by using standard FORTRAN-10 symbolic names. Variables are specified simply by name. Array elements are specified in the following format:

\[
\text{name}\ (S_1, \ldots, S_n)
\]

where

\[
\begin{align*}
\text{name} & = \text{a FORTRAN variable or array name} \\
(S_1, \ldots, S_n) & = \text{the subscripts of the particular array.}
\end{align*}
\]

An entire array may be referenced simply by its unsubscripted name; a range of array elements may be specified by inputting the first and last array elements of the desired range, separated by a dash (−).

#### Examples

- ALPHA
- ALPHA(7)
- ALPHA(Pi)
- ALPHA(2)–ALPHA(5)
F.1.2 Numeric Conventions

FORDDT accepts optionally signed numeric data in the standard FORTRAN-10 input formats:

1. INTEGER – A string of decimal digits.
2. FLOATING POINT – A string of decimal digits optionally including a decimal point. Standard engineering and double precision exponent formats are also accepted.
3. OCTAL – A string of octal digits optionally preceded by a double quote (".
4. COMPLEX – An ordered pair of integer or real constants separated by a comma and enclosed in parentheses.

F.1.3 Statement Labels and Source Line Numbers

FORTRAN statement labels are input and output by straightforward numeric reference (i.e., 1234). However, source line numbers must be input to FORDDT with a number sign (#) preceding them. This mandatory sign distinguishes statement labels from source line numbers.

F.2 NEW USER TUTORIAL

The new FORDDT user can rely on the commands described below as a basis for debugging FORTRAN programs. The new user will find these commands easy to understand and apply.

F.2.1 Basic Commands

The easiest method of loading and starting FORDDT is

```
.DEBUG filename.ext (DEBUG)/F10,SYS:FORDDT.REL
```

FORDDT will respond with

```
ENTERING FORDDT
>>
```

Just as an asterisk (*) signifies FORTRAN-10's readiness, the two angle brackets signify that FORDDT is awaiting one of the following commands:

- **OPEN**
  - Makes available to FORDDT the symbol names in a particular program unit of the FORTRAN program. When a program unit symbol table is opened, the previously open program unit is automatically closed. When FORDDT is entered, the MAIN program is automatically opened. The command format is

  ```
  OPEN name
  ```

  This will open the particular program unit named and allow all variables within that subprogram to be accessible to FORDDT.

- **OPEN**
  - with no arguments will reopen the symbol table of the main program unit.

- **START**
  - Starts the user program at the main program entry point. The command format is

  ```
  START
  ```
STOP

Terminates program execution, causes all files to be closed, and exits to the monitor. The command format is

STOP

MODE

Defines the display format for succeeding FORDDT TYPE commands. Only the first character of the mode need be typed to identify it to FORDDT. The modes are

<table>
<thead>
<tr>
<th>Mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>FLOATING POINT</td>
</tr>
<tr>
<td>D</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>C</td>
<td>COMPLEX</td>
</tr>
<tr>
<td>I</td>
<td>INTEGER</td>
</tr>
<tr>
<td>O</td>
<td>OCTAL</td>
</tr>
<tr>
<td>A</td>
<td>ASCII (left-justified)</td>
</tr>
<tr>
<td>R</td>
<td>RASCI (right-justified)</td>
</tr>
</tbody>
</table>

Unless the MODE command is given, the default typeout mode is the floating point format.

The command format is

MODE list

where list contains one or more of the mode identifiers separated by commas. The current setting can be changed by issuing another MODE command. If more than one mode is given, the values are typed out in the order: F,D,C,I,O,A,R

MODE

with no arguments will reset FORDDT to the original setting of floating point format.

TYPE

Allows the user to display the contents of one or more data locations. They are displayed on the user terminal formatted according to the last MODE specification. The command format is

TYPE list

where list may contain one or more arrays, variables, array elements, or array element ranges separated by commas. For example

TYPE I, ALPHA, BETA(2), J(3)–J(5)

Each item will be displayed in each of the currently active typeout modes as set by the last MODE command.

ACCEPT

Allows the user to change the contents of a FORTRAN variable, array, array element, or array element range. The command format is

ACCEPT name/mode value
where

name = the name of the variable, array, array element, or array element range to be modified. If the field contains an unsubscripted array name or an element range, it causes all of the elements to be set to the given value (see special case for ASCII in section F.6).

mode = the format of the data value to be entered. If given, it must be preceded by a slash (/) and immediately follow the name. (Note that /mode does not apply to FORMAT modification.)

value = the new value to be assigned. It must correspond in format to the given mode.

Data Modes

Only the first character of a data mode need be typed to identify it to FORDDT. If not specified, the default mode is REAL. The following input modes are available:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>ASCII (left-justified)</td>
<td>/FOO/</td>
</tr>
<tr>
<td>C</td>
<td>COMPLEX</td>
<td>(1.25, -78.E+9)</td>
</tr>
<tr>
<td>D</td>
<td>DOUBLE PRECISION</td>
<td>123.4567890</td>
</tr>
<tr>
<td>F</td>
<td>REAL</td>
<td>123.45678</td>
</tr>
<tr>
<td>I</td>
<td>INTEGER</td>
<td>1234567890</td>
</tr>
<tr>
<td>O</td>
<td>OCTAL</td>
<td>76543210</td>
</tr>
<tr>
<td>R</td>
<td>RASCII (right-justified)</td>
<td>\BAR\</td>
</tr>
<tr>
<td>S</td>
<td>SYMBOLIC</td>
<td>PSI(2,4)</td>
</tr>
</tbody>
</table>

An example of the ACCEPT command format is

```
ACCEPT ALPHA 100.6
```

This changes the value of the variable ALPHA to 100.6 with the default input mode of REAL, since mode was not specified.

PAUSE

Allows the user to set a breakpoint at any label, line number, or subroutine entry in the user program. Up to 10 pauses may be set at one time. When one of these pauses is encountered, execution of the FORTRAN program is suspended and control is transferred to FORDDT. Also, when a pause is encountered, the symbol table of that subprogram is automatically opened. The command format is

```
PAUSE P
```

where P is a statement label number, line number, or routine entry point name; for example

```
PAUSE 100
```

will cause a breakpoint at statement label 100 of the currently open program unit.

Note that subprogram parameter values will be displayed when a pause is encountered at a subprogram entry point.
CONTINUE

Allows the program to resume execution after a FORDDT pause. After a CONTINUE is executed, the program either runs to completion, or it runs until another pause is encountered. If a value is included with this command, the program will run until the nth occurrence of the given pause or until a different pause is encountered. The command formats are

CONTINUE
or
CONTINUE n

Example
CONTINUE 15

will continue execution until the fifteenth occurrence of the pause.

REMOVE

Used to remove those pauses from the program previously set up by the PAUSE command. The command format is

REMOVE P

where P is the number of the statement label where the pause was set, i.e.,

REMOVE 100

will remove the pause at statement label 100.

Note that REMOVE with no arguments will remove all pauses; therefore, no abbreviation of the command is allowed in this instance. This precaution prevents the accidental removal of all pauses.

WHAT

Displays on the user terminal the name of the currently open program unit and any currently active pause settings. The command format is

WHAT

F.3 FORDDT AND THE FORTRAN-10 /DEBUG SWITCH

Most facilities of FORDDT are available without the FORTRAN-10 /DEBUG features; however, if the /DEBUG switch is not used when compiling a FORTRAN program, the trace features (NEXT command) will not be available, and several of the other commands will be restricted.

Using the /DEBUG switch tells FORTRAN-10 to compile extra information for FORDDT. (See Appendix C Using the Compiler for a complete description of each feature.) The additional features include

1. /DEBUG:DIMENSIONS which will generate dimension information to the REL file for all arrays dimensioned in the subprogram. The dimension information will automatically be available to FORDDT if the user wishes to reference an array in a TYPE or ACCEPT command. This feature eliminates the need to specify dimension information for FORDDT by using the DIMENSION command.
2. /DEBUG:LABELS which will generate labels for every executable source line in the form “line-number L”. If these labels are generated, they may be used as arguments with the FORDDT commands PAUSE and GOTO.

This switch will also generate labels at the last location allocated for a FORMAT statement so that FORDDT can detect the end of the statement. These labels have the form “format-label F”. If they are generated, the user will be able to display and modify his FORMAT statements via the TYPE and ACCEPT commands.

Note that the :LABELS switch is automatically activated with the :TRACE switch since labels are needed to accomplish the trace features.

3. /DEBUG:TRACE which will generate a reference to FORDDT before each executable statement. This switch is required in order for the trace command NEXT to function.

Note that if more than one FORTRAN statement has been placed on a single input line, only the first statement will have a FORDDT reference and line-number label associated with it. This also applies to the :LABELS switch.

4. /DEBUG:INDEX which will force the compiler to store, in its respective data location as well as a register, the index variable of all DO loops at the beginning of each loop iteration. The user will then be able to examine DO loops by using FORDDT. If a user modifies a DO loop index using FORDDT, he will not affect the number of loop iterations because a separate loop count is used (see section D.1.5).

Note that this switch has no direct affect on any of the commands in FORDDT.

F.4 LOADING AND STARTING FORDDT

1. The simplest form of loading and starting FORDDT is with the following command string:

```
.DEBUG filename.ext(/DEBUG)/F10,SYS:FORDDT,REL
```

FORDDT responds with

```
ENTERING FORDDT
>>
```

The angle brackets indicate that FORDDT is ready to receive a command, just like an asterisk (*) signifies FORTRAN-10's readiness.

The DEBUG command to the monitor will also load DDT (standard system debugging program). DDT can be used or ignored but it does require an extra 2K (octal) of core.

2. The user may wish to load his compiled program and FORDDT directly with the LINK-10 loader. (Loading with LINK-10 was accomplished implicitly in the previous command string.) The command sequence is as follows:

```
.R LINK
*/L, filename.ext, SYS:FORDDT
*/SYMSEG/G
```

If the total FORTRAN program consists of many subroutines and insufficient core is available to complete loading with symbols, it is possible to load with symbols just those sections expected to give trouble. The remaining routines need not be loaded.
F.5 SCOPE OF NAME AND LABEL REFERENCES

Each program unit has its own symbol table. When the user initially enters FORDDT, he automatically opens the symbol table of the main program. All references to names or labels via FORDDT must be made with respect to the currently open symbol table. If the user has given the main program a name other than MAIN by using the PROGRAM statement (see Chapter 5, section 5.2), FORDDT will ask for the defined program name. After the user enters the program name, FORDDT will open the appropriate symbol table. At this point, symbol tables in programs other than the main program can be opened by using the OPEN command (see section F.5).

References to statement labels, line numbers, FORMAT statements, variables, and arrays must have labels that are defined in the currently open symbol table. However, FORDDT will accept variable and array references outside the currently open symbol table providing the name is unique with respect to all program units in the given load module.

F.6 FORDDT COMMANDS

This section gives a detailed description of all commands in FORDDT. The commands are given in alphabetical order.

**ACCEPT**

Allows the user to change the contents of a FORTRAN variable, array, array element, array element range, or FORMAT statement. The command format is

```
ACCEPT name/mode value
```

where

- **name** = the variable array, array element, array element range, or FORMAT statement to be modified.
- **mode** = the format of the data value to be entered. The mode keyword must be preceded by a slash (/) and immediately follow the name. Intervening blanks are not allowed. (Note that /mode does not apply to FORMAT modification.)
- **value** = the new value to be assigned. The format of the input value must correspond to the specified mode.

**DATA LOCATION MODIFICATION**

**Data Modes**

The following data modes are accepted:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>ASCII (left-justified)</td>
<td>/FOO/</td>
</tr>
<tr>
<td>C</td>
<td>COMPLEX</td>
<td>(1.25,-78,E+9)</td>
</tr>
<tr>
<td>D</td>
<td>DOUBLE PRECISION</td>
<td>123.4567890</td>
</tr>
<tr>
<td>F</td>
<td>REAL</td>
<td>123.45678</td>
</tr>
<tr>
<td>I</td>
<td>INTEGER</td>
<td>1234567890</td>
</tr>
<tr>
<td>O</td>
<td>OCTAL</td>
<td>76543210</td>
</tr>
<tr>
<td>R</td>
<td>RASCII (right-justified)</td>
<td>\BAR\</td>
</tr>
<tr>
<td>S</td>
<td>SYMBOLIC</td>
<td>PSI(2,4)</td>
</tr>
</tbody>
</table>

If not specified, the default mode is REAL.
Two Word Values

For the data modes ASCII, RASCI, OCTAL, and SYMBOLIC, FORDDT will accept a "/LONG" modifier on the mode switch. This modifier indicates that the variable and the value are to be interpreted as two words in length.

Example

ACCEPT VAR/RASCI/LONG '1234567890'

will assume that VAR is two words long and store the given 10 character literal into it.

Initialization of Arrays

If the name field of an ACCEPT contains an unsubscripted array name or a range of array elements, all elements of the array or the specified range will be set to the given value.

Example

ACCEPT ARRAY/F 1.0
or
ACCEPT ARRAY(5)-ARRAY(10)/F 1.0

Note that this applies only to modes other than ASCII and RASCI.

Long Literals

When the value field of an ACCEPT contains an unsubscripted array name or range of array elements, and the specified data mode is ASCII or RASCI, the value field is expected to contain a long literal string. ACCEPT will store the string linearly into the array or array range. If the array is not filled, the remainder of the array or range will be set to zero. If the literal is too long the remaining characters will be ignored.

Example

ACCEPT ARRAY/RASCI 'ABCDEFHIJKLMNOPQRSTUVWXYZ'

FORMAT STATEMENT MODIFICATION

When the name field of an ACCEPT contains a label, it expects this label to be a FORMAT statement label and that the value field contains a new FORMAT specification.

Example

ACCEPT 10 (1H0,F10.2,3(I2))

The new specification cannot be longer than the space originally allocated to the FORMAT by the compiler. The remainder of the area is cleared if the new specification is shorter.

Note that FOROTS performs some amount of encoding of FORMAT statements when it processes them for the first time. If any I/O statement referencing the given FORMAT has been executed, the FORTRAN program has to be restarted (re-initializing FOROTS).
CONTINUE

Allows the program to resume execution after a FORDDT pause. After a CONTINUE is executed, the program either runs to completion or until another pause is encountered. The command format is

CONTINUE n

where the n is optional and if omitted will be assumed to be one. If a value is provided, it may be a numeric constant or program variable but it will be treated as an integer. When the value n is specified, the program will continue execution until the nth occurrence of this pause. For example,

CONTINUE 20

will continue execution after the 20th occurrence of the pause.

DDT

Transfers control of the program to DDT, the standard system debugging program (if loaded). Any files currently opened by FOROTS are unaffected and return to FORDDT is possible so that program execution may be resumed.

.F10 is the global symbol used to return control to FORDDT. The command format is

.F10SG

Where $ represents altmode or escape. The user program will be in the same condition as before unless the user has modified his core image with DDT.

DIMENSION

Sets the user defined dimensions of an array for FORDDT access purposes. These dimensions need not agree with those declared to the compiler in the source code. FORDDT will allow the user to redimension an array to have a larger scope than that of the source program. If this is done a warning is given. The command format is

DIMENSION S

For example:

DIMENSION ALPHA(7,5/6.10)

where S is the name of the array specified.

FORDDT will remember the dimensions of the array until it is redefined or removed.

The command

DIMENSION

will give a full list of all the user defined dimensions for all arrays.

DIMENSION ALPHA

will display the current information for the array ALPHA only.

DIMENSION ALPHA/REMOVE

will remove any user defined array information for the array ALPHA.