PDP-11 FORTRAN-77/RT-11

User's Guide

AA-BR70A-TC

March 1984

This document contains the information necessary to create, link and execute PDP-11 FORTRAN-77 programs on a PDP-11 processor. Programming information is provided for the RT-11 operating system.

SUPERSESSION/UPDATE INFORMATION:	This is a new document for this release.
OPERATING SYSTEM AND VERSION:	RT-11 V5.1
SOFTWARE VERSION:	FORTRAN-77/RT-11 V5.0

Adapted from PDP-11 FORTRAN-77/RSX by Multiware, Inc.
139 G Street, Suite 161, Davis, California 95616

digital equipment corporation · maynard, massachusetts

First Printing, March 1984

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 responsibilty for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

Printed in U.S.A.

The following are trademarks of Digital Equipment Corporation:

DIGITAL	DECsystem-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EduSystem	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAX	TYPESET-8
DDT	LAB-8	TYPESET-11
DECCOMM	DECSYSTEM-20	TMS-11
ASSIST-11	RTS-8	ITPS-10
VAX	VMS	SBI
DECnet	IAS	PDT
DATATRIEVE	TRAX	digital

MW-F77-006

HOW TO ORDER ADDITIONAL DOCUMENTATION

In Continental USA and Puerto Rico call 800-258-1710

In New Hampshire, Alaska, and Hawaii call 603-884-6660

In Canada call 613-234-7726 (Ottawa-Hull)
800-267-6146 (all other Canadian)

DIRECT MAIL ORDERS (USA & PUERTO RICO)*

Digital Equipment Corporation
P.O. Box CS2008
Nashua, New Hampshire 03061

*Any prepaid order from Puerto Rico must be placed with the local Digital subsidiary (809-754-7575)

DIRECT MAIL ORDERS (CANADA)

Digital Equipment of Canada Ltd.
940 Belfast Road
Ottawa, Ontario K1G 4C2
Attn: A&SG Business Manager

DIRECT MAIL ORDERS (INTERNATIONAL)

Digital Equipment Corporation
A&SG Business Manager
c/o Digital's local subsidiary or approved distributor

International orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Northboro, Massachusetts 01532.

CONTENTS

TABLE OF CONTENTS

PREFACE

MANUAL OBJECTIVES

The purpose of this document is to help programmers create, link, and execute PDP-11 FORTRAN-77/RT-11 programs under the RT-11 operating system. These operating systems must run on a machine with a Floating-Point Processor or a floating-point microcode option.

The PDP-11 FORTRAN-77/RT-11 language elements are described in the PDP-11 FORTRAN-77 Language Reference Manual.

INTENDED AUDIENCE

This manual is intended for programmers who have a working knowledge of the fundamental elements and interrelationships of the FORTRAN programming language. A detailed knowledge of the PDP-11 FORTRAN-77/RT-11 version of FORTRAN is not essential. A familiarity with RT-11 naming conventions and file specifications is assumed.

Building complex FORTRAN-77 applications which run under RT-11 and which are SYSTEM or VIRTUAL jobs certainly requires a detailed knowledge of RT-11 and its operation as will those which make full use of the RT-11 XM monitor. You are refered to the appropriate RT-11 reference manual for this information.

STRUCTURE OF THIS DOCUMENT

This manual is organized as follows:

- Chapter 1 contains the information needed to compile, link, and execute a PDP-11 FORTRAN-77/RT-11 program on the RT-11 operating system.
- Chapter 2 provides information about PDP-11 FORTRAN-77/RT-11 input/output, including details on file characteristics, record structure, and the use of certain OPEN statement keywords.
- Chapter 3 describes the PDP-11 FORTRAN-77/RT-11 run-time environment, including the calling conventions, error processing, and program section usage.
- Chapter 4 describes PDP-11 FORTRAN-77/RT-11 implementation concepts, with particular emphasis on data types, generic functions, DO loops, and floating-point data representation.
- Chapter 5 covers programming considerations relevant to typical PDP-11 FORTRAN-77/RT-11 applications.

- Chapter 6 discusses the use of character data, including character I/O and the character library functions.
- Appendixes A through G summarize internal data representation, diagnostic messages, system-supplied functions, compatibility between PDP-11 FORTRAN-77 and other DIGITAL FORTRAN implementations, and language extensions incorporated in PDP-11 FORTRAN-77/RT-11.

ASSOCIATED DOCUMENTS

The following documents are relevant to FORTRAN-77 programming:

- PDP-11 FORTRAN-77/RT-11 Language Reference Manual
- PDP-11 FORTRAN-77/RT-11 Object Time System Reference Manual
- PDP-11 FORTRAN-77/RT-11 Installation Guide/Release Notes for RT-11
- RT-11 System User's Guide
- RT-11 Software Support Manual
- RT-11 System Utilities Manual
- RT-11 Programmer's Reference Manual

For a complete list of software documents, see the host operating system documentation directory.

CONVENTIONS USED IN THIS DOCUMENT

The following conventions are observed in this manual:

- Uppercase words and letters used in examples indicate that you should type the word or letter exactly as shown.
- Lowercase words and letters used in examples indicate that you are to substitute a word or value of your choice.
- Brackets ([]) indicate optional elements.
- Braces ( ) are used to enclose lists from which one element is to be chosen.
- Ellipses (...) indicate that the preceding item(s) can be repeated one or more times.
- <RET> represents a carriage return.

SUMMARY OF TECHNICAL CHANGES

The PDP-11 FORTRAN-77/RSX compiler has been modified to execute under RT-11 Version 5.0 and 5.1. Several compile-time switches have been made available to the user. Code generated by this compiler is the same as that generated by its RSX counterpart. However, no support for RMS file access is supported under RT-11. Thus, access to indexed and relative files, and the ability to extend files is not present in the RT-11 version.

CHAPTER 1

USING RT-11 FORTRAN-77

DIGITAL's PDP-11 FORTRAN-77/RT-11 operates on PDP-11 minicomputers and Professional-300 series personal computers equipped with an FP-11 floating point unit and running the RT-11 operating system.

The FORTRAN-77/RT-11 software consists of two principal components:

- A FORTRAN-77 compiler, that translates a source program into object code.
- A collection of routines (facilities and services) that a program may need while it is executing. This collection of routines is called the Object Time System (OTS).

NOTE

Unless otherwise noted, the term FORTRAN-77 is used in this manual to mean PDP-11 FORTRAN-77/RT-11.

1.1 OVERVIEW

To transform a FORTRAN-77 source program into an executing job, you need to perform three steps:

1. Compile the program, to create a relocatable object module.
2. Link the program, to bind the object module with necessary external routines.
3. Execute the program (and debug it if necessary).

You compile a program by invoking the FORTRAN-77 compiler and specifying the source files to be processed; then you link it into an executable form, called a memory image by invoking the RT-11 linker and specifying the object module to be processed. Finally, you execute the memory image using the RUN, FRUN, or SRUN command.

Figure 1-1 illustrates the process of transforming a FORTRAN-77 source program into an executing program.

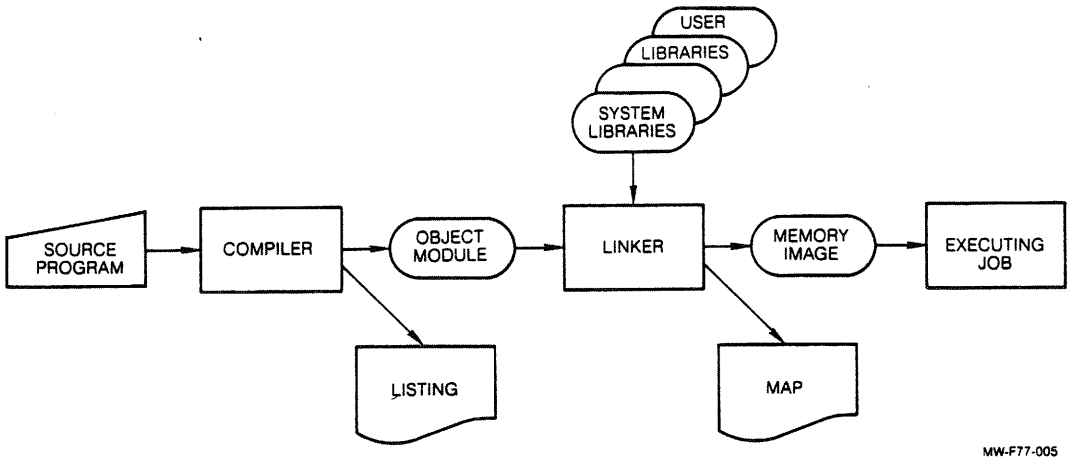


Figure 1-1 Preparing a FORTRAN-77 Program for Execution

You invoke the compiler or the linker by entering a keyboard command line that specifies the desired function, the input files, the output files, and any desired command options. Command lines are either written in Keyboard Monitor format for commands processed directly the Keyboard Monitor, or in Command String Interpreter (CSI) format for commands processed directly by utility programs or language processors. Keyboard Monitor Language is the RT-11 implementation of Digital Command Language (DCL).

Input files and output files are specified in command lines by file specifications.

Optional command inputs are specified with special command mnemonics called options or switches. Switches are appended to command words and file specifications.

1.2 RT-11 FILE SPECIFICATIONS AND SWITCHES

This section details the naming conventions for files used by the RT-11 operating system. It also describes the use of various command line modifiers, called switches.

1.2.1 RT-11 FILE SPECIFICATIONS

For each RT-11 utility program you use, you must specify the input files to be processed and (optionally for the FORTRAN-77 compiler and the linker) the output files to be produced.

The format of a file specification for an RT-11 program is as follows:

```
device:filename.filetype
```

device

The device where a file is stored or is to be written.

filename

The name of the file. A filename value can be from zero to six characters in length.

filetype

An indication of the kind of information stored in the file. A filetype value can be up to three characters long.

You need not explicitly state all the elements of a file specification each time you compile, link, or execute a program. The only part of a file specification that is usually required is the file name. If you omit any other part of the file specification, a default value is used. Table 1-1 summarizes the file specification default values.

If you request compilation of a source program specified only by a file name, the compiler searches for a file with the specified file name that

- is stored on the default device DK:
- has a filetype of FOR

For example, assume that your default device is DL0:, and that you supply the following input or output file specification to the compiler:

```
CIRCLE
```

For input, the compiler searches device DL0: for CIRCLE.FOR. For output, the compiler generates the file CIRCLE.OBJ and stores it on device DL0:.

Frequently a command will allow a list of files. In many cases, this list contains only one file specification; however, File Factoring allows for the simple expression of several different input files that reside on the same device as in:

```
DL1:(TEST1,TESTA,TESTC).FOR
```

The command shown above has the same meaning as and is easier to use than the next command:

```
DL1:TEST1.FOR,DL1:TESTA.FOR,DL:TESTC.FOR
```

Factoring is a general method of string replacement. (Note that the file factoring feature can only be used when commands are typed as Keyboard Monitor Commands. File Factoring cannot be used with the Command String Interpreter.) When you use factoring, the device name outside the parentheses applies to each file specification inside the parentheses. Without factoring, the system interprets each file specification to be "DK:filespec" unless you explicity specify another device name.

The example can be further reduced by the command

```
DL1:TEST(1,A,C).FOR
```

which has an identical meaning.

Care must be taken when using command factoring since the expanded command is limited in length to 80 characters. No error message is generated when a command line is truncated, but an error message may

be generated when the truncated command line is executed. This error message will most likely not be indicative of the true nature of the problem. The following example shows how a command line expands after factoring.

Original command line:

```
FORTRAN/SHOW:3/LIST:TT:  DY1:TEST(1,A,B)
```

Resulting command line (after factoring):

```
FORTRAN/SHOW:3/LIST:TT:  DY1:TEST1,DY1:TESTA,DY1:TESTB
```

NOTE

If the "FORTRAN" command is to be used as in these examples, then the compiler file F77.SAV or F77XM.SAV must first be renamed to FORTRA.SAV. See section 2.11 of the PDP-11 FORTRAN-77/RT-11 Installation Guide/Release Notes.

If all the file names you specify are six characters long, you should specify no more than five files. You can read more about factoring in the RT-11 System User's Guide.

Table 1-1
RT-11 Default File Specifications

Optional Element	Default Value	
device	User's current default device	DK:
filetype	Depends on usage:	
	Command file	COM
	Input to compiler	FOR
	Output from compiler	OBJ
	Input to linker	OBJ
	Output from linker	SAV
	Input to RUN command	SAV
	Compiler source listing	LST
	linker map listing	MAP
	linker library input	OBJ
	Input to executing program	DAT
	Output from executing program	DAT

1.2.2 COMMAND OPTIONS (SWITCHES)

Command switches are option qualifiers you can use in command lines to specify optional command instructions or inputs -- for example, to specify that the compiler should produce a listing file. There are two distinct forms of switches.

One type is used for Keyboard Monitor Commands while the other type is used with commands that are interpreted by the Command String Interpreter (CSI). Command switches are appended to other entities in a command line and have the form:

```
/switch[:val]
```

switch

is a mnemonic that specifies a certain instruction to the compiler or linker.

val

is a parameter consisting of an octal or decimal number, or a string of characters.

1.2.2.1 KEYBOARD MONITOR OPTION SWITCHES

You use Keyboard Monitor switches to specify optional instructions to a Keyboard Monitor command or to specify special attributes or processing commands for input or output files. A Keyboard Monitor switch consists of a slash followed by an option word and an optional qualifier separated from the option word by a colon.

For example, the FORTRAN compiler accepts a switch that causes a listing to be made. The switch used to create this listing is /LIST. /LIST has an optional qualifier that specifies the listing's filename. A Keyboard Monitor command line that uses this switch follows:

```
.FORTRAN/LIST:LSTFIL
```

Since neither a device nor a filetype were specifically entered, the default device DK: and the default listing filetype .LST are used to create the output file DK:LSTFIL.LST.

Certain compiler switches may be followed by a numeric value. This value is always interpreted as a decimal number. There is no method of specifiying an octal number in a Keyboard Monitor command without first converting it into its decimal equivalent.

1.2.2.2 CSI COMMAND SWITCHES

The CSI format for switches is more restrictive than Keyboard Monitor format: the mnemonic for the switch is limited to a single letter and the qualifier is restricted to signed numeric values or three character ASCII strings. Numeric arguments are assumed to be octal unless a decimal point is included in the number. You can use a minus sign (-) to denote negative octal or decimal numbers.

For example, the CSI compiler switch /C is used to specify the number of continuation lines the FORTRAN-77 compiler will accept. If you wish the compiler to limit the number of continuation lines to 10 (decimal) lines then you can use either

```
/C:10.  or /C:12
```

In the first case the decimal point establishes 10 as a decimal value while in the second case the absence of the decimal point causes 12 to be interpreted as an octal number.

Some examples of valid CSI compiler switches are:

```
/I
/S:ALL
/C:25.
```

1.3 COMPILING A FORTRAN-77 PROGRAM

This section contains information for the user who wants to compile a FORTRAN-77 program on an RT-11 system. The compiler can be run by using one of three distinct Keyboard Monitor commands:

- the FORTRAN command
- the R command
- the RUN command

You should first determine the filename given to the compiler when it was installed. It may be stored as F77.SAV, F77XM.SAV, or as FORTRA.SAV. It must be named FORTRA.SAV in order to use the FORTRAN command. In the R and RUN command examples below, you should substitute your compiler's file name in place of "F77". Also note that the R and RUN commands give you access to all of the compiler's switches, some of which have no equivalent qualifiers when the FORTRAN command is used.

1.3.1 COMPILING WITH THE FORTRAN COMMAND

You can use the keyboard monitor's FORTRAN command to compile programs with FORTRAN-77 if the compiler is stored on the system device under the name FORTRA.SAV. The Monitor Command switches available for FORTRAN-77 are a subset of those available for RT-11 FORTRAN-IV. Use of FORTRAN keyboard command switches not defined in Table 1-2 can lead to unpredictable results.

You invoke the FORTRAN-77 compiler with the FORTRAN command as follows:

```
FORTRAN [/options] filespec[/option...] [...filespec[/option...]]
```

or

```
FORTRAN [/options]
FILES? filespec[/options...] [...filespec[/option]]
```

/options (or switches)

optionally included to control the output files and the compiler.

filespec

Specifies an RT-11 file or files that contain the source program. The default file type is .FOR. A concatenated set of input files may be specified by using the plus (+) operator. The action of this operator is the same as if all the files were

copied together and submitted to the compiler as one file. The default name of the output file and list file will be that of the first file named.

There are many compiler option switches. Some of these switches are available only through lines processed by the Command String Interpreter using the R or RUN commands. Table 1-2 lists each available switch and its action when the FORTRAN command is used.

In order to compile into separate object files a FORTRAN-77 program MAIN.FOR with subprogram units SUB1.FOR, SUB2.FOR, and UTILS.FOR you could enter

```
FORTRAN MAIN,SUB(1,2),UTILS
```

The result would be four files: MAIN.OBJ, SUB1.OBJ, SUB2.OBJ, and UTILS.OBJ.

If you wanted to compile the same source files into only one object file you could enter

```
FORTRAN MAIN+SUB(1,2)+UTILS
```

The result would be one file, MAIN.OBJ.

Table 1-2

FORTRAN Options Available From Keyboard Monitor Under RT-11

Option	Explanation
/ALLOCATE:n	Use this option with /LIST or /OBJECT to reserve space on a device for the output file. The argument size represents the number of blocks of space to allocate. The meaningful range for this value is from 1 to 65535. A value of -1 is a special case that creates the largest file possible on a device.
/EXTEND	Use this option to change the right margin for source input lines from column 72 to 132. If this switch is specified, the ANSI standard extension flagger will issue an informational diagnostic (one per line) for source lines extending beyond column 72
/I4	Use this option to allocate two words for the default integer data type so that integers with absolute values greater than 32767 may be represented. FORTRAN-77 normally uses one-word integers.
/NOLINENUMBERS	You can specify this option to eliminate the generation of traceback code in the object file. This is equivalent to using the /S:NON option in CSI command format.

Table 1-2

FORTRAN Options Available From Keyboard Monitor Under RT-11
(continued)

Option	Explanation
/LIST[:filespec]	You must specify this option to produce a FORTRAN-77 compilation listing. Anytime you type a colon after the /LIST option (/LIST:) you must specify a device or a file specification after the colon. The /LIST option has different meanings depending on where you place it in the command line. The /LIST option produces a listing on the device "LP:" when /LIST follows the FORTRAN command. For example, the following command line produces a line printer listing after compiling a FORTRAN source file: .FORTRAN/LIST MYPROG Remember that file options apply only to the file (or group of files that are separated by plus signs) that they follow in the command string. For example: .FORTRAN A/LIST,B This command compiles A.FOR, producing A.OBJ and A.LST. It also compiles B.FOR, producing B.OBJ. However, it does not produce any listing file for the compilation of B.FOR.
/OBJECT[:filespec]	Use this option to specify a file name or device for the object file. Because FORTRAN creates object files by default, the following two commands have the same meaning: .FORTRAN A .FORTRAN/OBJECT A Both commands compile A.FOR and produce A.OBJ as output. The /OBJECT option functions like the /LIST option; it can be either a command option or a file qualifier.
/NOOBJECT	Use this option to suppress creation of an object file. As a command option, /NOOBJECT suppresses all object files; as a file option, it suppresses only the object file produced by compilation of the related input files.

Table 1-2

FORTRAN Options Available From Keyboard Monitor Under RT-11 (continued)

Option	Explanation
	In this command, for example, the system compiles A.FOR and B.FOR together, producing files A.OBJ and B.LST. It also compiles C.FOR and produces C.LST, but does not produce C.OBJ .FORTRAN A+B/LIST,C/NOOBJECT/LIST
/ONDEBUG	Use this option to include debug lines (those that have a D in column one) in the compilation. Therefore, you do not have to edit the file to include these lines in the compilation or to logically remove them. This option is useful in debugging a program. You can include messages, flags, and conditional branches to help you trace program execution and find an error.
/RECORD:length	Use this option to override the default record length for sequentially formatted input and output, usually 136 characters. The meaningful range for length is from 4 to 4095.
/SHOW[:type]	Use this option to control FORTRAN listing output. The argument type represents a code that indicates which listings the compiler is to produce. type has the following allowable values: 0 Minimal listing file: diagnostic messages and program section summary only. 1 Source listing and program section summary. 2 Source listing, program section summary, and storage map (default). 3 Source listing, assembly code, program section summary, and storage map. If you specify no code, the default value is 2.

Table 1-2

FORTRAN Options Available From Keyboard Monitor Under RT-11 (continued)

Option	Explanation
/STATISTICS	Use this option to obtain compilation statistics in the listing. The statistics include workfile access statistics and the number of pages of dynamic memory available and used.
/SWAP	Use this option to permit the /USR (User Service Routine) to swap over the FORTRAN-77 OTS in memory. This is the default operation.
/NOSWAP	This option keeps the USR resident during execution of a FORTRAN program. This may be necessary if the FORTRAN program uses some of the RT-11 System Subroutine Library calls (see Chapter 4 of the RT-11 Programmer's Reference Manual). If the program frequently updates or creates a large number of different files, making the USR resident can improve program execution. However, the cost for making the USR resident is 2K words of memory.
/UNITS:n	Use this option to override the default number of logical units (6) that can be open at one time. The maximum value you can specify for n is 16.
/WARNINGS	Use this option to include warning messages in FORTRAN compiler diagnostic error messages. These messages call certain conditions to your attention, but do not interfere with the compilation. A warning message is printed, for example, if you specify a variable name with more than six characters.
/NOWARNINGS	Use this option to exclude warning messages in FORTRAN compiler diagnostic error messages. This is the default setting.

1.3.2 COMPILING WITH THE R COMMAND

You can invoke the FORTRAN-77 compiler with the Keyboard Monitor Command

```
R F77
```

The R command requires that the compiler reside on the system device SY: and does not allow for an argument list to be present on the command line. The presence of an argument list in an R command line

will result in an error message from the Keyboard Monitor. When the FORTRAN-77 compiler is ready to accept a command it will display the following prompt

```
*
```

This prompt is issued by the RT-11 Command String Interpreter (CSI). When you use the compiler in this fashion, entering command to the * prompt from the CSI, the compiler is said to be in "Interactive Mode". To enter a succession of compilation commands to the compiler in interactive mode, type one command line after each prompt, followed by a carriage return, until all commands are entered. Each command line must specify the appropriate input and output files for the program module to be compiled, and any optional switches desired. Type CTRL/C to return to the Keyboard Monitor.

For example, to compile the FORTRAN programs WINKN, BLINKN, and NOD into separate object modules, you can enter a succession of commands as follows:

```
.R F77 <RET>  (From this point on, the compiler issues the *
                 prompt.)

*WINKN,WINKNP=WINKN <RET>
*BLINKN,BLINKN=BLINKN <RET>
*NOD,NOD=NOD <RET>
*^C
```

Note that the compiler issues the * prompt each time you enter a command until a CTRL/C (^C) is typed to return control to the Keyboard Monitor.

1.3.3 COMPILING WITH THE RUN COMMAND

The RUN command is similar to the R command, but there are some differences. First, you can use the RUN command to use the compiler when it resides on a device other than the system device SY:. It also allows you to enter a full command line, in the form accepted by the Command String Interpreter as part of the RUN command. Finally, the RUN command can compose a CSI command string from command line text entered in CCL format.

Note that the RUN command cannot be used on Professional 300 series computers running the F77XM version of the compiler. Users of these systems should use the FORTRAN or R commands, described earlier.

The RUN command should be used to invoke the FORTRAN-77 compiler if:

- The compiler is not located on the system disk.
- A command in Command String Interpreter format is desired on the same command line.
- A command in Concise Command Language (CCL) format is on the same command line.

If the FORTRAN-77 compiler is not on the system disk, use the RUN command to start the compiler. You can use it in exactly the same way as you would if it were invoked using the R keyboard command.

For example, to run the compiler in interactive mode from the current default device, you enter:

```
RUN F77
```

or to run the extended memory version of the F77 compiler from the virtual disk VM:

```
RUN VM:F77XM
```

In either case the compiler responds with the prompt:

```
*
```

from the Command String Interpreter and then accepts commands in exactly the same manner as the R command. For example, if you want to compile FORTRAN-77 programs WINKN, BLINKN, and NOD into separate object modules when the compiler is located on the device DL2:, you could enter a sequence of commands such as

```
.RUN DL2:F77                    (From this point on, the compiler
*WINKN=WINKN                     issues the * prompt)
*BLINKN=BLINKN
*NOD=NOD
*^C
```

This succession of commands produces no listings and assumes the input and output device DK:.

The RUN command also allows a single command string to share the same command line. To compile the file WINKN into an object module of the same name, and to produce a listing on the line printer LP:, you could enter

```
.RUN SY:F77 WINKN,LP:=WINKN
```

All of the functionality of the interactive mode is preserved except that only one command line is allowed.

If the compiler is located on the system device, Concise Command Language (CCL) allows an even shorter command sequence where the RUN command can be omitted entirely. The system device is searched for a memory image and the command line treated as a CSI string. This reduces the command from the example above to:

```
.F77 WINKN,LP:=WINKN
```

There is one side effect to the use of CCL that should be understood. CCL attempts to give some of the functionality of DCL. If you were to type

```
.F77 WINKN WINKN.FOR
```

CCL would construct a command line reversing the positions of WINKN.OBJ and WINKN.FOR, then inserting an "=" between them, resulting in an equivalent command of:

```
.RUN SY:F77 WINKN.FOR=WINKN
```

The effect would be to replace the FORTRAN source program, WINKN.FOR with another file, named WINKN.FOR containing the object code. For further discussion of CCL constructs refer to RT-11 System User's Guide.

Table 1.3 describes the action of the various compiler switches and the syntax for their modifiers.

Table 1-3

RT-11 FORTRAN-77 Command String Switches

Interactive Mode Switch	Explanation
[n]	Use this option to reserve space on a device for the output files. It follows the file name specification. The argument n represents the number of blocks of space to allocate. The meaningful range for this value is from 1 to 32767. A value of -1 is a special case that creates the largest possible file on the device.
/A	Produces a statistics report at the end of each module listing.
/B	Specifies that the compiler is to provide symbol table information for use by the PDP-11 FORTRAN-77 symbolic debugger. When you use the /B qualifier, you should also use the /O qualifier to disable compiler optimization. The symbolic debugger is not currently supported under RT-11.
/C:n	Specifies the maximum number of continuation lines in the program. (You may have fewer than n continuation lines.) The value of n may range from 0 to 99; the default value is 19. Note that each level of nesting of an INCLUDE statement causes the maximum number of continuation lines to be decreased by two.
/D	Requests compilation of lines with a D in column one. These lines are treated as comment lines by the default mode. (see the PDP-11 FORTRAN-77 Language Reference Manual for further information).
/E	Specifies that the compiler interprets FORTRAN source text that extends beyond column 72, into column 132 of an input record. If /E is specified, and the ANSI standard extension flagger is invoked by the command switch /Y:SRC, the compiler issues an informational diagnostic (one per line) for those source lines extending beyond column 72. This flag is normally off.

Table 1-3

RT-11 FORTRAN-77 Command String Switches
(continued)

Interactive Mode Switch	Explanation
/F:n	Sets the length of the workfile in disk blocks. Default is 128 (decimal). Use this switch to increase the size of the workfile for larger compilations. The value n is interpreted as an octal value unless a decimal point is used after the number. This value may be adjusted when the compiler is installed. See the PDP-11 FORTRAN-77/RT11 INSTALLATION GUIDE for complete details.
/I	Specifies that array references are to be checked to ensure that they are within the array address boundaries specified. However, array upper bounds checking is not performed for arrays that are dummy arguments for which the last dimension bound is specified as * or 1. For example: DIMENSION B(0:10,0:*) or DIMENSION A(1) The switch is OFF by default.
/K	Causes the current switch settings to be retained (latched) for subsequent compilations in interactive mode. Normally, switch settings are restored to their default values before processing each command line. This switch is convenient for compiling a series of programs in interactive mode with the same switch settings. Disabled by default.
/L:n	Specifies listing options. The value of n may range from 0 to 3. The meaning of each value is as follows: 0 Minimal listing file: diagnostic messages and program section summary only. 1 Source listing and program section summary. 2 Source listing, program section summary, and storage map (default). 3 Source listing, assembly code, program section summary, and storage map.

Table 1-3

RT-11 FORTRAN-77 Command String Switches
(continued)

Interactive Mode Switch	Explanation
/N:n	Select the number of logical units available to the compiled program to be n. See chapter 2 for a discussion of logical unit numbers. The default is 6 logical units available.
/O	Precludes optimization of compiler generated code. Use this switch to defeat any optimizations the compiler may be making to your code.
/Q	Produces a wide 132-column map listing instead of an 80-column listing. The compiler normally produces the narrower listing.
/R:n	Specifies the maximum record length (in bytes) for run time I/O (4 < n < 4095).
/S:xxx	Controls the amount of extra code included in the compiled output for use by the OTS during error traceback. This code is used in producing diagnostic information and in identifying which statement in a source program caused an error during execution. /S:xxx can have the following forms:
	/S — Same as /S:NON
	/S:ALL — Error traceback information is compiled for all source statements and function and subroutine entries.
	/S:LIN — Same as /S:ALL.
	/S:BLO — Traceback information is compiled for subroutine and function entries and for selected source statements. The source statements selected by the compiler are initial statements in sequences called blocks (see Section 5.2.3 for the definition of a block).
	/S:NAM — Traceback information is compiled only for subroutine and function entries.
	/S:NON — No traceback information is produced.
	The default value is /S:BLO.

Table 1-3

RT-11 FORTRAN-77 Command String Switches
(continued)

Interactive Mode Switch	Explanation
	The /S:ALL setting is generally advisable during program development and testing. The default setting /S:BLO is appropriate for most programs in regular use. The setting /S:NON may be used for obtaining fast execution and minimal code, but it provides no information to the OTS for diagnostic message traceback.
/T	Allocates two words for the default length of integer and logical variables. Normally, single storage words are the default allocation for all integer or logical variables not given an explicit length definition (such as INTEGER*2, LOGICAL*4). The default setting allocates one word of storage for integers and logical variables. See Section 4.2 for further information.
/U	Inhibits swapping of the USR routines over the compiled program when it is executed.
/V	Types the FORTRAN-77 compiler identification and version number on your terminal.
/W	Disables compiler warning diagnostics (W-class messages; see Section C.1.1). If /W is set, no warning messages are issued by the compiler. The default is /W not set.
/X	Disables the compiler's FORTRAN-77 features. See Chapter 3 for further discussion of this flag's action.
/Y:xxx	Directs the compiler to check your source code for extensions to ANSI standard (X3.9-1978) FORTRAN at the full-language level. If the compiler finds extensions, it flags them and produces informational diagnostics about them. (To receive informational diagnostics, you must set the warning switch /W.) Although RT-11 FORTRAN-77 conforms to the ANSI FORTRAN standard at the subset level, the compiler flags only those features that are extensions to the full language. See Appendix G for a list of the flagged extensions. The /Y:xxx switch can take the following forms: /Y Informational diagnostics for syntax extensions /Y:ALL Informational diagnostics for all detected extensions

Table 1-3

RT-11 FORTRAN-77 Command String Switches
(continued)

Interactive Mode Switch		Explanation
	/Y:NON	No informational diagnostics
	/Y:SRC	Informational diagnostics for lowercase letters and tab characters in source code
	/Y:SYN	Same as /Y The default value is /Y:NON See Section C.2 for a list of compiler diagnostic messages.
/Z		Directs the compiler to specify pure code and pure data sections as read-only. See Section 3.3 for a description of program section attributes. /Z is off by default.

The default settings of the compiler switches can be summarized as:

/C:19./F:128./L:2/N:6/R:136./S:BLO/Y:NON

which expands to:

- 19 continuation lines
- a work file of 128 blocks
- a source listing with program section summary and storage map
- default maximum formatted record length of 136 characters
- traceback for module names and code at block level
- 6 logical units available
- no special ANSI standards checking
- no debug symbol table is produced
- lines with D in column 1 are not compiled
- source characters beyond column 72 are not compiled

(continued on next page)

Default Compiler Switch Settings
(continued)

- array references are not checked to be within defined bounds
- object code is optimized
- integers occupy 1 16-bit word (INTEGER*2)
- the USR may swap over compiler generated code
- warning messages are not produced
- FORTRAN-77 syntax is accepted

1.4 LINKING A FORTRAN-77 PROGRAM

The linker is a system program that binds relocatable object modules to form an executable memory image.

You invoke the linker with the Keyboard Monitor's LINK command, or you may use the R or RUN command. The LINK program is described in Section 1.4.1.

The object modules to be linked can come from user-specified input files, user libraries, or system libraries. The linker resolves references to symbols defined in one module and referred to in other modules. Should any symbols remain undefined after all user-specified input files are processed, the linker automatically searches the system object library SY:SYSLIB.OBJ to resolve them.

The default FORTRAN-77 object time system library is normally part of either the system object library, SYSLIB.OBJ, or it is a separate object library that may be named either F77OTS.OBJ or FORLIB.OBJ.

You can also use the linker to build memory images with overlay structures. For additional information about the linker and linker options, refer to the RT-11 System User's Guide.

1.4.1 LINKING WITH THE LINK COMMAND

The RT-11 LINK program combines one or more user-written program units with selected routines from any user libraries and the default FORTRAN-77 OTS Library to form an executable memory image file. LINK generates a single runnable memory image file and an optional load map from one or more object files created by the FORTRAN-77 compiler, MACRO assembler, or other language processor.

The default types for the executable file are .SAV for a background or mapped environment program, and .REL for a foreground program. The default output device is DK:.

The default name of the .SAV or .REL file is that of the first concatenated input object file specified. When F77OTS resides in SYSLIB, the required elements of the FORTRAN library will be linked

automatically since any undefined global references are correlated and resolved through SYSLIB.

The LINK command adheres to the following syntax:

```
LINK[/option...] filespec[/option...][,...filespec[/option...]]
```

or

```
LINK[/option...]
FILE?  filespec[/option...][,...filespec[/option]]
```

where "filespec" represents the file to be linked and "options" are those described in Table 1-4.

Table 1-4

Linker Options Available from the RT-11 Keyboard Monitor

Option	Explanation
/ALLOCATE:n	Guarantee space for a maximum file of n blocks.
/ALPHABETIZE	Lists program's global symbols alphabetically in the load map.
/BITMAP	Creates a memory usage bitmap (default setting).
/BOTTOM:n	Specifies a bottom address for a background program.
/BOUNDARY:value	Starts a specific program section on a particular address boundary. Argument value, must be a power of 2. Prompts you: Boundary Section? Enter name of section, then <ret> .
/DEBUG[:filespec]	Links ODT or other debugging utility to the linked program.
/EXECUTE[:filespec]	Designates the executable file.
/EXTEND:n	Extends a program section to octal value n. Prompts you: Extend Section?
/FILL:n	Initializes unused locations in the load module to n (an octal value).
/FOREGROUND [:stacksize]	Generates a .REL file for a foreground link.

Table 1-4

Linker Options Available from the RT-11 Keyboard Monitor

Option	Explanation
/INCLUDE	Allows subsequent entry at the keyboard of global symbols to be taken from any library and included in the linking process. When the /INCLUDE option is typed, the linker prints: Library search? Reply with the list of global symbols to be included in the load module. Press the carriage return key <RET> to enter each symbol in the list.
/LDA	Produces executable file in LDA format for use with the Absolute Loader.
/LIBRARY	same as /LINKLIBRARY. (Included for compatibility with other systems).
/LINKLIBRARY:[filespec]	This option is ignored unless a file specification is typed. The file specification is included as an object module library in the linking operation.
/MAP[:filespec]	Produces a link map on the listing device LP: or in the file specified.
/NOEXECUTE	Does not create a .SAV file.
/PROMPT	Causes the linker and librarian to prompt for CSI formatted commands. The linker treats the command strings as continuation lines until a // is seen. /PROMPT is equivalent to // mode of continuation. Use this option to specify overlays. For example: .LINK/PROMPT ROOT *OVR1/O:1 *OVR2/O:1 *OVR3/O:2 *OVR4/O:2// This creates two overlay regions with two segments each.
/ROUND:n	Rounds up a section so that the root is a whole number multiple of of n (a power of 2). Prompt: Round section:
/RUN	For background jobs only, executes the resulting .SAV file.

Table 1-4

Linker Options Available from the RT-11 Keyboard Monitor

Option	Explanation
/SLOWLY	Allows largest memory area for symbol table.
/STACK[:n]	Modifies the stack address (default is loc. 42). Give an octal value (:nnnnnn) or else system prompts for a global symbol: Stack Symbol?
/SYMBOLTABLE[:filespec]	Creates a file containing symbol definitions for all global symbols. Enter the symbol table file specification as the third output specification in the LINK command.
/TOP:value	Specifies the highest address to be used by relocatable code. The argument value represents an unsigned, even, octal number.
/TRANSFER[:n]	Prompts for a global symbol to be used as the starting address of the program. The user can specify a starting address (represented by n).
/WIDE	Sets the number of columns for the width of the link map to 6 global name definitions per line. The default width is normally 3 for an 80 column wide listing.
/XM	Enables special .SETTOP features in the XM monitor. This option allows a virtual job to map a scratch region in extended memory with the .SETTOP program request. See the RT-11 Programmer's Reference Manual for further information on these special .SETTOP features.

Examples of linker options under RT-11 are

1) LINK A,B,C	Links A.OBJ, B.OBJ and C.OBJ on DK: and creates A.SAV on DK:
2) LINK/MAP A	Links A.OBJ and creates A.SAV on DK: and a map on LP:

Examples of linker options under RT-11 (continued)

3) LINK/MAP:RK1:/EXE:RK0: A,B,C

Links A.OBJ, B.OBJ, and C.OBJ. The map A.MAP goes to RK1: and the executable file A.SAV goes to RK0:.

4) LINK/MAP/EXE:F00 B,C,D,E,LIB

Links B.OBJ, C.OBJ, D.OBJ,E.OBJ, and the Library LIB.OBJ to create F00.SAV on DK: and a map on LP:.

1.4.2 LINKING USING THE R COMMAND

You can run the linker using the R command instead of using the Keyboard Monitor LINK command. The format of the R command is

```
R LINK
```

The R command searches the system disk SY: for the LINK program and starts it executing. The linker returns with a prompt when it is ready to accept input from your terminal:

```
*
```

To enter a succession of link commands to the CSI you type one command line after each prompt, followed by a carriage return, until all commands are entered. Each command line must specify appropriate files of the program and subprogram modules to be linked, and any optional switches desired. After the linker has finished linking a job it prints another asterisk prompt. You can then type CTRL/C to exit the linker, or you can issue another link command.

For example, if you want the object files WINKN, BLINKN, and NOD linked into a executable memory image file, you can enter a succession of commands as follows:

```
R LINK <RET>                    (From this point on the compiler
                                 issues the * prompt.)

*WINKN,WINKN=WINKN,BLINKN,NOD
*^C
```

Note that the linker types the asterisk (*) prompt whenever it awaits user input. The result in the example is two files, WINKN.SAV, an executable memory image, and WINKN.MAP, a load map of the memory image file. Both are placed on the default device DK:.

When invoked with the R command, or the RUN command with no arguments, the RT-11 linker accepts the first command string in the form

```
[bin-filespec][,map-filespec][,stb-filespec] = [infiles-list]
```

Option switches may be included on the command line. Table 1-5 lists the various switches and their required location within a command line.

After the first command line, ensuing lines have the form:

infiles-list[/option]

bin-filespec

The file specification of the linker's output load module file. This may be a memory image file for execution under RT-11 or it may be an absolute loader file.

map-file

The file specification of the map output file. This file specification may be omitted if no memory image map file is desired. A file type value of MAP is assumed if no file type is specified.

stb-filespec

The file specification for an optional symbol definition file. This file, also called the STB file, contains all global symbol table definitions. See the RT-11 Software Support Manual for more information on the content and structure of the STB file.

infiles-list

The list of input files that contain compiled FORTRAN-77 object modules. (This list may also contain compiled or assembled libraries and modules that were written in a language other than FORTRAN-77, such as MACRO.) In many cases, this list contains only one file specification; however, when there is more than one specification, you must separate the individual specifications with commas. Only a file name is normally required; a file type value of OBJ is assumed.

The form of the infiles-list is:

obj-filespec[/option...][,...obj-filespec[/option]]

If you do not specify an output file, the linker assumes that you do not want the associated output. For example, if you do not specify the load module or the load map (by using one comma in front of the equal sign) the linker prints only error messages, if any errors occur.

If the linker detects a syntax error in a command string, it prints an error message. You can then retype a new command string following the asterisk. Similarly, if you specify a nonexistent file, a warning diagnostic is printed. Control then returns to the Command String Interpreter, an asterisk prompt is printed, and you can reenter the command string.

Table 1-5 lists the options associated with the linker. You must precede the letter representing each option by the slash character. Options must appear on the line indicated if you continue the input on more than one line, but you can position them anywhere on the line. Under the column titled Command Line is listed the line in the command string in which the option can appear.

Table 1-5

Linker Options Available from RT-11 Interactive Mode

Option (Switch)	Command Line	Meaning
/A	First	Lists global symbols in program sections in alphabetical order.
/B:n	First	Changes the bottom address of a program to n. (invalid with /H and /R)
/C	Any	Continues input specification on another command line. Used also with /O.
/D	First	Allows the global symbol you specify to be defined once in each segment that references that symbol. These symbols must be defined in library modules.
/E:n	First	Allows a specified program section to be extended to the value given. When the /E switch is specified, the linker prints: EXTEND SECTION? Reply with the name of the program section, whose length then becomes greater than or equal to the value given. It will be "greater than" when the object code requires a space larger than the value specified.
/F	First	Instructs the linker to use the default FORTRAN library, FORLIB.OBJ, to resolve any undefined global references. Note that this option should not be specified in the command line when FORLIB (F77OTS) has been incorporated into SYSLIB.OBJ.
/G	First	Adjusts the size of the linker's library directory buffer to accommodate the largest multiple definition library directory.
/H:n	First	Specifies the top (highest) address to be used by the relocatable code in the load module. The high value must be specified or the error message /H NO VALUE will be returned. The high value must be an unsigned even octal number. If the value is odd, /H ODD VALUE error is returned. If the value is not large enough for the relocatable code, /H VALUE TOO LOW error message is returned.

Table 1-5

Linker Options Available from RT-11 Interactive Mode

Option (Switch)	Command Line	Meaning
		Use care with the /H switch because most RT-11 programs use the free memory above the relocatable code as a dynamic working area for I/O buffers, device handlers, symbol tables, etc. The size of this area varies with different memory configurations as programs linked to a high address may not run in a system with less physical memory. /R, /B, and /H are mutually exclusive and give the error /x-BAD SWITCH. /H is the counterpart to /B.
/I	First	Includes in the memory image the library object modules that declare the specified global symbols.
/K:n	First	Puts the specified value in word 56 of the image file block 0. This value states that the program requires nK words of memory. Range for the required value is 1 through 28.
/L	First	Produces an output file in LDA format
/M or /M:n	First	Specifies the stack address at the terminal keyboard or via n.
/N	First	Produces a cross-reference in the load map of all global symbols defined during the linking process.
/O:n	Any but the first	Indicates that the program has an overlay structure: n specifies the overlay region to which the module belongs. Invalid with /L.
/P:n	First	Allows the linker to maintain as many as "n" library routines.
/Q	First	Lets you specify the base addresses of up to eight root program sections. Invalid with /H or /R.
/R[:n]	First	Produces an output file in relocatable image format for execution as a foreground or system job and optionally indicates stack size for a foreground job. Invalid with /B, /H, /K, and /L.
/S	First	Allows the maximum amount of space in memory to be available for the linker's symbol table. (This switch should only be used when a particular link operation causes a symbol table overflow.)

Table 1-5

Linker Options Available from RT-11 Interactive Mode

Option (Switch)	Command Line	Meaning
/T[:n]	First	Cause the linker to prompt you for a global symbol that represents the transfer address or that sets the transfer address to the value n.
/U:n	First	Prompts the user with "ROUND SECTION": The user replies with the name of the program section to be rounded up, that must be in the root segment. The value given must be a power of 2. The specified section will be rounded up to a size that is a whole number multiple of n. For example, to make the first overlay start on a block boundary so that the root section and the first overlay region can be read into physical memory with only one device read operation you would use: /U:1000. If the specified section is not found, the error message will be "ROUND SECTION NOT FOUND".
/V	First	Enables special .SETTOP and .LIMIT features provided by the XM monitor. Invalid with /L.
/V:n[:m]	Any but first	Indicates that an extended memory overlay overlay segment is to be mapped in virtual region n, and optionally in partition m.
/W:n	First	Specifies the width of the map to be produced. The value is the number of ENTRY or ADDR combinations to print across the page. If no /W is given, the default is 3 (normal for 80 column paper). If only /W is given, n defaults to 6, that effectively utilizes a 132 column page. The useful range is 1 through 7.
/X	First	Intended for RSTS/E; not normally used for RT-11. Meaning: do not output (Xmit) the bitmap if code below 400. Locations 360-377 in block 0 of the load module are used for the bitmap. The linker nomally stores the program usage bits in these eight words. Each bit represents 256-word block of memory. This information is used by the R, RUN, an GET commands when loading the program. Therefore care should be exercised in using this switch.
/Y:n	First	Starts a specific program section in the root on a particular address boundary. Invalid with /H
/Z:n	First	Sets unused locations in the load module to the value n.

Table 1-5

Linker Options Available from RT-11 Interactive Mode

Option (Switch)	Command Line	Meaning
//	First and last	This method provides an alternative to the /C (Continue) switch that must be given on every line except the last. The "//" switch allows additional lines of command string input without the need for a /C at the end of each continuation line. "//" is typed on the first command line and the linker will continue to request input until the next occurrence of "//". The second occurrence terminates specification of command string input, and may be at the end of the last object file name or on a command line by itself. CAUTION: The use of /C and // cannot be mixed in a link command string input sequence. Example: .R LINK *LINK,LP:=MYPROG/B:500/W// *MODOV1/O:1 *MODOV2/O:1 *MODOV3/O:1 *MODOV4/O:1 *MODOV5/O:1//

1.4.3 LIBRARY USAGE ON RT-11 SYSTEMS

You can create a library of commonly used assembly language and FORTRAN-77 functions and subroutines through the system program LIBR, which provides for library creation and modification. The Librarian chapter of the RT-11 System User's Guide describes the LIBR program in detail.

Include a library file in the LINK command string simply by adding the file specification to the input file list. LINK recognizes the file as a library file and links only the required routines. The LINK command string

```
*LOAD=MAIN,LIB1
```

requests LINK to combine MAIN.OBJ with any required functions or subroutines contained in LIB1.OBJ. Finally, any unresolved GLOBALS are resolved through SYSLIB. The entire memory image is output to the file LOAD.SAV.

If the /F option or switch is used, all user-created libraries are searched before the default FORTRAN library, FORLIB.OBJ is searched. Consult the linker chapter of the RT-11 System User's Guide for a detailed description of multilibrary global resolution.

If the linker fails for lack of symbol table space, use the /S linker option in your next attempt. This could slow the linking process, but it allows the maximum possible symbol table space.

To maintain the integrity of the distributed FORTRAN and system libraries, you should create separate user library files rather than modifying or adding to the FORTRAN-77 Library (F77OTS or FORLIB) or to the System Library (SYSLIB). An exception is the explicit addition of F77OTS to SYSLIB as described in the RT-11 FORTRAN-77 Installation Guide and Release Notes.

1.4.4 OVERLAY USAGE

Often the size of available physical memory imposes a constraint on the size and complexity of a FORTRAN program. You can use the overlay feature of the linker to segment the memory image so that the entire program is not memory-resident at one time. This frequently allows the execution of a program that is otherwise too large for the available memory.

An overlay structure consists of a root segment and one or more overlay regions. The root segment contains the FORTRAN-77 main program COMMON, subroutines, function subprograms, and any .PSECT that has the GBL attribute and is referenced from more than one segment. An overlay region is an area of memory allocated for two or more overlay segments, only one of which can be resident at a time. An overlay segment consists of one or more subroutines or function subprograms.

When a call is made at run time to a routine in an overlay segment, the overlay handler verifies that the segment is resident in its overlay region. If the segment is in memory, control passes to the routine. If the segment is not resident, the overlay handler reads the overlay segment from the memory image file into the specified overlay region. This replaces the previous overlay segment in that overlay region destroying the contents of all variables stored there. Control then passes to the routine.

You must give careful consideration to placing routines into a root segment and overlay regions, and subsequently divide each overlay region into overlay segments. Remember that it is illegal to call a routine located in a different segment but in the same region, or a region with a lower numeric value (as specified by the linker overlay /O:n) than the calling routine. Divide each overlay region into overlay segments that never need to be resident simultaneously.

The FORTRAN-77 main program unit must be placed in the root segment.

In an overlay environment, subroutine calls and function subprogram references must be limited to the following:

- A FORTRAN-77 library routine (for example, ASSIGN or COS).
- A FORTRAN-77 or assembly language routine contained in the root segment.

- A FORTRAN-77 or assembly language routine contained in the same overlay segment as the calling routine.
- A FORTRAN-77 or assembly language routine contained in an overlay segment whose region number is greater than that of the calling routine.

In an overlay environment, you must place the COMMON blocks so that they are resident when you reference them. Blank COMMON is always resident because it is always placed in the root segment. You must place all named COMMON either in the root segment or in the segment whose region number is lowest of all the segments that reference the COMMON block. A named COMMON block cannot be referenced by two different segments in the same region unless the COMMON block appears in a segment of a lower region number. The linker automatically places a COMMON block into the root segment if it is referenced by the FORTRAN main program or by a subprogram that is located in the root segment. Otherwise, the linker place a COMMON block in the first segment encountered in the linker command sting that references that COMMON block.

All COMMON blocks that are data-initialized (by use of DATA statements) must be so initialized in the segment in which they are placed.

The entire overlay initialization process is handled by LINK. The command format outlined below (and further explained in the linker chapter of the RT-11 System User's Guide) is used to describe the overlay structure to the linker. LINK combines the runtime overlay handler with the user program, making the overlay process completely transparent to the user's program.

The size of the overlay region is automatically computed to be large enough to contain the largest overlay segment in that overlay region. The root segment and all overlay segments are contained in the memory image file generated by the linker.

Two options are used to specify the overlay structure to LINK. The overlay option is of the form:

```
/O:n
```

where n is an octal number specifying the overlay region number. The command continuation option has two forms:

```
/C  and //
```

Placing "/C" at the end of command lines allows you to continue long command strings on the next line of input. The "//" notation is used at the end of the first line and again at the end of the last line of input (see Table 1-5).

The first line of the LINK overlay structure command string should contain, as the input list, all object modules that are to be included in the root segment. This line should be terminated with the "/C" or "//" option. The /O:n option cannot appear in the first line of the command string. If all modules that are to be placed in the root segment cannot be specified on on the first command line, additional modules can be specified on subsequent command lines, each ending with a /C (but not "//"). The entire root segment must be specified before any overlays.

All subsequent lines of the command string should be terminated with the /O:n option specifying an overlay region and/or the /C option.

The presence of only the /C option (switch) specifies that this is a continuation of the previous line, and therefore a continuation of the specification of that overlay segment. The object modules on each line, or set of continuation lines, constitute an overlay segment and share the specified overlay region with all other segments in the same numeric value overlay region. All but the last line of the command string should contain the /C option (switch).

For example, assuming that F77OTS has been built into SYSLIB, and given the following overlay structure description:

1. A main program and the object module SUB1 are to occupy the root segment.

2. The object module SUB2 is to share an overlay region with the object module SUB3

3. The object modules SUB4 and SUB5 are to share a second overlay region with the object modules SUB6 and SUB7.

The following command sequence could be used:

```
.LINK/PROMPT MAIN,SUB1
*SUB2/O:1
*SUB3/O:1
*SUB4/O:2
*SUB5
*SUB6/O:2
*SUB7//
```

1.4.5 EXTENDED MEMORY OVERLAYS

You can use LINK to create an overlay structure that uses extended memory for privileged or virtual FORTRAN jobs. You will need an XM monitor and a hardware configuration that includes a Memory Management Unit to run a program having overlays in extended memory, but you can link such a program on any RT-11 system.

The extended-memory overlay structure is different from the low-memory overlay structure in that extended-memory overlays can reside concurrently in extended memory. This difference allows for speedier execution because, once a program is read in, it requires fewer I/O transfers with the auxiliary mass-storage volume. In fact, if all program data is resident, and the program is loaded, the program may be able to run without an auxiliary mass-storage volume.

Note that you must observe with extended-memory overlays the same restrictions that apply to low-memory overlays, especially those pertaining to return paths.

The following command string illustrates the use of extended-memory overlays in a privileged FORTRAN-77 job instead of the low-memory overlays used in the previous example.

```
.LINK/PROMPT/EXE:LOAD MAIN,SUB1
*SUB2/V:1
*SUB3/V:1
*SUB4/V:2
*SUB5
*SUB6/V:2
*SUB7//
```

Refer to the RT-11 System User's Guide for more information on Extended Memory Overlays.

1.4.6 FORTRAN PROGRAMS RUN AS VIRTUAL JOBS

Under RT-11's Extended Memory (XM) monitor, you can develop FORTRAN-77 programs that can access a full 32K words of address space, and which you can run as virtual jobs. (Virtual Jobs cannot access the I/O page.) When a FORTRAN-77 job becomes a virtual job, the FORTRAN-77 OTS initialization code uses the special features of the .SETTOP programmed request to allocate a full 32K words of address space. The initialization code then places the OTS work area in the extended memory at the high limit returned by the .SETTOP request. This allocation method differs from that of privileged FORTRAN jobs. Even if they use extended memory overlays, privileged jobs allocate all the free space in low memory for the OTS work area.

To make a FORTRAN-77 program a virtual job, you compile the source-program units as usual. You then link the program with the linker's /XM switch.

The following command links object files VIRFOR.OBJ and SUBS.OBJ to make a program that can be run as a virtual job:

```
.LINK/XM  VIRFOR,SUBS
```

For more information on virtual and privileged jobs see the RT-11 System User's Guide and the RT-11 Software Support Manual.

1.4.7 USING VIRTUAL ARRAYS

VIRTUAL arrays are specially managed arrays that occupy memory outside a program's normal address space. Programs that use VIRTUAL arrays can run only on RT-11 XM monitors because they utilize the system's memory management directives. Thus, when linking a program that uses VIRTUAL arrays, you must use the /XM switch and the distributed module VIRTXM.OBJ. The following example links a program called MYPROG with user subroutines in SUBS.OBJ, the VIRTXM module, and the FORTRAN-77 Object Time System library:

```
.LINK/XM MYPROG,SUBS,SY:VIRTXM,SY:F77OTS
```

For more details on the use of VIRTUAL arrays see Chapter 3.

1.5 EXECUTING A FORTRAN-77 PROGRAM

Use the monitor RUN command to start execution of the memory image file generated by LINK. The command

```
.RUN dev:filnam
```

causes the file on the device "dev:" to be loaded into memory and executed. The file specification of the memory image file is "dev:filnam.SAV".

You can end a job before its normal completion by typing CTRL/C (^C).

A job that terminates as a result of a CALL EXIT statement or by reaching the end of the main program does not produce any output to indicate that it is terminating, other than the RT-11 monitor prompt (".").

1.6 EXAMPLES OF FORTRAN-77 COMMAND SEQUENCES

For a FORTRAN-77 job consisting of:

- A main program in file MAIN.FOR
- Subroutines in file SUBR1.FOR
- Several subprograms in the file UTILTY.FOR

you can use the following sequence of commands for compiling, linking, and executing:

```
.R F77
*JOB=JOB,SUBR1,UTILTY
^C
.LINK JOB
.RUN JOB
```

The R command is used to run the FORTRAN-77 compiler from the system device. The command line shown compiles three FORTRAN source modules into a single object file named JOB.OBJ. The LINK keyboard command is used to produce an executable memory image from the object module JOB.OBJ. The F77OTS library is built into SYSLIB in this case, so no explicit reference to it is required. The RUN command is then used to load the newly linked JOB.SAV program into memory and start execution.

If the compiler were named FORTRA.SAV on the system device SY:, and the F77OTS library existed separately, also on the system device, then a DCL command sequence to include a listing and load map at the terminal might be:

```
.FORTRAN/LIST:TT: JOB+SUBR1
.FORTRAN/LIST:TT: UTILTY
.LINK/MAP:TT: JOB,UTILTY,SY:F77OTS
.RUN JOB
```

In this case, JOB and SUBR1 are compiled into a single module called JOB.OBJ, while UTILTY is compiled into a separate module called UTILTY.OBJ. All input and output files except those preceded by "SY:" are on the default device DK:.

1.7 DEBUGGING A FORTRAN-77 PROGRAM

FORTRAN-77 provides several aids for finding and reporting errors:

- DEBUG lines in source programs

  FORTRAN-77 statements containing a "D" in column 1 can be added for debugging purposes. During program development, you can use these statements with the /D switch to type out intermediate values and results. After the program runs correctly, you can treat these statements as comments by recompiling without the /D switch.

- Traceback facility

  The compiled code and the OTS provide information on the program unit and line number of a run-time error. A list following the error message shows the sequence of calling program units and line numbers. The amount of information provided in the list is determined by the /S switch during compilation. See Section C.3.1 for the exact format and content of the traceback.

- The debugging program ODT, a user-interactive debugging aid

  You can include ODT in a job by linking it with your program. The LINK/DEBUG:filespec command may be used to specify the debug utility. When using ODT, you should obtain the machine language code listing of the program (specify the /L:3 compiler switch) and the link map of the memory image.

CHAPTER 2

FORTRAN-77 INPUT/OUTPUT

This chapter describes input/output (I/O) as implemented in RT-11 FORTRAN-77. In particular, it provides information about FORTRAN-77 I/O in relation to the RT-11 file system.

2.1 FORTRAN-77 I/O CONVENTIONS

Certain conventions exist for logical device and file name assignments, and for implied logical units. These conventions are outlined in this section.

2.1.1 Device and File Name Conventions

There is no specific relationship between FORTRAN logical unit numbers and RT-11 channel numbers. Channels are mapped to logical units on a first come first served basis starting with channel 0. If logical unit 4 is the first logical unit assigned then it will correspond to channel 0. The next logical unit will correspond to channel 1, and so on.

Listed in Table 2-1 are the default logical device and file name assignments. You can change default device assignments at the following times: (1) prior to execution, by using the appropriate operating system command; (2) at execution time, by using the ASSIGN system subroutine (see Section D.2) or an OPEN statement.

If a filename is not specified in an ASSIGN or OPEN statement, then a default name is used. The default file name conventions hold for logical units not listed below; for example, unit number 12 has a default file name of FOR012.DAT. The default device assignment for logical units not listed is the default disk, DK:.

You may use any combination of valid logical unit numbers; however, there is an imposed maximum number of units that can be active simultaneously. This number depends on the number of buffers allocated and the number of buffers required for each logical unit (usually 1). The number of active logical units can be set at compile time by using the /UNITS:n or /N:n switches. The default value used is six.

When a logical unit is closed, the default file name assignment that existed at the start of task execution is reestablished; the default device assignment becomes undefined.

Table 2-1
FORTRAN Default Logical Device Assignments

Logical Unit No.	Device		Default File Name	Carriage Control
1	System device	SY:	FOR001.DAT	LIST
2	Default device	DK:	FOR002.DAT	LIST
3	Default device	DK:	FOR003.DAT	LIST
4	Default device	DK:	FOR004.DAT	LIST
5	User's terminal	TT:	FOR005.DAT	FORTRAN
6	System printer	LP:	FOR006.DAT	FORTRAN
7	User's terminal	TT:	FOR007.DAT	FORTRAN
8	Default device	DK:	FOR008.DAT	LIST
.	.	.	.	.
.	.	.	.	.
99	Default device	DK:	FOR099.DAT	LIST

NOTE

The device assignment to a logical unit is not affected by a CLOSE operation. However, this convention is subject to change in future releases and should not be relied on. If the device assignment of a unit is changed by a CALL ASSIGN or an OPEN statement, it is recommended that all CALL ASSIGN or OPEN statements referencing that unit explicitly specify the device to be used.

2.1.2 Implied-Unit Number Conventions

Certain I/O statements do not require explicit logical unit specifications. These statements, and their equivalent forms, are listed in Table 2-2.

From Table 2-2, you can see that a formatted READ statement of the form:

```
READ f,list
```

is equivalent to:

```
READ(1,f)list
```

In a program, these two forms function identically. If logical unit number 1 is assigned to a terminal, input comes from this terminal no matter which of the above READ formats you use.

The PRINT, ACCEPT, and TYPE statements implicitly refer to logical units 6, 5, and 5, respectively.

On output, logical units 5, 6, and 7 employ FORTRAN carriage control. The first character in each record is used to determine whether the printer should be advanced by a line or page before the record is printed. All other logical units employ the LIST attribute, meaning that the first character of each record is passed to the printer, and not used for interpretation. These default attributes may be changed by using an OPEN statement with a CARRIAGECONTROL specification, or by calling the ASSIGN subroutine.

Table 2-2
Implied Unit Numbers

Statement Type		Equivalent Form	
READ	f, list	READ	(1,f) list
PRINT	f, list	WRITE	(6,f) list
ACCEPT	f, list	READ	(5,f) list
TYPE	f, list	WRITE	(5,f) list

2.2 FILES AND RECORDS

This section discusses file structures, record access modes, and record formats in the context of the capabilities of the RT-11 file system.

2.2.1 File Structure

A clear distinction must be made between the way files are organized and the way records are accessed.

The term "file organization" refers to the way records are arranged within a file; the term "record access" refers to the method by which records are read from a file or written to a file. A file's organization is specified when the file is created, and cannot be changed. Record access is specified each time a file is opened, and can be different each time the same file is opened. This section discusses file organization; Section 2.2.2 discusses record access. Table 2-3 shows the valid record access modes for each file organization.

2.2.2 Access to Records

You can select records for processing by the following methods:

- Sequential access
- Direct access
- Unformatted access

2.3 OPEN STATEMENT KEYWORDS

The following sections supplement the OPEN statement description that appears in the RT-11 FORTRAN-77 Language Reference Manual. In particular, implementation-dependent and/or system-dependent aspects of certain OPEN statement keywords are described. This section does not discuss all the keywords that apply to the OPEN statement.

2.3.1 BLANK

BLANK in an OPEN statement controls the interpretation of blanks in numeric input fields. The default is BLANK='NULL' (blanks in numeric input fields are ignored).

If a logical unit is opened by means other than an OPEN statement, a default equivalent to BLANK='ZERO' is assumed (that is, blanks in numeric input fields are treated as zeros).

The BLANK keyword affects the treatment of blanks in numeric input fields read with the D, E, F, G, I, O, and Z field descriptors. If BLANK='NULL' is in effect for these descriptors, embedded and trailing blanks are ignored; the value affected is converted as if the nonblank characters were right justified in the field. If BLANK='ZERO' is in effect, embedded and trailing blanks are treated as zeros.

The /X switch determines whether a default of BLANK='NULL' or BLANK='ZERO' is assumed, as illustrated below:

```
      OPEN(UNIT=1, STATUS='OLD')
      READ(1,10)I,J
10    FORMAT(2I5)
      END

          Data record:   1 2    12

      Assigned values:

           normal              /X
           I=  12          I=  1020
           J=  12          J=    12
```

If your program treats blanks in numeric input fields as zeros, and you do not want to use the /X switch, include BLANK='ZERO' in the OPEN statement or use the BZ edit descriptor in the FORMAT statement.

2.3.2 BLOCKSIZE

BLOCKSIZE specifies the physical I/O transfer size for a file. A BLOCKSIZE specification has the form:

```
BLOCKSIZE = bks
```

BLOCKSIZE is ignored by RT-11. All channels have a block size of BUFFERCOUNT*512. bytes.

2.3.3 BUFFERCOUNT

BUFFERCOUNT specifies the number of memory buffers. A BUFFERCOUNT specification has the form:

```
BUFFERCOUNT = bc
```

The value of bc may be 1 or 2.

The default value is one.

2.3.4 DISPOSE

DISPOSE specifies the disposition of a file at the time the file is closed. A DISPOSE specification has the form:

```
                'SAVE'
DISPOSE=        'KEEP'
DISP=           'DELETE'
```

DISPOSE cannot be used to save or print a scratch file, or to delete or print a read-only file. A DISPOSE parameter in a CLOSE statement always supersedes a disposition specified in an OPEN statement.

2.3.5 INITIALSIZE and EXTENDSIZE

INITIALSIZE specifies the initial storage allocation for a disk file, and EXTENDSIZE specifies the amount by which a disk file is extended each time more space is needed for the file. EXTENDSIZE cannot be used in RT-11. See note 1 below.

INITIALSIZE is effective only at the time a file is created.

If there is not enough space available to hold the initial size of a file an error message is issued.

2.3.6 KEY

The KEY keyword cannot be used in RT-11. See note 1 below.

2.3.7 ORGANIZATION

ORGANIZATION specifies the type of organization a file has or is to have. An ORGANIZATION specification has the form:

```
ORGANIZATION = 'SEQUENTIAL'
```

All files are organized as sequential, contiguous files on RT-11. The ORGANIZATION keyword cannot be used in RT-11. See note 1 below.

2.3.8 READONLY

READONLY specifies that write operations are not to be allowed on the file being opened.

2.3.9 SHARED

The SHARED keyword cannot be used in RT-11. See note 1 below.

Note 1: This keyword is recognized by the FORTRAN-77 compiler for purposes of compatibility with PDP-11 FORTRAN-77/RSX and VAX FORTRAN. However, keyed, indexed, and shared access to files is not supported under the RT-11 system and a run-time error diagnostic will be generated by the OTS.

2.3.10 USEROPEN

USEROPEN provides access to features of the supporting I/O system not directly supported by the FORTRAN-77 I/O system.

USEROPEN is intended for experienced users.

2.4 BACKSPACE AND ENDFILE IMPLICATIONS

This section describes implications of the BACKSPACE and ENDFILE I/O statements, which are supported only for sequential files.

A BACKSPACE operation cannot be performed on a file that is opened for append access, because under append access the current record count is not available to the FORTRAN-77 I/O system; backspacing from record n is done by rewinding to the start of the file and then performing n-1 successive reads to reach the previous record.

The ENDFILE statement writes an end-file record. Because the concept of an embedded end-file is unique to FORTRAN, the following convention has been adopted: An end-file record is a 1-byte record that contains the octal code 32 (CTRL/Z). An end-file record can be written only to sequentially organized files that are accessed as formatted sequential or unformatted segmented sequential. End-file records should not be written in files that are read by programs written in a language other than FORTRAN.

CHAPTER 3

PDP-11 FORTRAN-77/RT-11 OPERATING ENVIRONMENT

This chapter discusses aspects of the PDP-11 FORTRAN-77/RT-11 compiler and OTS operating environment. Information is provided on the following:

- The PDP-11 calling sequence convention
- FORTRAN program sections
- FORTRAN COMMON blocks
- FORTRAN-77 OTS error processing
- Compiler listing-file format

3.1 FORTRAN-77 OBJECT TIME SYSTEM

The FORTRAN-77 Object Time System (OTS) is composed of the following routines:

- Math routines, including the FORTRAN-77 library functions and other arithmetic routines (for example, exponentiation routines)
- Miscellaneous utility routines (ASSIGN, DATE, ERRSET, and so forth)
- Routines that handle FORTRAN-77 input/output
- Error-handling routines that process arithmetic errors, I/O errors, and system errors
- Miscellaneous routines required by the compiled code

The FORTRAN-77 OTS is a collection of many small modules that allows you to omit unnecessary routines during linking. For example, if a program performs only sequential formatted I/O, none of the direct-access I/O routines is included in the job.

3.2 FORTRAN-77 CALLING SEQUENCE CONVENTION

The PDP-11 FORTRAN-77/RT-11 calling sequence convention is compatible with all PDP-11 processor options and provides both reentrant and nonreentrant forms.

3.2.1 The Call Site

The MACRO-11 form of the call is:

```
; IN INSTRUCTION-SPACE

          MOV #LIST,R5    ;ADDRESS OF ARGUMENT LIST TO
                          ;REGISTER 5
          JSR PC,SUB      ;CALL SUBROUTINE
          ...

; IN DATA-SPACE

LIST:     .BYTE N,0       ;NUMBER OF ARGUMENTS
          .WORD ADR1      ;FIRST ARGUMENT ADDRESS
          ...
          .WORD ADRN      ;N'TH ARGUMENT ADDRESS
```

The argument list must reside in DATA-SPACE and, except for subprograms and statement label arguments, all addresses in the list must also refer to DATA-SPACE. The argument list itself cannot be modified by the called program.

The byte at address LIST+1 should be considered undefined and not referenced. This byte is reserved for use as defined by DIGITAL.

The called program is free to refer to the arguments indirectly through the argument list. This argument-passing mechanism is known as call-by-reference.

3.2.2 Return

Control is returned to the calling program by restoring (if necessary) the stack pointer to its value on entry and executing the following:

```
RTS PC
```

3.2.3 Return Value Transmission

Function subprograms return a single result in the processor general registers. The register assignments for returning the different variable types are:

Type	Result
INTEGER*2 LOGICAL*1 LOGICAL*2	R0
INTEGER*4 LOGICAL*4	R0 -- low-order result R1 -- high-order result
REAL	R0 -- high-order result R1 -- low-order result
DOUBLE PRECISION	R0 -- highest-order result R1 -- R2 -- R3 -- lowest-order result

Type	Result
COMPLEX	R0 -- high-order real result R1 -- low-order real result R2 -- high-order imaginary result R3 -- low-order imaginary result

3.2.4 Register Usage Conventions

Before making a call, a calling program must save any values in general-purpose registers R0 through R4 that it needs after a return from a subprogram. After a return, a calling program cannot assume that the argument list pointer value in register R5 is valid.

Conventions for floating-point registers are similar to those for general-purpose registers. If a Floating Point Processor (FP11) or the floating-point microcode option (KEF11A) is present on a system, the calling program must save and restore any floating-point registers in use by a calling program. The calling program cannot assume that the floating-point status bits I/L (integer/long integer) or F/D (floating/double precision) are restored by the called routine. Note that a floating point option is not required for execution of FORTRAN-77 programs which do not generate floating point code. The production of code which does refer to floating point instructions is flagged in the FORTRAN-77 listing file.

A subprogram that is called by a FORTRAN-77 program can freely use processor registers R0-R5, FPP registers F0-F5, and the FPP status register. When returning from a subroutine (when the RTS PC is executed), the initial (routine entry) value must be restored to the contents of the processor hardware stack pointer SP.

3.2.5 Nonreentrant Example

In nonreentrant forms, the argument list can be placed either in line with the call or out of line in an impure data section. (The latter is recommended and illustrated here, and is the form produced by the FORTRAN-77 compiler.) Example 3-1 illustrates assembly language code implementing a small FORTRAN-77 FUNCTION subprogram that uses the nonreentrant form of a call. Note that the nonreentrant form is shorter and generally faster than the reentrant form because addresses of simple variables can be assembled into the argument list.

Example 3-1: Call Sequence Conventions: Nonreentrant Example

```
      INTEGER FUNCTION FNC(I,J)
      INTEGER FNC1
      FNC=FNC1(I+J,5)+I
      RETURN
      END

      .PSECT
      .GLOBL    FNC,FNC1
FNC:  MOV       R5,-(SP)        ;SAVE ARG LIST POINTER
      MOV       @2(R5),-(SP)    ;FORM I+J ON STACK
      ADD       @4(R5),@SP
      MOV       SP,LIST+2       ;ADDRESS OF I+J TO
                                ;ARG LIST
```

(continued on next page)

Example 3-1 (Cont.): Call Sequence Conventions: Nonreentrant Example

```
            MOV     #LIST,R5
            JSR     PC,FNC1
            ADD     #2,SP           ;DELETE TEMPORARY I+J
            MOV     (SP)+,R5        ;RESTORE R5
            ADD     @2(R5),R0       ;ADD I TO FNC1 RESULT
            RTS     PC              ;RETURN VALUE IN R0

            .PSECT  DATA            ;DATA AREA
LIST:       .BYTE   2,0             ;TWO ARGUMENTS
            .WORD   0               ;DYNAMICALLY FILLED IN
            .WORD   LIT5            ;ADDRESS OF CONSTANT 5
;
LIT5:       .WORD   5,0             ;CONSTANT 5
            .END
```

3.2.6 Reentrant Example

The PDP-11 FORTRAN-77/RT-11 calling convention has a reentrant form in which the argument list is constructed at run time on the execution stack. Note that the argument addresses must be pushed backwards on the stack to be correctly arranged in memory for the subroutine that uses the list. Basically, the technique consists of:

```
MOV     #ADRn,-(SP)     ;ADDRESS OF NTH ARGUMENT
...
MOV     #ADR2,-(SP)
MOV     #ADR1,-(SP)     ;ADDRESS OF 1ST ARGUMENT
MOV     #n,-(SP)        ;NUMBER OF ARGUMENTS
MOV     SP,R5
JSR     PC,SUB          ;CALL SUBROUTINE
ADD     #2*n+2,SP       ;DELETE ARGUMENT LIST
```

Example 3-2 illustrates assembly language code that uses reentrant call forms for the same example shown in Example 3-1.

The FORTRAN-77 compiler does not produce reentrant call forms.

Example 3-2: Call Sequence Convention: Reentrant Example

```
            INTEGER FUNCTION FNC(I,J)
            INTEGER FNC1
            FNC=FNC1(I+J,5)+I
            RETURN
            END

            .PSECT
            .GLOBL  FNC,FNC1
FNC:        MOV     R5,-(SP)        ;SAVE ARG LIST POINTER
            MOV     @2(R5),-(SP)    ;FORM I+J
            ADD     @4(R5),@SP      ;ON STACK
            MOV     SP,R4           ;REMEMBER WHERE
            MOV     #CON5,-(SP)     ;BUILD ARG LIST ON STACK
            MOV     R4,-(SP)        ;ADDRESS OF TEMPORARY
            MOV     #2,-(SP)        ;ARGUMENT COUNT
            MOV     SP,R5           ;ADDRESS OF LIST TO R5
```

(continued on next page)

Example 3-2 (Cont.): Call Sequence Convention: Reentrant Example

```
            JSR       PC,FNC1          ;CALL FNC1
            ADD       #10,SP           ;DELETE ARG LIST AND TEMP I+J
            MOV       (SP)+,R5         ;RESTORE ARG LIST POINTER
            ADD       @2(R5),R0        ;ADD I TO RESULT OF FNC1
            RTS       PC               ;RETURN RESULT IN R0

            .PSECT    DATA             ;DATA AREA
CON5:       .WORD     5,0
            .END
```

3.2.7 Null Arguments

Null arguments are represented in an argument list with an address of -1 (177777 octal). This address is chosen to ensure that using null arguments in calling routines not prepared to handle null arguments will result in an error when the routine is called at execution time. The errors most likely to occur are illegal memory references and/or word reference to odd byte addresses.

Note that null arguments are included in the argument count, as follows:

FORTRAN Statement	Resulting Argument List
CALL SUB	.BYTE 0,0
CALL SUB( )	.BYTE 1,0 .WORD -1
CALL SUB(A,)	.BYTE 2,0 .WORD A .WORD -1
CALL SUB(,B)	.BYTE 2,0 .WORD -1 .WORD B

3.3 PROGRAM SECTIONS

Program sections (PSECTs) are named segments of code and/or data. Attributes associated with each program section (see Table 3-1) direct the linker when the linker is combining separately compiled FORTRAN program units, assembly language modules, and library routines into an executable memory image.

3.3.1 Compiled-Code PSECT Usage

The compiler uses PSECTs to organize compiled output into the following six sections:

1. Section $CODE1 contains all of the executable code for a program unit.

2. Section $PDATA contains pure data, such as constants, that cannot change during program execution.

3. Section $IDATA contains impure data, such as argument lists, that can change during program execution.

4. Section $VARS contains storage allocated for variables and arrays used in a program.

5. Section $TEMPS contains temporary storage allocated by the compiler.

6. Section $SAVE contains global storage for entities specified in a SAVE statement.

The attributes associated with each of these sections are shown in Table 3-1.

Table 3-1
Program Section Attributes

Section Name	Attributes
$CODE1	RW, I, LCL, REL, CON
$PDATA	RW, D, LCL, REL, CON
$IDATA	RW, D, LCL, REL, CON
$VARS	RW, D, LCL, REL, CON
$TEMPS	RW, D, LCL, REL, CON
$SAVE	RW, D, GBL, REL, CON, SAV

NOTE

The RO/RW attributes for the sections $CODE1 and $PDATA are controlled by the compiler /Z command qualifier (see Section 1.3.3.1).

Section attributes are as follows:

- RW, RO -- read/write, read only
- I, D -- instructions, data
- CON, OVR -- concatenated, overlaid
- LCL, GBL -- local within overlay segment, global across segments
- SAV -- unconditionally place PSECT in root segment

Virtual arrays are allocated into a special control section, $VIRT, that the linker allocates into the mapped array area of a job.

3.3.2 FORTRAN COMMON and RT-11 System Common

Storage for a common block is placed into a PSECT of the same name as that of the common block. PSECTs used for common blocks are given the attributes RW, D, GBL, REL, OVR, and, for saved named common blocks and blank common, SAV. (The /X switch must be not set for the blank common block PSECT to have the SAV attribute; named common block PSECTS have the SAV attribute under /X.) For example, the statement

```
COMMON /X/A,B,C
```

produces the equivalent of the following MACRO-11 code:

```
        .PSECT X,RW,D,GBL,REL,OVR, SAV
A:      .BLKW 2
B:      .BLKW 2
C:      .BLKW 2
```

A blank common uses the section name .$$$$. Therefore, without setting the /X switch, the statement

```
COMMON T,U,V
```

produces the equivalent of:

```
        .PSECT .$$$$.,RW,D,GBL,REL,OVR,SAV
T:      .BLKW 2
U:      .BLKW 2
V:      .BLKW 2
```

When named PSECTs with the OVR attribute are combined by the linker, all PSECTs with the same name are allocated to begin at the same address. The resulting PSECT has the length of the largest of the combined PSECTs.

An example of common communication between a FORTRAN-77 main program and an assembly language subroutine is shown in Examples 3-3 and 3-4. In the example, the variable ISTRNG in blank common is filled with Hollerith data. This variable is copied to OSTRNG (with space characters removed) in the labeled common DATA, and the actual length is returned in the variable LEN.

Note that one word is allocated for each integer in the assembly language subroutine; this allocation convention is necessary for compatibility with FORTRAN storage allocation under the default switch setting for compilation.

Example 3-3 shows the FORTRAN main program compiled with the /S:NON option. The assembly language subroutine COMPRS is shown in Example 3-4.

Example 3-3: Establishing a FORTRAN COMMON Area and Assembly Language Subroutine CALL

```
            LOGICAL*1 ISTRNG(80),OSTRNG(80)
            COMMON ISTRNG
            COMMON /DATA/ LEN, OSTRNG

C           GET INPUT STRING
C
            READ 1, ISTRNG
1           FORMAT( 80A1 )

C           COMPRESS THE STRING
C
            CALL COMPRS

C           TYPE OUT THE COMPRESSED STRING
C
            TYPE 2, LEN, (OSTRNG(I),I=1,LEN)
2           FORMAT(1X,I3,6X,80A1)
            END
```

Example 3-4: Use of FORTRAN COMMON Area by Assembly Language Subroutine

```
            .TITLE  COMPRS
            .IDENT  /01/

;           COMPRESS THE HOLLERITH STRING IN BLANK COMMON
;           COPYING THE STRING TO LABELLED COMMON DATA AND
;           RETURNING THE ACTUAL LENGTH AS WELL.
;
            .PSECT  .$$$$.,D,GBL,OVR
I:          .BLKB   80.             ; INPUT BUFFER

            .PSECT  DATA,D,GBL,OVR
L:          .BLKW   1               ; ACTUAL LENGTH
O:          .BLKB   80.             ; OUTPUT BUFFER

            .PSECT
COMPRS::
            MOV     #I,R0           ; INPUT POINTER
            MOV     #O,R1           ; OUTPUT POINTER
            MOV     #80.,R2         ; INPUT LENGTH
            CLR     L               ; OUTPUT LENGTH
1$:         MOVB    (R0)+,R3        ; GET INPUT CHARACTER
            CMPB    #' ,R3          ; IS THIS CHAR A SPACE?
            BEQ     2$              ; IGNORE IF SO
            MOVB    R3,(R1)+        ; OUTPUT THE CHARACTER
            INC     L               ; COUNT THE CHARACTER
2$:         DEC     R2              ; COUNT DOWN THE INPUT
            BGT     1$              ; LOOP IF MORE DATA
            RTS     PC
            .END
```

3.3.3 OTS PSECT Usage

All OTS modules consist of at least two program sections: $$OTSI and $$OTSD. Section $$OTSI contains pure-code sequences and section $$OTSD contains pure-data information.

The OTS module $OTV declares the following sections that are used as impure working storage by the OTS:

- Section $$AOTS contains a general work area.

The handling and conversion routines for formatted records are contained in the following sections: $$FIOC, $$FIOD, $$FIO2, $$FIOI, $$FIOL, $$FIOZ, $$FIOS, and $$FIOR. Special conventions are used so that the conversion routines are loaded only if they are required by FORMAT statements in a source program.

3.4 OTS ERROR PROCESSING

The Object Time System detects certain errors in a program (for example, I/O, arithmetic, and invalid argument errors) and reports these errors on the user's terminal. An error-control table within the OTS then determines what action the system is to take for each error reported; for example, it may call for the system to terminate the job. The default action for each FORTRAN-specific error is shown in Table 3-2 (in Section 3.5.1.3).

Three system subroutines (ERRSNS, ERRTST, and ERRSET) are provided to enable you to control OTS error processing: that is, to obtain information on specific errors and/or to specify an action to be taken when a specific error occurs.

The ERRSNS subroutine provides you with information about the error that has most recently occurred during program execution. It also provides detailed information on errors detected by the file system.

The ERRTST subroutine allows you to test for the occurrence of a specific error during program execution.

The ERRSET subroutine allows you to modify the continuation action the system is to take when an error is detected by the OTS. In many cases, the particular continuation action to be taken may be changed from the one specified in the error-control table (see Table 3-2).

The subroutines ERRSNS, ERRTST, and ERRSET are described in detail in Appendix D. OTS error codes and the format of the OTS diagnostic messages are shown in Appendix C.

3.4.1 Recovering From OTS-Detected Errors

You can use three methods to control recovery from errors detected by the OTS:

- ERR= and END= transfers
- The ERRSNS subroutine
- The ERRSET subroutine

The following three sections discuss these methods.

3.4.1.1 **Using ERR= and END= Transfers** - By including an ERR=label or END=label specification in an I/O statement, you can transfer control to error-processing code or to any other desired point in a program. If you use an END= or ERR= specification to process an I/O error, execution continues at the statement specified by a label. However, if you do not use an END= or ERR= specification to process an I/O error, the system by default prints an error message and halts execution.

For example, suppose the following statement is in your program:

```
WRITE(8,50,ERR=400)
```

If an error occurs during the write operation specified, control transfers to the statement at label 400.

When an ERR= transfer occurs, file status and record position become undefined.

You can use the END=label specification to handle an end-of-file condition. For example, if an end-of-file condition is detected while the statement

```
READ(12,70,END=550)
```

is being executed, control transfers to statement 550.

If an end-of-file is detected while a READ statement is being executed, and you did not specify END=label, an error condition occurs. If you specified ERR=label, control is transferred to the specified statement.

3.4.1.2 **Using the ERRSNS Subroutine** - You can use the ERRSNS system subroutine to process errors as they are encountered by a program. When one of the errors listed in Table 3-2 occurs in a program, you can obtain the number of the error by calling the ERRSNS subroutine; then, in most situations, you can provide code to react to this number.

To determine the number of an error, use the ERRSNS routine as demonstrated in the following example:

```
      CHARACTER*40 FILN
10    ACCEPT 1, FILN

1     FORMAT (A)
      OPEN (UNIT=INF, STATUS='OLD', FILE=FILN, ERR=100)
      .
      . (process input file)
      .

100   CALL ERRSNS(IERR)
      IF (IERR .EQ. 43) THEN
         TYPE *, 'FILE NAME WAS INCORRECT; ENTER NEW FILE NAME'
      ELSE IF(IERR .EQ. 29) THEN
         TYPE *, 'FILE DOES NOT EXIST; ENTER NEW FILE NAME'
      ELSE
         TYPE *, 'FAILURE ON INPUT FILE; ERROR=', IERR
      ENDIF
      STOP
      END
```

In this example, the OPEN statement contains an ERR=100 specification that causes a branch to the ERRSNS subroutine if an error occurs during execution of the OPEN. The ERRSNS subroutine returns an error-number value in the integer variable IERR. The program then uses the value of IERR to print a message that explains the nature of the error and to determine whether the program should continue.

3.4.1.3 Using the ERRSET Subroutine - You can alter the default continuation action to be taken upon OTS detection of a particular error by using the ERRSET subroutine.

Processing each of the errors detected by the OTS is controlled by six control bits associated with each error. These bits are preset (see Table 3-2); however, you may alter some of the initial settings -- and thereby the continuation action to be taken upon the detection of a particular error -- by using the ERRSET subroutine.

The six control bits and what they control are as follows:

1. Continuation Bit -- If the Continuation Bit is not set, the job encountering the error exits. If this bit is set, the job continues (if the next two conditions permit continuation).

2. Count Bit -- If the Count Bit is set, the error encountered is counted against the job error-count limit unless an ERR=transfer is specified. If the error-count limit is exceeded, the job exits.

3. Continuation Type Bit -- The Continuation Type Bit provides for one of the following two types of action for a particular error:

   a. Return to the routine that reported the error, for appropriate recovery action, then proceed.

   b. Take an ERR= transfer in an I/O statement. (If the Continuation Type Bit specifies an ERR= transfer, and no ERR=label was included in the I/O statement, the job exits).

Each of the error-control-bit checks above must be satisfied for the job to continue.

4. Log Bit -- If a job continues after an error is encountered (that is, if continuation is permitted by each of the above three control bits), then the Log Bit is tested. If the Log Bit is set, an error message is produced before the job continues; if the Log Bit is not set, the job continues without a message.

If processing any of the first three control bits does not permit continuation, the job exits and the system prints an error message.

Two additional control bits are used to specify the acceptability of arguments to the ERRSET subroutine.

5. Return Permitted Bit -- If the Return Permitted Bit is set, ERRSET may set the Continuation Type Bit to specify a return.

6. ERR= Permitted Bit -- If the ERR= Permitted Bit is set, ERRSET may set the Continuation Type Bit to specify that an ERR= transfer is to occur.

At least one of these two additional bits must be set in order for the Continuation Bit to be set.

All four of the possible combinations of these two bits occur in the OTS; however, most errors occur as the following:

- I/O errors that generally permit ERR= continuation type but not return continuation
- Errors that permit return continuation but not ERR= transfer continuation (even if they occur during I/O statement processing)

Notable exceptions are the synchronous system-trap errors (3 through 10) and the recursive I/O error (40), all of which always result in job termination. The format processing and format conversion errors (59, 61, 63, 64, 68) allow both types of continuation.

The initial setting of all six control bits -- the two permitted bits as well as the Continuation Bit, the Count Bit, the Continuation Type Bit, and the Log Bit -- is shown in Table 3-2. You can use the ERRSET subroutine to change the settings for "CONTINUE?", "COUNT?", "CONTINUE TYPE", and "LOG?". The ERRSET subroutine is described in detail in Appendix D.

Table 3-1
Initial Error Control Bit Settings

ERROR NUMBER	CONTINUE?	COUNT?	CONTINUE TYPE	LOG?	PERMITTED ERR=?	PERMITTED RETURN?	TEXT
1	NO	NO	FATAL	YES	NO	NO	INVALID ERROR CALL
2	NO	NO	FATAL	YES	NO	NO	NOT ENOUGH MEMORY FOR OTS TABLES
3	NO	NO	FATAL	YES	NO	NO	ODD ADDRESS TRAP
4	NO	NO	FATAL	YES	NO	NO	SEGMENT FAULT
5	NO	NO	FATAL	YES	NO	NO	T-BIT OR BPT TRAP
6	NO	NO	FATAL	YES	NO	NO	IOT TRAP
7	NO	NO	FATAL	YES	NO	NO	RESERVED INSTRUCTION TRAP...
8	NO	NO	FATAL	YES	NO	NO	NON-FORTRAN ERROR CALL
9	NO	NO	FATAL	YES	NO	NO	TRAP INSTRUCTION TRAP
10	NO	NO	FATAL	YES	NO	NO	PDP-11/40 FIS TRAP
11	NO	NO	FATAL	YES	NO	NO	FPP HARDWARE FAULT
12	NO	NO	FATAL	YES	NO	NO	FPP ILLEGAL OPCODE TRAP
13	NO	NO	FATAL	YES	NO	NO	FPP UNDEFINED VARIABLE TRAP
14	NO	NO	FATAL	YES	NO	NO	FPP MAINTENANCE TRAP
20	YES	YES	ERR=	YES	YES	NO	INVALID LOGICAL UNIT NUMBER
21	YES	YES	ERR=	YES	YES	NO	NO AVAILABLE CHANNELS
22	YES	YES	ERR=	YES	YES	NO	INPUT RECORD TOO LONG
23	YES	YES	ERR=	YES	YES	NO	BACKSPACE ERROR
24	YES	YES	ERR=	YES	YES	NO	END-OF-FILE
25	YES	YES	ERR=	YES	YES	NO	RECORD NUMBER OUTSIDE RANGE
26	YES	YES	ERR=	YES	YES	NO	ACCESS MODE NOT SPECIFIED
27	YES	YES	ERR=	YES	YES	NO	TOO MANY RECORDS IN I/O STATEMENT
28	YES	YES	ERR=	YES	YES	NO	CLOSE ERROR
29	YES	YES	ERR=	YES	YES	NO	NO SUCH FILE
30	YES	YES	ERR=	YES	YES	NO	OPEN FAILURE

(continued on next page)

Table 3-1 (Cont.)
Initial Error Control Bit Settings

ERROR NUMBER	CONTINUE?	COUNT?	CONTINUE TYPE	LOG?	PERMITTED ERR=?	PERMITTED RETURN?	TEXT
31	YES	YES	ERR=	YES	YES	NO	MIXED FILE ACCESS MODES
32	YES	YES	ERR=	YES	YES	NO	DUPLICATE FILE SPECIFICATIONS
33	YES	YES	ERR=	YES	YES	YES	ENDFILE ERROR
34	YES	YES	ERR=	YES	YES	NO	UNIT ALREADY OPEN
35	YES	YES	ERR=	YES	YES	NO	RANDOM I/O TO NON-FILE...
36	YES	YES	ERR=	YES	YES	NO	ATTEMPT TO ACCESS NON-EXIST...
37	YES	YES	ERR=	YES	YES	YES	INCONSISTENT RECORD LENGTH
38	YES	YES	ERR=	YES	YES	NO	ERROR DURING WRITE
39	YES	YES	ERR=	YES	YES	NO	ERROR DURING READ
40	NO	NO	FATAL	YES	NO	NO	RECURSIVE I/O OPERATION
41	YES	YES	ERR=	YES	YES	NO	NO BUFFER ROOM
42	YES	YES	ERR=	YES	YES	NO	NO SUCH DEVICE
43	YES	YES	RETURN	YES	NO	YES	FILE NAME SPECIFICATION ERROR
44	YES	YES	ERR=	YES	YES	NO	INCONSISTENT RECORD TYPE
45	YES	YES	ERR=	YES	YES	NO	KEYWORD VALUE ERROR IN OPEN...
46	YES	YES	ERR=	YES	YES	NO	INCONSISTENT OPEN/CLOSE...
47	YES	YES	ERR=	YES	YES	NO	WRITE TO READONLY FILE
48	YES	YES	ERR=	YES	YES	NO	UNSUPPORTED I/O OPERATION
49	YES	YES	ERR=	YES	YES	NO	REWIND ERROR
50	YES	YES	ERR=	YES	YES	NO	HARD I/O ERROR
51	YES	YES	ERR=	YES	YES	NO	LIST DIRECTED I/O SYNTAX ERR..
52	YES	YES	ERR=	NO	YES	NO	INFINITE FORMAT LOOP
53	YES	YES	ERR=	YES	YES	NO	FORMAT/VARIABLE-TYPE MISMATCH
54	YES	YES	ERR=	YES	YES	NO	SYNTAX ERROR IN FORMAT
55	YES	YES	ERR=	YES	YES	NO	OUTPUT CONVERSION ERROR
56	YES	YES	ERR=	YES	YES	NO	INPUT CONVERSION ERROR
57	YES	YES	ERR=	YES	YES	NO	FORMAT TOO BIG FOR 'FMTBUF'
58	YES	YES	ERR=	YES	YES	NO	OUTPUT STATEMENT OVERFLOWS...
59	YES	YES	ERR=	YES	YES	NO	RECORD TOO SMALL FOR I/O LIST
70	YES	YES	RETURN	YES	NO	YES	INTEGER OVERFLOW
71	YES	YES	RETURN	YES	NO	YES	INTEGER ZERO DIVIDE
72	YES	YES	RETURN	YES	NO	YES	FLOATING OVERFLOW
73	YES	YES	RETURN	YES	NO	YES	FLOATING ZERO DIVIDE
74	YES	NO	RETURN	NO	NO	YES	FLOATING UNDERFLOW
75	YES	YES	RETURN	YES	NO	YES	FPP FLOATING TO INTEGER...
80	YES	YES	RETURN	YES	NO	YES	WRONG NUMBER OF ARGUMENTS
81	YES	YES	RETURN	YES	NO	YES	INVALID ARGUMENT
82	YES	YES	RETURN	YES	NO	YES	UNDEFINED EXPONENTIATION
83	YES	YES	RETURN	YES	NO	YES	LOGARITHM OF ZERO OR NEGATIVE...
84	YES	YES	RETURN	YES	NO	YES	SQUARE ROOT OF NEGATIVE VALUE
86	YES	YES	RETURN	YES	NO	YES	INVALID ERROR NUMBER

(continued on next page)

Table 3-1 (Cont.)
Initial Error Control Bit Settings

ERROR NUMBER	CONTINUE?	COUNT?	CONTINUE TYPE	LOG?	PERMITTED ERR=?	PERMITTED RETURN?	TEXT
91	YES	NO	RETURN	NO	NO	YES	COMPUTED GOTO OUT OF RANGE
92	YES	YES	RETURN	YES	NO	YES	ASSIGNED LABEL NOT IN LIST
93	YES	YES	RETURN	YES	NO	YES	ADJUSTABLE ARRAY DIMENSION...
94	YES	YES	RETURN	YES	NO	YES	ARRAY REFERENCE OUTSIDE ARRAY
95	NO	NO	FATAL	YES	NO	NO	INCOMPATIBLE FORTRAN OBJECT...
96	NO	NO	FATAL	YES	NO	NO	MISSING FORMAT CONVERSION...
101	NO	NO	FATAL	YES	NO	NO	VIRTUAL ARRAY INITIALIZATION...
102	YES	YES	RETURN	YES	NO	YES	VIRTUAL ARRAY MAPPING ERROR

3.5 FORTRAN-77 COMPILER LISTING FORMAT

There are three optional sections that you may include in a compiler listing file: the source program, the generated machine code, and the storage map. The source program and storage map are included in a list file by default. The generated machine language code is excluded by default. A description of each of these sections is given below.

3.5.1 Source Listing

The source code of a compiled program is written into the source listing section of the compiler listing file in the same format as that in which the source code appears in the input file, except that the compiler adds internal sequence numbers to facilitate ease of reference. Comment lines and uncompiled debug statements, however, do not receive internal sequence numbers.

If the text editor you use generates line numbers, these numbers also appear in the source listing. They appear in the left margin, with the compiler-generated sequence numbers shifted to the right. Diagnostic messages always refer to the compiler-generated sequence numbers.

3.5.2 Generated Code Listing

The generated code listing section of the compiler listing file contains symbolic representations of object code generated by the compiler. These representations are similar to a MACRO-11 source listing, but they are not in a form that can be directly assembled by MACRO-11.

Labels that correspond to FORTRAN source labels are printed with an initial dot. For example, the source label "300" would appear in a generated code listing as ".300". Not all labels appearing in a source program necessarily appear in the corresponding generated code listing. In particular, labels not referenced in a source program are ignored by the compiler and are not used in resulting generated code.

References to variables and arrays defined in a source program are shown in the corresponding generated code listing by their FORTRAN names.

PDP-11 general registers 0 through 5 are represented in a generated code listing by R0 through R5, general register 6 is represented by SP (for Stack Pointer), and general register 7 is represented by PC (for Program Counter); the floating-point registers are represented by F0 through F5. These representations are the conventional PDP-11 register names and are used despite the fact that you can also use these names as FORTRAN variable names.

In some cases, the compiler generates labels for its own use. These labels are shown in a generated code listing as "L$xxxx", where "xxxx" is a unique symbol for each label within a program unit.

Addresses for other than labels, registers, and variables are represented by the name of the program section plus the offset within that section. Program section names used by the compiler are summarized in Section 3.3.1. Changes from one program section to another are shown as .PSECT lines. The left column of a listing shows the offset within the current section to which the remainder of the line applies.

All numbers are in octal radix.

The first line of a generated code listing contains a .TITLE directive; for SUBROUTINE and FUNCTION subprograms, the title is the same as the subprogram name. If a PROGRAM statement is used in a main program, the name in that statement is used as the title; otherwise, the title .MAIN. is used. If a name is included in a BLOCKDATA statement, this name is used for the title; otherwise, the title .DATA. is used.

The second line of a generated code listing contains an .IDENT directive in which the date of the compilation is represented.

The lines that follow the second line describe the contents of storage initialized for FORMAT statements, DATA statements, constants, subprogram call argument lists, and so forth.

Machine instructions are represented in a generated code listing with MACRO-11 mnemonics and syntax.

3.5.3 Storage Map Listing

The storage map contains summaries of the following:

- Program sections
- Entry points
- Variables

- Arrays
- Virtual arrays
- Labels
- Functions and subroutines referenced
- Total memory allocated

Figure 3-1 illustrates a typical storage map listing.

In each of the following descriptions, when a size is given, this size is printed as octal bytes followed by decimal words (except for virtual arrays). For example:

```
000006     3
```

A data address is given as a program section number followed by the octal offset from the beginning of that program section.

For example, in the data address that follows, 1 is the program section number and 000626 is the offset (in octal) from the beginning of program section 1:

```
1-000626
```

A dummy argument is represented with an F instead of a program section number, and the offset is the offset from the argument pointer (R5).

The symbol * following an address field specifies that the program section number (or F), plus the offset, points to the address of the data rather than to the data itself.

The PROGRAM SECTIONS summary in a storage map contains information -- one line per program section -- about each of the program sections (PSECTs) generated by the compiler. Each line contains the number of the PSECT being summarized (used by most of the other summaries), the name of the section, the size of the section, and the attributes of the section. The size is shown twice: first, as the number of bytes in octal radix; and, second, as the number of words in decimal radix. See Section 3.3.1 for definitions of the section attributes.

The ENTRY POINTS summary contains a list of all declared entry points and their addresses. If the routine containing an entry point being listed is a function, the declared data type of this entry point is also included.

The VARIABLES summary contains a list of each simple variable, together with its data type and address.

The ARRAYS summary is the same as the VARIABLES summary, except that it supplies total array size information and detailed dimension information. If the array is an adjustable array or assumed-size array, the size of the array is specified as **, and each adjustable-dimension bound or assumed-size bound is specified as *.

The VIRTUAL ARRAYS summary is similar to the array summary. The address of a virtual array is shown as an offset, in 64 byte units, from the start of virtual array storage. The size is specified as the number of array elements, not the number of bytes.

```
PDP-11 FORTRAN-77 V5.0          08:07:08      5-Mar-84          Page 3
DK :DEMO   .FOR      /A/C:19/F:128/L:3/N:6/R:136/S:BLO
PROGRAM SECTIONS

Number    Name        Size                  Attributes

  1       $CODE1   000124     42           RW,I,CON,LCL
  2       $PDATA   000002      1           RW,D,CON,LCL
  4       $VARS    002530    684           RW,D,CON,LCL
  8       .$$$$.   000260     88           RW,D,OVR,GBL,SAV
  9       ISTUFF   000316    103           RW,D,OVR,GBL
 10       FLAGS    000007      4           RW,D,OVR,GBL

VARIABLES

 Name    Type  Address     Name    Type  Address     Name    Type  Address

 A       C*8   8-000000    ILARGE  I*4   9-000052    ISDONE  L*1   10-000006
 ISMALL  I*2   9-000000    ISOK    L*2   10-000004   ISOVER  L*4   10-000000
 U       R*8   4-000000    X       R*4   8-000130    Z       R*4   8-000254

ARRAYS

 Name    Type  Address        Size          Dimensions

 B       C*8   8-000010   000120     40     (10)
 IARRAY  I*2   9-000002   000050     20     (2,10)
 IBYTE   L*1   9-000176   000120     40     (80)
 JARRAY  I*4   9-000056   000120     40     (2,10)
 V       R*8   4-000010   001700    480     (8,15)
 W       R*8   4-001710   000620    200     (50)
 Y       R*4   8-000134   000120     40     (20)

LABELS

 Label    Address          Label    Address          Label    Address

 10          **

FUNCTIONS AND SUBROUTINES REFERENCED

 EXIT     $SIN

Total Space Allocated = 003463   922

Workfile reads:  3
Workfile writes: 5
Size of workfile required: 24 blocks
Size of core pool: 14 blocks
```

Figure 3-1 Storage Map Example

The LABELS summary contains a list of all user-defined statement labels. If a label is marked with an apostrophe, the label is a format label. If the label address field contains **, the label is neither referenced nor used by the compiled code.

The FUNCTIONS AND SUBROUTINES REFERENCED summary contains a list of all external-routine references made by the source program.

If the text NO FPP INSTRUCTIONS GENERATED appears in the storage map, the FORTRAN-77 object module may not require the Floating Point Processor (FPP) for execution. See Section 5.4.1 for further information.

At the end of the above summaries, the total amount of memory allocated by the compilation for all program sections is printed as follows:

```
TOTAL SPACE ALLOCATED = 000502    161
```

If any virtual arrays are declared in the program, the total size in 64-byte units is given as follows:

```
TOTAL VIRTUAL ARRAY STORAGE = 632
```

If a summary section has no entries in a particular compilation, the summary headings are not printed.

3.6 VIRTUAL ARRAY OPTIONS

The VIRTUAL statement declares arrays that are assigned space outside a program's address space and that are manipulated through the VIRTUAL array facility of PDP-11 FORTRAN-77/RT-11. The VIRTUAL array facility allows arrays to be stored in large data areas that are accessed at high speed.

NOTE

VIRTUAL arrays are supported only on the RT-11 Extended Memory Monitor (XM). See PDP-11 FORTRAN-77/RT-11 Installation Guide and Release Notes for information regarding systems support of VIRTUAL arrays.

3.6.1 Limits on VIRTUAL Elements

VIRTUAL arrays are limited by the number of elements, not by the available storage. The maximum number of elements in a VIRTUAL array is 65535; there is no limit to the total size of the VIRTUAL arrays a program can access. The limit on elements is 65535 because PDP-11 FORTRAN-77/RT-11 requires that the number of elements in an array not exceed the size of an unsigned integer*2, which is 2**16-1.

The largest LOGICAL*1 VIRTUAL array is 32K words, or 65535 bytes; and the largest REAL*8 VIRTUAL array is 256K words, or 624280 bytes.

3.6.1.1 VIRTUAL and DIMENSION Statements - The syntax of the VIRTUAL statement is identical to that of the DIMENSION statement. However, there is a significant semantic difference between the two because of the limitations imposed on the DIMENSION statement. Local arrays declared by the DIMENSION statement are limited by the maximum memory available to the program. Section 3.7.2 demonstrates how to use the VIRTUAL feature in an existing program.

3.6.1.2 Memory Allocation for VIRTUAL Arrays - The linker allocates a mapped array area below a job's header; this mapped array area is large enough to contain all the VIRTUAL arrays declared in a program.

A window of 4K words initially maps the first 4K words of the VIRTUAL array region. When a VIRTUAL array element lies outside the window, a Memory Management directive causes a remap operation to allow access.

3.6.1.3 Execution Time of Virtual Arrays - Using VIRTUAL arrays increases the execution time of a job because VIRTUAL array elements must be mapped to memory addresses. In general, the larger the VIRTUAL array, the greater the number of times mapping occurs; therefore, larger arrays generally take longer to execute than do smaller arrays.

The following example illustrates how using VIRTUAL arrays increases execution time:

```
      PARAMETER       N=3500
      VIRTUAL         A(N), B(N), C(N)
      DO 10 I= I,N
      A(I)=1234.
      B(I)=5678
10    C(I)=A(I)/B(I)
      STOP
      END
```

As declared in the program above, the VIRTUAL arrays A, B, and C are each too large (7000 words) to fit within a 4K-word window of memory. Each time an element outside the 4K-word window is accessed, remapping occurs. Thus, executing the DO loop requires 17,500 (3500*5) mappings. If only array C were VIRTUAL, however, then only two mappings would be needed to execute the loop.

The operations in the program above can require as long as 14.1 seconds for execution on a PDP-11/60. By contrast, if arrays A, B, and C were declared with a DIMENSION statement in directly addressable memory, the same operations could require as little as 0.12 seconds in the same operating environment.

You can reduce the mapping of VIRTUAL arrays by breaking large arrays into smaller ones and/or by keeping consecutive accesses of array elements within the current 4K-word window.

3.6.2 Converting a Program to VIRTUAL Array Usage

You can convert an existing program to use VIRTUAL arrays simply by declaring the array with VIRTUAL statements instead of DIMENSION statements. In doing this, however, be sure to observe the usage restrictions for VIRTUAL arrays described in the PDP-11 FORTRAN-77 Language Reference Manual.

The following example illustrates general program conversion.

1. Identify the non-VIRTUAL arrays that are to be converted to VIRTUAL arrays.

2. Locate the DIMENSION and the type declaration statements in which these arrays are declared. Replace DIMENSION statements with equivalent VIRTUAL statements. Replace array-declarative type declaration statements with VIRTUAL statements to define the array dimension, and remove the dimensioning information from the type declaration statements.

3. Compile the program and observe all compilation errors. These errors occur where the syntax restrictions outlined in the PDP-11 FORTRAN-77 Language Reference Manual have been violated. In some cases, to use VIRTUAL arrays effectively you may need to reformulate the data structures.

4. Check the code to ensure that VIRTUAL array parameters are passed correctly to subprograms.

   a. If the argument list of a subprogram call includes an unsubscripted VIRTUAL array name, the argument list of the SUBROUTINE or FUNCTION statement must have an unsubscripted VIRTUAL array name in its corresponding dummy argument. This corresponding VIRTUAL array name establishes access to the VIRTUAL array for the subprogram. The declaration of the VIRTUAL array in the subprogram must be dimensionally compatible with the VIRTUAL declaration in the calling program. All changes to the VIRTUAL array that occurred during subprogram execution are retained when control returns to the calling program.

      When you pass entire arrays as subprogram parameters, be certain that the matching arguments are defined as both VIRTUAL or both non-VIRTUAL. Mismatches of array types are not detectable at either compilation or execution time, and the results are undefined.

   b. If the argument list of a subprogram reference includes a reference to a VIRTUAL array element, the matching formal parameter in the SUBROUTINE or FUNCTION statement must be a non-VIRTUAL variable. Value assignments to the formal parameter occurring within the subprogram do not alter the stored value of the VIRTUAL array element in the calling program. To alter the value of that element, the calling program must include a separate assignment statement that references the VIRTUAL array element directly.

The process of changing non-VIRTUAL arrays to VIRTUAL arrays is demonstrated below.

The following program contains two arrays, A and B.

```
      DIMENSION A(1000,20)
      INTEGER*2 B(1000)
      DATA B/1000*0/
      CALL ABC(A,B,1000,20)
      WRITE(2,*)(A(I,1),I=1,1000)
      END

      SUBROUTINE ABC(X,Y,N,M)
      DIMENSION X(N,M)
      INTEGER*2 Y(N)
      DO 10, I=1,N
10    X(I,1)=Y(I)
      RETURN
      END
```

Array A is declared in a DIMENSION statement and is of the default data type. Therefore, substituting the keyword VIRTUAL for the keyword DIMENSION is sufficient for its conversion.

Note, however, that array B and its dimensions are declared in a type declaration statement (in the second line of the program).

To convert B into a VIRTUAL array, its declarator must be moved to a VIRTUAL statement; also, the variable B must remain in the type declaration statement, but without a dimension specification.

A and B are both passed to subroutine ABC as arrays, rather than array elements. Therefore, the associated subroutine parameters must also be converted to VIRTUAL arrays.

The following listing shows the program after the conversion is completed.

```
      VIRTUAL A(1000,20), B(1000)
      INTEGER*2 B
      DO 5 I=1,1000
5     B(I) = 0
      CALL ABC(A,B,1000,20)
      WRITE(2,*)(A(I,1),I=1,1000)
      END

      SUBROUTINE ABC(X,Y,N,M)
      VIRTUAL Y(N), X(N,M)
      INTEGER*2 Y
      DO 10, I=1,N
10    X(I,1)=Y(I)
      RETURN
      END
```

3.6.3 Virtual Arrays in the RT-11 Environment

If you want to use VIRTUAL arrays in a program under the RT-11 operating system, you must be running the RT11XM monitor. This is because the virtual array indexing code in the OTS employs the extended monitor's Memory Management Directives.

When linking a program that uses VIRTUAL arrays, you must include the distributed module VIRTXM.OBJ in the LINK command line. This module allocates extra queue elements in your job and sets the virtual bit in RT-11's Job Status Word. If VIRTXM.OBJ is not present on your system device, you should locate it on the FORTRAN-77/RT-11 distribution volume and make a copy of it on SY:, for use when you perform the LINK operation.

In addition, because programs that use virtual arrays are linked as virtual jobs, you must use the linker's /V switch or the LINK command's /XM switch on the command line. An example follows:

```
LINK/XM MYPROG,MYSUBS,SY:VIRTXM,SY:F77OTS
```

This command links user-written routines in MYPROG.OBJ and MYSUBS.OBJ with the distributed module VIRTXM.OBJ and the FORTRAN-77 Object Time System Library. The distributed modules are assumed to be stored on the system device SY:.

When you RUN a program that uses VIRTUAL arrays, you must be aware of the physical memory requirement you impose, based on the dimensions of your VIRTUAL statements. Sufficient extended memory must be available or an error message will be printed. It might be necessary to ABORT and UNLOAD foreground or system jobs that are not currently being used to free required space.

If you are using the VM: virtual memory device, you must make sure that it is not occupying all of extended memory. You can use the SET VM BASE=nnnn command to alter the amount of extended memory that is governed by the VM handler. Or you can disable VM: entirely by typing UNLOAD VM and REMOVE VM. See the RT-11 System User's Guide for more information on this SET command and the VM handler in general.

Note that users of Professional 300 series computers might find it necessary to run large virtual jobs by using the 'R' command. This requires that the job's SAV file be placed on the system device SY:.

See section 1.4.7 in this manual for additional information about how to link programs that use VIRTUAL arrays.

CHAPTER 4

PDP-11 FORTRAN-77/RT-11 IMPLEMENTATION CONCEPTS

This chapter discusses several of the fundamental design and implementation concepts of PDP-11 FORTRAN-77/RT-11 that are different from those of other FORTRAN systems, or that are likely to be new to many FORTRAN programmers.

4.1 INTRINSIC FUNCTIONS

As it processes a program unit, the compiler determines (without any information about other program units that may be added later) whether a function referenced in the program unit is an intrinsic function (processor-defined) or a user-defined function. The compiler invokes an intrinsic function with a symbolic name, called an internal name, that is different from any name the user can define. For example, the intrinsic real-valued sine function is invoked by the compiler with the internal name $SIN.

In general, an internal name is a FORTRAN name with a dollar sign prefixed. Where the FORTRAN name is six characters long, a 5-character contraction is combined with the dollar sign. A complete list of the intrinsic names and their corresponding internal names appears in Table 4-1.

Using the IMPLICIT statement to change the default data type rules has no effect on the data type of intrinsic functions.

4.1.1 Using EXTERNAL and INTRINSIC Statements

The EXTERNAL statement identifies symbolic names as user-supplied functions and subroutines. The INTRINSIC statement identifies symbolic names as system-supplied functions or subroutines. For example, the statement

```
EXTERNAL INVERT
```

identifies a subroutine named INVERT as user-supplied, and

```
INTRINSIC ABS
```

identifies a function named ABS as system-supplied.

Once a symbolic name has been identified in an EXTERNAL statement, it is no longer available in the same program unit for use in an INTRINSIC statement.

4.2.1 Representation and Relationship of INTEGER*2 and INTEGER*4 Values

INTEGER*2 values are stored as two's complement binary numbers in one word of storage. INTEGER*4 values are represented in two's complement binary form in two words of storage: the first word (lower address) contains the low-order part of the value, and the second word (higher address) contains the high-order part of the value (including sign).

An INTEGER*2 value is, then, a subset of an INTEGER*4 value. Therefore, the address of an INTEGER*4 value within the range -32768 to +32767 can be treated as the address of an INTEGER*2 value; and conversion from INTEGER*4 to INTEGER*2 (without overflow checking) consists simply of ignoring the high-order word of the INTEGER*4 value. (In certain situations where you can determine at compile time that the results will not be affected, you can generate INTEGER*2 code to perform INTEGER*4 operations.)

Table 4-1
Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Namel	Type of Argument	Type of Result
Square Root(2)	1	SQRT	SQRT	Real	Real
			DSQRT	Double	Double
a(1/2)			CSQRT	Complex	Complex
Natural Logarithm(3)	1	LOG	ALOG	Real	Real
			DLOG	Double	Double
log(e)a			CLOG	Complex	Complex
Common Logarithm(3)	1	LOG10	ALOG10	Real	Real
			DLOG10	Double	Double
log(10)a					
Exponential	1	EXP	EXP	Real	Real
			DEXP	Double	Double
e(a)			CEXP	Complex	Complex
Sine(4)	1	SIN	SIN	Real	Real
			DSIN	Double	Double
sin a			CSIN	Complex	Complex
Cosine(4)	1	COS	COS	Real	Real
			DCOS	Double	Double
cos a			CCOS	Complex	Complex
Tangent(4)	1	TAN	TAN	Real	Real
			DTAN	Double	Double
tan a					
Arc Sine(5,6)	1	ASIN	ASIN	Real	Real
			DASIN	Double	Double
arc sin a					
Arc Cosine(5,6)	1	ACOS	ACOS	Real	Real
			DACOS	Double	Double
arc cos a					

(continued on next page)

Table 4-1 (Cont.)
Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Namel	Type of Argument	Type of Result
Arc Tangent(6)	1	ATAN	ATAN	Real	Real
			DATAN	Double	Double
arc tan a					
Arc Tangent(6,7)	2	ATAN2	ATAN2	Real	Real
			DATAN2	Double	Double
arc tan a(1)/a(2)					
Hyperbolic Sine	1	SINH	SINH	Real	Real
			DSINH	Double	Double
sinh a					
Hyperbolic Cosine	1	COSH	COSH	Real	Real
			DCOSH	Double	Double
Cosh a					
Hyperbolic Tangent	1	TANH	TANH	Real	Real
			DTANH	Double	Double
Tanh a					
Absolute value(8)	1	ABS	ABS	Real	Real
			DABS	Double	Double
[a]			CABS	Complex	Real
			IIABS	Integer*2	Integer*2
			JIABS	Integer*4	Integer*4
		IABS	IIABS	Integer*2	Integer*2
			JIABS	Integer*4	Integer*4
Truncation(9)	1	INT	IINT	Real	Integer*2
			JINT	Real	Integer*4
[a]			IIDINT	Double	Integer*2
			JIDINT	Double	Integer*4
		IDINT	IIDINT	Double	Integer*2
			JIDINT	Double	Integer*4
		AINT	AINT	Real	Real
			DINT	Double	Double
Nearest Integer(9)	1	NINT	ININT	Real	Integer*2
			JNINT	Real	Integer*4
[a+.5*sign(a)]			IIDNNT	Double	Integer*2
			JIDNNT	Double	Integer*4
		IDNINT	IIDNNT	Double	Integer*2
			JIDNNT	Double	Integer*4
		ANINT	ANINT	Real	Real
			DNINT	Double	Double

(continued on next page)

Table 4-1 (Cont.)
Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Namel	Type of Argument	Type of Result
Fix(10) (real-to-integer conversion)	1	IFIX	IIFIX JIFXI	Real Real	Integer*2 Integer*4
Float(10) (integer-to-real conversion)	1	FLOAT	FLOATI FLOATJ	Integer*2 Integer*4	Real Real
Double Precision Float(10) (integer-to-double conversion)	1	DFLOAT	DFLOTI DFLOTJ	Integer*2 Integer*4	Double Double
Conversion to Single Precision(10)	1	SNGL	- SNGL FLOATI FLOATJ	Real Double Integer*2 Integer*4	Real Real Real Real
Conversion to Double Precision(10)	1	DBLE	DBLE - - DFLOTI DFLOTJ	Real Double Complex Integer*2 Integer*4	Double Double Double Double Double
Real Part of Complex or Conversion to Single Precision(10)	1	REAL	REAL FLOATI FLOATJ SNGL SNGL	Complex Integer*2 Integer*4 Real Double	Real Real Real Real Real
Imaginary Part of Complex	1	-	AIMAG	Complex	Real
Conversion to Complex or Complex from Two Arguments(11)	1,2 1,2 1,2 1,2 1,2 1	CMPLX	- - - CMPLX - -	Integer*2 Integer*4 Real Real Double Complex	Complex Complex Complex Complex Complex Complex
Complex Conjugate (if a=(X,Y) CONJG (a)=(X,Y)	1	-	CONJG	Complex	Complex
Double Product of Reals a(1)*a(2)	2	-	DPROD	Real	Double
Maximum max(a(1),a(2),...a(n)) (returns the maximum value from among the argument list; there must be at least two arguments)	n	MAX	AMAX1 DMAX1 IMAX0 JMAX0	Real Double Integer*2 Integer*4	Real Double Integer*2 Integer*4
		MAX0	IMAX0 JMAX0	Integer*2 Integer*4	Integer*2 Integer*4
		MAX1	IMAX1 JMAX1	Real Real	Integer*2 Integer*4
		AMAX0	AIMAX0 AJMAX0	Integer*2 Integer*4	Real Real

(continued on next page)

Table 4-1 (Cont.)
Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Namel	Type of Argument	Type of Result
Minimum	n	MIN	AMIN1	Real	Real
			DMIN1	Double	Double
min(a(1),a(2),..a(n))			IMIN0	Integer*2	Integer*2
(returns the minimum value among the argument list;			JMIN0	Integer*4	Integer*4
there must be at least two arguments)		MIN0	IMIN0	Integer*2	Integer*2
			JMIN0	Integer*4	Integer*4
		MIN1	IMIN1	Real	Integer*2
			JMIN1	Real	Integer*4
		AMIN0	AIMIN0	Integer*2	Real
			AJMIN0	Integer*4	Real
Positive Difference	2	DIM	DIM	Real	Real
			DDIM	Double	Double
a(1)-(min(a(1),a(2)))			IIDIM	Integer*2	Integer*2
(returns the first argument minus the minimum of the			JIDIM	Integer*4	Integer*4
two arguments)		IDIM	IIDIM	Integer*2	Integer*2
			JIDIM	Integer*4	Integer*4
Remainder	2	MOD	AMOD	Real	Real
			DMOD	Double	Double
a(1)-a(2)*[a(1)/a(2)]			IMOD	Integer*2	Integer*2
(returns the remainder when the first argument			JMOD	Inetger*4	Integer*4
is divided by the second)					
Transfer of Sign	2	SIGN	SIGN	Real	Real
			DSIGN	Double	Double
a(1) *Sign a(2)			IISIGN	Integer*2	Integer*2
			JISIGN	Integer*4	Integer*4
		ISIGN	IISIGN	Integer*2	Integer*2
			JISIGN	Integer*4	Integer*4
Bitwise AND	2	IAND	IIAND	Integer*2	Integer*2
(performs a logical AND on corresponding bits)			JIAND	Integer*4	Integer*4
Bitwise OR	2	IOR	IIOR	Integer*2	Integer*2
(performs an inclusive OR on corresponding bits)			JIOR	Integer*4	Integer*4
Bitwise Exclusive OR	2	IEOR	IIEOR	Integer*2	Integer*2
(performs an exclusive OR on corresponding bits)			JIEOR	Integer*4	Integer*4
Bitwise Complement	1	NOT	INOT	Integer*2	Integer*2
(complements each bit)			JNOT	Integer*4	Integer*4

(continued on next page)

Table 4-1 (Cont.)
Generic and Intrinsic Functions

Functions	Number of Arguments	Generic Name	Specific Namel	Type of Argument	Type of Result
Bitwise Shift (a(1) logically shifted left a(2) bits)	2	ISHFT	IISHFT JISHFT	Integer*2 Integer*4	Integer*2 Integer*4
Random Number (12) (returns the next number from a sequence of pseudo-random numbers of uniform distribution over the range 0 to 1)	1	-	RAN	Integer*4	Real
	2	-	RAN	Integer*2	Real
Length (returns length of the character expression)	1	-	LEN	Character	Integer*2
Index (C(1),C(2)) (returns the position of the substring c(2) in the character expression c(1))	2	-	INDEX	Character	Integer*2
ASCII Value (returns the ASCII value of the argument; the argument must be a character expression that has a length of 1)	1	-	ICHAR	Character	Integer*2
Character relationals (ASCII collating sequence)	2	-	LLT	Character	Logical*2
	2	-	LLE	Character	Logical*2
	2	-	LGT	Character	Logical*2
	2	-	LGE	Character	Logical*2

1. See Section 4.2.4 for definitions of "I" and "J" forms.

2. The argument of SQRT and DSQRT must be greater than or equal to 0. The result of CSQRT is the principal value with the real part greater than or equal to 0. When the real part is 0, the result is the principal value with the imaginary part greater than or equal to 0.

3. The argument of ALOG, DLOG, ALOG10, and DLOG10 must be greater than 0. The argument of CLOG must not be (0.,0.).

4. The argument of SIN, DSIN, COS, DCOS, TAN, and DTAN must be in radians. The argument is treated modulo 2*pi.

5. The absolute value of the argument of ASIN, DASIN, ACOS, and DACOS must be less than or equal to 1.

6. The result of ASIN, DASIN, ACOS, DACOS, ATAN, DATAN, ATAN2, and DATAN2 is in radians.

7. The result of ATAN2 and DATAN2 is 0 or positive when a(2) is less than or equal to 0. The result is undefined if both arguments are 0.

8. The absolute value of a complex number, (X,Y), is the real value:

   (X(2)+Y(2))(1/2)

9. [x] is defined as the largest integer whose magnitude does not exceed the magnitude of x and whose sign is the same as that of x. For example [5.7] equals 5. and [-5.7] equals -5.

10. Functions that cause conversion of one data type to another type provide the same effect as the implied conversion in assignment statements. The function SNGL with a real argument and the function DBLE with a double precision argument return the value of the argument without conversion.

11. When CMPLX has only one argument, this argument is converted into the real part of a complex value, and zero is assigned to the imaginary part. When CMPLX has two arguments, the first argument is converted to the real part of a complex value, the second to the imaginary part.

12. The argument for this function must be an integer variable or integer array element. The argument should initially be set to 0. The RAN function stores a value in the argument that it later uses to calculate the next random number. Resetting the argument to 0 regenerates the sequence. Alternate starting values generate different random-number sequences.

4.1.2 Generic Function References

A generic function is similar to an intrinsic function, but instead of being a single function it is a set of similar functions called specific functions. The specific functions in a generic set differ from each other only in that each function manipulates data of one specific type. For example, SIN() is a generic function that includes the specific functions SIN, DSIN, and CSIN, where SIN manipulates real data, DSIN double-precision data, and CSIN complex data. The data type of the argument in a generic reference determines which specific function is actually invoked. For example, SIN(X) invokes SIN if X is real and DSIN if X is double precision. The compiler makes a separate determination of the specific function to be referenced each time it encounters the same generic reference.

Those intrinsic functions that can be referenced by generic references are listed in Table 4-1 under the heading "Generic Name." Many generic function names are also intrinsic function names. However, in a few cases (for example, the generic function name MIN), the generic function name is not an intrinsic function name.

4.2 INTEGER*2 AND INTEGER*4

PDP-11 FORTRAN-77/RT-11 provides two integer data types: INTEGER*4 , for purposes of high precision; and INTEGER*2 , for purposes of efficiency. INTEGER*4 operations are performed to 32 bits of significance; however, because these operations require more instructions and storage than INTEGER*2 operations, they are less efficient in terms of both time and memory.

To encourage efficiency, the FORTRAN-77 compiler assumes all integer variables to be of INTEGER*2 types unless you explicitly declare them to be INTEGER*4 within a program, or unless you set the /T compiler switch (see Section 1.3.3.1).

When in INTEGER*4 mode, the compiler treats all integer (and logical) variables as INTEGER*4 (and LOGICAL*4) types unless you explicitly declare them otherwise within a program.

The FORTRAN rules state that corresponding actual and dummy arguments must agree in type. In the following example, however, if the compiler supplies an INTEGER*2 constant as the actual argument, SUB executes correctly even if its dummy argument is of INTEGER*4 data type:

```
CALL SUB(2)
```

4.2.2 Integer Constant Typing

In general, typing integer constants as either INTEGER*2 or INTEGER*4 is based on the magnitude of the constant; and in most contexts, INTEGER*2 and INTEGER*4 variables and integer constants may be freely mixed. However, the programmer is responsible for ensuring that integer overflow conditions that might adversely affect the program do not occur. Consider the following example:

```
INTEGER*2   I
INTEGER*4   J
I = 32767
J = I + 3
```

In this example, I and 3 are INTEGER*2 values, and an INTEGER*2 result is computed. The 16-bit addition, however, overflows the valid INTEGER*2 range, and the resulting bit pattern represents -32766, a valid INTEGER*2 value that is converted to INTEGER*4 type and assigned to J. This overflow is not detected.

Compare the above example with the following apparently equivalent program, which produces an entirely different, and logically correct, result:

```
INTEGER*4  J
PARAMETER  I = 32767
J = I + 3
```

In this example, the compiler adds the constant 3 and the parameter constant 32767 and produces a resulting constant of 32770. The compiler recognizes this constant as an INTEGER*4 value and assigns it to J.

4.2.3 Octal Constant Typing

Octal constants can take either of two forms:

```
'C1 C2 C3...Cn'O

"C1 C2 C3...Cn
```

Octal constants of the form 'C1 C2 C3...Cn'O are typeless numeric constants that assume data types on the basis of the way they are used. See the PDP-11 FORTRAN-77 Language Reference Manual for the rules on the typing of octal constants of this form.

Octal constants of the form "C1 C2 C3...Cn, however, are typed as either INTEGER*2 or INTEGER*4 , and are typed on the basis of the magnitude of the constant.

An octal constant of the form "C(1) C(2) C(3)...C(n) is typed as INTEGER*2 if bits 16 through 31 of the value are the same as bit 15; otherwise, it is typed as INTEGER*4 . Because octal constants are treated as unsigned values, they are interpreted as positive values unless bit 31 is set. The octal constants "100000 through "177777 are typed as INTEGER*4 and interpreted as the decimal values 32768 through 65535, rather than as the negative signed decimal values -32768 through -1.

Because octal constants are positive values, you must take care when you compare octal constants with negative signed INTEGER*2 values.

Consider the following example:

```
INTEGER*2 I
IF (I .EQ.  "105132) STOP
```

The comparison made here always results in an inequality (and the STOP statement is not executed). The reason for this is that the INTEGER*2 value of I is converted to INTEGER*4 before the comparison (to conform with the type of "105132); therefore, whenever I contains the bit pattern "105132, this pattern will be interpreted after conversion as the negative decimal value -30118.

The above example is equivalent to:

```
INTEGER*2 I
IF (I .EQ.  35418) STOP
```

If INTEGER*2 values must be compared with octal constants of the form "1xxxxx, the octal constant should be assigned to an INTEGER*2 temporary. An INTEGER*2 temporary could be used in our example as follows:

```
INTEGER*2 I, ICONST
DATA ICONST/"105312/
IF (I .EQ.  ICONST) STOP
```

4.2.4 Integer-Valued Intrinsic Functions

A number of the intrinsic functions provided by FORTRAN-77 (for example, IFIX) produce integer results from real or double-precision arguments. These intrinsic functions are called "result generic functions." Because the compiler operates in two different modes, INTEGER*2 mode and INTEGER*4 mode, the system provides two internal versions of each of these integer-producing functions: an INTEGER*2 version and an INTEGER*4 version. The compiler selects the proper version on the basis of the current compiler mode setting rather than -- as it does for the other intrinsic functions -- on the basis of the data type of arguments in the function reference.

In some cases, you may need to use the version of an integer intrinsic function that is the opposite of the one that would be invoked under the current compiler mode setting. For example, a program that predominantly uses INTEGER*2 values may at some point need to get an INTEGER*4 result from a intrinsic function. To satisfy this need, the system provides an additional pair of intrinsic function names that can reference the two internal versions of each integer-producing intrinsic function no matter what the current compiler mode setting may be. By convention, these additional names are created by

prefixing I and J to the intrinsic function name. For example, I is prefixed to IFIX to create the INTEGER*2 version of this function name, and J is prefixed to create the INTEGER*4 version. IIFIX references the INTEGER*2 internal function $IFIX, and JIFIX references the INTEGER*4 internal function $JFIX.

The complete set of names and corresponding internal routines is shown in Table 4-1 (in Section 4.1).

4.2.5 Implementation-Dependent Integer Typing

The FORTRAN-77 compiler performs a number of integer-typing optimizations by taking advantage of certain properties of the PDP-11 and/or the operating system. These optimizations are generally transparent to a FORTRAN user and include the following:

- Array addressing calculations

  Because the entire virtual address space of the PDP-11 can be represented in one word, array bounds expressions and array subscript expressions are always converted to INTEGER*2 before being used in an array address calculation. Therefore, even when the compiler is operating in /T mode, the code generated for array addressing is performed with INTEGER*2 operations.

- Input/output logical unit numbers

  Because logical unit numbers can always be represented by a 1-word integer, the compiler converts all unit numbers to INTEGER*2 when producing calls to the I/O section of the OTS.

- Direct access record numbers

  For simplicity of implementation, and to provide to programs that predominantly use 1-word integers the capability of using very large files, all direct access record numbers are processed as INTEGER*4 values.

4.3 BYTE (LOGICAL*1) DATA TYPE

FORTRAN-77 provides the byte data type (BYTE) to take advantage of the byte-processing capabilities of the PDP-11. Although LOGICAL*1 is a synonym for BYTE, a BYTE value is actually a signed integer. In addition to storing small integers, the byte data type is useful for storing and manipulating Hollerith information.

In general, when data of two different types are used in a binary operation, the lower-ranked type is converted, before any computations, to the higher-ranked type. However, in the case of a byte variable and an integer constant that can be represented as a byte variable, the integer constant is treated as a byte constant; therefore, the result of the operation is of type byte rather than of type integer, as it would be under the more general convention. The overflow possibilities under this convention, however, are similar to those previously discussed in Section 4.2.2 for mixed INTEGER*2 and INTEGER*4 variables and constants.

4.4 ITERATION COUNT MODEL FOR DO LOOPS

FORTRAN-77 provides an extended form of the DO statement. This statement has the following features:

- The control variable may be an INTEGER*2 , INTEGER*4 , REAL, or DOUBLE PRECISION variable.
- The initial value, step size, and final value of the control variable can be represented by any expressions whose resulting types are INTEGER*2 , INTEGER*4 , REAL, or DOUBLE PRECISION.
- The number of times the loop is executed (the iteration count) is determined when the DO statement is initialized and is not reevaluated during successive executions of the loop. Thus, the number of times the loop is executed is not affected by changing the values of the parameter variables used in the DO statement.

4.4.1 Cautions Concerning Program Interchange

Three common practices associated with the use of DO statements on other FORTRAN systems may not have the intended effects when used with FORTRAN-77. These are as follows:

- Assigning a value to the control variable within the body of the loop that is greater than the final value does not always cause early termination of the loop.
- Modifying a step size variable or a final value variable within the body of the loop does not modify the loop behavior or terminate the loop.
- Using a negative step size (for example, DO 10 I = 1,10,-1) in order to cause an arbitrarily long loop that is terminated by a conditional control transfer within the loop results in zero iterations of the loop body. A zero step size may result in an infinite loop at run time.

4.4.2 Iteration Count Computation

Given the following generic DO statement:

```
DO label V=m1,m2,m3
```

(where m1, m2, and m3 are any expressions), the iteration count is computed as follows:

```
count= MAX(INT(m2-m1+m3)/m3,0)
```

This computation does the following:

- Provides that the body of the DO loop will be executed zero times if the iteration count given by the above formula is zero (Under the /X switch, the loop is executed one time if the iteration count is zero.)
- Permits the step size (m3) to be negative or positive, but not zero

- Gives a well-defined and predictable value of an iteration count that results from any combination of values of the allowed result types

Be aware, however, that overflow of INTEGER*2 control variables is not detected and can result in an infinite loop at run time. Consider the following program unit:

```
      DO 10 I=1,32767
            .
            .
   10 CONTINUE
```

This program unit always results in an infinite loop when I is of INTEGER*2 type. See Section 4.2.2 for more information on integer overflow conditions.

You should also be aware that the effects of round-off error inherent in any floating-point computation, when real or double-precision values are used, may cause the count to be greater than, or less than, desired.

Under certain conditions, it is not necessary actually to compute the iteration count to obtain the required number of iterations; if all the parameters in an iteration computation are of type integer, and the step size is a constant (so that the sign of the increment value is known), the FORTRAN-77 compiler generates the necessary code to compare the control variable directly with the final value in order to control the number of iterations of the loop.

4.5 USING EQUIVALENCE WITH MIXED DATA TYPES

You can readily foresee the effects of EQUIVALENCE statements involving variables and/or arrays of mixed type when you consider the actual storage (in bytes) of each type of variable involved.

Example 4-1 illustrates the relationships that result when an EQUIVALENCE statement uses byte, integer, real, and complex elements.

Character data must not be equivalenced to data of any type other than character, BYTE, or LOGICAL*1.

Example 4-1: EQUIVALENCE Using Mixed Data Types

```
BYTE B (0:9)
COMPLEX C(4)
REAL R(3)
INTEGER*2 I(3)
EQUIVALENCE (C(2),R(3),I),(I(3),B(9))

Address         Storage Alignment

n          C(1) R(1)
n+1          .    .
n+2          .    .
n+3          .    .              B(0)
n+4          .  R(2)             B(1)
n+5          .    .              B(2)
```

(continued on next page)

Example 4-1 (Cont.): EQUIVALENCE Using Mixed Data Types

Address	Storage Alignment			
n+6	.	.		B(3)
n+7	.	.		B(4)
n+8	C(2)	R(3)	I(1)	B(5)
n+9	.	.	.	B(6)
n+10	.	.	I(2)	B(7)
n+11	.	.	.	B(8)
n+12	.		I(3)	B(9)
n+13	.		.	
n+14	.			
n+15	.			
n+16	C(3)			

4.6 EQUIVALENCE, BYTE DATA, AND STORAGE ALIGNMENT

The PDP-11 hardware requires that storage for all data elements except byte elements begin at an even address. This requirement can be satisfied in all except the following two cases:

- Equivalence relationships involving byte elements and nonbyte elements can make it logically impossible to allocate variables in a manner that satisfies the even-byte alignment constraint for all elements involved in an equivalence. An example of such an equivalence relationship is as follows:

```
BYTE B(2)
INTEGER*2 I,J
EQUIVALENCE (B(1),I),(B(2),J)
```

- Using a COMMON block in more than one program unit constitutes an implied relationship of equivalence among the sets of elements declared in that block. If a strict interpretation of the sequence of variable allocations causes a nonbyte variable to start at an odd address, a compiler adjustment is not made because it could destroy alignment properties expected in another program unit.

The compiler begins allocating each common block, and each group of equivalenced variables that are not in common, at an even address. If an allocation results in an element not of type byte being stored beginning at an odd address, an error message is produced. If this happens, to avoid fatal errors during execution, you must modify the common and/or EQUIVALENCE statements to eliminate the odd-byte addressing.

Variables and arrays not in common and not used in EQUIVALENCE statements are always correctly aligned.

4.7 ENTRY STATEMENT ARGUMENTS

The FORTRAN-77 implementation of argument association in ENTRY statements varies from that of some other FORTRAN systems.

As mentioned in Chapter 3 of this manual, FORTRAN-77 uses the call-by-reference method of passing arguments to called procedures. Some other FORTRAN implementations use the call-by-value/result method. This difference in approach is important to keep in mind when you reference dummy arguments in ENTRY statements.

Although standard FORTRAN allows you to use the same dummy arguments in different ENTRY statements, it allows you to reference only those dummy arguments that are defined for the ENTRY point being called. For example, given the subprogram unit

```
SUBROUTINE SUB1(X,Y,Z)
   .
   .
   .
ENTRY ENT1(X,A)
   .
   .
   .
ENTRY ENT2(B,Z,Y)
```

you can make the following references:

CALL	Valid References
SUB1	X Y Z
ENT1	X A
ENT2	B Z Y

FORTRAN implementations that use the call-by-value/result method, however, permit you to reference dummy arguments that are not defined in the ENTRY statement being called. For example, consider the following device for initializing dummy variables for subsequent referencing:

```
SUBROUTINE INIT(A,B,C)
RETURN
ENTRY CALC(Y,X)
Y = (A*X+B)/C
END
```

You can use this nonstandard device in call-by-value/result implementations because a separate internal variable is allocated for each dummy argument in the called procedure. When the procedure is called, each scalar actual-argument value is assigned to the corresponding internal variable, and these internal variables are then used whenever there is a reference to a dummy argument within the procedure. On return from the procedure, modified dummy arguments are copied back to the corresponding actual-argument variables.

When an entry point is referenced, all the dummy arguments of the entry point are defined with the values of the corresponding actual arguments and can be referenced on subsequent calls to the subprogram. However, you should avoid such subsequent referencings in programs that are to be compiled under FORTRAN-77, as they will not have the intended effect will produce programs that are not transportable to other systems that use the call-by-reference method.

FORTRAN-77 creates associations between dummy and actual arguments by passing the address of each actual argument to the called procedure. Each subsequent reference to a dummy argument generates an indirect address reference through the actual-argument address. When control returns from the called procedure, the association between actual and dummy arguments ends. The dummy arguments do not retain their values, and therefore cannot be referenced on subsequent calls. Therefore, to perform the kind of nonstandard references shown in the previous example, the subprogram would have to copy the values of the dummy

arguments to other variables. For example, if subroutine INIT is rewritten as follows, it will work on FORTRAN-77 as well as on systems that use the call-by-value/result method:

```
SUBROUTINE INIT(A1,B1,C1)
SAVE A,B,C
A = A1
B = B1
C = C1
RETURN
ENTRY CALC(Y,X)
Y = (A*X+B)/C
END
```

CHAPTER 5

PDP-11 FORTRAN-77/RT-11 PROGRAMMING CONSIDERATIONS

This chapter discusses techniques for writing effective FORTRAN-77 programs. Topics discussed are as follows:

- Efficient use of program statements and data types
- Compiler optimizations
- Program size and speed considerations
- Optional OTS capabilities

5.1 CREATING EFFICIENT SOURCE PROGRAMS

The following sections discuss the use of the PARAMETER, INCLUDE, OPEN, and CLOSE statements in relation to writing efficient source programs; they also discuss the efficient use of the INTEGER*2 and INTEGER*4 data types.

5.1.1 PARAMETER Statement

The PARAMETER statement provides a way for you to write programs containing easily modified parameters, such as array bounds and iteration counts, without losing the efficiency of using constant expressions to manipulate these parameters. Because the FORTRAN-77 compiler can optimize constants more efficiently than it can optimize variables (see Section 5.2.2), programs that use PARAMETER statements are generally more efficient than programs that initialize parameters with DATA or assignment statements. For example, the first program fragment below compiles into more efficient code than the second or third:

```
(1)       PARAMETER(M=50,N=100)
          DIMENSION X(M),Y(N)
          DO 5, I=1,M
          DO 5, J=1,N
     5    X(I) = X(I)*Y(J) + X(M)*Y(N)

(2)       DIMENSION X(50),Y(100)
          DATA M,N/50,100/
          DO 5, I=1,M
          DO 5, J=1,N
     5    X(I) = X(I)*Y(J) + X(M)*Y(N)
```

```
(3)       DIMENSION X(50),Y(100)
          M = 50
          N = 100
          DO 5, I=1,M
          DO 5, J=1,N
     5    X(I) = X(I)*Y(J) + X(M)*Y(N)
```

5.1.2 INCLUDE Statement

The INCLUDE statement provides a way for you to eliminate duplication of source code and to facilitate program maintenance. Because of the availability of the INCLUDE statement, you can create and maintain a separate file for a section of program text used by several different program units, and then include this text in the individual program units at compile time. For example, rather than duplicate the specification for a common block referenced by several program units, you can write the specification a single time in a separate file; then each program unit referencing the common block merely executes an INCLUDE statement to incorporate the specification into the unit. In addition to increasing programming efficiency, using the INCLUDE statement fosters reliability, modular programming, and ease of maintenance.

The following example shows the use of the INCLUDE statement.

The file COMMON.FTN defines the size of the blank common block and the size of the arrays X,Y, and Z.

```
          Main Program File                File COMMON.FTN

          INCLUDE 'COMMON.FTN'             PARAMETER M=100
          DIMENSION Z(M)                   COMMON X(M),Y(M)
          CALL CUBE
          DO 5 I=1,M
     5    Z(I)=X(I)+SQRT(Y(I))

               .
               .
               .

          SUBROUTINE CUBE
          INCLUDE 'COMMON.FTN'
          DO 10 I=1,M
    10    X(I)=Y(I)**3
          RETURN
          END
```

5.1.3 OPEN and CLOSE Statements

The OPEN and CLOSE statements provide you with precise and explicit -- as well as efficient -- control of I/O devices and files. Some examples follow:

- OPEN (UNIT=1, STATUS='NEW', INITIALSIZE=200)

  This statement creates a sequential file and allocates the space required for the file. Dynamic extension of a file is not allowed in RT-11.

- OPEN (UNIT=J, STATUS='NEW'...)
  .
  .

```
IF (IERR) CLOSE(UNIT=J, STATUS='DELETE')
     .
     .
CLOSE (UNIT=J, STATUS='SAVE')
```

If an error (denoted by IERR) occurs that makes the file created by the OPEN statement invalid or useless, the file is efficiently deleted.

```
      CHARACTER*14 FILNAM
1     TYPE 100
100   FORMAT('$INPUT  FILE?')
      ACCEPT 101,FILNAM
101   FORMAT (A)

      OPEN (UNIT=3, FILE=FILNAM, STATUS='OLD', ERR=9)
      ...

9     TYPE 102, FILNAM
102   FORMAT (' ERROR OPENING FILE ',A)
      GO TO 1
```

This program fragment reads a file specification into the character variable FILNAM. The specified file is then opened for processing.

5.1.4 INTEGER*2 and INTEGER*4

Because the PDP-11 is a 16-bit computer, the code sequences generated for INTEGER*4 computations are larger and slower than those for their INTEGER*2 counterparts. Therefore, the use of INTEGER*4 should be limited to those data items requiring 32-bit representation; INTEGER*2 should be used elsewhere. In general, it is advisable to minimize use of the /T compiler option.

5.2 COMPILER OPTIMIZATIONS

Optimization is producing the greatest amount of processing with the least amount of time and memory.

The primary goal of FORTRAN-77 optimization is to produce an object program that executes faster than an unoptimized version of the same source program. A secondary goal is to reduce the size of the object program.

The language elements you use in a source program directly affect the compiler's ability to optimize the object program. Therefore, you should be aware of the ways in which you can assist compiler optimization. The FORTRAN-77 compiler performs the following optimizations:

- Constant folding: Integer constant expressions are evaluated at compile-time.
- Compile-time constant conversion.
- Compile-time evaluation of constant subscript expressions in array calculations.
- Argument-list merging: If two function or subroutine references have the same arguments, a single copy of the argument list is generated.

- Branch instruction optimizations for arithmetic and logical IF statements.
- Eliminating unreachable ("dead") code: An optional warning message is issued to indicate unreachable statements in a source program.
- Recognizing and Replacing common subexpressions.
- Removing invariant computations from DO loops.
- Local register assignment: Frequently referenced variables are retained (if possible) in registers to reduce the number of load and store instructions required.
- Assigning frequently used variables and expressions to registers across DO loops.
- Constant pooling: Storage is allocated for only one copy of a constant in the compiled program. Constants, including most numeric constants, used as immediate-mode operands are not allocated storage.
- Inline code expansion for some intrinsic functions.
- Fast calling sequences for the real and double-precision versions of some intrinsic functions.
- Reordering the evaluation of expressions to minimize the number of temporary values required.
- Delaying unary minus and .NOT. operations to eliminate unary negation and complement operations.
- Partially evaluating Boolean expressions. For example, if e1 in the following expression has the value .FALSE., e2 is not evaluated:

  ```
  IF (e1.AND.e2) GO TO 20
  ```

  The order in which e1 and e2 appear in the source statement has no effect on partial evaluation.
- Peephole optimization of instruction sequences: examining code on an instruction-by-instruction basis to find operations that can be replaced by shorter, faster operations.

5.2.1 Characteristics of Optimized Programs

An optimized FORTRAN-77 program is computationally equivalent to an unoptimized program; therefore, identical numerical results are obtained and equivalent (in meaning, not quantity) run-time diagnostic messages are produced. An optimized program, however, can produce fewer run-time diagnostic messages and the diagnostics can occur at different statements in the source program.

Example 5-1: Effects of Optimization on Error Reporting

Unoptimized Program

```
      A = X/Y
      B = X/Y
      DO 10, I = 1,10
10    C(I) = C(I) * (X/Y)
```

Optimized Program

```
      t  = X/Y
      A  = t
      B  = t
      DO 10, I = 1,10
10    C(I) = C(I) * t
```

In Example 5-1, if Y has the value 0.0, the unoptimized program produces 12 zero-divide errors at run time; the optimized program, however, produces only one zero-divide error because the calculation that produces the error has been moved out of a loop. (Note that t is a temporary variable created by the compiler.)

Note that optimizations such as eliminating redundant calculations and moving invariant calculations out of loops can affect the use of the ERRTST system subroutine. For example, in the above program, a call to ERRTST from inside the loop does not detect a zero-divide error in the loop calculation because the compiler has moved the error-producing part of the calculation outside the loop.

5.2.2 Compile-time Operations on Constants

The compiler performs the following computations on expressions involving constants (including PARAMETER constants):

- Negation of constants: Constants preceded by unary minus signs are negated at compile time. For example:

  ```
  X = -10.0
  ```

  is compiled as a single move operation.

- Type conversion of constants: Lower-ranked constants are converted to the data type of the higher-ranked operand at compile time. For example:

```
X = 10*Y
```

  is compiled as:

```
X = 10.0*Y
```

- Integer arithmetic on constants: Expressions involving +, -, *, / or ** operators are evaluated at compile time. For example:

```
PARAMETER (NN=27)
I = 2*NN+J
```

  is compiled as:

```
I = 54+J
```

Array subscript calculations involving constants are simplified at compile time where possible. For example:

```
DIMENSION I(10,10)
I(1,2) = I(4,5)
```

is compiled as a single move instruction.

5.2.3 Source Program Blocks

FORTRAN-77 performs some optimizations only within the confines of a single "block" of a source program. A block is a sequence of one or more source statements. The start of a new block is generally defined by a labeled statement that is the target of a control transfer from another statement (for example, a GO TO, an arithmetic IF, or an ERR= option). An ENTRY statement also defines a new block. Some occurrences of statement labels do not define the start of a new block; these occurrences are as follows:

- Unreferenced statement labels.
- A label terminating a DO loop, provided the only references to the label occur in DO statements.
- Labels of FORMAT statements. FORMAT statements must be labeled, but control cannot be transferred to a FORMAT statement.
- Labels such that the only reference to the label occurs in the immediately preceding arithmetic IF statement. For example:

```
    IF(A) 10,20,20
10  X = 1.
```

- Singly referenced labels. A jump to a singly referenced label may be equivalent to an IF THEN/ENDIF structure. If it is, the IF THEN/ENDIF structure is used and the block is extended past the labeled statement.

The compiler imposes a limitation on the size of a single block. Therefore, a very long straight-line sequence of FORTRAN statements can be treated as several "blocks" during optimization.

A block can contain one or more DO loops, provided none of the labels within the loops defines the start of a new block. Therefore, the following are considered single blocks and are optimized as complete units:

Example 1

```
      X = B*C
      DO 10, I=1,N
10    A(I) = A(I)/(B*C)
      DO 20, J=1,N
20    Y(J) = Y(J)+B*C
```

Example 2

```
      DO 20, I=1,N
      DO 20, J=1,N
      SUM = 0.0
      DO 10, K=1,N
10    SUM = SUM+A(I,K)*B(K,J)
20    C(I,J) = SUM
```

If the label specified as the target of a GOTO in a logical IF is referenced only once, the structure may be equivalent to a block IF. For example, the following examples are equivalent:

Example 1

```
      IF (I .LT. J) GOTO 20
      A(I) = A(I)*J
      J=J-1
20    I=I+1
```

Example 2

```
      IF (I .GE. J) THEN
          A(I) = A(I)*J
          J=J-1
      ENDIF
      I=I+1
```

However, even though these two examples are equivalent, Example 2 is more easily optimized. Therefore, as long as Example 1 is valid (that is, as long as both the GOTO and the label are in the same block, and the nesting rules are not violated), FORTRAN-77 transforms Example 1 into the form shown in Example 2.

Optimizations can be done most effectively over complete structures. Therefore, if a block would otherwise be ended within either a block IF or DO structure, the block is instead ended at the beginning of the DO structure or the conditional block of the block IF structure.

Also, a more thoroughly optimized object program is produced if the number of separate blocks is minimized. The common-subexpression, code motion, and register allocation optimizations are performed only within single blocks.

Multiple block IF structures, as well as nested DO and block IF structures, can occur within a single block.

5.2.4 Eliminating Common Subexpressions

Often a subexpression appears in more than one computation within a program. If the values of the operands of such a subexpression are not changed between computations, the value of the subexpression can be computed once and substituted for each occurrence of the subexpression. For example, B*C is a common subexpression in the following sequence:

```
A = B*C+E*F
    .
    .
H = A+G-B*C
    .
    .
IF((B*C)-H)10,20,30
    .
    . The preceding sequence is compiled as:
```

```
t = B*C
A = t+E*F
    .
    .
H = A+G-t
    .
    .
IF((t)-H)10,20,30
    .
    .
```

where t is a temporary variable created by the compiler. Two computations of the subexpression B*C are eliminated from the sequence.

In the above example, you can modify the source program to eliminate the redundant calculation of (B*C). In the following example, however, you cannot reasonably modify the source program to achieve the same optimization ultimately effected by the compiler. The statements

```
DIMENSION A(25,25), B(25,25)
A(I,J)= B(I,J)
```

are compiled, without optimization, to a sequence of instructions of the form:

```
t1 = J*25+I
t2 = J*25+I
A(t1) = B(t2)
```

where the variables t1 and t2 represent equivalent expressions. Recognizing the redundancy, the compiler optimizes the sequence into the following shorter, faster sequence:

```
t = J*25 + I
A(t) = B(t)
```

If a common subexpression is created within a conditional block of a block IF, this subexpression can be used anywhere within the conditional block in which it was created, including within any nested inner blocks; but it cannot be used outside that conditional block.

5.2.5 Removing Invariant Computations From Loops

Execution speed is enhanced if invariant computations are moved out of loops. For example, in the sequence

```
      DO 10, I=1,100
10    F = 2.0*Q*A(I)+F
```

the value of the subexpression 2.0*Q is the same during each iteration of the loop. Transformation of the sequence to:

```
      t = 2.0*Q
      DO 10, I=1,100
10    F = t*A(I)+F
```

moves the calculation 2.0*Q outside the body of the loop and eliminates 99 multiply operations.

However, invariant computations cannot be moved out of a zero-trip DO loop. For example, in the sequence

```
      DO 10, I=1,N
10    F=2.0*Q*A(I)+F
```

statement 10 is not executed for certain values of n; therefore, the invariant computation 2.0*Q cannot be moved out of the loop.

5.3 RUN-TIME PROGRAMMING CONSIDERATIONS

You can often reduce the execution time of programs by making use of the following facts relevant to the FORTRAN-77 run-time environment.

- Unformatted I/O is substantially faster and more accurate than formatted I/O. The unformatted data representation usually occupies less file storage space as well. Therefore, you should use unformatted I/O for storing intermediate results on secondary storage.

- Specifying an array name in an I/O list is more efficient than using an equivalent implied DO list. A single I/O transmission call passes an entire array; however, an implied DO list can pass only a single array element for each I/O call.

- Implementing the BACKSPACE statement involves repositioning the file and scanning previously processed records. If a reread capability is required, it is more efficient to read the record into a temporary array and DECODE the array several times than to read and backspace the record.

- Array subscript checking is time-consuming and requires additional compiled code. It is primarily useful during program development and debugging.

- To obtain minimum direct access I/O processing, the record length should be an integer factor or multiple of the device block size of 512 bytes (for example, 32 bytes, 1024 bytes, and so on).

- Using run-time formats should be minimized. The compiler preprocesses FORMAT statements into an efficient internal form. Run-time formats must be converted into this internal form at run-time. In many cases, variable format expressions allow the format to vary at run time as needed.

5.4 FORTRAN-77 OPTIONAL CAPABILITIES

The FORTRAN-77 system, as distributed, contains several optional capabilities supported by alternate OTS modules. These capabilities include:

- Running FORTRAN-77 without a Floating Point Processor

- Choosing alternate run-time error reporting

- Obtaining an alternate floating-point output conversion routine

- Choosing an alternate random-number generator for compatibility with previous versions of the OTS (see Appendix B).

These options are described below. You can choose which options are

available on your system when you install F77. See the PDP-11 FORTRAN-77/RT-11 Installation Guide and Release Notes for complete information. None of these options is required for normal use of the FORTRAN-77 system.

5.4.1 Non-FPP Operation (F77EIS.OBJ)

The FORTRAN-77 compiler does not require a floating-point processor (FP11 or KEF11A) to compile a FORTRAN-77 program; the compiler can run on any PDP-11 with the EIS instruction set. However, the code generated by the FORTRAN-77 compiler is intended to run on a PDP-11 with FPP and may therefore contain FPP instructions.

A FORTRAN-77 source program containing no real, double-precision, or complex constants, variables, arrays, or function references is compiled into a PDP-11 program that contains no FPP instructions. If this program is linked using the module F77EIS.OBJ and the standard FORTRAN-77 OTS, as shown below, the resulting job executes no FPP instructions. Such programs can therefore run on any PDP-11 with the EIS instruction set.

```
LINK INT,SY:F77EIS.OBJ
```

If a compiled program unit contains no FPP instructions, the program listing contains the statement: NO FPP INSTRUCTIONS GENERATED.

5.4.2 Optional OTS Error Reporting (F77NER.OBJ)

An optional OTS module that does not perform any run-time diagnostic message reporting is available; it is several hundred words smaller than the standard error-reporting module. Error processing and calls to ERRSET, ERRSNS, and ERRTST continue to operate normally, only the logging of the diagnostic message to the user terminal being suppressed. If this option is used, STOP and PAUSE messages are not produced.

5.4.3 Short Error Text (SHORT.OBJ)

The error message text for run-time error reports is contained in memory and requires over 1000 words. An alternative version is available that requires only one word. If the alternative is used, the error report is complete except for the 1-line English text description of the error. This module, $SHORT, is included in the job at link time. For example:

```
LINK MAIN,SY:SHORT.OBJ
```

5.4.4 Intrinsic Function Name Mapping (F77MAP.OBJ)

As discussed in Section 4.1, references to FORTRAN intrinsic functions are transformed at compile time into calls that use internal names. Therefore, if a program written in MACRO-11 uses a FORTRAN name instead of an internal name to reference an intrinsic function, an unresolved reference results during link.

To prevent such unresolved references during the linking of a MACRO program, a set of concatenated object modules is provided for transforming FORTRAN-77 intrinsic-function names into internal names

at link time. For example, the name SIN is transformed at link time by means of the following module:

```
        .TITLE  $MSIN
SIN::   JMP     $SIN
        .END
```

The object module similar to the one for SIN is available for each intrinsic-function name.

An F77MAP library may be necessary to provide function mapping.

5.4.5 Floating-point Output Conversion (F77CVF.OBJ)

An alternative module for performing formatted output of floating-point values under control of the D, E, F, and G format codes is provided. The standard module uses multiple-precision, fixed-point integer techniques to maintain maximum accuracy during the conversion. (FPP hardware is not used.) The alternative module performs the same functions using the FPP hardware; it is Substantially faster but in some cases less accurate than the standard module. The standard module is accurate to 16 decimal digits; the optional module is accurate to 15 digits.

CHAPTER 6

USING CHARACTER DATA

The character data type facilitates the manipulation of alphanumeric data. You can use character data in the form of character variables, arrays, constants, and substrings.

6.1 CHARACTER SUBSTRINGS

You can select certain segments (substrings) from a character variable or array element by specifying the variable name, followed by delimiter values that indicate the leftmost and/or rightmost characters in the substring. For example, if the character string NAME contains:

```
ROBERT WILLIAM BOB JACKSON
```

and you want to extract the substring BOB, specify the following:

```
NAME(16:18)
```

If you omit the first value, you are indicating that the first character of the substring is the first character in the variable. For example, if you specify:

```
NAME(:18)
```

the resulting substring is:

```
ROBERT WILLIAM BOB
```

If you omit the second value, you are specifying the rightmost character to be the last character in the variable. For example:

```
NAME(16:)
```

encompasses:

```
BOB JACKSON
```

6.2 CHARACTER CONSTANTS

Character constants are strings of characters enclosed in apostrophes. You can assign a character value to a character variable in much the same way you would assign a numeric value to a real or integer variable. For example, as a result of the statement

```
XYZ = 'ABC'
```

the characters ABC are stored in location XYZ. Note that if XYZ's length is less than three bytes, the character string is truncated on the right. Thus, if you specify:

```
CHARACTER*2 XYZ

XYZ = 'ABC'
```

the result is AB. If, on the other hand, the variable is longer than the constant, it is padded on the right with blanks. For example, the statements

```
CHARACTER*6 XYZ

XYZ = 'ABC'
```

result in having:

```
ABC
```

stored in XYZ. If the previous contents of XYZ were CBSNBC, the result would still be ABC because the previous contents are overwritten.

You can give character constants symbolic names by using the PARAMETER statement. For example, if you specify:

```
CHARACTER*17 TITLE
PARAMETER (TITLE = 'THE METAMORPHOSIS')
```

you can use the symbolic name TITLE anywhere a character constant is allowed.

You can include an apostrophe as part of the constant by specifying two consecutive apostrophes. For example, the statements

```
CHARACTER*15 TITLE
PARAMETER (TITLE = 'FINNEGANS''S WAKE')
```

result in the character constant FINNEGAN'S WAKE.

The value assigned to a character parameter can only be a character constant.

6.3 DECLARING CHARACTER DATA

To declare variables or arrays as character type, you use the CHARACTER type declaration statement, as demonstrated in the following example:

```
CHARACTER*10 TEAM(12),PLAYER
```

This statement defines a 12-element character array (TEAM), each element of which is 10 bytes long; and a character variable (PLAYER), which is also 10 bytes long.

You can specify different lengths for variables in a CHARACTER statement by including a length value for specific variables. For example:

```
CHARACTER*6 NAME,AGE*2,DEPT
```

In this example, NAME and DEPT are defined as 6-byte variables, and AGE is defined as a 2-byte variable.

You can place CHARACTER data in FORTRAN-77 COMMON blocks, however, numeric data and CHARACTER data must not be combined in the same named or unnamed COMMON block. You should make certain that all COMMON blocks contain either all numeric or all CHARACTER data.

6.4 INITIALIZING CHARACTER VARIABLES

Use the DATA statement to preset the value of a character variable. For example:

```
CHARACTER*10 NAME, TEAM(5)
DATA        NAME/' '/,TEAM/'SMITH','JONES',
1           'DOE','BROWN','GREEN'/
```

Note that NAME contains 10 blanks, but that each array element in TEAM contains a character value, right-padded with blanks.

To initialize an array so that each of its elements contains the same value, use a DATA statement of the following type:

```
CHARACTER*5 TEAM(10)
DATA TEAM/10*'WHITE'/
```

The result is a 10-element array in which each element contains WHITE.

6.5 CHARACTER DATA EXAMPLES

An example of character data usage is shown in Example 6-1. The example is a program that manipulates the letters of the alphabet. The results are shown in Example 6-2.

6.6 CHARACTER LIBRARY FUNCTIONS

The FORTRAN-77 system provides the following character functions:

- ICHAR
- INDEX
- LEN
- LGE, LGT, LLE, LLT

6.6.1 ICHAR Function

The ICHAR function returns an integer ASCII code equivalent to the character expression passed as its argument. It has the form:

```
ICHAR(c)
```

c

A character expression. If c is longer than one byte, the ASCII code equivalent to the first byte is returned and the remaining bytes are ignored.

Example 6-1: Character Data Usage

```
      CHARACTER C,ALPHA*26
      DATA ALPHA/'ABCDEFGHIJKLMNOPQRSTUVWXYZ'/
      WRITE(6,90)
90    FORMAT(' CHARACTER EXAMPLE PROGRAM OUTPUT')

      DO 10 I = 1,26
          WRITE(6,*) ALPHA
          C = ALPHA(1:1)
          ALPHA(1:25) = ALPHA(2:26)
          ALPHA(26:26) = C
10    CONTINUE

      CALL REVERS(ALPHA)
      WRITE(6,*) ALPHA

      CALL FIND('UVW',ALPHA)
      CALL FIND('AAA','DAAADHAJDAAAJAAA CEUEBCUEI')

      WRITE (6,*) ' END OF CHARACTER EXAMPLE PROGRAM'
      END

      SUBROUTINE REVERS(S)
      CHARACTER T*1,S*26

      K = 26
      DO 10 I = 1, K/2
             T = S(I:I)
             S(I:I) = S(K:K)
             S(K:K) = T
             K = K -1
10    CONTINUE
      RETURN
      END


      SUBROUTINE FIND(SUB,S)
      CHARACTER*3 SUB, S*26
      CHARACTER*132 MARKS

      I = 1
      MARKS = ' '
10    J = INDEX(S(I:),SUB)
      IF (J .NE. 0) THEN
      I = I + (J-1)
      MARKS(I:I) = '#'
      I = I+1
      IF (I .LE. LEN(S)) GO TO 10
      ENDIF

      WRITE(6,91) S, MARKS
91    FORMAT(2(/1X,A))
      RETURN
      END
```

Example 6-2: Output Generated by Example Program

```
CHARACTER EXAMPLE PROGRAM OUTPUT
ABCDEFGHIJKLMNOPQRSTUVWXYZ
BCDEFGHIJKLMNOPQRSTUVWXYZA
CDEFGHIJKLMNOPQRSTUVWXYZAB
DEFGHIJKLMNOPQRSTUVWXYZABC
EFGHIJKLMNOPQRSTUVWXYZABCD
FGHIJKLMNOPQRSTUVWXYZABCDE
GHIJKLMNOPQRSTUVWXYZABCDEF
HIJKLMNOPQRSTUVWXYZABCDEFG
IJKLMNOPQRSTUVWXYZABCDEFGH
JKLMNOPQRSTUVWXYZABCDEFGHI
KLMNOPQRSTUVWXYZABCDEFGHIJ
LMNOPQRSTUVWXYZABCDEFGHIJK
MNOPQRSTUVWXYZABCDEFGHIJKL
NOPQRSTUVWXYZABCDEFGHIJKLM
OPQRSTUVWXYZABCDEFGHIJKLMN
PQRSTUVWXYZABCDEFGHIJKLMNO
QRSTUVWXYZABCDEFGHIJKLMNOP
RSTUVWXYZABCDEFGHIJKLMNOPQ
STUVWXYZABCDEFGHIJKLMNOPQR
TUVWXYZABCDEFGHIJKLMNOPQRS
UVWXYZABCDEFGHIJKLMNOPQRST
VWXYZABCDEFGHIJKLMNOPQRSTU
WXYZABCDEFGHIJKLMNOPQRSTUV
XYZABCDEFGHIJKLMNOPQRSTUVW
YZABCDEFGHIJKLMNOPQRSTUVWX
ZABCDEFGHIJKLMNOPQRSTUVWXY
ZYXWVUTSRQPONMLKJIHGFEDCBA

ZYXWVUTSRQPONMLKJIHGFEDCBA


DAAADHAJDAAAJAAA CEUEBCUEI
 #       #   #
 END OF CHARACTER EXAMPLE PROGRAM
```

6.6.2 INDEX Function

The INDEX function is used to determine the starting position of a substring. It has the form:

```
INDEX(c1,c2)
```

c1

A character expression that specifies the string to be searched for a match with the value of c2.

c2

A character expression representing the substring for which a match is desired.

If INDEX finds an instance of the specified substring (c2), it returns an integer value corresponding to the starting location in the string (c1). For example, if the substring sought is CAT and the string that is searched contains DOGCATFISHCAT, the return value of INDEX is 4.

If INDEX cannot find the specified substring, it returns the value 0.

If there are multiple occurrences of the substring, INDEX locates the first (left-most) one. Use of the INDEX function is illustrated in Examples 6-1 and 6-2. Note that the FORTRAN-77 INDEX function is distinctly different from the RT-11 SYSLIB INDEX function. See section 6.6.5 below.

6.6.3 LEN Function

The LEN function returns an integer value that indicates the length of a character expression. It has the form:

```
LEN(c)
```

c

A character expression.

Note that the FORTRAN-77 LEN function is distinctly different from the RT-11 SYSLIB LEN function. See section 6.6.5 below.

6.6.4 LGE, LGT, LLE, LLT Functions

The lexical comparison functions (LGE, LGT, LLE, and LLT) compare two character expressions, using the ASCII collating sequence. The result is the logical value .TRUE. if the lexical relation is true, and .FALSE. if the lexical relation is not true. The functions have the forms:

```
LGE (c1,c2)
LGT (c1,c2)
LLE (c1,c2)
LLT (c1,c2)
```

c1,c2

Character expressions.

You may wish to include these functions in FORTRAN programs that can be used on computers that do not use the ASCII character set. In FORTRAN-77, the lexical comparison functions are equivalent to the .GE., .GT., .LE., .LT. relational operators. For example, the statement

```
IF (LLE (string1, string2)) GO TO 100
```

is equivalent to:

```
IF (string1.LE.string2) GO TO 100
```

6.6.5 RT-11 SYSLIB INDEX and LEN Functions

RT-11 provides in its system subroutine library (SYSLIB), string handling functions LEN and INDEX, as well as numerous others that can be used to manipulate NULL-terminated BYTE strings. The FORTRAN-77 INDEX and LEN functions are different functions, and generally employ in-line code rather thah calling external library subroutines. The two data structures are not compatible, as CHARACTER data is not terminated by a NULL byte.

The FORTRAN-77 compiler senses whether the reference to LEN or INDEX is to be made to its own routines or to external routines (SYSLIB or USER provided) by virtue of the calling argument type. If the argument is not of type CHARACTER, then an external function is assumed and an error 72 is generated. This error is really only a warning, and should be ignored if you intend to use the SYSLIB functions.

In a given program or subprogram module, you can use either SYSLIB's functions, with NULL-terminated BYTE strings, or FORTRAN-77's functions with CHARACTER data. However, you should not attempt to use both.

6.7 CHARACTER I/O

The character data type simplifies transmitting alphanumeric data. You can read and write character strings of any length from 1 to 255 characters.

For example; the statements

```
      CHARACTER*24 TITLE
           .
           .
           .
      READ (12,100) TITLE
100   FORMAT (A)
```

cause 24 characters to be read from logical unit 12 and stored in the 24-byte character variable TITLE. If instead of character data you were to use Hollerith data stored in numeric variables or arrays, the following code is necessary:

```
      INTEGER*4 TITLE(6)
           .
           .
           .
      READ (12,100) TITLE
100   FORMAT (6A4)
```

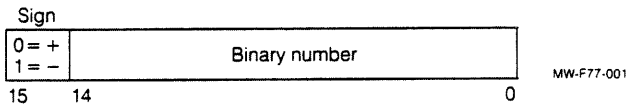
Note that you must divide the data into lengths suitable for real or (in this case) integer data, and specify I/O and FORMAT statements to match. In this example, a one-dimensional array comprising six 4-byte elements is filled with 24 characters from logical unit 12.

APPENDIX A

FORTRAN-77 DATA REPRESENTATION

A.1 INTEGER FORMATS

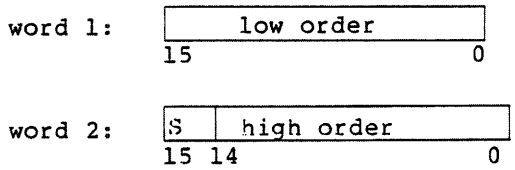
A.1.1 INTEGER*2 Format



Integers are stored in two's complement representation. INTEGER*2 values lie in the range -32768 to +32767. For example:

```
+22 = 000026   (Octal)
 -7 = 177771   (Octal)
```

A.1.2 INTEGER*4 Format



INTEGER*4 values are stored in two's complement representation. The first word contains the low-order part of the value; the second word contains the sign and high-order part of the value. Note that if the value is in the range of an INTEGER*2 value (-32768 to +32767), then the first word may be referenced as an INTEGER*2 value.

A.2 FLOATING-POINT FORMATS

The exponent for both 2-word and 4-word floating-point formats is stored in excess-128 notation. Binary exponents from -128 to +127 are represented by the binary equivalents of 0 through 255. Fractions are represented in sign-magnitude notation, with the binary radix point to the left. Numbers are assumed to be normalized; therefore, because it would be redundant, the most significant bit is not stored (the practice of not storing the most significant bit is called "hidden bit normalization"). The unstored bit is assumed to be a 1 unless the exponent is 0 (corresponding to 2**-128), in which case the unstored bit is assumed to be 0. The value 0 is represented by an exponent

field of 0 and a sign bit of 0. For example, +1.0 would be represented in octal by:

```
40200
0
```

in the 2-word format, or:

```
40200
0
0
0
```

in the 4-word format. The decimal number -5.0 is:

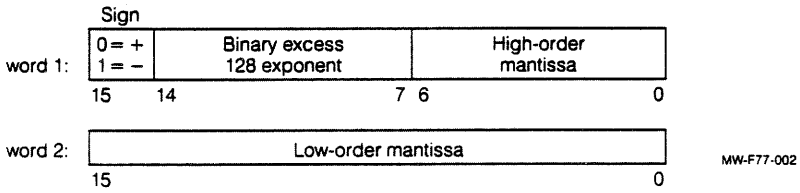
```
140640
0
```

in the 2-word format, or:

```
140640
0
0
0
```

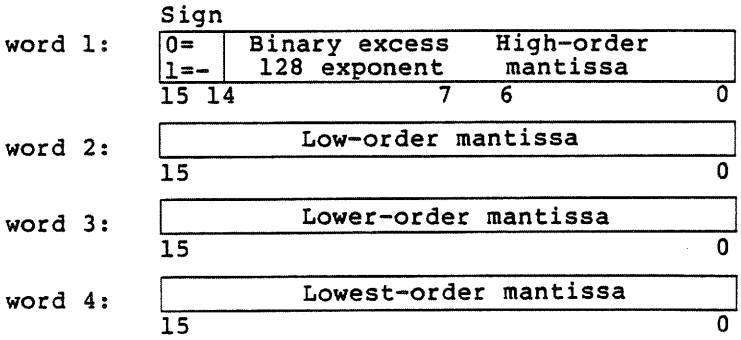
in the 4-word format.

A.2.1 REAL (REAL*4) Format (2-Word Floating Point)



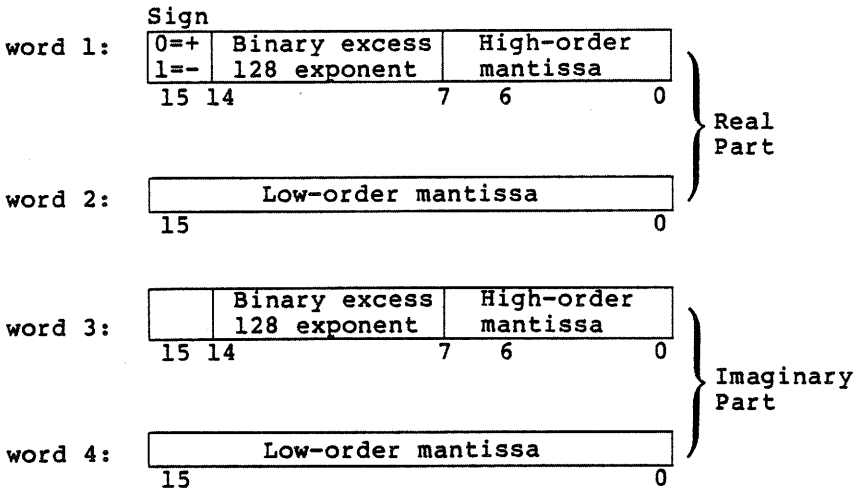
The form of a single-precision real number is sign magnitude, with bit 15 the sign bit, bits 14:7 an excess 128 binary exponent, and bits 6:0 and 15:0 in the second word a normalized 24-bit fraction with the redundant most significant fraction bit not represented. The value of a single-precision real number is in the approximate range .29*10**-38 through 1.7*10**38. The precision is approximately one part in 2**23--or typically seven decimal digits.

A.2.2 DOUBLE-PRECISION (REAL*8) Format (4-Word Floating Point)



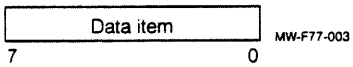
The form of a double-precision real number is identical to that of a single-precision real number except for an additional 32 low-significance fraction bits. The exponent conventions and approximate range of values are the same as for a single-precision real value. The precision is approximately one part in 2**55--or typically 16 decimal digits.

A.2.3 COMPLEX Format



The form of a complex number is an ordered pair of real numbers. The first real number represents the real part of the imaginary number; the second represents the imaginary part.

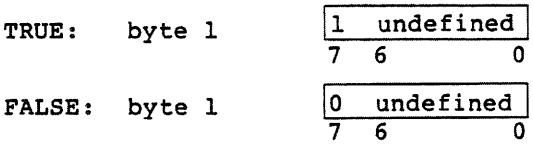
A.3 LOGICAL*1 (BYTE) FORMAT



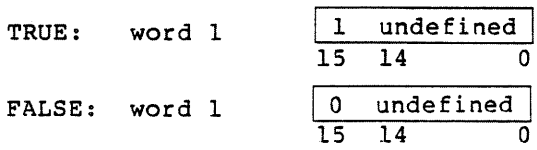
The logical values true or false (see Section A.4), a single Hollerith character, or integers in the range of numbers from +127 to -128 can be represented in LOGICAL*1 format. LOGICAL*1 array elements are stored in adjacent bytes.

A.4 LOGICAL FORMATS

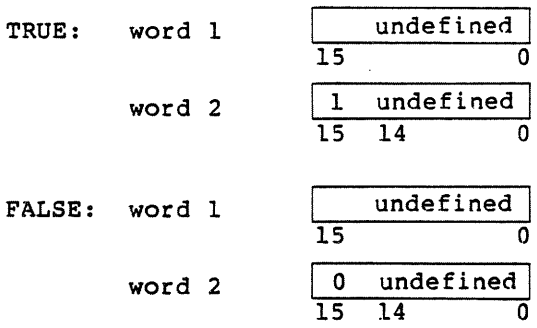
LOGICAL*1



LOGICAL*2

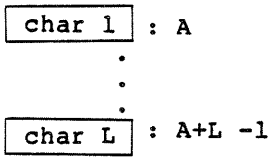


LOGICAL*4



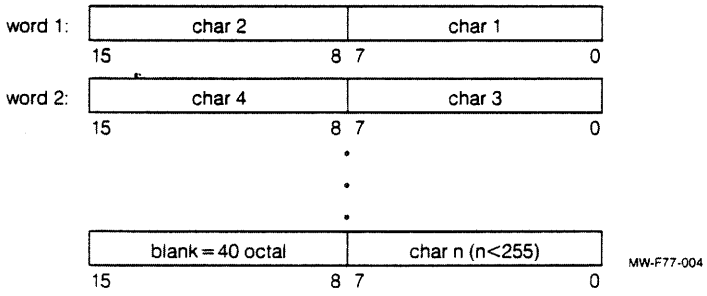
A.5 CHARACTER REPRESENTATION

A character string is a contiguous sequence of bytes in memory.



A character string is specified by two attributes: the address A of the first byte of the string, and the length L of the string in bytes. The length L of a string is in the range 1 through 255.

A.6 HOLLERITH FORMAT



Hollerith constants are stored one character per byte. Hollerith values are padded on the right with blanks, if necessary, to fill the associated data item.

A.7 RADIX-50 FORMAT

Radix-50 character set

Value (Octal)	Octal ASCII	Character Equivalent Radix-50
(space)	40	0
A-Z	101-132	1-32
$	44	33
.	56	34
(unused)		35
0-9	60-71	36-47

Radix-50 values are stored, up to three characters per word, by packing the Radix-50 values into single numeric values according to the formula:

```
((i*50+j)*50+k)
```

i, j, k

The code values of three Radix-50 characters.

The maximum Radix-50 value is, therefore:

```
47*50**2+47*50+47=174777(8)
```

The following table provides a convenient means of translating between the ASCII character set and Radix-50 equivalents. For example, given the ASCII string X2B, the Radix-50 equivalent is (arithmetic is performed in octal):

```
  X=113000
  2=002400
  B=000002
X2B=115402
```

Single Character or First Character	Second Character	Third Character
000000	000000	000000 (space)
A 003100	A 000050	A 000001
B 006200	B 000120	B 000002
C 011300	C 000170	C 000003
D 014400	D 000240	D 000004
E 017500	E 000310	E 000005
F 022600	F 000360	F 000006
G 025700	G 000430	G 000007
H 031000	H 000500	H 000010
I 034100	I 000550	I 000011
J 037200	J 000620	J 000012
K 042300	K 000670	K 000013
L 045400	L 000740	L 000014
M 050500	M 001010	M 000015
N 053600	N 001060	N 000016
O 056700	O 001130	O 000017
P 062000	P 001200	P 000020
Q 065100	Q 001250	Q 000021
R 070200	R 001320	R 000022
S 073300	S 001370	S 000023
T 076400	T 001440	T 000024
U 101500	U 001510	U 000025
V 104600	V 001560	V 000026
W 107700	W 001630	W 000027
X 113000	X 001700	X 000030
Y 116100	Y 001750	Y 000031
Z 121200	Z 002020	Z 000032
$ 124300	$ 002070	$ 000033
. 127400	. 002140	. 000034
132500	002210	000035 (unused)
0 135600	0 002260	0 000036
1 140700	1 002330	1 000037
2 144000	2 002400	2 000040
3 147100	3 002450	3 000041
4 152200	4 002520	4 000042
5 155300	5 002570	5 000043
6 160400	6 002640	6 000044
7 163500	7 002710	7 000045
8 166600	8 002760	8 000046
9 171700	9 003030	9 000047

APPENDIX B

ALGORITHMS FOR APPROXIMATION PROCEDURES

This appendix contains brief descriptions of the algorithms used in intrinsic functions that involve approximations.

Some of the descriptions below give relative error bounds. These relative error bounds are for the approximating polynomials involved in the algorithms, and assume exact arithmetic. Possible additional sources of errors not reflected in these error bounds are:

- Rounding and truncation errors that can occur when a given argument is reduced to the range in which approximations for a polynomial or rational fraction are valid
- Rounding errors that can occur as a result of using finite-precision, floating-point arithmetic in polynomial or rational-fraction computations

B.1 REAL-VALUE PROCEDURES

B.1.1 ACOS -- Real Floating-Point, Arc Cosine

ACOS(X) is computed as:

```
If X = 0, then ACOS(X) = pi/2
If X = 1, then ACOS(X) = 0
If X = -1, then ACOS(X) = pi
If 0 < X < 1, then ACOS(X) = ATAN(SQRT(1-X**2)/X)
If -1 < X <0, then ACOS(X) = ATAN(SQRT(1-X**2)/X) + pi
If 1 <  ABS(X) , error
```

B.1.2 DACOS -- Double-Precision Floating-Point Arc Cosine

DACOS(X) is computed as:

```
If X = 0, then DACOS(X) = pi/2
If X = 1, then DACOS(X) = 0
If X = -1, then DACOS(X) = pi
If 0 < X < 1, then DACOS(X) = DATAN(DSQRT(1-X**2)/X)
If -1 < X < 0, then DACOS(X) = DATAN(DSQRT(1-X**2)/X) + pi
If 1 <  ABS(X), error
```

B.1.3 ASIN -- Real Floating-Point Arc Sine

ASIN(X) is computed as:

```
If X = 0, then ASIN(X) = 0
If X = 1, then ASIN(X) = pi/2
If X = -1, then ASIN(X) = -pi/2
If 0 <  ABS(X) < 1, then ASIN(X) = ATAN(X/SQRT(1-X**2))
If 1 <  ABS(X), error
```

B.1.4 DASIN -- Double-Precision Floating-Point Arc Sine

DASIN(X) is computed as:

```
If X = 0, then DASIN(X) = 0
If X = 1, then DASIN(X) = pi/2
If X = -1, then DASIN(X) = -pi/2
If 0 <  ABS(X) < 1, then DASIN(X) = DATAN(X/DSQRT(1-X**2))
If 1 <  ABS(X), error
```

B.1.5 ATAN -- Real Floating-Point Arc Tangent

ATAN(X) is computed as:

1. If X < 0, then:
   Begin
   Perform Steps 2, 3, and 4 with arg = ABS(X)
   Negate the result since ATAN(X) = -ATAN(-X)
   Return End

2. If ABS(X) > 1, then:
   Begin
   Perform Steps 3 and 4 with arg = 1/ABS(X)
   Negate result and add a bias of pi/2 since
   ATAN(ABS(X)) = pi/2 - ATAN(1/ABS(X))
   Return End

3. At this point the argument is 1 >= X >= 0
   If ABS(X) > TAN(pi/12), then:
   Begin
   Perform Step 4 with arg = (X * SQRT(3) - 1)/
   (SQRT(3) + X)
   Add pi/6 to the result
   Return End

   Note: (X * SQRT(3) -1)/(X + SQRT(3)) <= TAN(pi/12) for
   ABS(X) >= TAN(pi/12)

4. Finally, the argument is ABS(X) <= TAN(pi/12)
   Begin
   ATAN(X) = X * SUM(C[i] * X**(2[i])), i = 0:4
   Return End

The coefficients C[i] are drawn from Hart #4941.[1]
The relative error is <= 10**-9.54.

---

[1] Hart, J. F. et al., Computer Approximations (John Wiley & Sons, 1968), P. 267.

B.1.6 ATAN2 -- Real Floating-Point Arc Tangent with Two Parameters

ATAN2(X,Y) is computed as:

```
If Y = 0 or X/Y > 2**25, ATAN(X,Y) = pi/2 * (sign X)
If Y > 0 and X/Y <= 2**25, ATAN2(X,Y) = ATAN(X/Y)
If Y < 0 and X/Y <= 2**25, ATAN2(X,Y) = pi * (sign X)
        + ATAN(X/Y)
```

B.1.7 DATAN -- Double-Precision Floating-Point Arc Tangent

DATAN(x) is computed as:

1. If X < 0, then:
   Begin
   Perform Steps 2, 3, and 4 with arg = ABS(X)
   Negate the result since DATAN(X) = -DATAN(-X)
   Return
   End

2. If ABS(X) > 1, then:
   Begin
   Perform Steps 3 and 4 with arg = 1/ABS(X)
   Negate result and add a bias of pi/2 since
   DATAN(ABS(X)) = pi/2 - DATAN(1/ABS(X))
   Return
   End

3. At this point the argument is 1 >= X >= 0
   If ABS(X) > DATAN(pi/12) then:
   Begin
   Perform Step 4 with arg = (X*DSQRT(3) - 1)/
   (DSQRT(3) + X)
   Add pi/6 to the result
   Return
   End

   Note: (X*DQRT(3) -1)/(X + DQRT(3)) <= DATAN(pi/12 for
   AB(X) >= DATAN(pi/12)

4. Finally, the argument is ABS(X) <= DATAN(pi/12):
   Begin
   DATAN(X) = X * SUM(C[i] * X**(2*i)), i = 0:8
   Return
   End

   The coefficient C[i]'s are drawn from Hart #4941.[1]
   The relative error is <= 10**-9.54.

B.1.8 DATAN2 -- Double-Precision Floating-Point Arc Tangent with Two Parameters

```
If Y = 0 or X/Y > 2**25, DATAN2(X,Y) = pi/2 * (sign X)
If Y > 0 and X/Y <= 2**25, DATAN2(X,Y) = DATAN(X/Y)
If Y < 0 and X/Y <= 2**25, DATAN2(X,Y) = pi * (sign X)
        + DATAN(X/Y)
```

---

[1] Hart, Computer Approximations p. 267.

B.1.9 ALOG10 -- Real Floating-Point Common Logarithm

ALOG10(x) is computed as:

```
ALOG10(E) * ALOG(X)
```

where:

E = 2.718, the base of the natural log system.

See the description of ALOG (Section B.1.21) for the complete algorithm.

B.1.10 DLOG10 -- Double-Precision Floating-Point Common Logarithm

DLOG10(X) is computed as:

```
DLOG10(E) * DLOG (X)
```

where:

E = 2.718, the base of the natural log system.

See the description of DLOG (Section B.1.22) for the complete algorithm.

B.1.11 COS -- Real Floating-Point Cosine

COS(X) is computed as:

```
SIN(X+pi/2)
```

See the description of SIN (Section B.1.23) for the complete algorithm.

B.1.12 DCOS -- Double-Precision Floating-Point Cosine

DCOS(X) is computed as:

```
DSIN(X+pi/2).
```

See the description of DSIN (Section B.1.24) for the complete algorithm.

B.1.13 EXP -- Real Floating-Point Exponential

EXP(X) is computed as:

```
If X > 88.028, overflow occurs
If X <= -88.5, EXP(X) = 0
If ABS(X) < 2**-28, EXP(X) = 1
```

Otherwise:

```
EXP(X) = 2**Y * 2**Z * 2**W
```

where:

```
Y = INTEGER(X*LOG2(E))
V = FRAC(X*LOG2(E)) * 16
Z = INTEGER(V)/16
W = FRAC(V)/16

          P+wQ
2**W =
          P-wQ
```

P and Q are first degree polynomials in W**2.
The coefficients of P and Q are drawn from Hart #1121.[1]

Powers of 2**(1/16) are obtained from a table. All arithmetic is done in double precision and then rounded to single precision at the end of calculation. The relative error is less than or equal to 10**-16.4.

B.1.14 DEXP -- Double-Precision Floating-Point Exponential

See the description of EXP (Section B.1.13). The approximation is identical except that there is no conversion to single precision at the end.

B.1.15 COSH -- Real Floating-Point Hyperbolic Cosine

COSH(X) is computed as:

```
If  ABS(X)  < 2**-11, COSH(X) = 1

If 2**-11 <= ABS(X)  < 0.25,
     COSH(X) = DIGITAL's approximation[2]
If 0.25 <= ABS(X) <= 87.0,
     COSH(X) = (EXP(X) + EXP(-X))/2
If 87.0 <  ABS(X)  and  ABS(X)  - LOG(2) < 87,
     COSH(X) = EXP(ABS(X)  - LOG(2))

If 87.0 <  ABS(X)  and  ABS(X)  - LOG(2) >= 87, then overflow
```

B.1.16 DCOSH -- Double Floating-Point Hyperbolic Cosine

DCOSH(X) is computed as:

```
If  ABS(X)  < 2**-27, DCOSH(X) = 1

If 2**-27 <= ABS(X) < 0.25,
     DCOSH(X) = DIGITAL's approximation[2]
```

---

[1] Hart, Computer Approximations, p. 206.

---

[2] This approximation is proprietary.

```
If 0.25 <= ABS(X) <= 87.0,
     DCOSH(X) = (DEXP(X) + DEXP(-X))/2

If 87.0 < ABS(X) and ABS(X) - LOG(2) < 87,
     DCOSH(X) = DEXP(ABS(X) - LOG(2))

If 87.0 < ABS(X) and ABS(X) - LOG(2) >= 87, then overflow
```

B.1.17 SINH -- Real Floating-Point Hyperbolic Sine

SINH(X) is computed as:

```
If  ABS(X)  < 2**-11, SINH(X) = X

If 2**-11 <= ABS(X) < 0.25,
     SINH(X) = DIGITAL's approximation[2]

If 0.25 <= ABS(X) <= 87.0,
     SINH(X) = (EXP(X) - EXP(-X))/2

If 87.0 < ABS(X) and ABS(X) - LOG(2) < 87,
     SINH(X) = sign(X) * EXP(ABS(X)  - LOG(2))

If 87.0 < ABS(X) and ABS(X) - LOG(2) >= 87, then overflow
```

B.1.18 DSINH -- Double-Precision Floating-Point Hyperbolic Sine

DSINH(x) is computed as:

```
If ABS(X) < 2**-27, DSINH(X) = X

If 2**-27 <= ABS(X) < 0.25,
     DSINH(X) = DIGITAL's approximation

If 0.25 <= ABS(X) <= 87.0,
     DSINH(X) = (DEXP(X) - DEXP(-X))/2

If 87.0 < ABS(X) and ABS(X) - LOG(2) < 87,
     DSINH(X) = sign(X) * DEXP(ABS(X) - LOG(2))

If 87.0 < ABS(X) and ABS(X) - LOG(2) >= 87, then overflow
```

B.1.19 TANH -- Real Floating-Point Hyperbolic Tangent

TANH(X) is computed as:

```
If ABS(X) <= 2**-14, then TANH(X) = X

If 2**-14 < ABS(X) <= 0.25, then TANH(X) = SINH(X) / COSH(X)

If 0.25 < ABS(X) < 16.0, then
     TANH(X) = (EXP(2*X) - 1)/(EXP(2*X) + 1)

If 16.0 <= ABS(X), then TANH(X) = sign(X) * 1
```

B.1.20 DTANH -- Double-Precision Floating-Point Hyperbolic Tangent

DTANH(X) is computed as:

If ABS(X) <= 2**-14, then DTANH(X) = X

If 2**-14 < ABS(X) <= 0.25, then DTANH(X) = DSINH(X)/DCOSH(X)

If 0.25 < ABS(X) < 16.0, then
DTANH(X) = (DEXP(2*X) - 1)/(DEXP(2*X) + 1)

If 16.0 <= ABS(X), then DTANH(X) = sign(X) * 1

B.1.21 ALOG -- Real Floating-Point Natural Logarithm

ALOG(x) is computed as:

If X <= 0, an error is signaled.

Therefore, let X = Y * (2**A)

where:

1/2 <= Y < 1

Then LOG(X) = A * LOG(2) + LOG(Y)

If ABS(X-1) <= 0.25, let W = (X-1)/(X+1)

Then, LOG(X) = W * SUM(C[i] * W**(2*i))

Otherwise, let W = (Y-SQRT(2)/2)/(Y+SQRT(2)/2)

Then, LOG(X) = A * LOG(2) - 1/2 * LOG(2) +
W * SUM C[i] * W**(2*i)

The coefficients are drawn from Hart #2662.[1]
The polynomial approximation used is of degree 4.

The relative error is less than or equal to 10**-9.9.

B.1.22 DLOG -- Double-Precision Floating-Point Natural Logarithm

DLOG(x) is computed as:

If X <= 0, an error is signaled.

Therefore, let X = Y * (2**A)

where:

1/2 <= Y < 1

Then, DLOG(X) = A * DLOG(2) + DLOG(Y)

---

[1] Hart, Computer Approximations, p. 227.

```
If  ABS(X-1)  <= 0.25, then let W = (X-1)/(X+1)

Then DLOG(X) = W * SUM (C[i] * W**(2*i))

Otherwise, let W = (Y - DSQRT(2)/2)/(Y + DSQRT(2)/2)

Then DLOG(X) = A * DLOG(2) - 1/2 * DLOG(2) +
     W * SUM(C[i] * W**(2*i)
```

The coefficients are drawn from Hart #2662.[1]
The polynomial approximation used is of degree 6.

The relative error is less than or equal to 10**-9.9.

B.1.23 SIN -- Real Floating-Point Sine

SIN(X) is computed as:

```
Let Q = INTEGER(ABS(X)/(pi/2))
```

where:

```
   Q = 0 for first quadrant
   Q = 1 for second quadrant
   Q = 2 for third quadrant
   Q = 3 for fourth quadrant
```

```
Let Y = FRACTION((ABS(X)/(pi/2))

If ABS(Y) < 2**-14, the sine is computed as:

   SIN(X) = S * (pi/2)

      S = Y          if Q = 0
      S = 1-Y        if Q = 1
      S = -Y         if Q = 2
      S = Y-1        if Q = 3
```

For all other cases:

```
    SIN(X) = P(Y*pi/2)            if Q = 0
    SIN(X) = P((1-Y)*pi/2)        if Q = 1
    SIN(X) = P(-Y*pi/2)           if Q = 2
    SIN(X) = P((Y-1)*pi/2)        if Q = 3
```

where:

```
P = Y*SUM(C[i]*(Y**(2*i)))   for i = 0:4
```

The coefficients are taken from Hastings.[2]
The polynomial approximation used is of degree 4.

The relative error is less than or equal to 10**-8. The result is guaranteed to be within the closed interval -1.0 to +1.0.

---

[1] Hart, Computer Approximations, p. 227.

---

[2] Hastings, C. et al., Approximation for Digital Computers (Princeton University Press, 1955), Sheet 16 (Part 2, p. 140).

B.1.24 DSIN -- Double-Precision Floating-Point Sine

DSIN(X) is computed as:

Let Q = INTEGER(ABS(X)/(pi/2))

where:

Q = 0 for first quadrant
Q = 1 for second quadrant
Q = 2 for third quadrant
Q = 3 for fourth quadrant

Let Y = FRACTION((ABS(X)/(pi/2))

If ABS(Y) < 2**-28, the sine is computed as:

DSIN(X) = S * (pi/2)

```
S = Y        if Q = 0
S = 1-Y      if Q = 1
S = -Y       if Q = 2
S = Y-1      if Q = 3
```

For all other cases:

```
DSIN(X) = P(Y*pi/2)          if Q = 0
DSIN(X) = P((1-Y)*pi/2)      if Q = 1
DSIN(X) = P(-Y*pi/2)         if Q = 2
DSIN(X) = P((Y-1)*pi/2)      if Q = 3
```

where:

P = Y*SUM(C[i]*(Y**(2*i))) for i = 0:8

The coefficients are taken from Hastings.

The polynomial approximation used is of degree 8.

The relative error is less than or equal to 10**-18.6. The result is guaranteed to be within the closed interval -1.0 to +1.0.

No loss of precision occurs if X < 2 * pi *256.

B.1.25 SQRT -- Real Floating-Point Square Root

SQRT(X) is computed as:

If X <= 0, an error is signaled. Therefore, let X = -X .

Let X = 2**K * F

where:

K is the exponential part of the floating-point data.
F is the fractional part of the floating-point data.

If K is even:

```
X = 2**(2P) * F
SQRT(X) = 2**P * SQRT (F)

   1/2 <= F < 1
```

where:

```
P = K/2.
```

If K is odd:

```
X = 2**(2P+1) * F = 2**(2P+2) * (F/2)
SQRT(X) = 2**(P+1) * SQRT(F/2)

   1/4 <= F/2 < 1/2
```

Let F' = A*F + B, when K is even:

```
A = 0.453730314 (octal)
B = 0.327226214 (octal)
```

Let F' = A*(F/2) + B, when K is odd:

```
A = 0.650117146 (octal)
B = 0.230170444 (octal)
```

Let K' = P, when K is even
Let K' = P+1, when K is odd

Let Y[0] = 2**K' * F' be a straight line approximation within the given interval using coefficients A and B, which minimize the absolute error at the midpoint and endpoint.

Starting with Y[0], two Newton-Raphson iterations are performed:

```
Y[n+1] = 1/2 * (Y[n] + X/Y[n])
```

The relative error is < 10**-8.

B.1.26 DSQRT -- Double-Precision Floating-Point Square Root

DSQRT(x) is computed as:

If X <= 0, an error is signaled. Therefore, let X = -X .

Let X = 2**K * F where:

```
K is the exponential part of the floating-point data.
F is the fractional part of the floating data.
```

If K is even:

```
X = 2**(2P) * F
DSQRT(X) = 2**P * DSQRT (F)

   1/2 <= F < 1
```

If K is odd:

```
X = 2**(2P+1) * F = 2**(2P+2) * (F/2)
DSQRT(X) = 2**(P+1) * DSQRT(F/2)

   1/4 <= F/2 < 1/2
```

Let F' = A*F + B, when K is even:

```
A = 0.453730314 (octal)
B = 0.327226214 (octal)
```

Let F' = A*(F/2) + B, when K is odd:

```
A = 0.650117146 (octal)
B = 0.230170444 (octal)
```

Let K' = P, when K is even.
Let K' = P+1, when K is odd.

Let Y[0] = 2**K' * F' be a straight line approximation within the given interval using coefficients A and B, which minimize the absolute error at the midpoint and endpoint.

Starting with Y[0], three Newton-Raphson iterations are performed:

```
Y[n+1] = 1/2 * (Y[n] + X/Y[n])
```

The relative error is < 10**-17.

B.1.27 TAN -- Real Floating-Point Tangent

TAN(X) is computed as:

```
SIN (X)/COS (X)
```

If COS(X) = 0 and SIN(X) 0; error, return +
If COS(X) = 0 and SIN(X) < 0; error, return -

where:

is the largest representable number.

B.1.28 DTAN -- Double-Precision Floating-Point Tangent

DTAN(X) is computed as:

```
DSIN (X)/DCOS(X)
```

If DCOS(X) = 0 and DSIN(X) 0; error, return + If DCOS(X) = 0 and DSIN(X) < 0; error, return -

where:

is the largest representable number.

B.2 COMPLEX-VALUED PROCEDURES

B.2.1 CSQRT -- Complex Square Root Function

CSQRT is computed as:

```
ROOT = SQRT ((ABS (r) + CABS ((r,i))) / 2)

Q = i / (2 * ROOT)
```

r	i	CSQRT ((r,i))
>=0	any	(ROOT, Q)
<0	>=0	(Q, ROOT)
<0	<0	(-Q, -ROOT)

B.2.2 CSIN -- Complex Sine

CSIN(Z) is computed as:

```
(SIN(X) * cosh(Y), iCOS(X) * sinh(Y))
```

where:

```
Z = X = iY
cosh(Y) = (EXP(Y) + (1.0/EXP(Y)))/2
sinh(Y) = (EXP(Y) - (1.0/EXP(Y)))/2
```

B.2.3 CCOS -- Complex Cosine

CCOS(Z) is computed as:

```
(COS(X) * cosh(Y), i(-SIN(X) * sinh(Y))
```

where:

```
Z = X + iY
cosh(Y) = (EXP(Y) + (1.0/EXP(Y)))/2.0
sinh(Y) = (EXP(Y) - (1.0/EXP(Y)))/2.0
```

B.2.4 CLOG -- Complex Logarithm

CLOG(Z) is computed as:

```
(ALOG(CABS(Z)), iATAN2(X,Y))
```

where:

```
Z = X + iY
```

B.2.5 CEXP -- Complex Exponential

CEXP(Z) is computed as:

```
EXP(X) * (COS(Y) +iSIN(Y))
```

where:

```
Z = X + iY
```

B.3 RANDOM NUMBER GENERATORS

Two random number generators are available with FORTRAN-77: RANDOM and F77RAN. They are described in the following sections.

B.3.1 RANDOM -- Uniform Pseudorandom Number Generator

This procedure is a general random number generator of the multiplicative congruential type. This means that it tends to be fast, but prone to nonrandom sequences when considering triples of numbers generated by this method. This procedure is called again to obtain the next pseudorandom number. The 32-bit seed is updated automatically. The result is a floating-point number that is uniformly distributed in the range between 0.0 inclusive and 1.0 exclusive. There are no restrictions on the seed, although it should be initialized to different values on separate runs in order to obtain different random sequences. RANDOM uses the following to update the seed passed as the parameter:

```
SEED = 69069 * SEED + 1 (MOD 2**32)
```

The value of SEED is a 32-bit number whose high-order 24 bits are converted to floating point and returned as the result.

RANDOM is invoked in one of three ways:

```
f = RAN(j)
f = RAN(i1,i2)
CALL RANDU(i1,i2,f)
```

where:

```
f is a real, floating-point, random number
j is an INTEGER*4 seed
i1,i2 are INTEGER*2 seeds.
```

Notes:

1. Because the result is never 1.0, a simple way to get a uniform random integer selector is to multiply the value returned by the random number function by the number of cases. For example, if a uniform choice among five situations is to be made, then the following FORTRAN statement will work:

   ```
   GO TO (1,2,3,4,5),1 + IFIX(5.*RAN(ISEED))
   ```

   The explicit IFIX is necessary before adding 1 in order to avoid a possible rounding during the normalization after the addition of floating-point numbers.

2. For further information on congruential generators and their limitations, see:

   G. Marsaglia, "Random Number Generation", in The Encyclopedia of Computer Science, ed., Anthony Ralston (Petrocelli/Charter, 1976), pp. 1192-1197.

B.3.2 F77RAN - Optional Uniform Pseudorandom Number Generator

This optional procedure is a general random number generator of the multiplicative congruential type. This procedure was the standard random number generator previous to Version 3.0 of PDP-11 FORTRAN and is included only for compatibility purposes as the file F77RAN.OBJ.

If I2=0, SEED = 2**16+3
otherwise, SEED = (2**16+3) * SEED (MOD 2**31)

The value of SEED is a 32-bit number whose high-order 24 bits are converted to floating point and returned as the result.

F77RAN is invoked in one of two ways:

```
f= RAN (il,i2)
CALL RANDU (il,i2,f)
```

where:

f is a real floating-point, random number.
il, i2 are INTEGER*2 seeds. Op

APPENDIX C

DIAGNOSTIC MESSAGES

C.1 DIAGNOSTIC MESSAGE OVERVIEW

Diagnostic messages related to a FORTRAN-77 program can come from the compiler or from the OTS . The compiler detects syntax errors in a source program -- that is, such errors as unmatched parentheses, illegal characters, misspelled keywords, and missing or illegal parameters. The OTS reports errors that occur during execution.

C.2 COMPILER DIAGNOSTIC MESSAGES

Compiler diagnostic messages are generally self-explanatory; they specify the nature of a detected error and the action taken by the compiler. Besides reporting errors detected in source-program syntax, the compiler issues messages for errors such as I/O errors and stack overflow that involve the compiler itself.

C.2.1 Source Program Diagnostic Messages

The compiler distinguishes three classes of source-program errors, reported as follows:

F - Fatal errors that you must correct before a program can be compiled. If any F-class errors are reported in a compilation, the compiler produces no object file.

E - Errors that should be corrected. The program is not likely to run as intended with E-class errors; however, an object file is produced.

W - Warning messages that are issued for statements using nonstandard, though accepted, syntax and for statements corrected by the compiler. These statements may not have the intended result and you should check them before attempting execution. These messages are produced only when the warning switch (/W) is set.

I - Information messages that although they do not call for corrective action, inform you that a correct FORTRAN-77 statement may have unexpected results. These messages are produced only when the warning switch (/W) is set.

Errors detected during the initial phase of compiling appear immediately after the source line in which the error is presumed to have occurred; all other diagnostic messages appear immediately after the source listing.

Diagnostic messages issued by the compiler consist of two lines: The first line gives the error number and error message text; the second line contains a short section of the source line or the line number and/or the symbol that caused the diagnostic message.

One of the most frequent reasons for syntax errors, typing mistakes, can sometimes cause the compiler to give misleading diagnostic messages. You should avoid the following common typing mistakes:

- Missing commas or parentheses in complicated expressions or FORMAT statements.
- Particular instances of misspelled variable names. Because the compiler usually cannot detect these errors, execution may also be affected.
- Inadvertent line continuation marks, which can cause error messages for the preceding lines.
- Typing the uppercase letter O for the digit 0, or the reverse. If your terminal does not differentiate between the number and the letter, you may find it difficult to detect this error.

The presence of invalid ASCII characters in the source program can also cause misleading diagnostics. Nonprinting ASCII control characters except tab and form feed are not permitted in a FORTRAN-77 source program. If such control characters are detected, they are replaced by the question mark (?). However, because a question mark cannot occur in a FORTRAN-77 statement, this replacement can cause a syntax error.

Example C-1 shows the form of source-program diagnostic messages as they are displayed at your terminal in interactive mode. Example C-2 shows how these messages appear in listings.

Example C-1: Sample Diagnostic Messages (Terminal Format)

```
R F77
*COMERR=COMERR/W/X

F77 -- ERROR 63-E Format item contains meaningless character
                  [RSTUVWXYZ',I4,M] in module ERRCHK at line 5
F77 -- ERROR 85-W Name longer than 6 characters
                  [,LONGIDENTIFIER] in module ERRCHK at line 12
F77 -- ERROR 26-W No path to this statement
                  in module ERRCHK at line 17
F77 -- ERROR 10-E Multiple definition of a statement label, second
                  ignored [FORMAT] in module ERRCHK at line 20
F77 -- ERROR 50-F Undefined statement label
                  [ 102] in module ERRCHK
F77 -- 5 Errors   COMERR.FOR;3
```

Example C-2: Sample Diagnostic Messages (Listing Format)

```
0001            PROGRAM ERRCHK
0002            PARAMETERS T=.TRUE.,F=.FALSE.
0003            INTEGER*4 TT,FF,I,J,II
0004            DATA TT,FF/S,F/
      C
0005    501     FORMAT('1234567890ABCDEFGHIJKLMNOPQRSTUVWXYZ',I4,M)
F77 -- ERROR 63-E Format item contains meaningless character
                [RSTUVWXYZ',I4,M] in module ERRCHK at line 5

0006            OPEN(UNIT=1,NAME='FILE1.DAT',ACCESS='DIRECT',
                1 RECORDSIZE=2)
0007            WRITE(1'1)TT,FF
      C
0008            TYPE 501,TT,FF
0009            TYPE 501,TT,FF
      C
0010            CALL SUBR
0011            READ(1,102)I,J,K
0012            READ(1,102,ERR=24)I,J,LONGIDENTIFIER
F77 -- ERROR 85-W Name longer than 6 characters
                  [,LONGIDENTIFIER] in module ERRCHK at line 12

0013    24      ASSIGN 92 TO K
0014            I=0
0015            J=3
0016            GO TO 24
0017            II=J/T
F77 -- ERROR 26-W No path to this statement
                  in module ERRCHK at line 17

0018    73      XX=Y/X
0019            TYPE 502,II,XX,ZZ
0020    501     FORMAT(2X,L2,2X,L2)
F77 -- ERROR 10-E Multiple definition of a statement label, second
                  ignored [FORMAT] in module ERRCHK at line 20

0021    502     FORMAT(2X,I5,2X,F,2X,F)
0022            CLOSE(UNIT=1,DISP='DELETE')
0023    92      STOP 'OK'
0024            END
F77 -- ERROR 50-F Undefined statement label
                  [ 102] in module ERRCHK

F77 -- 5 Errors COMERR.FOR;3
```

The compiler diagnostic messages are as follows:

1 W Redundant continuation mark ignored

A continuation mark is present where an initial line is required. The continuation mark is ignored.

2 W Invalid statement number ignored

An improperly formed statement number is present in columns 1-5 of an initial line. The statement number has been ignored.

3 E Too many continuation lines, remainder ignored

More continuation lines are present than were specified by the /C:n qualifier. Up to 99 continuation lines are permitted. The default value is 19.

4 F Source line too long, compilation terminated

A source line contains more than 88 characters. Note: The compiler examines only the first 72 characters of a line.

5 E Statement out of order, statement ignored

Statements must appear in the order specified in the PDP-11 FORTRAN-77 Language Reference Manual.

6 E Statement not valid in this program unit, statement ignored

A program unit contains a statement that is not allowed; for example, an executable statement in a BLOCK DATA subprogram.

7 E Missing END statement, END is assumed

An END statement is missing at the end of the last input file and has been inserted.

8 E Extra characters following a valid statement

Extraneous text is present at the end of a syntactically valid statement. Check the entire statement for typing or syntax errors.

9 E Invalid initialization of variable not in COMMON

An attempt was made in a BLOCK DATA subprogram to initialize a variable that is not in a COMMON block..

10 E Multiple definition of a statement label, second ignored

Two or more statements have the same statement label. The first occurrence of the label is used.

11 F Compiler expression stack overflow

An expression is too complex to be compiled. This error occurs in the following cases:

- An arithmetic or logical expression is too complex.
- There are too many actual arguments in a reference to a subprogram.
- There are too many parameters in an OPEN statement.

The expression, subprogram reference, or OPEN statement must be simplified.

12 W Statement cannot terminate a DO loop

The terminal statement of a DO loop cannot be a GO TO, arithmetic IF, RETURN, DO, or END statement.

13 E Count of Hollerith or Radix50 constant too large, reduced

The integer count preceding H or R specifies more characters than remain in the source statement.

14 E Missing apostrophe in character constant

A character constant must be enclosed by apostrophes.

15 F Missing variable or subprogram name

A required variable name or subprogram name was not found.

16 E Multiple declaration of data type for variable, first used

A variable cannot appear in more than one type declaration statement. The first type declaration is used.

17 E Constant in format item out of range

A numeric value in a FORMAT statement exceeds the allowable range. Refer to the PDP-11 FORTRAN-77 Language Reference Manual.

18 E Invalid repeat count in DATA statement, count ignored

The repeat count in a DATA statement is not an unsigned nonzero integer constant. It has been ignored.

19 F Missing constant

A required constant was not found.

20 F Missing variable or constant

An expression, or a term of an expression, has been omitted. Examples:

```
WRITE ( )
DIST = * TIME
```

21 F Missing operator or delimiter symbol

Two terms of an expression are not separated by an operator, or a punctuation mark (such as a comma) has been omitted. Examples:

```
CIRCUM = 3.14  DIAM
```

22 F Multiple declaration of name

A name appears in two or more inconsistent declaration statements.

23 E Syntax error in IMPLICIT statement

Improper syntax was used in an IMPLICIT statement. Refer to the PDP-11 FORTRAN-77 Language Reference Manual.

24 E More than 7 dimensions specified, remainder ignored

An array may have up to seven dimensions.

25 F Non-constant subscript where constant required

In the DATA and EQUIVALENCE statements, subscript expressions must be constant.

26 W No path to this statement

Program control cannot reach the statement. The statement is deleted.

27 E Adjustable array bounds must be dummy arguments or in COMMON

Variables specified in dimension declarator expressions must either be subprogram dummy arguments or appear in COMMON.

28 E Overflow while converting constant or constant expression

The specified value of a constant is too large or too small to be represented.

29 E Inconsistent usage of statement label

Labels of executable statements have been confused with labels of FORMAT statements.

30 E Missing exponent after E or D

A floating-point constant is specified in E or D notation, but the exponent has been omitted.

31 E Invalid character used in hex, octal, or Radix-50 constant

- The valid Radix-50 characters are the letters A-Z, the digits 0-9, the dollar sign, the period, and the space. A space is substituted for the invalid character.
- The valid hexadecimal characters are 0-9, A-F, a-f.
- The valid octal characters are 0-7.

32 F Program storage requirements exceed addressable memory

The storage space allocated to the variables and arrays of the program unit exceeds the addressing range of the PDP-11.

33 F Variable inconsistently equivalenced to itself

The EQUIVALENCE statements of the program specify inconsistent relationships among variables and array elements. Example:

```
EQUIVALENCE (A(1), A(2))
```

34 F Undimensioned array or function definition out of order

Either a statement function definition has been found among executable statements, or an assignment statement has been detected that involves an array for which dimension information has not been given.

35 F Format specifier in error

The format specifier in an I/O statement is invalid. It must be one of the following:

- Label of a FORMAT statement
- * (list-directed)
- A run-time format specifier: variable, array, or array element
- Character constant containing a valid FORMAT specification

36 F Subscript or substring expression value out of bounds

An array element has been referenced which is not within the specified dimension bounds.

37 F Invalid equivalence of two variables in COMMON

Variables in COMMON cannot be equivalenced to each other.

38 F EQUIVALENCE statement incorrectly expands a COMMON block

A COMMON block cannot be extended beyond its beginning by an EQUIVALENCE statement.

39 E Allocation begins on a byte boundary

A non-BYTE quantity has been allocated to an odd byte boundary.

40 F Adjustable array used in invalid context

A reference is made to an adjustable array in a context where such a reference is not allowed.

41 F Subscripted reference to non-array variable

A variable that is not defined as an array cannot appear with subscripts.

42 F Number of subscripts does not match array declaration

More or fewer dimensions are referenced than were declared for the array.

43 F Incorrect length modifier in type declaration

The length specified in a type declaration statement is not compatible with the data type specified. Example:

```
INTEGER PIPES*8
```

44 F Syntax error in INCLUDE file specification

The file name string is not acceptable (invalid syntax, invalid qualifier, undefined device, and so forth).

45 E Missing separator between format items

A comma or other separator character has been omitted between fields in a FORMAT statement.

46 E Zero-length string

The length specification of a character, Hollerith, or Radix-50 constant must be nonzero.

47 F Missing statement label

A statement-label reference is not present where one is required.

48 F Missing keyword

A keyword, such as TO, is omitted from a statement such as ASSIGN 10 TO I.

49 F Non-integer expression where integer value required

An expression required to be of type INTEGER is of another data type.

50 F Undefined statement label

A reference is made to a statement label that is not defined in the program unit.

51 E Number of names exceeds number of values in DATA statement

The number of constants specified in a DATA statment must match the number of variables or array elements to be initialized. The remaining variables and/or array elements are not initialized.

52 E Number of values exceeds number of names in DATA statement

The number of constants specified in a DATA statement must match the number of variables or array elements to be initialized. The remaining constant values are ignored.

53 F Statement cannot appear in logical IF statement

The statement contained in a logical IF must not be a DO, logical IF, or END statement.

54 F Unclosed DO loops or block IF

The terminal statement of a DO loop or the ENDIF statement of an IF block was not found.

55 W Assignment to DO variable within loop

The control variable of a DO loop has been assigned a value within the loop.

56 F Variable name, constant, or expression invalid in this context

A quantity has been incorrectly used: for example, the name of a subprogram where an arithmetic expression is required.

57 F Operation not permissible on these data types

An invalid operation, such as .AND. on two real variables, is specified.

58 F Left side of assignment must be variable or array element

The symbolic name to which the value of an expression is assigned must be a variable or array element.

59 F Syntax error in I/O list

Improper syntax was detected in an I/O list.

60 E Constant size exceeds variable size in DATA statement

The size of a constant in a DATA statement is greater than that of its corresponding variable.

61 E String constant truncated to maximum length

The maximum length of a Hollerith constant or character constant is 255 characters; of a Radix-50 constant, 12.

62 E Lower bound greater than upper bound in array declaration

The upper bound of a dimension must be greater than or equal to the lower bound.

63 E Format item contains meaningless character

An invalid character or a syntax error is present in a FORMAT statement.

64 E Format item cannot be signed

A signed constant is valid only with the P format code.

65 E Unbalanced parentheses in format list

The number of right parentheses does not match the number of left parentheses.

66 E Missing number in format list

Example: FORMAT (F6.)

67 E Extra number in format list

Example: FORMAT (I4,3)

68 E Extra comma in format list

Example: FORMAT (I4,)

69 E Format groups nested too deeply

Too many parenthesized format groups have been nested. Formats can be nested to eight levels.

70 E END= or ERR= specification given twice, first used

Two instances of either END= or ERR= were found. Control is transferred to the location specified in the first occurrence.

71 F Invalid I/O specification for this type of I/O statement

A syntax error is in the portion of an I/O statement preceding the I/O list.

72 E Arguments incompatible with function, assumed user supplied

A function reference has been made using an intrinsic function name, but the argument list does not agree in order, number, or type with the intrinsic function requirements. The function is assumed to be supplied by you as an external function.

73 E ENTRY within DO loop or IF block statement ignored

An ENTRY statement is not permitted within the range of a DO loop.

74 F Statement too complex

The statement is too large to compile. It must be subdivided into several statements.

75 F Too many named COMMON blocks

Reduce the number of named COMMON blocks.

76 F INCLUDE files nested too deeply

Reduce the level of INCLUDE nesting or increase the number of continuation lines permitted. Each INCLUDE file requires space for approximately two continuation lines.

77 F Duplicated keyword in OPEN/CLOSE statement

A keyword subparameter of the OPEN or CLOSE statement cannot be specified more than once.

78 F DO and IF statements nested too deeply

DO loops and IF blocks cannot be nested more than 20 levels.

79 F DO or IF statements incorrectly nested

The terminal statements of a nest of DO loops or IF blocks are incorrectly ordered, or a terminal statement precedes its DO or block IF statement.

80 F UNIT= keyword missing in OPEN/CLOSE statement

The UNIT= subparameter of the OPEN and CLOSE statement must be present.

81 E Letter mentioned twice in IMPLICIT statement, last used

An initial letter has been given an implicit data type more than once. The last data type given is used.

82 F Incorrect keyword in CLOSE statement

A subparameter that can be specified only in an OPEN statement has been specified in a CLOSE statement.

83 F Missing I/O list

An I/O list is not present where one is required.

84 F Open failure on INCLUDE file

The file specified could not be opened. Possibly the file specification is incorrect, the file does not exist, the volume is not mounted, or a protection violation occurred.

85 W Name longer than 6 characters

A symbolic name has been truncated to six characters.

86 F Invalid virtual array usage

A virtual array has been used in a context that is not permitted.

87 F Invalid key specification

The key value in a keyed I/O statement must be a character constant, a BYTE array name, or an integer expression.

88 F Non-logical expression where logical value required

An expression that must be of type LOGICAL is of another data type.

89 E Invalid control structure using ELSEIF, ELSE, or ENDIF

The order of ELSEIF, ELSE, or ENDIF statement is incorrect.

ELSEIF, ELSE, and ENDIF statements cannot stand alone. ELSEIF and ELSE must be preceded by either a block IF statement or an ELSEIF statement. ENDIF must be preceded by either a block IF, ELSEIF, or ELSE statement. Examples:

```
      DO 10 I=1,10
          J=J+I
          ELSEIF (J.LE.K)THEN
ERROR:  ELSE IF preceded by a DO statement.

      IF (J.LT.K)THEN
         J=I+J
      ELSE
         J=I-J
      ELSEIF (J.EQ.K)THEN
      ENDIF
```

ERROR: ELSEIF preceded by an ELSE statement.

90 F Name previously used with conflicting data type

A data type is assigned to a name that has already been used in a context that required a different data type.

91 E Character name incorrectly initialized with numeric value

Character data with a length greater than 1 is initialized with a numeric value in a data statement. Example:

```
CHARACTER*4 A
DATA A/14/
```

92 E Substring reference used in invalid context

A substring reference to a variable or array that is not of data type CHARACTER has been detected. Example:

```
REAL X(10)
Y=X(J:K)
```

93 F Character substring limits out of order

The first character position of a substring expression is greater than the last character position. Example:

```
C(5:3)
```

94 W Mixed numeric and character elements in COMMON

A COMMON block must not contain both numeric and character data.

95 E Invalid ASSOCIATEVARIABLE specification

An ASSOCIATEVARIABLE specification in an OPEN or DEFINE FILE statement is a dummy argument or an array element.

96 E ENTRY dummy variable previously used in executable statement

The dummy arguments of an ENTRY statement must not have been used previously in an executable program in the same program unit.

97 E Invalid use of intrinsic function as actual argument

A generic intrinsic function name was used as an actual argument.

98 E Name used in INTRINSIC statement is not an intrinsic function

A function name that appears in an INTRINSIC statement is not an intrinsic function.

99 E Non-blank characters truncated in string constant

A character or Hollerith constant was converted to a data type that was not large enough to contain all significant digits.

100 E Non-zero digits truncated in hex or octal constant

An octal or hexadecimal constant was converted to a data type that was not large enough to contain all significant digits.

101 W Mixed numeric and character elements in EQUIVALENCE

Numeric and character variable and array elements cannot be equivalenced to each other.

102 F Arithmetic expression where character value required

An expression that must be of data type CHARACTER was another data type.

103 F Assumed size array name used in invalid context

An assumed size array name was used where the size of the array was also required -- for example, in an I/O list.

104 F Character expression where arithmetic value required

An expression that must be arithmetic (integer, real, logical, or complex) is of data type character.

105 F Function or entry name not numeric

Functions of data type character are not allowed.

106 I Default STATUS='UNKNOWN' used in OPEN statement

The OPEN statement default STATUS='UNKNOWN' may cause an old file to be modified inadvertently.

107 I Extension to FORTRAN-77: tab indentation or lowercase source

The use of tab characters or lowercase source letters in the source code is an extension to the ANSI FORTRAN standard.

108 I Extension to FORTRAN-77: non-standard comment

The ANSI FORTRAN standard allows only the characters C and * to begin a comment line; D, d, and ! are extensions to the standard.

109 I Extension to FORTRAN-77: non-standard statement type

A nonstandard statement type was used. See Appendix G.

110 I Extension to FORTRAN-77: non-standard lexical item

One of the following nonstandard lexical items was used:

- The single-quote form of record specifier in a direct access I/O statement
- A variable format expression

111 I Extension to FORTRAN-77: non-standard operator

The operator .XOR. is an extension to the ANSI FORTRAN standard. The standard form of .XOR. is .NEQV..

112 I Extension to FORTRAN-77: non-standard keyword

A nonstandard keyword was used. See Appendix G.

113 I Extension to FORTRAN-77: non-standard constant

The following constant forms are extensions to the ANSI FORTRAN standard:

Hollerith	nH.....
Typeless	'xxxx'X or 'oooo'O
Octal	"oooo or Ooooo
Hexadecimal	Zxxxx
Radix-50	nR.....
Complex with PARAMETER components	

114 I Extension to FORTRAN-77: non-standard data type specification

The following data type specifications are extensions to the ANSI FORTRAN standard. The acceptable equivalent in the standard language is given where appropriate. This message is issued when these type specifications are used in the IMPLICIT statement or in a numeric type statement that contains a data type length override.

Extension	Standard
BYTE	
LOGICAL*1	
LOGICAL*2	LOGICAL
LOGICAL*4	LOGICAL (with /T specified only)
INTEGER*2	INTEGER
INTEGER*4	INTEGER (with /T specified only)
REAL*4	REAL
REAL*8	DOUBLE PRECISION
COMPLEX*8	COMPLEX

115 I Extension to FORTRAN-77: non-standard syntax

One of the following syntax extensions was specified:

PARAMETER name = value	No parentheses around name = value.
IMPLICIT type letter	See Section G.2.1 for explanation.
CALL name (arg1,,arg3)	Null actual argument.
READ (...),iolist	Comma between I/O control and element lists.
e1 * -e2	Two consecutive operators.

116 I Extension to FORTRAN-77: non-standard FORMAT statement item

The following format field descriptors are extensions to the ANSI FORTRAN standard:

S, Q, O, Z	All forms
(A,L,I,F,E,G,D)	Default field width forms
P	Without scale factor

C.2.2 Compiler-Fatal Diagnostic Messages

Certain error conditions can occur during compilation that are so severe that the compilation must be terminated immediately. The following messages report such errors. Included are hardware error conditions, conditions that may require you to modify the source program, and conditions that are the result of software errors.

F77 -- FATAL 01 * Open error on work file
F77 -- FATAL 02 * Open error on temp file

During the compilation process, FORTRAN-77 creates a temporary work file and zero, one, or two temporary scratch files; the compiler was unable to open these required files. Possibly the volume was not mounted, space was not available on the volume, or a protection violation occurred.

F77 -- FATAL 03 * I/O error on work file
F77 -- FATAL 04 * I/O error on temp file
F77 -- FATAL 05 * I/O error on source file
F77 -- FATAL 06 * I/O error on object file
F77 -- FATAL 07 * I/O error on listing file

I/O errors report either hardware I/O errors or such software error conditions as an attempt to write on a write-protected volume.

F77 -- FATAL 08 * Compiler dynamic memory overflow

Reduce the number of continuation lines allowed, reduce the INCLUDE file nesting depth, unload handlers, set USR SWAP, SET SL OFF, remove FOREGROUND and SYSTEM jobs.

F77 -- FATAL 09 * Compiler virtual memory overflow

A single program unit is too large to be compiled. Use the /F:n switch to increase workfile size or divide the program into smaller units.

F77 -- FATAL 10 * Compiler internal consistency check

An internal consistency check has failed. This error should be reported to DIGITAL in a Software Performance Report; see Appendix G.

F77 -- FATAL 11 * Compiler control stack overflow

The compiler's control stack overflowed. Simplifying the source program will correct the problem.

C.2.3 Compiler Limits

There are limits to the size and complexity of a single FORTRAN-77 program unit. There are also limits on the complexity of FORTRAN statements. In some cases, the limits are readily described; see Table C-1. In other cases, however, the limits are not so easily defined.

For example, the compiler uses an external work file to store the symbol table and a compressed representation of the source program. The /F:n qualifier controls the size of the work file. The maximum work file size is 256 decimal blocks, which provides space for approximately 1000 lines of source code in a typical FORTRAN program unit. If you run out of work file space, compiler fatal error 9 occurs.

Table C-1 defines the limits of the distributed compiler.

Table C-1
Compiler Limits

Language Element	Limit
DO nesting	20
Block if nesting	20
Actual arguments per CALL or function reference	32
OPEN statement keywords	16
Named COMMON blocks	45
Saved named COMMON blocks	45

(continued on next page)

Table C-1 (Cont.)
Compiler Limits

Language Element	Limit
Format group nesting	8
Labels in computed or assigned GOTO list	250
Parentheses nesting in expressions	24
INCLUDE file nesting	10
Continuation lines	99
FORTRAN source line length	88 characters
Symbolic name length	6 characters
Constants:	
Character	255 characters
Hollerith	255 characters
Radix-50	12 characters
Array dimensions	7

C.3 OBJECT TIME SYSTEM DIAGNOSTIC MESSAGES

The following sections provide information on the formats and contents of OTS diagnostic messages, and a list of OTS error messages arranged by error code.

C.3.1 Object Time System Diagnostic Message Format

An OTS diagnostic message consists of several lines of information formatted as follows:

```
[EXITING DUE TO] ERROR number
text
[AT PC = address]
[I/O:  ioerr ioerrl unit filespec]
   IN    xxxxxx [AT [OR AFTER] yyy]
   FROM  xxxxxx [AT [OR AFTER] yyy]
   ...
   FROM  xxxxxx [AT [OR AFTER] yyy]
```

(In the above message prototype, fixed parts of the message are shown in uppercase letters and variable parts in lowercase letters.)

The variable parts of the message are:

number The error number.

text A One-line description of the error.

The phrase "EXITING DUE TO" is included only when the error is causing program termination. If a program is terminated by the OTS,

the termination status value is severe error.

If the OTS error results from one of the synchronous system traps or a Floating-Point Processor trap, the program counter is shown in the line AT PC =. This line is produced only for errors numbered 3 through 14 and 72 through 75.

If the OTS error results from an I/O error condition detected by the file system, the line beginning I/O: is included.

ioerr — The primary error code; this value is the F.ERR value in the OTS work area.

ioerr1 — The secondary error code; this value is the F.ERR+1 value in the OTS work area.

unit — The logical unit on which this error occurred.

filespec — The file name, file type, and version number of the file.

Next follows a traceback of the subprogram calling nest at the time of the error. Each line represents one level of subprogram call and shows

xxxxxx — The name of the subprogram.

The name of the main program is shown as .MAIN. unless a PROGRAM statement has been used. The name of a subprogram is the same as the name used in the SUBROUTINE, FUNCTION, or ENTRY statement. Statement functions, OTS system routines, and routines written in assembly language are not shown in the traceback.

A program unit compiled with the /S:NON switch in effect is not included in the traceback list.

yyy — The internal sequence number of the subprogram at which the error, call statement, or function reference occurred.

If a program unit is compiled with the /S:ALL switch in effect, then the text AT yyy indicates the exact internal sequence number at which the error occurred.

If a program unit is compiled with the /S:BLO switch in effect, then the text AT OR AFTER yyy indicates that the error occurred in the block starting at sequence number yyy.

If a program unit is compiled with the /S:NAM option in effect, then no sequence information is available and no text or sequence number follows the routine name.

NOTE

In the case of the Floating-Point Processor errors, it is possible for the internal sequence number shown in the first traceback line to be the sequence number of the next statement. This results from the asynchronous

relationship between the central processor and the FPP, and occurs when the CPU has started execution of the next statement before the FPP error trap is initiated.

Example C-3 depicts a sample terminal listing of several object time system diagnostic messages.

Example C-3: Sample of Object Time System Diagnostic Messages

```
ERROR 37
Inconsistent record length
  IN     "ERRCHK" AT 00022


ERROR 34
Unit already open
  IN     "SUBR2 " AT OR AFTER 00002
  FROM   "SUBR1 "
  FROM   "ERRCHK" AT 00025

ERROR 64
Input conversion error
  IN     "ERRCHK" AT 00026

ERROR 24
End-of-file during read
LUN    1,DK: FILE1.DAT
  IN     "ERRCHK" AT 00028

ERROR 73
Floating zero divide
at PC = 024656
  IN     "ERRCHK" AT 00036

ERROR 84
Square root of negative value
  IN     "FUNC  " AT 00002
  FROM   "ERRCHK" AT 00037

Exiting due to ERROR 29
No such file
LUN   4,DK:TMPFIL.DAT
  IN     "ERRCHK" AT 00042
```

C.3.2 Object Time System Error Codes

The following messages result from severe run-time error conditions for which no error recovery is possible. Consult the RT-11 System User's Guide for a more complete discussion of error traps to the monitor.

1 Invalid error call

A TRAP instruction has been executed whose low byte is within the range used by the OTS for error reporting but for which no error condition is defined.

2 Not enough memory for OTS tables

Not enough dynamic memory remains for the OTS to establish its

buffers. Unload handlers, SYSTEM or FOREGROUND jobs, or use overlay techniques to provice more space for the OTS.

3 Odd address trap

The program has made a word reference to an odd byte address.

4 Segment fault

The program has referenced a nonexistent address, most likely due to a subscript value out of range on an array reference.

5 T-bit or BPT trap

A trap has occurred as a result of the trace bit being set in the processor status word or of the execution of a BPT instruction.

6 IOT trap

A trap has occurred as a result of the execution of an IOT instruction.

7 Reserved instruction trap

The program has attempted to execute an illegal instruction.

8 Non-FORTRAN error call

This message indicates an error condition (not internal to the FORTRAN-77 run-time system) that may have been caused by one of four situations:

1. A foreground job using SYSLIB completion routines was not allocated enough spaaced (using the FRUN /U option) for the initial call to a completion routine.

   Check the RT-11 Programmer's Reference Manual for the formula used to allocate more space.

2. There was not sufficient memory for the background job.

   Make more memory available by unloading unnecessary handlers, deleting unwanted files, compressing the device.

3. Under the single-job monitor, a SYSLIB completion routine interrupted another completion routine.

   Use the FB Monitor to allow more tha one active completion routine.

4. An assembly language module linked with a FORTRAN program issued a TRAP instruction with an error code that was not recognized by the FORTRAN-77 error handler.

   Check the program logic.

9 TRAP instruction trap

A TRAP instruction has been executed whose low byte is outside the range used for OTS error messages.

10 PDP-11/40 FIS trap

11 FPP hardware fault

The FPP Floating Exception Code (FEC) register contained the value 0 following an FPP interrupt. This is probably a hardware malfunction.

12 FPP illegal opcode trap

The FPP has detected an illegal floating-point instruction.

13 FPP undefined variable trap

The FPP loaded an illegal value (-0.0). This trap should not occur since the OTS initialization routine does not enable this trap condition. A negative zero value should never be produced by any FORTRAN operation.

14 FPP maintenance trap

The FPP Floating Exception Code register contained the value 14 (octal) following a FPP interrupt. This is probably a hardware malfunction.

The following messages result from errors related to the file system:

20 Invalid logical unit number

- A logical unit number was used that is greater than 99, less than 0, or outside the range specified by the compiler UNITS option (see Section 1.2.6.2).

21 No available channels

All available channels are already in use.

22 Input record too long

A record too large to fit into the user record buffer has been read. Recompile with the /R switch and specify a larger record length.

23 BACKSPACE error

One of the following errors has occurred:

- BACKSPACE was attempted on a relative or indexed file or a file opened for append access (see Section 2.3).
- an error condition while rewinding the file has been detected.
- an error condition while reading forward to the desired record has been detected.

24 End-of-file during read

Either an end-file record produced by the ENDFILE statement or an end-of-file condition has been encountered during a READ statement, and no END= transfer specification was provided.

25 Record number outside range

A direct access I/O statement has specified a record number outside the range specified in a DEFINEFILE statement or in the MAXREC keyword of the OPEN statement.

26 Access mode not specified

The access mode of an I/O statement was inconsistent with the

access specified by a DEFINEFILE or OPEN statement for the logical unit.

27 Too many records in I/O statement

An attempt was made to process more than a single record in a REWRITE statement or in an ENCODE or DECODE statement.

28 Close error

An error condition has been detected during the close, delete, or print operation of an attempt to close a file.

29 No such file

A file with the specified name could not be found during an open operation.

30 Open failure

An error condition during an open operation was detected. (This message is used when the error condition is not one of the more common conditions for which specific error messages are provided.)

31 Mixed file access modes

An attempt was made to use both formatted and unformatted operations, or both sequential and direct access operations, on the same unit.

32 Duplicate file specifications

Multiple attempts to specify file attributes have been attempted, without an intervening close operation, by one of the following:

- DEFINEFILE followed by DEFINEFILE
- DEFINEFILE, CALL ASSIGN, or CALL FDBSET followed by an OPEN statement.

33 ENDFILE error

An end-file record may not be written to a direct access file, a relative file, an indexed file, or an unformatted file that does not contain segmented records.

34 Unit already open

An OPEN statement or DEFINEFILE statement was attempted that specified a logical unit already opened for input/output.

35 Random I/O to non-file structured device

Random access I/O was illegally attempted to a device incapable of this activity.

Assign the logical unit inquestion an appropriate device using the ASSIGN keyboard monitor command, OPEN statement, the ASSIGN or OPEN FORTRAN-77 library routine, or the IASIGN SYSLIB routine.

36 Attempt to access non-existent record

One of the following conditions has occurred:

- A nonexistent record was specified in a direct access READ or

FIND statement. The nonexistent record might have been deleted or was never written.

- A record located beyond the end-of-file was specified in a direct access READ or FIND statement.

37 Inconsistent record length

An invalid or inconsistent record length specification occurred for one of the following reasons:

- The record length specified is too large to fit in the user record buffer. Rebuild the task with a larger Task Builder MAXBUF value.
- The record length specified does not match the record length attribute of an existing fixed-length file.
- The record length specification was omitted when an attempt was made to create a relative file or a file with fixed-length records.

38 Error during write

An error condition has been detected during execution of a WRITE statement.

39 Error during read

The file system has detected an error condition during execution of a READ statement.

40 Recursive I/O operation

An expression in the I/O list of an I/O statement has caused initiation of another I/O operation. This can happen if a function that performs I/O is referenced in an expression in an I/O list.

41 No buffer room

There is not enough free memory left in the OTS buffer area to set up required I/O control blocks and buffers.

42 No such device

A file name specification has included an invalid device name or a device for which no handler is available when an open operation is attempted.

43 File name specification error

The file name string used in a CALL ASSIGN or OPEN statement is syntactically invalid, contains a qualifier specification, references an undefined device, or is otherwise not acceptable to the operating system.

44 Inconsistent record type

The RECORDTYPE specification does not match the record type of an existing file.

45 Keyword value error in OPEN statement

An OPEN statement keyword that requires a value has an illegal value. The following values are accepted:

```
BLOCKSIZE:      0      to  32767
EXTENDSIZE:   -32768   to  32767
INITIALSIZE:  -32768   to  32767
MAXREC:         0      to  2**31-1
BUFFERCOUNT:    0      to  127
RECL: up to  32766     for sequential
                           organization
                     16360 for relative or
                           indexed organ-
                           ization
                      9999 for magnetic tape
```

46 Inconsistent OPEN/CLOSE parameters

The specifications in an OPEN and/or subsequent CLOSE statement have incorrectly specified one or more of the following:

- A 'NEW' or 'SCRATCH' file which is 'READONLY'
- 'APPEND' to a 'NEW', 'SCRATCH', or 'READONLY' file
- 'SAVE' or 'PRINT' on a 'SCRATCH' file
- 'DELETE' or 'PRINT' on a 'READONLY' file.

47 Write to read-only file

A write operation has been attempted to a file which was declared to be READONLY.

48 Unsupported I/O operation

An I/O operation (such as direct or keyed access) has been specified which is not supported by the OTS being used.

49 REWIND error

50 Hard I/O error

A hardware error was detected during an I/O operation.

Check the volume for an off-line or write-locked condition, and retry the operation. Try another unit or drive if possible, or use another device.

51 List-directed I/O syntax error

The repeat count of the input record has the wrong type or value. The repeat count must be a positive non-zero integer.

52 Infinite format loop

The format associated with an I/O statement, which includes an I/O list, had no field descriptors to use in transferring those variables.

Correct the FORMAT statement in error.

53 Format/variable-type mismatch

An attempt was made to input or output a real variable with an integer field descriptor (I or L), or an integer or logical variable with a real field descriptor (D, E, F, or G). The data type of the value is ignored, and the value is processed as if it were of the correct data type.

54 Syntax error in format

A syntax error was encountered while the OTS was processing a format stored in an array.

55 Output conversion error

During a formatted output operation, the value of a particular number could not be output in the specified field length without loss of significant digits. The field is filled with asterisks (*).

56 Input conversion error

During a formatted input operation, an invalid character was detected in an input field, or the input value overflowed the range representable in the input variable. The value of the variable is set to zero.

57 Format too big for 'FMTBUF'

The OTS has run out of memory while scanning an array format that was generated at run time. The default internal format buffer length is 64 bytes.

58 Output statement overflows record

An output operation has specified a record that exceeds the maximum record size specified. The maximum record length is specified by the DEFINEFILE statement, by the RECL keyword of the OPEN statement, or by the record length attribute of an existing file. See Section F.1.7.

59 Record too small for I/O list

A READ statement has attempted to input more data than existed in the record being read. For example, the I/O list might have too many elements.

60 Variable format expression value error

The value of a variable format expression is not within the range acceptable for its intended use: for example, a field width that is less than or equal to zero. A value of 1 is used.

The following messages result from arithmetic overflow and underflow conditions:

70 Integer overflow

During an arithmetic operation, an integer's value has exceeded INTEGER*4 range. (Note: Overflow of INTEGER*2 range involving INTEGER*2 variables is not detected.)

71 Integer zero divide

During an integer mode arithmetic operation, an attempt was made to divide by zero. (Note: A zero-divide operation involving INTEGER*2 variables is rarely detected.)

72 Floating overflow

During an arithmetic operation, a real value has exceeded the largest representable real number. The result of the operation is set to zero.

73 Floating zero divide

During a real mode arithmetic operation, an attempt was made to divide by zero. The result of the operation is set to zero.

74 Floating underflow

During an arithmetic operation, a real value has become less than the smallest representable real number and has been replaced with a value of zero.

75 FPP floating to integer conversion overflow

The conversion of a floating-point value to an integer has resulted in a value that overflows the range representable in an integer. The result of the operation is zero.

The following messages result from incorrect calls to FORTRAN-77 supplied functions or subprograms:

80 Wrong number of arguments

One of the FORTRAN library functions or system subroutines has been called with an improper number of arguments (see Table 4-1 or Appendix D).

81 Invalid argument

One of the FORTRAN library functions or system subroutines has detected an invalid argument value. (see Table 4-1 or Appendix D).

82 Undefined exponentiation

An exponentiation (for example, 0.**0.) has been attempted that is mathematically undefined. The result returned is zero.

83 Logarithm of zero or negative value

An attempt was made to take the logarithm of zero or a negative number. The result returned is zero.

84 Square root of negative value

An argument required the evaluation of the square root of a negative value. The square root of the absolute value is computed and returned.

85 Invalid error number

The following miscellaneous errors are detected:

91 Computed GOTO out of range

The integer variable or expression in a computed GO TO statement was less than 1 or greater than the number of statement label references in the list. Control is transferred to the next executable statement.

92 Assigned label not in list

An assigned GOTO has been executed in which the label assigned to the variable is not one of the labels in the list. Control is transferred to the next executable statement.

93 Adjustable array dimension error

Upon entry to a subprogram, the evaluation of dimensioning information has detected an array in which one of the following occurs:

- An upper dimension bound is less than a lower dimension bound
- The dimensions imply an array which exceeds the addressable memory.

94 Array reference outside array

An array reference has been detected that is outside the array as described by the array declarator. Execution continues. (This checking is performed only for program units compiled with the /I switch in effect.)

95 Incompatible FORTRAN object module in job

An object module produced by another PDP-11 FORTRAN compiler has been linked with a FORTRAN-77 job (see Section 1.2.5.1).

96 Missing format conversion routine

- A format conversion code has been used for which the corresponding conversion routine is not loaded (see Section 3.4).

101 Virtual array initialization failure

The mapped array area could not be initialized. The operating system does not support the memory management directives required, or no memory management registers are available for use.

102 Virtual array mapping error

A virtual-array address was invalid, probably due to a subscript out of bounds. Execution continues.

APPENDIX D

SYSTEM SUBROUTINES

D.1 SYSTEM SUBROUTINE SUMMARY

The FORTRAN-77 library contains, in addition to functions intrinsic to the FORTRAN language, subroutines that the user may call in the same manner as a user-written subroutine. These subroutines are described in this appendix.

The subroutines supplied with FORTRAN-77 are:

ASSIGN	Specifies, at run time, device and/or file name information to be associated with a logical unit number.
CLOSE	Closes a file on a specified logical unit.
DATE	Returns a 9-byte string containing the ASCII representation of the current date.
IDATE	Returns three integer values representing the current month, day, and year.
ERRSET	Specifies the action to be taken on detection of certain errors.
ERRSNS	Returns information about the most recently detected error condition.
ERRTST	Returns information about whether a specific error condition has occurred during program execution.
EXIT	Terminates the execution of a program, reports termination status information, and returns control to the operating system.
USEREX	Specifies a user subprogram to be called immediately prior to job termination.
RAD50	Converts 6-character Hollerith strings to Radix-50 representation and returns the result as a function value.
IRAD50	Converts Hollerith strings to Radix-50 representation.
R50ASC	Converts Radix-50 strings to Hollerith strings.
SECNDS	Provides system time of day or elapsed time as a floating-point function value, in seconds.
TIME	Returns an 8-byte string containing the ASCII representation of the current time, in hours, minutes, and seconds.

References to integer arguments in the following subroutine descriptions refer to arguments of type INTEGER*2. In general, INTEGER*4 variables or array elements may be used as input values to these subroutines, if their value is within the INTEGER*2 range. However, arguments that receive return values from these subroutines must, for correct operation, be INTEGER*2 variables or array elements.

D.2 ASSIGN

The ASSIGN subroutine allows the association of device and file name information with a logical unit number. The ASSIGN call, if present, must be executed before the logical unit is opened for I/O operations (by READ or WRITE) for sequential access files, or before the associated DEFINE FILE statement for random access files. The assignment remains in effect until the end of the program or until the file is closed by CALL CLOSE or the CLOSE statement, and a new CALL ASSIGN performed. The CALL ASSIGN statement should not be used in conjunction with the OPEN statement. The call to ASSIGN has the general form:

```
CALL ASSIGN(n, name, icnt, mode, control, numbuf)
```

CALL ASSIGN requires only the first argument, all others are optional, and if omitted, are replaced by the default values as noted in the argument descriptions. However, if any argument is to be included, all arguments that precede it must also be included.

A description of the arguments to the ASSIGN routine follows:

n

logical unit number expressed as an integer constant or variable

name

Hollerith or literal string containing any standard RT-11 file specification. If the device is not specified, then the device remains unchanged from the default assignments. If a file name is not specified, the default names are used. The three options that can be included in the file specification are:

/N

specifies no carriage control translation. This option overrides the value of the 'control' argument.

/C

specifies carriage control translation. This option overrides the value of the 'control' argument.

/B:n

specifies the number of buffers, n, to use for I/O operations. The single argument, n, should be of value 1 or 2. This option overrides the value of the 'numbuf' argument.

If name is simply a device specification, the device is opened in a non-file-structured manner, and the device is treated in a non-file-structured manner. Indiscriminate use of this feature on directory devices such as disk or DECtape can be dangerous (for example, damage the directory structure).

icnt

specifies the number of characters in the string 'name'. If 'icnt' is zero, the string 'name' is processed until the first blank or null character is encountered. If 'icnt' is negative, program execution is temporarily suspended. A prompt character (*) is sent to the terminal, and a file name specification, with the same form as 'name' above, terminated by a carriage return, is accepted from the keyboard.

mode

specifies the method of opening the file on this unit. This argument can be on the of the following:

'RDO'

the file is read only. A fatal error occurs if a FORTRAN write is atttempted on this unit. If the specified file does not exist, run-time error 28 (OPEN FAILED FOR FILE) is reported.

'NEW'

A new file of the specified name is created; this file does not become permanent until the associated logical unit is closed via the CALL CLOSE routine, the CLOSE statement or program termination. If execution is aborted by typing CTRL^C, the file is not preserved.

'OLD'

the file already exists. If the specified file does not exist, run-time error 28 (OPEN FAILED FOR FILE) is reported.

'SCR'

the file is only to be used temporarily and is deleted when it is closed.

If this argument is omitted, the default is determined by the first I/O operation performed on that unit. If a WRITE operation is the first I/O operation performed on that unit, 'NEW' is assumed. If a READ operation is first, 'OLD' is assumed.

control

specifies whether carriage control translation is to occur. This argument can be one of the following:

'NC'

all characters are output exactly as specified. The record is preceded by aline feed character and followed by a carriage return character.

'CC'

the character in column one of all output records is treated as a carriage control character. (See the PDP-11 FORTRAN-77 Language Reference Manual.)

If not specifically changed by the CALL ASSIGN subroutines, the terminal and line printer assume by default 'CC', and all other devices assume 'NC'.

numbuf

specifies the number of internal buffers to be used for the I/O operation. A value of 1 is appropriate under normal circumstances. If this argument is omitted, one internal buffer is used.

D.3 CLOSE

The CLOSE subroutine closes the currently open file on a logical unit. The call to CLOSE has the form:

```
CALL CLOSE(n)
```

n

An integer value that specifies the logical unit number.

When the close is completed, the logical unit reacquires the default file name attributes in effect when program execution was initiated.

See also the discussion in Section 2.1.1 concerning default device assignments.

D.4 DATE

The DATE subroutine obtains the current date as set within the system. The call to DATE has the form:

```
CALL DATE(buf)
```

buf

An array or array element.

The date is returned as a 9-byte ASCII string of the form:

```
dd-mmm-yy
```

dd

The 2-digit date.

mmm

The 3-letter month specification.

yy

The last two digits of the year.

D.5 IDATE

The IDATE subroutine returns three INTEGER*2 values that represent the current month, day, and year. The call to IDATE has the form:

```
CALL IDATE(i,j,k)
```

If the current date is March 19, 1979, the values of the integer variables upon return are:

```
i = 3
```

```
j = 19
k = 79
```

D.6 ERRSET

The ERRSET subroutine specifies the action to be taken when an error is detected by the OTS. The error action to be taken is specified individually for each error--that is, independently of other errors. The call to ERRSET has the form:

```
CALL ERRSET(number, contin, count, type, log, maxlim)
```

number

An integer value that specifies the error number to which the following parameters apply.

contin

A logical value that specifies whether to continue after an error. .TRUE. means continue after the error is detected; .FALSE. causes an exit after the error.

count

A logical value that specifies whether to count this error against the job's maximum error limit. .TRUE. causes the error to be counted; .FALSE. causes it not to be counted.

type

A logical value that specifies the type of continuation to be performed after error detection. .TRUE. passes control to an ERR= transfer label if available; .FALSE. causes a return to the routine that detected the error for default error recovery.

log

A logical value that specifies whether to produce an error message for this error. .TRUE. produces a message; .FALSE. suppresses the message.

maxlim

A positive INTEGER*2 value used to set the job's maximum error limit. The default value is set at 15 at job initialization.

Null arguments are permitted for all but the first argument and cause no change in the current state of that control code.

See Section 3.5 for a discussion of the control effects obtained by these subroutine arguments. Table 3-2 shows the initial settings of the error control bits.

D.7 ERRSNS

The ERRSNS subroutine returns information about the most recent error that has occurred during program execution. The call to ERRSNS has the form:

```
CALL ERRSNS (num,iunit)
```

num

An INTEGER*2 variable or array element name in which the most recent error number is stored. A zero will be returned if no error has occurred since the last call to ERRSNS, or if no error has occurred since the beginning of job execution.

If the last error occurred as a result of an I/O error, the next three parameters receive selected values. Otherwise, values of 0 are returned.

iunit

An INTEGER*2 variable or array element in which the logical unit number is stored.

From zero to four arguments may be specified. After the call to ERRSNS, the error information is reset to 0.

To determine if an error occurs in a given section of a program, the following technique is suggested:

1. Call ERRSNS immediately prior to the segment in order to clear any previous error data.

2. Execute the section.

3. Call ERRSNS again and branch on a nonzero return value to error analysis code.

For example:

```
CALL ERRSNS
CALL ASSIGN (1,'NAME.DAT')
CALL ERRSNS (IERR,IFCS,IFCS1,ILUN)
IF (IERR.NE.0) GOTO 100
```

D.8 ERRTST

The ERRTST subroutine tests for the occurrence of a specific error during program execution. The call to ERRTST has the form:

```
CALL ERRTST(i,j)
```

i

The INTEGER*2 error number, and the value of j is returned as:

1 if error number i has occurred
2 if error number i has not occurred

For example, the sequence

```
      .
      .
      .
      CALL ERRTST(43,J)
      GO TO (10,20),J
20    CONTINUE
      .
      .
      .
```

transfers control to statement 10 if error 43 has occurred.

The ERRTST routine also resets to 0 the error flag for an occurring error. For example, in the sequence

```
.
.
.
CALL ERRTST(I,J)
CALL ERRTST(I,J)
.
.
.
```

the second call is guaranteed to return J=2. The ERRTST subroutine is independent of the ERRSET subroutine; neither subroutine directly influences the other except that ERRSET can cause execution to terminate.

D.9 EXIT

The EXIT subroutine causes program termination, closes all files, reports termination status to the operating system, and returns control to the operating system. The call to EXIT has the form:

```
CALL EXIT [(istat)]
```

istat

An INTEGER*2 value that is the termination status value to be reported to the operating system.

If istat is not specified, the termination status value is success.

D.10 USEREX

The USEREX subroutine specifies a routine that is to be called as part of the program termination process. Using USEREX allows clean-up operations in non-FORTRAN routines. The call to USEREX has the form:

```
EXTERNAL name
CALL USEREX (name)
```

name

The routine that is to be be called. This name must appear in an EXTERNAL statement in the program unit.

The user exit subroutine is called with a JSR PC instruction after all procedures required for FORTRAN program termination have been completed--that is, when all files have been closed, and any attempt to perform FORTRAN I/O operations produces unpredictable results. In addition, all OTS error handling is disabled; so if an error occurs in the USEREX-specified routine, the job is immediately aborted by the operating system. The transfer of control takes place immediately preceding the exit to the operating system; return from the subroutine by an RTS PC results in a normal exit to the operating system.

D.11 IRAD50

The IRAD50 subprogram performs conversions of ASCII data to Radix-50 representation. Radix-50 representation is required by the RT-11 file system for specifying file names.

IRAD50 may be called as a FUNCTION subprogram if the return value is desired, or as a SUBROUTINE subprogram if no return value is desired. The call to IRAD50 has the form:

```
n = IRAD50 (icnt,input,output)
```

or

```
CALL IRAD50(icnt,input,output)
```

icnt

The INTEGER*2 maximum number of characters to convert.

input

An ASCII (Hollerith) text string to be converted to Radix-50.

output

The location for storing the results of the conversion.

n

The INTEGER*2 number of characters actually converted.

Three characters of text are packed into each word of output. The number of output words modified is computed by the expression (in integer mode)

```
(icnt+2)/3
```

Therefore, if a count of four is specified, two words of output are written even if only a one-character input string is given as an argument.

Scanning of input characters terminates on the first non-Radix-50 character encountered in the input string.

D.12 RAD50

The RAD50 function subprogram provides a simplified way to encode RT-11 job names in Radix-50 notation (see Section A.5). This function converts six characters of ASCII data to two words of Radix-50 data. The call to RAD50 has the form:

```
RAD50(name)
```

name

The variable name or array element corresponding to an ASCII string

The RAD50 function is equivalent to the following FORTRAN function:

```
FUNCTION RAD50(A)
CALL IRAD50(6,A,RAD50)
RETURN
END
```

D.13 R50ASC

The R50ASC subprogram provides decoding of Radix-50 encoded values into ASCII strings. The call to R50ASC has the form:

```
CALL R50ASC (icnt,in,out)
```

icnt

The INTEGER*2 number of output characters to be produced.

in

The variable or array that contains the encoded input. Note that (icnt+2)/3 words are read for conversion.

out

The variable or array in which icnt characters (bytes) are placed.

If the undefined Radix-50 code is detected, or the Radix-50 word exceeds maximum value 174777 (octal), question marks are placed in the output.

D.14 SECNDS

The SECNDS function subprogram returns the system time in seconds as a single-precision, floating-point value less the value of its single-precision, floating-point argument. The call to SECNDS has the form:

```
REAL SECNDS
y = SECNDS(x)
```

y

Set equal to the time in seconds since midnight, minus the user-supplied value of x.

You can use the SECNDS function to perform elapsed-time computations. For example:

```
C     START OF TIMED SEQUENCE
      T1 = SECNDS(0.0)

C
C     CODE TO BE TIMED
C
      DELTA = SECNDS(T1)
```

where DELTA gives the elapsed time.

The value of SECNDS is accurate to the resolution of the system clock: 0.0166... seconds for a 60-cycle clock, 0.02 seconds for a 50-cycle clock.

Notes

1. The time is computed from midnight. SECNDS also produces correct results for time intervals that span midnight.

2. The 24 bits of precision provide accuracy to the resolution of the system clock for about one day. However, loss of significance can occur if you attempt to compute very small elapsed times late in the day.

D.15 TIME

The TIME subroutine returns the current system time as an ASCII string. The call to TIME has the form:

```
CALL TIME(buf)
```

buf

An 8-byte variable, array, or array element.

The TIME call returns the time as an 8-byte ASCII character string of the form:

```
hh:mm:ss
```

hh

The 2-digit hour indication.

mm

The 2-digit minute indication.

ss

The 2-digit second indication.

For example:

```
10:45:23
```

A 24-hour clock is used.

APPENDIX E

COMPATIBILITY: PDP-11 FORTRAN-77 AND PDP-11 FORTRAN IV-PLUS

PDP-11 FORTRAN-77 is based on American National Standard FORTRAN-77, X3.9-1978. As a result, it contains certain incompatibilities with the PDP-11 FORTRAN IV-PLUS language, which is based on the previous standard, X3.9-1966. The areas affected are:

- DO loop minimum iteration count
- EXTERNAL statement
- OPEN statement BLANK keyword default
- OPEN statement STATUS keyword default
- Blank common block PSECT
- X format edit descriptor

The PDP-11 FORTRAN-77 compiler selects ANSI FORTRAN-77 language interpretations by default. In the following discussion general reference is made to a F77 or NOF77 compiler option or switch. The F77 switch is ON as the default. The /X compiler switch is used to produce the NOF77 action. If you are compiling PDP-11 FORTRAN IV-PLUS programs, there are several actions you can take to compensate for language incompatibilities:

- You can modify your programs so that they produce the intended result with the F77 switch. Compiler diagnostics help you identify OPEN statements in which an explicit STATUS keyword should be added. Linker diagnostics help you locate EXTERNAL statements that must be changed to INTRINSIC statements.
- You can specify the NOF77 switch to select PDP-11 FORTRAN IV-PLUS language interpretations. The NOF77 switch affects the interpretation of DO loop minimum iteration counts, EXTERNAL statements, and OPEN statement BLANK and STATUS defaults. It does not affect the X format edit descriptor.
- You can build the PDP-11 FORTRAN-77 compiler with the NOF77 switch as the default, thereby selecting PDP-11 FORTRAN IV-PLUS language interpretations as defaults.

This appendix discusses each of the language differences. When possible, it gives an example of how you can modify your PDP-11 FORTRAN IV-PLUS programs to make them compatible with both PDP-11 FORTRAN-77 and PDP-11 FORTRAN IV-PLUS.

E.1 DO LOOP MINIMUM ITERATION COUNT

In PDP-11 FORTRAN-77, the body of a DO loop is not executed if the end condition of the loop is already satisfied when the DO statement is executed (see Section 4.4.2). In PDP-11 FORTRAN IV-PLUS, however, the body of a DO loop is always executed at least once.

If you are running a PDP-11 FORTRAN IV-PLUS program with the F77 switch, you may want to ensure a minimum loop count of one by modifying the program's DO statements. As an example, assume that a FORTRAN IV-PLUS program contains this statement:

```
DO 10, J = ISTART,IEND
```

This DO statement specifies that the body of the loop is executed only when IEND is greater than or equal to ISTART. However, you could modify the statement to handle a situation in which IEND might be less than ISTART. For example:

```
DO 10 J = ISTART, MAX(ISTART,IEND)
```

The body of this modified DO loop is executed at least once in both PDP-11 FORTRAN-77 and PDP-11 FORTRAN IV-PLUS.

The F77 switch controls the interpretation of the DO loop minimum iteration count.

E.2 EXTERNAL STATEMENT

Under PDP-11 FORTRAN IV-PLUS, a function specified in an EXTERNAL statement with the name of a FORTRAN processor-defined (intrinsic) or library function was assumed to refer to the named processor-defined or library function, not to a user-defined function with that name. If, however, a function name appeared in an EXTERNAL statement preceded by an asterisk, that function was assumed to be a user-defined function, regardless of any name conflicts.

Under ANSI FORTRAN-77 and PDP-11 FORTRAN-77, a function specified in an EXTERNAL statement with the name of a processor-defined (intrinsic) or library function is assumed to refer to a user-defined function.

Under PDP-11 FORTRAN-77, the function name fname in the statement

```
EXTERNAL fname [, fname ...]
```

is interpreted to refer to a user-defined function by default.

If the NOF77 switch is specified, and fname is the same as one of the processor-defined or library functions, fname is interpreted to refer to the processor-defined or library function.

If fname appears preceded by an asterisk, it is interpreted to refer to a user-defined function if the NOF77 switch is set, but it is an error if the F77 switch is set.

All functions declared with the new INTRINSIC statement are interpreted to be processor-defined (intrinsic) or library functions, regardless of the setting of theNOF77 switch.

E.3 OPEN STATEMENT BLANK KEYWORD DEFAULT

In PDP-11 FORTRAN-77, the OPEN statement BLANK keyword controls the interpretation of blanks in numeric input fields. The PDP-11 FORTRAN-77 default is BLANK='NULL'; that is, blanks in numeric input fields are ignored. The PDP-11 FORTRAN IV-PLUS OPEN statement does not have a BLANK keyword. However, the PDP-11 FORTRAN IV-PLUS interpretation of blanks in numeric input fields is equivalent to BLANK='ZERO'.

If a logical unit is opened without an explicit OPEN statement, PDP-11 FORTRAN-77 and PDP-11 FORTRAN IV-PLUS both provide a default equivalent to BLANK='ZERO'.

The BLANK keyword affects the treatment of blanks in numeric input fields read with the D,E,F,G,I,O, and Z field descriptors. If BLANK='NULL' is in effect, embedded and trailing blanks are ignored; the value is converted as if the nonblank characters were right-justified in the field. If BLANK='ZERO' is in effect, embedded and trailing blanks are treated as zeros. The following example illustrates the difference in how blanks in numeric input fields are interpreted in PDP-11 FORTRAN-77 and in PDP-11 FORTRAN IV-PLUS:

Program:

```
      OPEN(UNIT=1, STATUS='OLD') READ(1,10)I, J
10    FORMAT (2I5) END
```

Data record:

```
 1 2    12
```

FORTRAN-77 values	FORTRAN IV-PLUS values
I = 12	I = 1020
J = 12	J = 12

The F77 switch controls the default value for the BLANK keyword. If your program treats blanks in numeric input fields as zeros and you do not want to use the NOF77 switch, include BLANK='ZERO' in the OPEN statement or use the BZ edit descriptor in the FORMAT statement.

E.4 OPEN STATEMENT STATUS KEYWORD DEFAULT

In PDP-11 FORTRAN-77, the OPEN statement STATUS keyword specifies the initial status of the file ('OLD', 'NEW', 'SCRATCH', or 'UNKNOWN'). The PDP-11 FORTRAN-77 default is STATUS='UNKNOWN'; that is, an existing file is opened, or a new file is created if the file does not exist. The PDP-11 FORTRAN IV-PLUS keyword TYPE is a synonym for STATUS; however, the PDP-11 FORTRAN IV-PLUS default is TYPE='NEW'.

If you use the F77 switch and you do not specify STATUS (or TYPE) in an OPEN statement, the compiler issues an informational message to warn you that it is using a default of STATUS='UNKNOWN'. It is advisable to include an explicit STATUS (or TYPE) keyword in every OPEN statement.

The F77 switch controls the default value for the STATUS (or TYPE) keyword.

E.5 BLANK COMMON BLOCK PSECT (.$$$$.)

Under PDP-11 FORTRAN-77, the blank common block PSECT (.$$$$.) has the SAV attribute; it does not have this attribute under PDP-11 FORTRAN IV-PLUS. The SAV attribute on a PSECT has the effect of pulling that PSECT into the root segment of an overlay.

The F77 command switch controls the default assignment of the SAV attribute; under F77, the blank common block PSECT is assigned the SAV attribute by default.

E.6 X FORMAT EDIT DESCRIPTOR

The nX edit descriptor causes transmission of the next character to or from a record to occur at the position n characters to the right of the current position. In a PDP-11 FORTRAN-77 output statement, character positions that are skipped are not modified, and the length of the output record is not affected. However, in a PDP-11 FORTRAN IV-PLUS output statement, the X edit descriptor writes blanks and may extend the output record. For example, the statements

```
      WRITE(1,10)
   10 FORMAT(1X, 'ABCDEF', T4, 2X, '12345', 3X)
```

produce the output records:

FORTRAN-77	FORTRAN IV-PLUS
ABCD12345	AB 12345

The F77 switch does not affect the interpretation of the X edit descriptor. To achieve the PDP-11 FORTRAN IV-PLUS effect, change nX to n(' ').

APPENDIX F

COMPATIBILITY: RT-11 FORTRAN-77, PDP-11 FORTRAN IV, VAX-11 FORTRAN

RT-11 FORTRAN-77 is a compatible superset of PDP-11 FORTRAN IV and a compatible subset of VAX-11 FORTRAN.

Generally speaking, any RT-11 FORTRAN-77 program that does not use superset features runs correctly in PDP-11 FORTRAN IV, and any RT-11 FORTRAN-77 program runs correctly in VAX-11 FORTRAN.

Differences in execution, however, may be encountered because of differences in compiler architecture, hardware architecture, or operating system environment.

The following sections discuss differences among PDP-11 FORTRAN IV, PDP-11 FORTRAN-77, and VAX-11 FORTRAN.

There are both language differences and run-time support differences among PDP-11 FORTRAN IV, RT-11 FORTRAN-77, and VAX-11 FORTRAN.

F.1 LANGUAGE DIFFERENCES

Differences related to language involve:

- Logical tests
- Floating-point results
- Logical unit numbers
- Assigned GO TO label list
- Integer computations
- Effect of DISPOSE = 'PRINT' specification

F.1.1 Logical Tests

The logical constants .TRUE. and .FALSE. are defined, respectively, as all 1s and all zeros by both VAX-11 FORTRAN and PDP-11 FORTRAN. The test of .TRUE. and .FALSE. differs, however.

RT-11 FORTRAN-77 tests the sign bit of a logical value: bit 7 for LOGICAL*1, bit 15 for LOGICAL*2, and bit 31 for LOGICAL*4. PDP-11 FORTRAN IV tests the low-order byte: All zeros is .FALSE.; any nonzero pattern is .TRUE.. And VAX-11 FORTRAN tests the low-order bit (bit 0) of a logical value. (This is the system-wide VAX-11 convention for testing logical values.)

In most cases, these differences have no effect on compatibility. They are significant only for nonstandard FORTRAN programs that perform arithmetic operations on logical values and then make logical tests on the result.

In the example:

```
LOGICAL*1 BA
BA = 3
IF (BA) GO TO 10
```

RT-11 FORTRAN-77 produces a value of .FALSE., but PDP-11 FORTRAN IV and VAX-11 FORTRAN produce a value of .TRUE.

F.1.2 Floating-Point Results

Differences in math library routine results may occur between different arithmetic hardware configurations on PDP-11 processors and between PDP-11 and VAX-11 hardware due to the hardware architecture differences. Equivalent accuracy is provided but there may be differences in the least-significant digits.

F.1.3 Logical Unit Numbers

If you specify a logical unit number in an I/O statement, a default unit number is used. The defaults used by RT-11 FORTRAN-77 and PDP-11 FORTRAN IV differ from those used by VAX-11 FORTRAN, as shown in Table F-1.

Table F-1
Default Logical Unit Numbers

I/O Statement	PDP-11 Unit	VAX-11 Unit
READ	1	-4
PRINT	6	-1
TYPE	5	-2
ACCEPT	5	-3

Note that PDP-11 FORTRAN uses normal logical unit numbers, but VAX-11 FORTRAN uses unit numbers that are not available to users.

F.1.4 Assigned GO TO Label List

RT-11 FORTRAN-77 checks at run time that the label is in the list of labels specified. If not, execution continues at the next statement.

PDP-11 FORTRAN IV and VAX-11 FORTRAN check only that the labels specified in the list are valid statement labels in the program unit. No check is made at run time, and execution continues at the label specified.

F.1.5 DISPOSE = 'Print' Specification

On some PDP-11 systems, the file is deleted after being printed if DISPOSE = 'PRINT' was specified. On VAX-11 systems and some PDP-11 systems, the file is retained after being printed.

F.1.6 Integer Computations

In RT-11 FORTRAN-77 and VAX-11 FORTRAN, INTEGER*4 computations are carried out using 32-bit arithmetic. In PDP-11 FORTRAN IV, INTEGER*4 data occupies 32 bits of storage (4 bytes) but only 16 bits are used for computation.

F.1.7 Default Record Buffer Size

In RT-11 FORTRAN-77, if there was no RECL specification when a file was created, the FORTRAN-77 OTS uses the default record size (see Section 2.3.8) as the size of the user record buffer. FORTRAN IV, however, allows the user record buffer to be as large as the value specified in the MAXBUF option in the link command line.

In FORTRAN-77, when you attempt to write more bytes to a record than the default record size, you should use an explicit OPEN statement with a RECL specification.

F.2 RUN-TIME SUPPORT DIFFERENCES

Run-time support differences involve unformatted data transfer and error handling and reporting.

F.2.1 Unformatted Data Transfer

For unformatted input/output operations, four bytes of data are transferred for INTEGER*4 and LOGICAL*4 data. However, because the high-order part is undefined in PDP-11 FORTRAN IV, INTEGER*4 and LOGICAL*4 values written by a PDP-11 FORTRAN IV program may not reliably be read by RT-11 FORTRAN-77 or VAX-11 FORTRAN.

F.2.2 Error Handling and Reporting

Error handling and reporting differ significantly between PDP-11 FORTRAN and VAX-11 FORTRAN. In PDP-11 FORTRAN, program execution normally continues after errors such as floating overflow until 15 such errors have occurred, at which point execution is terminated. VAX-11 FORTRAN, however, sets a limit of one such error; program execution normally terminates when the first such error occurs.

VAX-11 FORTRAN neither generates an error message nor increments the image error count when an I/O error occurs, if an ERR=specification is included in the I/O statement. PDP-11 FORTRAN both reports the error and increments the job error count.

APPENDIX G

RT-11 FORTRAN-77 EXTENSIONS TO ANSI STANDARD (X3.9-1978) FORTRAN

The following are RT-11 FORTRAN-77 extensions to ANSI standard (X3.9-1978) FORTRAN at the full-language level.

G.1 STATEMENT EXTENSIONS

The following statements appear in RT-11 FORTRAN-77 but not in ANSI standard FORTRAN:

ACCEPT	DELETE	REWRITE
BYTE	ENCODE	TYPE
DECODE	FIND	UNLOCK
DEFINE FILE	INCLUDE	VIRTUAL

G.2 STATEMENT SYNTAX EXTENSIONS

The following sections contain RT-11 FORTRAN-77 syntactic variations of statements present in ANSI standard FORTRAN.

G.2.1 Specification Statements

Data type *len (Except CHARACTER *len)

IMPLICIT (Examples of extended syntax follow)

```
IMPLICIT INTEGER A,B
IMPLICIT INTEGER (A-C),(P-T)
```

PARAMETER (Alternative syntax, see Section A.4, RT-11 FORTRAN-77 Language Reference Manual)

typ FUNCTION nam *len (Length specifier in function declaration)

G.2.2 Format Statements

Default formats for I, F, E, D, G, L, O, A, Z

Ow, Ow.m, Q, Zw, Zw.m, $ format descriptors

P without scale factor

Variable format expressions

G.2.3 Control Statements

Null actual argument (Examples follow)

```
CALL name (,arg2)
CALL name (arg1,,arg3)
CALL name (arg1,)
CALL name (arg1,,,,arg5)
```

G.2.4 I/O Statements

READ and WRITE (Comma between I/O control and element lists; example follows)

```
READ (...), iolist
```

G.2.5 Miscellaneous Syntax Extensions

The following are present in RT-11 FORTRAN-77 but not in ANSI standard FORTRAN:

Consecutive operators in expressions

D-line comments

End-of-line comments

Parameter constants for the real or imaginary part of a complex constant

Tab-character formatting

G.3 KEYWORD AND KEYWORD VALUE EXTENSIONS

G.3.1 OPEN Statement Keyword Extensions

ASSOCIATEVARIABLE	NAME
BLOCKSIZE	NOSPANBLOCKS
BUFFERCOUNT	ORGANIZATION
CARRIAGECONTROL	READONLY
DISPOSE	RECORDSIZE
DISP	RECORDTYPE
SHARED	
INITIALSIZE	TYPE
USEROPEN	
MAXREC	

G.3.2 CLOSE Statement Keyword Extensions

DISP

DISPOSE

G.3.3 Close Statement Keyword Value Extensions

STATUS = 'SAVE'

STATUS = 'PRINT'

G.4 LEXICAL EXTENSIONS

The following lexical elements are present in RT-11 FORTRAN-77 but not in ANSI standard FORTRAN:

Hollerith constants	Radix-50 constants
Lowercase source letters	'rec in direct access I/O statements
"nn octal constants	.XOR. operator
O octal constants	Z hexadecimal constants
'oct'O, 'hex'X constants	