# Available tables:



---

# Contents

(316 entries)

CONTENTS

EXAMPLES

FIGURES



TABLES

# Available tables:



---

# Contents

(887 entries)

CONTENTS

EXAMPLES

FIGURES



TABLES

# Available tables:



---

# Examples

(887 entries)

EXAMPLES

# Available tables:



---

# Figures

(887 entries)

FIGURES

# Available tables:



---

# Tables

(887 entries)

TABLES

# Available tables:



---

# Index

(887 entries)

INDEX

A



See also Bracket operators ([ ])

Copying files

function

- void
- functions
- Dereferencing

See also Pointers

DIGITAL Command Language

See also Command Languages

Directives

- #define
- #elif
- #else
- #endif
- #error
- # if
- #ifdef
-
- #include
- #line
- #module
- #pragma
- #pragma charset
- #pragma linkage

F



Files



Floating-point

L

M



Main function

Operating Systems

P

- declarations
- function prototypes
- main function
- rules governing
- Parameters in #define preprocessor macros

PDP-11 C language

- aggregates
- arrays
- character strings
- elements
- list of operators
- members
- scalars
- structures and unions
- PDP-11 C Run-Time Library (RTL)
- linking to
- portability concerns
- PDP11C$INCLUDE logical name
- Performance Issues
-
- PDP-11 C work file

Pointer

Portability concerns

- #endif
- #error
- 
- #ifdef
- #ifndef
- #include
- token substitution
- #line
- #linkage
- #list
- #module
- #pragma
- #pragma [no]standard
- #psect
- # undef
- Preprocessor substitutions
- Primary expressions

See also Expressions

- array reference
- function call
- lvalues
- parentheses

S

Storage



qualifiers

String data type



See also Arrays

[next] [previous] [contents]

---



**PDP-11 C**
**Guide to PDP-11 C**

**January 1992**
This guide describes how to create, link, and execute PDP-11 C programs. It contains information on PDP-11 C program development in the PDP-11 and VMS environments and cross-system portability concerns.

**Revision/Update Information:** This is a revised manual.
**Operating System and Version:** Micro /RSX Version 4.3 or higher
RSTS/E Version 10.0 or higher
RSX-11M (mapped) Version 4.6 or higher
RSX-11M-PLUS Version 4.3 or higher
RT-11 Version 5.5 or higher
VMS Version 5.4 or higher

**Software Version:** PDP-11 C Version 1.2

[next] [previous] [contents]

---

---


The Reader's Comment form at the end of this document requests your critical evaluation to assist in preparing future documentation.

This document is available on CDROM.

**Preface**

This guide combines reference information for the PDP-11 C programming language with information necessary for developing and debugging PDP-11 C programs on PDP-11 and VMS environments. The guide also includes information concerning the porting of C programs to and from PDP-11 and other environments, as well as the differences between PDP-11 C and other implementations of the C programming language. For additional information concerning porting programs to and from other operating systems, refer to the *PDP-11 C Run-Time Library Reference Manual* .

**Intended Audience**

This guide is intended for experienced programmers who need to learn PDP-11 C, or for users who need to know the difference between PDP-11 C and other implementations. Readers should be familiar with one high-level language, the DIGITAL Command Language (DCL) and their operating systems.

**Document Structure**

This guide has eight chapters, four appendixes, and a glossary. They are as follows:

- Chapter 1 explains how to edit, compile, link, and run a PDP-11 C program. It also describes how to use debugging aids.
- Chapter 2 explains program structure.
- Chapter 3 describes PDP-11 C statements.
- Chapter 4 discusses expressions and operators used in PDP-11 C.
- Chapter 5 explains data types and declarations.
- Chapter 6 describes storage classes and allocation.

- Chapter 7 explains preprocessor directives.
- Chapter 8 explains features of the PDP-11 C implementation.
- Appendix A lists PDP-11 C compiler messages.
- Appendix B describes PDP-11 C definition modules.
- Appendix C describes compiler and run-time internationalization.
- Appendix D provides a summary of all PDP-11 C language features.
- The Glossary provides an alphabetical listing of key terms used in this manual.

**Associated Documents**

You may find the following documents useful when programming in PDP-11 C:

- *PDP-11 C Installation Guide* -For system programmers who install the PDP-11 C software.
- *PDP-11 C Run-Time Library Reference Manual* -For programmers who wish to use the PDP-11 C Run-Time Library functions and who need additional information concerning porting programs to and from other operating systems.
- *RSX-11M/M-PLUS and Micro /RSX Task Builder Manual* -For programmers who need information about using the Task Builder on RSX systems.
- *RSTS/E Task Builder Reference Manual* -For programmers who need information about using the Task Builder on RSTS/E systems.
- *RT-11 System Utilities Manual* -For programmers who

need information about using the linker on RT-11.

.

*The C Programming Language*

**1**

-For those who need a more intensive tutorial than that provided in Chapter 2.

**Conventions**

---

**Convention Meaning**

---

---

---

xxx

---

---

The symbol

---

---

xxx

---

---

represents a single stroke of a key on a terminal. For example,

---

---

Tab

---

indicates that you should press the key labeled Tab.

Ctrl/ *x* The symbol Ctrl/ *x* , where letter *x* represents a terminal control character, is generated by holding down the Ctrl key while pressing the key of the specified terminal character.

**. . .** Horizontal ellipsis indicates that you can enter additional parameters, values or other information. A comma that precedes the ellipsis indicates that successive items must be separated by commas.

.
.
.

A vertical ellipsis indicates that not all the text of a program or program output is illustrated. Only relevant material is shown in the example.

[ ] Brackets usually indicate optional syntax. However, brackets that are part of directory names and brackets that are used to delimit the dimensions of a multidimensional array in PDP-11 C source code do not indicate optional syntax.

UPPERCASE WORDS Uppercase words and letters in syntax formats indicate that you enter the word or letter exactly as shown.

lowercase words Lowercase words or letters in syntax formats indicate that you substitute a word or value of your choice.

**boldface Boldface** type in interactive examples is used to show user input. **Boldface** type in the text identifies language keywords and the names of PDP-11 C Run-Time Library functions.

*italic Italic* type is used to identify variable names and the names of definition

modules.

sc-specifier ::=
**auto**
**static**
**extern**
**register**

In syntax definitions, items appearing on separate lines are mutually exclusive alternatives.

¡ A delta symbol is used in some contexts to indicate a single ASCII space character.

---

Unless otherwise stated, all commands are followed by pressing the Return key.

[next] [previous] [contents]

---

**1. Developing PDP-11 C Programs**
This chapter describes how to create, compile, link, and run a PDP-11 C program using DCL commands as well as alternative MCR and CCL commands where applicable.

The host operating systems are as follows:

- RSX-11M-PLUS
- RSX-11M (mapped)
- Micro /RSX
- RSTS/E (RSX RT)
- RT-11 (XM Monitor only)
- VMS

The target operating systems are as follows:

- RSX-11M-PLUS
- RSX-11M
- RSX-11S
- Micro /RSX
- RSTS/E (RSX RT)
- RSTS/E (RT-11 RT)
- RT-11
- VAX-11 RSX

## 1.1 DCL Commands for Program Development

This section briefly describes the Concise Command Language (CCL), DIGITAL Command Language (DCL), and Monitor Console Routine (MCR) commands used to create, compile, link, and run a PDP-11 C program. Figure 1-1 shows these commands. For a more detailed description of each command on the VMS, RSX, RSTS/E, and RT-11 operating systems, see the sections that follow.

The following example shows each of the commands shown in Figure 1-1 executed in sequence. For the specific compiler and linker command formats and qualifiers on your operating system, see the section on linking for that system.

```
$ edit ave.c
$ cc ave
$ link ave,lb:[1,1]cfpursx/library
$ run ave
```

Throughout this chapter, the PDP-11 C compile command will be *CC* in sections that are not referring to a specific operating system. However, note that different operating systems require different compile commands. Refer to Section 1.3 for the different command formats found on each specific operating system.

To create a PDP-11 C source program at DCL level, you must invoke a text editor. In the previous example, Digital's standard editing utility, EDT, is invoked to create the source program AVE.C. You can use the EDT editor on RSX, RSTS/E, and VMS systems. Other editors are available on specific operating systems. By convention, the file type for a PDP-11 C source program is the letter C.

When you compile your program using PDP-11 C, you do not have to specify the file type; by default, PDP-11 C searches for a file with a C file type.

If your source program compiles successfully, the PDP-

11 C compiler creates an object file with the file type OBJ. However, if the PDP-11 C compiler detects errors in your source program, the compiler displays each error on your screen and then returns to the operating system prompt. You can then reinvoke your text editor to correct the errors. Object files will be created if the error severity is either a warning level or an informational level. If the error severity is an error level, no object file will be created.

You can include command qualifiers when invoking the compiler. Command qualifiers cause the PDP-11 C compiler to perform additional actions. In the following example, the /LIST qualifier causes the PDP-11 C compiler to produce a listing file:
$ **cc/list ave**

For a complete list and explanation of all the command qualifiers supported by the PDP-11 C compiler, see Section 1.3.4.

Once your program has compiled successfully, you invoke the Task Builder (for RSTS/E or RSX target systems) or the RT-11 Linker (for RT-11 or RSTS/E target systems) to create an executable image file. The Task Builder and RT-11 Linker use the object file produced by PDP-11 C as input to produce an executable file.

You can specify command qualifiers with the DCL command LINK or with the MCR or CCL command TKB. For a list and explanation of the most commonly used command qualifiers available with the LINK or TKB commands, see Section 1.5.5.3.

Once the executable file has been created, you can run your program with the RUN command.

[next] [previous] [contents]

---

### 1.2 Creating a PDP-11 C Program

To create or modify a PDP-11 C program, you must invoke a text editor. The following table shows which editors are available for each operating system.

---

**System Editors**

---

RSX EDT
RSTS/E EDT
VMS EDT or VAXTPU
RT-11 KED

---

### 1.2.1 Using EDT

The Digital Editor (EDT) is an interactive general-purpose text editor that offers three editing modes: keypad, nokeypad, and line. With keypad mode, you issue commands by using the numeric keypad that appears on the right of your main keyboard. With nokeypad mode, you enter commands on a command line, which EDT processes when you press the Return key. With line mode, you issue commands at the line mode asterisk prompt (

*

). Line mode focuses on the line as the unit of text.

EDT is available for use on RSTS/E, RSX, and VMS systems. The editor available on RT-11 is KED, referenced in Section 1.2.3 of this guide.

Keypad mode and nokeypad mode continually display the contents of the file on your screen. When you begin your editing session, editing in line mode is the default. Unlike keypad and nokeypad mode, line mode displays only one line

of text on your screen.

Use the following syntax to invoke the EDT editor and create a file.

On **RSTS/E** :
Under DCL, enter:
edit [/qualifier **. . .** ] file-specification

When you are working under RSTS/E DCL, you have the option of using the CCL EDT command with its qualifiers and specifiers.

Under CCL, enter:
edt [[output_file_spec] [,journal-file]=jrn_file_spec] input_file_spec [,command-file] [/qualifier **. . .** ]

On **RSX** :
Under DCL, enter:
edit/edt [/qualifiers **. . .** ] input_file_spec

Under MCR, enter:
edt [[output_file_spec][,journal-file=jrn_file_spec]] input_file_spec[,com_ file_spec] [/qualifier **. . .** ]

On **VMS** :
Use the following command:
edit/edt file_spec

Use these keys to move between types of editing modes:

.
To change from line mode to keypad mode, enter the CHANGE command at the asterisk prompt.

.
To return to line mode from keypad mode, press Ctrl/Z.

.
To change from line mode to nokeypad mode, use the SET NOKEYPAD command, and then enter the CHANGE command at the asterisk prompt.

When you invoke EDT to create a file, a journal file is created

automatically. You can use this journal file to recover your edits if the system fails during an editing session. To recover your edits, use the EDIT/RECOVER command followed by the name of the file you were editing.

EDT provides an online help facility that you can access during an editing session.

- In line mode, use the HELP command. EDT displays general information on EDT as well as detailed information on both line mode editing and nokeypad mode editing.
- In keypad mode, press the HELP key or the PF2 key. EDT displays a keypad diagram on your screen and a list of keypad editing keys. For help on a specific editing key, press that key.

On VMS, you can define a global symbol for the EDIT /EDT command by placing a symbol definition in your LOGIN.COM file. For example:
$ EDT == "EDIT/EDT"

After this command line is executed, you can enter EDT at the DCL prompt followed by the name of the file you want to modify or create.

For more information on using the advanced features of EDT on VMS, see the *Guide to VMS Text Processing* . For more information on using the advanced features of EDT on RSTS/E and RSX, see the *EDT Editor Manual* .

### 1.2.2 Using VAXTPU

The VAX Text Processing Utility (VAXTPU) is a high-performance, programmable utility. VAXTPU provides the Extensible VAX Editor (EVE) editing interface. You can also create your own interfaces.

Like EDT, VAXTPU provides you with an online help facility that you can access during your editing session. When you invoke VAXTPU to create a file, a journal file is created

automatically. You can use this journal file to recover your edits if the system fails during an editing session. To recover your edits, use the EVE/RECOVER command.

Unlike EDT, VAXTPU provides multiple windows. This feature allows you to view two files on your screen at the same time. VAXTPU also provides you with other advanced features, such as two editing interfaces.

The following section describes how to use the EVE interface.

**The EVE Interface**

EVE is an interactive text editor that allows you to execute common editing functions using the EVE keypad, or to execute more advanced functions by typing commands on the EVE command line. The following command line invokes the EVE editor and creates the file, AVE.C:

$ **edit/tpu ave.c**

You can define a global symbol for the EDIT/TPU command by placing a symbol definition in your LOGIN.COM file. For example:

$ EVE == "EDIT/TPU"

After this command line is executed, you can enter EVE at the DCL prompt followed by the name of the file you want to modify or create.

VAXTPU uses a buffer, a temporary holding area, to manage the editing session. The contents of the edit session are shown in an area of the screen that is called a window. The [End of file] message defines the end of the workspace. It is only visible on the screen and is not interpreted. A highlighted status line, located at the bottom of the window, shows the buffer name, current mode (insert or overstrike), and the current direction (forward or reverse).

VAXTPU manages the buffers with commands that do the following:

.

List all of the buffers used in this edit session

- Delete a specified buffer
- Change the buffer displayed in the window
- Create a new buffer that contains the contents of a specified file
- Write the contents of a buffer to a specified file

The EVE editing interface allows you to view more than one window on your terminal screen at the same time. For example, you can edit the source code in one window and display the listing file in another window. To help you manage the windows, VAXTPU commands are available to do the following:

- Split the screen into more than one window
- Put the cursor in the next, previous or other window
- Restore the current window as a single, large window
- Enlarge or shrink the current window by a specified number of lines

For more information about windows, buffers, and the VAXTPU commands, access the online help utility for the EVE editor. Press the Do or PF4 key, or enter Ctrl/B to reveal the VAXTPU prompt and enter the HELP command. *Guide to VMS Text Processing* has more information on using the advanced features of EVE.

### 1.2.3 Using KED

The PDP-11 Keypad Editor (KED) is a program that you can use to create, inspect, and edit files. When you use the keypad editor, you control the different editing processes by using a set of functions and a set of commands.

The following command line invokes the editor and creates the file AVE.C:

. **edit ave.c**

The keypad editor provides a help function. When the keypad editor fails, it signals you by loading a one line explanation of the signal in an internal message buffer. When you use the help function, the editor temporarily erases the bottom three screen lines and displays the explanation.

For more information about KED, see the *PDP-11 Keypad Editor User's Guide* .

### 1.3 Compiling a PDP-11 C Program

The PDP-11 C compiler can compile any program conforming to the ANSI Standard for the C language. The PDP-11 C compiler is highly compatible with VAX C.

The PDP-11 C compiler performs the following functions:

- Detects errors in your source program
- Displays each error on your screen and writes the errors to the listing file (if selected)
- Generates machine language instructions from the source statements
- Groups these language instructions into an object module for the Task Builder or the RT-11 Linker

#### 1.3.1 The Compile Command

To invoke the PDP-11 C compiler, use the compile command. The compile command has the following format:
command[/qualifier . . . ] [file-spec [/qualifier . . . ]], . . .

The command used to invoke the compiler differs depending on the specific operating system and command line interpreter you are using. See the following sections for the specific command you will need.

---

**Note**

---

All user input is converted to uppercase unless enclosed by quotation marks.

---

**/qualifier**
Specifies an action to be performed by the compiler on all

files or specific files listed. When a qualifier appears directly after the compile command, it affects all the files listed. However, when a qualifier appears after a file specification in a comma-separated list, it affects only the file that immediately precedes it. When files are concatenated, the qualifier affects all the files in the concatenation.

**file-spec**
Specifies an input source file that contains the program or module to be compiled. You are not required to specify a file type if you have given your file a C file type; the PDP-11 C compiler adopts the default file type C.

You can include more than one file specification on the same command line by separating the file specifications with either a comma ( , ) or a plus sign ( + ). If you separate the file specifications with commas, you can control which source files are affected by each qualifier. Using the comma separator also causes the compiler to generate individual output files for each source file specified. In the following example, the PDP-11 C compiler creates an object file for each source file but creates only a listing file for the source files PROG1 and PROG3.
$ **cc /list prog1, prog2/nolist, prog3**

If you separate file specifications with plus signs, the PDP-11 C compiler concatenates each of the specified source files to form a compilation unit and creates one object file and one listing file. In the following example, only one object file is created, PROG3.OBJ, and only one listing file is created, PROG2.LST. Unlike VAX C, the names of default object and listing files are taken from the last source file in the list.
$ **cc prog1 + prog2/list + prog3**

Note that any qualifiers specified for a single file within a list of files separated with plus signs affect all the files in the list.

### 1.3.1.1 Compiling a Program on RSX Systems

On RSX systems, you can invoke the PDP-11 C compiler from either DCL or MCR. You can invoke the compiler from DCL by entering the following command line:
CC[/qualifier **. . .** ] [file_spec[/qualifier **. . .** ]], **. . .**

PDP-11 C is installed with the MCR task name CCC. You can invoke the compiler from MCR by entering the following command line:
CCC[/qualifier . . . ] [file_spec[/qualifier . . . ]], . . .

The following example produces an object module format that can be read by the RSX Task Builder:
$ **ccc prog1**

#### 1.3.1.2 Compiling a Program on RSTS/E Systems

You can invoke the compiler on RSTS/E using either the CCL CCC command or the DCL CC command.

To use CCL, enter:
CCC[/qualifier . . . ] [file_spec[/qualifier . . . ]], . . .

To use DCL, enter:
CC[/qualifier . . . ] [file_spec [/qualifier . . . ]], . . .

The following examples produce object module formats that can be read by the RSTS/E Task Builder:
$ **ccc prog1**
$ **cc prog1**

---

**Note**

---

The OBJ formats for RSX and RSTS/E are identical.

---

#### 1.3.1.3 Compiling a Program on RT-11 Systems

On RT-11 systems, you invoke the compiler by using the CC command:
CC [/qualifier . . . ] [file_spec[/qualifier . . . ]], . . .

The following example produces an object module format that can be linked by the RT-11 Linker on RT-11 or RSTS/E systems:
$ **cc /list prog1**

**Note**

On RT-11 systems, you must include at least a single space character between the CC command and the first qualifier or file specification.

See Section 1.3.4 for more details on command qualifiers and how to avoid using parentheses on the command line.

#### 1.3.1.4 Compiling a Program on VMS Systems

To invoke the PDP-11 C compiler on VMS systems, use the PDPCC command:
PDPCC [/qualifier . . . ] [file_spec[/qualifier . . . ]], . . .

The resulting object file will be in the correct format for the PDP-11 target systems.

The following example produces an object file (OBJ file type) that can be linked by the RSX Task Builder on either RSX systems or under the RSX emulator on RSTS/E systems:
$ **pdpcc/environment=(pic)/list prog2**

If you have the VAX-11 RSX emulator installed on your VMS system, you can also link this object file on VMS.

### 1.3.2 Prompt Mode

The PDP-11 C compiler supports a prompting mode that enables you to create an environment to compile one or more programs. In prompting mode, you can set the qualifiers once and until you reset the qualifiers or exit the prompting mode, those qualifiers will remain activated. This mode is invoked whenever the compiler is called without specifying any filespec.

The format of the command level interface is exactly the same as that of DCL. For example:
$ **cc**
CC> **prog1/list/environment=fpu**

To return to DCL, enter Ctrl/Z.

In this example, the file PROG1 is compiled, a listing file is created, and the compiler generates code using the Floating Point Processor.

To continue a line in prompting mode, use a hyphen ( - ). The command line processor treats this exactly like DCL does. Note that any prompt line that contains an input file specification and does not end in a hyphen will start a compilation. The next CC> prompt will be displayed only after that compilation has finished. To exit from prompt mode, enter Ctrl/Z. As with other PDP-11 layered products, any command line that is terminated by Ctrl/Z is not executed. Incomplete command lines consisting only of qualifiers may be used to establish defaults for the remainder of the compilation.

The following example illustrates prompting mode and the informational messages that are displayed.

```
$ cc
CC> /define=(check=1,debug=1)
CC> prog1
CC> prog2,prog3
CC> prog4+prog5+prog6-
_CC> ,prog7
```

To return to DCL, enter Ctrl/Z.

The previous example equates to the following DCL commands:

```
$ cc/define=(check=1,debug=1) prog1
$ cc/define=(check=1,debug=1) prog2,prog3
$ cc/define=(check=1,debug=1) prog4+prog5+prog6,prog7
```

### 1.3.3 Indirect Command Files

The compiler can receive input from an indirect command file. This file, which has a default file extension CMD, contains the same type of information as required by the prompting mode. The only difference is that there is no Ctrl/Z terminating the input. The compiler stops processing command lines when the end-of-file is reached.

You can invoke an indirect command file in one of two ways:

.
Specify the indirect command-file specification preceded by the at sign ( @ ) character. (Not available on RT-11 systems.)

.
Specify the indirect command file specification as a value to the /COMMAND qualifier.

You can use either method in the command line or in prompting mode.

---

**Note**

---

You cannot specify indirect files using the at sign ( @ ) character on RT-11 command lines; use the /COMMAND qualifier.

---

You may also use preceding qualifiers to establish defaults for the remainder of the compilations in the command file. Within an indirect command file, the exclamation point ( ! ) delimits a comment that extends from ! to the end of the line. Indirect command-file invocations may be nested. The maximum nesting depth is determined by available resources in the host environment.

The following example illustrates the use of PDP-11 C indirect command files:

```
$ type myccsetup.cmd
! Set up file for compilation under PDP-11 C
/DEFINE=("CHECK=1","DEBUG=1") ! Enable CHECK and DEBUG variants
/INCLUDE_DIRECTORY=PROJ$:[HEADERS]
/LIST
/ENVIRONMENT=(NOFPU,NOPIC)
$ type build.cmd
@MYCCSETUP
PROG1
PROG2,PROG3
PROG4+PROG5+PROG6-
```

,PROG7
$ **cc @build**

The effect of executing the previous command yields the
following results for the last two lines in the BUILD.CMD
command:

```
CC/DEFINE = ("CHECK=1", "DEBUG=1")/INCLUDE_DIRECTORY=PROJ$:[HEADERS]-
   /LIST/ENVIRONMENT=(NOFPU,NOPIC)-
   PROG1,PROG2,PROG3,PROG4+PROG5+PROG6,PROG7
```

### 1.3.4 The PDP-11 C Command Qualifiers

The following list shows all the command qualifiers and their
defaults. A description of each qualifier follows the list.

**Command Qualifiers Default**

/COMMAND=file-spec /COMMAND
/[NO]DEFINE[=(definition list)] /NODEFINE
/ENVIRONMENT=([NO]FPU, [NO]PIC) /ENVIRONMENT=(FPU,NOPIC)
/[NO]ERROR_LIMIT /ERROR_LIMIT=30
/[NO]INCLUDE_DIRECTORY=(pathname [, . . . ])
/NOINCLUDE_DIRECTORY
/[NO]LIST[=file-spec] /NOLIST
/[NO]MACRO /NOMACRO
/[NO]MEMORY /NOMEMORY
/[NO]MODULE /MODULE
/[NO]OBJECT[=file-spec] /OBJECT
/SHOW[=(option, . . . )] (See description for default values)
/[NO]STANDARD[=(option, . . . )] /NOSTANDARD
/[NO]TERMINAL /TERMINAL=NOSOURCE
/[NO]TITLE /NOTITLE
/[NO]UNDEFINE[=(undefine list)] /NOUNDEFINE
/[NO]WARNINGS[=(option, . . . )] /WARNINGS
/[NO]WORK_FILE_SIZE /NOWORK_FILE_SIZE

---

**Note**

---

Using [NO] before any qualifier prohibits specifying
any values for the qualifier.

---

You can place command qualifiers either on the command

line itself or on individual file specifications. If placed on a file specification, the qualifier affects only the compilation of the specified source file and all subsequent source files in the compilation unit. If placed on the command used to invoke PDP-11 C, the qualifier affects all source files in all compilation units unless it is overridden by a qualifier on an individual file specification.

Several command qualifiers accept a comma-separated list of values enclosed within parenthesis. However on RT-11 systems, RT-11 factors commands that contain parenthesis resulting in an incorrect or inappropriate PDP-11 C command line. Use left and right curly braces ({ }) in place of left and right parenthesis on RT-11 command lines.

The rest of this section describes the command qualifiers.

**/COMMAND=file-spec**
Specifies an indirect command file. You must specify a file-spec value. The default file type is CMD. Refer to Section 1.3.3 for more information on command files.

**/[NO]DEFINE=( " identifier[(param, . . . )] token-string " [, . . . ])**

**/[NO]UNDEFINE=( " identifier " [, . . . ])**
Performs, from the command line, the same functions performed by the **#define** and **#undef** preprocessor directives. The /DEFINE qualifier defines a token string or macro to be substituted for every occurrence of a given identifier in the compilation units; /UNDEFINE cancels a previous definition. When /DEFINE is specified multiple times for a compilation unit, only the last /DEFINE is effective; the same is true for the /UNDEFINE qualifier. When both /DEFINE and /UNDEFINE are specified for a compilation unit, /DEFINE is evaluated before /UNDEFINE.

Each string literal specified with the /DEFINE and /UNDEFINE qualifiers is processed as though the string (without quotes) was specified as the right-hand portion of the **#define** and **#undef** preprocessor directives, respectively. Thus, the following command-line qualifiers are equivalent to the preprocessor directives that follow them:

```
/DEFINE=("DUMP","CLEAR(x) (x=0)")
/UNDEFINE=("DEBUG","TEST")
# define DUMP
# define CLEAR(x) (x=0)
# undef DEBUG
# undef TEST
```

You can specify quotation marks within a macro definition by placing one quotation adjacent to another. For instance, the following command-line qualifier and preprocessor directive are equivalent:

```
/DEFINE="COMPLAIN (fprintf(stderr, ""Unrecognized option\n""))"
# define COMPLAIN (fprintf(stderr, "Unrecognized option\n"))
```

The /UNDEFINE qualifier is useful for undefining the predefined PDP-11 C preprocessor constants. For example, if you use a preprocessor constant (such as _ _pdp11c or _ _PDP11C) to conditionally compile segments of PDP-11 C code, you can undefine that constant to see how the portable sections of your program execute. Consider the following program:

```
#include <stdio.h>
int main()
{
#ifdef __PDP11C
printf("I'm being compiled with PDP-11 C.");
#else
printf("I'm being compiled on some other compiler.");
#endif
}
```

For example, on RSTS/E systems, output from the program is as follows:

```
$ cc exampl.c
$ link/cc exampl.obj
$ run exampl.tsk
I'm being compiled with PDP-11 C.
$ cc/undefine="__pdp11c" exampl
$ link/cc exampl.obj
$ run exampl.tsk
I'm being compiled on some other compiler.
```

Since /DEFINE and /UNDEFINE are not part of the source file, they are not associated with a listing line number or source line number. Therefore, when an error occurs in a command line definition, the message displayed at the terminal does not indicate a line number. In the listing file, these diagnostic messages are printed after the source listing in the order that they were encountered. When the expansion of a definition causes an error at a specific source line in the program, the diagnostics-both at the terminal and in the listing file-are associated with that source line.

A command line containing the /DEFINE and the /UNDEFINE qualifiers can be 256 characters long. Continuation characters cannot appear within quotes or they will be included in the token stream. The length of a command line cannot exceed the maximum length allowed by DCL.

The defaults are /NODEFINE and /NOUNDEFINE.

**/ENVIRONMENT=([NO]FPU, [NO]PIC)**
Specifies the type of environment in which the generated code is to execute. You can specify the following values: [NO]FPU, [NO]PIC. If you specify this qualifier, you must provide at least one value, or an error message will be generated.

If you specify or default to FPU, floating-point processor instructions will be generated as appropriate. If you specify NOFPU, floating-point processor instructions will not be generated.

Do not specify FPU if you are going to link against the Extended Instruction Set (EIS) run-time library. If any of the modules are compiled for FPU, you should link to the FPU run-time library. For more information, see Section 1.5.8.1.

If you specify PIC, PDP-11 C produces position-independent code. If you specify or default to NOPIC, code may be generated that is not position-independent. For information about position-independent code, see the discussion in the *PDP-11 MACRO-11 Language Reference Manual* .

The default is /ENVIRONMENT=(FPU,NOPIC)

**/[NO]ERROR_LIMIT[=value]**
Specifies the maximum error count. If the number of errors encountered (exclusive of informationals and warnings) exceeds the integer value specified, the compilation is aborted without further source code analysis. The default is /ERROR_LIMIT=30. If you specify the /NOERROR_LIMIT qualifier, compilation proceeds regardless of the number of errors encountered.

**/[NO]INCLUDE_DIRECTORY=(pathname [, . . . ])**
Provides an additional level of search for include files. Each pathname argument can be either a logical name or a legal directory specification.

The forms of inclusion affected are the # **include** `` file-spec" and # **include** <file-spec> forms. The quoted form is generally used with user-defined header files. The bracketed form is generally used with header files supplied with PDP-11 C.

The default is /NOINCLUDE_DIRECTORY.

**/[NO]LIST[=file-spec]**
Directs the compiler to produce a listing file. You must specify this qualifier to get any type of listing. See the /SHOW qualifier for an explanation of the options available for the contents of the listing file.

When /LIST is in effect, the compiler, by default, creates a listing file with the same name as the source file and with the LST file extension. If you include a file specification with the /LIST qualifier, the compiler uses that specification to name the listing file.

The default is /NOLIST.

**/[NO]MACRO[=file-spec]**
Specifies a MACRO-11 file specification. If you do not specify a file name, the default file name is used, which is the file name of the last file in the compilation unit. If you do not specify a file type, the type MAC is used. A legal MACRO-11

source program corresponding to the translation of the source program is placed in the specified file.

The /MACRO qualifier differs from the /SHOW=MACHINE_CODE qualifier as follows: the /MACRO qualifier places a MACRO-11 source program in a separate file rather than the listing file. The MACRO-11 source program may be assembled under MACRO-11 without modification. The /SHOW=MACHINE_CODE qualifier places a machine code listing similar to the list file output produced by MACRO-11 into the listing file.

The default is /NOMACRO.

**/[NO]MEMORY[=value]**
You use this qualifier to determine the amount (in 8Kb regions) of extended memory to allocate in a PDP-11 host environment. This qualifier is ignored in VMS host environments and on PDP-11 host systems that do not support the I- and D-space feature.

This switch only affects compiler performance. The specified integer value determines the number of 8192-byte regions that are to be allocated. You can specify an integer between 0 and 511 to allocate up to the 4Mb architectural limit of the PDP-11. If the specified amount of extended memory is not available, the largest number of available 8192-byte regions are allocated. In general, the greater amount of extended memory allocated, the less work file activity and the faster the performance of the compiler. However, using more extended memory reduces the amount of remaining memory for other tasks or jobs while PDP-11 C is operating. The default value is /MEMORY=8. The /NOMEMORY qualifier is equivalent to /MEMORY=0.

---

**Note**

---

If this qualifier is used, it must be specified with the first compilation unit.

---

**/[NO]MODULE=(identifier | " string " [,identifier | " string " ])**
The module identifier is the name used by the object librarian, and the identifier appears in object libraries, object librarian listings, and link maps. The last-file-spec-name is the last file name specified for a given compilation unit (that is, the last file in a list separated by plus ( + ) signs). By default, PDP-11 C uses the last name as the module identifier. The module qualifier can be used to override the default module identifier or the module identifier specified by the **#module** or **#pragma module** preprocessor directives. This qualifier will accept at most two identifier or string values. If this qualifier is asserted, the user must supply at least the first value.

The default is /MODULE=(last-file-spec-name, ``V1.0").

**/[NO]OBJECT[=file-spec]**
Directs the compiler to produce an object module. By default, /OBJECT creates an object module file with the same name as the last source file of a compilation unit and with the OBJ file extension. If you include a file specification with /OBJECT, the compiler uses that specification instead. See Section 1.3.1 for more information about file specifications.

The compiler executes faster if it does not have to produce an object module. Use the /NOOBJECT qualifier when you need only a listing of a program or when you want the compiler to check a source file for errors.

The default is /OBJECT.

**/SHOW=(option, . . . )**
The qualifier /SHOW sets or cancels listing options. You must use the /LIST qualifier with the /SHOW qualifier to select or cancel any of the following options:

---

**Option Usage**

---

ALL The ALL option prints all listing information.

[NO]CONDITIONALS The CONDITIONALS option causes conditional program segments that were not compiled (due to **#if** type preprocessor directives) to appear in the listing. Specifying NOCONDITIONALS causes conditional program segments that were not compiled to be omitted in the listing. The default is CONDITIONALS.

[NO]EXPANSION The EXPANSION option prints final macro expansions in the program listing. When you specify this option, the macro nesting level of the last macro expanded on the line prints next to each line. The NOEXPANSION option is the default.

[NO]INCLUDE The INCLUDE option prints the contents of **#include** files in the program listing. The NOINCLUDE option is the default.

[NO]INTERMEDIATE The INTERMEDIATE option prints all intermediate and final macro expansions in the program listing. The NOINTERMEDIATE option is the default.

[NO]MACHINE_CODE The MACHINE_CODE option directs the compiler to list the generated machine code in the listing file. The NOMACHINE_CODE option is the default.

NONE The NONE option creates an empty listing file, with only the header.

[NO]SOURCE The SOURCE option places the source program statements in the program listing. The SOURCE option is the default.

---

**/[NO]STANDARD[=(option, . . . )]**

Determines what language features will be allowed. If you specify the ANSI option, this instructs the compiler to compile and generate code according to ANSI C Standard syntax and semantics. If you specify /STANDARD without an option, the default is /STANDARD=ANSI.

The default is /NOSTANDARD, which implies PDP-11 C native syntax and semantics.

When specifying the default /NOSTANDARD qualifier, PDP-11 C allows the use of **$** in identifier names. However, PDP-11 C does not support the use of **$** in identifier names when specifying the /STANDARD=ANSI qualifier.

**/[NO]TERMINAL[=[NO]SOURCE]**
Determines whether compiler messages are displayed at the terminal. If you specify either /TERMINAL or /TERMINAL=NOSOURCE, compiler messages are displayed on the user terminal or in the batch log, but associated user source text is not displayed. If you specify /NOTERMINAL, only the summary message is displayed on the user terminal or in the batch log.

If you specify /TERMINAL=SOURCE, the compiler displays the source line of each error, as well as the compiler messages associated with the error.

The default is /TERMINAL=NOSOURCE.

**/[NO]TITLE=["]identifier["]**
Controls the compiler-produced output list header for the program file. The identifier is the list title name for a given compilation unit. This list title name will override the **#pragma list title** .

This qualifier, if asserted, must be supplied with a string value. Use quotation marks around the identifier to retain lowercase characters or if the identifier contains a space character. By default, if no quotation marks are used, the identifier is converted to uppercase.

The default is /NOTITLE.

**/[NO]WARNINGS[=(option, . . . )]**
Controls whether the compiler prints warning diagnostic messages, informational diagnostic messages, neither, or both. The default qualifier, /WARNINGS, causes the compiler to print all diagnostic messages. The /NOWARNINGS qualifier

suppresses both the informational and the warning messages. Note, however, that error and fatal messages cannot be suppressed.

The two options are as follows:

---

**Option Usage**

---

NOINFORMATIONALS The NOINFORMATIONALS option causes the compiler to suppress informational messages.

NOWARNINGS The NOWARNINGS option causes the compiler to suppress all warning messages.

---

The informational message, SUMMARY, cannot be suppressed with /NOWARNINGS or /WARNINGS=NOINFORMATIONALS.

The default is /WARNINGS.

**/[NO]WORK_FILE_SIZE=value**
The value is an integer value between 1 and 65535 representing the number of 512-byte disk blocks to allocate for the work file. This qualifier is ignored on VMS systems. If the /WORK_FILE_SIZE qualifier is not specified, a default work file size of 2048 is used. The size of the work file determines the PDP-11 C compiler's capacity for processing source input. Note that the /MEMORY qualifier affects both the performance and capacity of the PDP-11 C compiler.

Specifying the WORK_FILE_SIZE qualifier with large values can impact compiler performance in varying degrees. See the Implementation Notes in Chapter 8 of this guide for more detail.

The work file is placed on the SY: device. PDP-11 C first attempts to open the work file contiguously. If insufficient contiguous disk storage is available, PDP-11 C then attempts to open the work file using noncontiguous disk storage. If

insufficient non-contiguous disk storage is available, PDP-11 C issues a diagnostic message and aborts. PDP-11 C does not extend the work file: if work file storage is exhausted during compilation, a diagnostic message is issued and PDP-11 C aborts.

---

**Note**

---

If this qualifier is used, it must be specified with the first compilation unit.

---

### 1.3.5 Compiler Error Messages

If there are errors in your source file when you compile your program, the PDP-11 C compiler signals these errors and displays diagnostic messages. Reference the diagnostic message, locate the error, and, if necessary, correct the error. Diagnostic messages displayed by PDP-11 C have the following format:

The following messages pertain to file
n: %PDP11C-s-ident, message-text

The parts of this message are described as follows:

**%PDP11C**

Is the facility name of the PDP-11 C compiler. This portion indicates that the message is being issued by PDP-11 C.

**s**

Is the severity of the error, represented as follows:

F Fatal error. The compiler stops executing when a fatal error occurs and does not produce an object, listing, or macro file. You must correct the error before you can compile the program.

E Error. The compiler continues, but does not produce an object or macro file. You must correct the error before you can successfully compile the program. Produces a listing file, if specified.

W Warning. The compiler produces an object module and macro file, if specified. It attempts to correct the error in the state-

ment, but you should verify that the compiler's action is acceptable. Otherwise, your program may produce unexpected results.

I Information. This message usually appears with other messages to inform you of specific actions taken by the compiler. No action is necessary on your part.

**ident**
Is the message identification. This is a descriptive abbreviation (mnemonic) of the message text.

**message-text**
Is the compiler's message. In many cases, it consists of more than one line of output. A message generally provides you with enough information to determine the cause of the error so that you can correct it.

**file**
The name of the source file in which the error occurred.

**n**
Gives you the number of the line where the error occurs. The number is relative to the beginning of the file specified by *file* . You can use the **#line** preprocessor directive to change both the line number and name that appear in the message.

The messages produced by the PDP-11 C compiler are listed in Appendix A.

The compiler command gives you control over the display of messages. The /NOWARNINGS qualifier, discussed previously, suppresses warning messages generated by the compiler.

### 1.3.6 Compiler Listings

A compiler listing provides information that can help you debug your PDP-11 C program. To generate a listing file, specify the /LIST qualifier when you compile your PDP-11 C program.

Under DCL on RSX-11M/M-PLUS systems:
$ **cc/list**

Under MCR on RSX-11M/M-PLUS systems:
> **cc/list**

Under CCL on RSTS/E systems:
$ **cc/list**

On RT-11 systems:
. **cc/list**

On VAX systems:
$ **pdpcc/list**

By default, the name of the listing file is the name of the source program with a file type of LST. You can include a file specification with the /LIST qualifier to override this default.

When used with the /LIST qualifier, the compiler command qualifier /SHOW supplies additional information in the compiler listing. See Section 1.3.4 for a description of each qualifier's function.

If the compiler command line contains the /LIST qualifier but does not contain the /SHOW qualifier, you are given the default listing. The default listing includes the following:

- Margin information
- PDP-11 C source text
- Errors encountered during the compilation
- Command line used to invoke the compiler

The left-hand margin of the source listing produced by the PDP-11 C compiler contains several items of information, arranged into fields in the following format:
nnnnn i x mm

**nnnnn**
Is the compiler-generated listing line number; it starts at

1 and is incremented by one for each line in the function, including lines read from included files (whether or not the /SHOW=INCLUDE qualifier was specified in the command line). All lines appearing before a function that do not belong to another function are included in the line numbering of the function.

**i**
Is the level of nesting of lines read from included files; this field is present only if /SHOW=INCLUDE is specified on the command line. Level 0, which appears as a blank, indicates lines read from the source file, or files, specified on the command line.

**x**
If the source line is ignored by the compiler as a result of the evaluation of a previous **#if** , **#ifdef** , or **#ifndef** preprocessor directive, this field appears as an ``x''; otherwise it is blank.

**mm**
Is the level of nesting of the last macro expanded in the line; this field is present only if the /SHOW=EXPANSIONS or /SHOW=INTERMEDIATE qualifier is specified on the command line. Level 0 corresponds to the original source line and appears as a blank. When this field is nonzero, however, the fields ``nnnnn'', ``i'', and ``ss'' all appear as blanks.

In all cases, the numbers listed are right justified in their fields with no leading zeros.

---

**Note**

---

The spacing within the compiler listings in this chapter may not be consistent with the spacing in actual compiler listings. These listings are condensed to fit on the page.

---

Example 1-1 shows the default compiler listing.

Key to Example 1-1:

**1** The name of the module appears at the top left of the listing, followed by the title string, the right-most 38 characters of the name of the source file, the version of the compiler, and the page number. The module name is specified with the **#module** or **#pragma module** preprocessor directive (see item 5) or is defaulted from the file name. The title string (if any) is specified with the **#pragma list title** preprocessor directive (see item 4).

**2** The module message identification appears on the second line of the listing followed by the listing subtitle string and the date and time of compilation. The module message identification is specified with the **#module** or **#pragma module** preprocessor directive or is defaulted to ``V1.0." The subtitle string is specified with the **#pragma list subtitle** preprocessor directive.

**3** The compiler generates listing line numbers. The generated line number is reset to zero at the end of each function.

**4** A title and subtitle string may be specified with the **#pragma list title** and **#pragma list subtitle** preprocessor directives, respectively. The title string has effect for the entire compilation unit and may be specified only once. A subtitle string has effect starting with the next page of the listing and may be specified any number of times.

**5** The internal object module title and message identification used by the librarian and linker or task builder may be specified with the **#module** or **#pragma module** preprocessor directive. The default object module title is taken from the first six characters of the object file name; the default object module message identification is ``V1.0."

**6** Compiler messages are generally cited against a point of interest in the source program. To indicate the point of interest, a digit is placed on the following line immediately beneath the point of interest. The corresponding message follows preceded by the point of interest digit, followed by an indication of the file specification and line number (relative to the start of the file, not the listing line number) of the source file to which the message pertains.

**7** Source lines that are excluded from the compilation with the **#if** preprocessor directive appear with an X in the

left margin. Such lines may also be excluded from the listing with the /SHOW=NOCONDITIONALS qualifier.

**8** A summary of the number of informational, warning, and error messages at the bottom of the listing, followed by the command used to compile the module.

Example 1-2 shows all compiler listing options.

Key to Example 1-2:

**1** Source lines included with the **#include** preprocessor directive appear in the listing with the /SHOW=INCLUDE and /SHOW=ALL qualifiers. The nesting level of the include file appears in the left margin.

**2** Final macro expansions are shown with the /SHOW=EXPANSION qualifier. Intermediate and final macro expansions are shown with the /SHOW=INTERMEDIATE and /SHOW=ALL qualifiers. Regardless of qualifiers specified, the final macro expansion is always shown for a line that has an associated compiler message. The macro nesting level of the last macro expanded on the line appears in the left margin.

**3** The macro ERROR_RECOVERY is defined as TRUE through the command line (see item 8). The intermediate expansion to TRUE is shown in the listing before the final expansion to 1.

**4** A machine code listing similar in format to that produced by MACRO-11 is included in the listing after each function with the /SHOW=MACHINE and /SHOW=ALL qualifiers.

**5** The relocatable object module memory location and the machine code instructions are listed.

**6** The assembly language code is shown beside its corresponding machine code instruction.

**7** Comments annotate the assembly language listing and correlate the assembly language statements with the PDP-11 C source language statements.

**8** The /DEFINE qualifier is used to define the ERROR_RECOVERY macro and drive the conditional compilation of the module.

### 1.4 Copying Files Among Target Environments

To copy text and object files among RSTS/E, RSX, RT-11, or VMS systems you can use DECnet or physical media. For more information about DECnet, refer to the DECNET documenation for your operating system.

Use Table 1-1 and the following sections to determine the appropriate tools to transfer files using physical media. You may need to refer to the specific documentation for more detail.

#### 1.4.1 File Transfer (FIT) Program

FIT transfers files between RSTS/E directory-structured devices and RT-11 directory-structured devices.

Using FIT, you can:

- Transfer files between RSTS/E-structured devices and RT-11-structured devices
- List the directory of an RT-11-structured device, including RX01 and RX02 flexible diskettes
- Delete files on an RT-11-structured device
- Initialize (zero) an RT-11-structured device
- Compress (squeeze) the files on an RT-11-structured device

To run the FIT program, use the following syntax:
[output[/switch]=]input[/switch]

The following is a sample FIT session:

```
$ run auxlib$:fit
FIT V9.0-14 RSTS V9.6-11 GNAT
FIT> dl0:*.obj/rt11/li
Directory of DL0:??????.OBJ (RT11 Format)
```

```
Name .Typ Size Date RT Pos
COINIT.OBJ 1 06-Jan-89 RT11 68
COFINI.OBJ 1 06-Jan-89 RT11 69
PRINTF.OBJ 1 06-Jan-89 RT11 70
WRITE .OBJ 1 06-Jan-89 RT11 71
COCSAL.OBJ 1 06-Jan-89 RT11 72
COCSAV.OBJ 1 06-Jan-89 RT11 73
COMAIN.OBJ 1 06-Jan-89 RT11 78
COMAI .OBJ 1 06-Jan-89 RT11 79
CRT .OBJ 5 06-Jan-89 RT11 80
XBL201.OBJ 2 06-Jan-89 RT11 185
XBL .OBJ 2 06-Jan-89 RT11 260
Total of 17 blocks in 11 files in DL0:
Total of 15869 free blocks in DL0:
FIT> *.*=dl0:write.obj/rt11
```

To exit the FIT program, enter Ctrl/Z. For more information on the FIT program, see the *RSTS/E Utilities Reference Manual* .

### 1.4.2 File Transfer Utility (FLX)

FLX allows you to use foreign volumes (not in Files-11 format) in DOS-11 or RT-11 format. FLX converts the format of a file to the format of the volume to which the file is being transferred.

You can use FLX interactively or by means of an indirect command file. FLX allows only one level of indirect command file specification.

You can invoke FLX by either specifying FLX or by specifying FLX and a command line. The format for entering FLX command lines follows:
devicespec/sw=infile/sw, **. . .** ,infilen/sw

For more information on FLX, see the *RSX-11M/M-PLUS Utilities Manual* .

### 1.4.3 VMS EXCHANGE Utility

The VMS EXCHANGE Utility performs file transfers and format conversions on RT-11 block-addressable volumes and DOS-format tapes. EXCHANGE recognizes RT-11

volumes on any VMS block-addressable device. However, RT-11 supports only some of the devices that are recognized by EXCHANGE.

For more information on the EXCHANGE Utility, see the *VMS Exchange Utility Manual* .

## 1.5 Linking a PDP-11 C Program

After you compile a PDP-11 C source program or module, invoke the Linker or Task Builder to combine your PDP-11 system object modules into one executable image. The executable image can then be executed on a PDP-11 system or a VMS system with an RSX emulator. A source program or module cannot run until you link it with the Task Builder or Linker. The following sections show methods you can use to invoke the Linker or Task Builder.

The RT-11 Linker and RSX Task Builder are system programs that link relocatable object modules to form an executable task image. Use the RT-11 Linker to build tasks that execute on the RT-11 operating system or to build tasks on the RSTS/E system that execute under the RT-11 Run-Time System. Use the RSX Task Builder to build executable tasks for the following systems.

- RSX-11M/M-PLUS
- Micro /RSX
- RSX-11S
- VAX-11 RSX operating systems
- Tasks built on RSTS/E operating systems that execute under the RSX Run-Time System

When you execute the LINK command or TKB command, the Task Builder or RT-11 Linker performs the following functions:

- Resolves local and global symbolic references in the object code
- Assigns values to the global symbolic references

- Signals an error message for any unresolved symbolic reference
- Allocates memory space for the executable image

The Task Builder also resolves references to resident common blocks and resident libraries.

The object modules to be linked can come from user-specified input files, user libraries, or system libraries. The Task Builder 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 Task Builder automatically searches the appropriate system object library to attempt to resolve them. For additional information about libraries, refer to the Section 1.5.8.

You can also use the Task Builder to build tasks with overlay structures. For additional information about the Task Builder and Task Builder options, refer to the *RSX-11M/M-PLUS and Micro /RSX Task Builder Manual* or the *RSTS/E Task Builder Reference Manual* and Section 1.5.9.

### 1.5.1 Linking a Program on RSX Systems

Use the DCL LINK command to invoke the Task Builder. For example, to link the PDP-11 C program PROG1 on RSX, use the following command line:

$ **link prog1,lb:[1,1]cfpursx.olb/library**

You can also use the following format to link a program, using a command file as follows:

LINK @command-file

For example, if you want to link using the command file PROG1.CMD, you enter the following command:

$ **link @prog1.cmd**

Alternatively, you can invoke the Task Builder (under either the DCL or MCR command line interpreters) by typing a RUN command in the following format:

$ **run tkb**

Or, if your system manager has installed TKB, you can enter the following:
$ **tkb**

In either case, after you press the Return key, the Task Builder prints the TKB> prompt. You then enter the TKB command.

After you press the Return key, the Task Builder prints another TKB> prompt. You then:

1. Enter additional input files, if any.
2. Enter a line containing only two slashes (//) to tell the Task Builder to create a task image and to exit.
3. Press the Return key.

See the *RSX-11M/M-PLUS and Micro /RSX Task Builder Manual* for detailed instructions.

### 1.5.2 Linking a Program on RSTS/E Systems

There are two ways you can link programs on RSTS/E. You can invoke the RSX Task Builder or the RT-11 Linker. The following sections describe these two methods.

#### 1.5.2.1 Invoking the RSX Task Builder on RSTS/E

There are three ways to invoke the RSX Task Builder on RSTS/E. You can invoke the Task Builder by using the DCL LINK command as follows:
$ **link/cc prog1**

You can run the Task Builder by entering a RUN command in the following format:
$ **run tkb**

Or, if your system manager has installed TKB as a CCL command, you can enter the following:
$ **tkb**

In each case, after you press the Return key, the Task Builder prints the TKB> prompt. You then enter the TKB command.

After you press the Return key, the Task Builder prints another TKB> prompt. You then:

1. Enter additional input files, if any.
2. Enter a line containing only two slashes (//) to tell the Task Builder to create a task image and to exit.
3. Press the Return key.

The following example shows how to taskbuild FOO.C on RSTS/E, using the taskbuilder FOO.CMD and FOO.ODL files.

FOO.C contains the source code for FOO.C, which copies the contents of one file to another file. FOO.ODL is the overlay description file.

```
/*Sample FOO.C program.*/
#include <stdio.h>
#include <errno.h>
int main ()
{
FILE *in;
FILE *out;
int c;
char inname[133];
char outname[133];
printf ("\nInput file?:\n");
gets(inname);
printf ("\nOutput file?:\n");
gets(outname);
if (in = fopen(inname, "r"))
        {
        if (out = fopen(outname, "w"))
                {
                while ((c = getc(in)) != EOF)
                        putc(c,out);
                fclose (out);
                }
        else
                {
                printf ("Could not open file %s\n",outname);
                printf ("Error was %d\n", errno);
                }
        fclose (in);
```

```
        }
else
        {
        printf ("Could not open file %s\n",inname);
        printf ("Error was %d\n", errno);
        }
}
```

The following file, FOO.CMD, calls the file FOO.ODL.

```
;Sample TKB CMD file to build program FOO.C for RSTS/E
SY:FOO=SY:FOO/MP
//
```

FOO.ODL is linked with FOO.C to produce an executable file.

```
;Sample TKB ODL file to build program FOO.C for RSTS/E
        .ROOT USER
USER: .FCTR SY:FOO-LB:CEISRE/LB:$PRMXF-RMSROT-LIBR,RMSALL
LIBR: .FCTR LB:CEISRE/LB
@LB:RMS11S
        .END
```

After compiling FOO.C, taskbuild the files using the following command:

$ **tkb @foo**

See the *RSTS/E Task Builder Reference Manual* for more detailed instructions.

### 1.5.2.2 Invoking the RT-11 Linker on RSTS/E

You can link your program on RSTS/E by invoking the RT-11 Linker. You invoke the RT-11 Linker as follows:

$ **run link.sav**

See Section 1.5.3 for additional information on the RT-11 Linker.

### 1.5.3 Linking a Program on RT-11 Systems

You can invoke the RT-11 Linker in either of two ways: using the Keyboard Monitor LINK command or using the RUN command.

Using the Keyboard Monitor LINK command adheres to the

following syntax, where ``filespec'' represents the file to be linked:
LINK[/option . . . ]filespec[/option . . . ][, . . . filespec[/option . . . ]]

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

To run the Linker using the RUN command, use the following format:
RUN SY:LINK

The RUN 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:
*

For example, if you want the object files WINKN, BLNKN, and NOD linked into an executable memory image file, you can enter a succession of commands as follows:
. **run sy:link**
(From this point on the linker issues the * prompt.)
* **winkn,winkn=winkn,blnkn,nod**

To exit the linker, enter Ctrl/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 RUN command either with or without arguments, the RT-11 Linker accepts the first command string in the form:
[bin-filespec][,map-filespec][,stb-filespec] = [infiles-list]

To make a job an RT-11 virtual job, use the $VIRTUAL$JOB macro in one of the routines in the job:

```
#include <RTSYS.H>
$VIRTUAL$JOB
```

The following is an example of how to link on RT-11:

```
. run sy:link
* test/m:3000/b:3000,test=test,sy:ceisrt
```

To exit the linker, enter Ctrl/C.

### 1.5.4 Linking a Program on VMS Systems

To link a program using VAX-11 RSX on the VMS operating system, invoke the Task Builder as follows:

```
$ mcr tkb
```

The Task Builder can then be used as shown in Section 1.5.1.

The following example shows how to taskbuild TEST.C on VMS and RSX using the taskbuilder TEST.CMD and TEST.ODL files.

TEST.C contains the source code which copies the contents of one file to another file.

```
/*Sample TEST.C program.*/
#include <stdio.h>
#include <errno.h>
int main ()
{
FILE *in;
FILE *out;
int c;
char inname[133];
char outname[133];
printf ("\nInput file?:\n");
gets(inname);
printf ("\nOutput file?:\n");
gets(outname);
if (in = fopen(inname, "r"))
        {
        if (out = fopen(outname, "w"))
                {
                while ((c = getc(in)) != EOF)
                        putc(c,out);
                fclose (out);
                }
```

```
        else
                {
                printf ("Could not open file %s\n",outname);
                printf ("Error was %d\n", errno);
                }
        fclose (in);
        }
else
        {
        printf ("Could not open file %s\n",inname);
        printf ("Error was %d\n", errno);
        }
}
```

The following file, TEST.CMD, calls the file TEST.ODL.

```
;Sample TKB CMD file to build program TEST.C for RSX or VMS
SY:TEST/CP=SY:TEST/MP
//
```

TEST.ODL is linked with TEST.C to produce an executable file.

```
;Sample TKB ODL file to build program TEST.C for RSX or VMS
        .ROOT USER
USER: .FCTR SY:TEST-LB:[1,1]CEISRSX/LB:$PRMXF-RMSROT-LIBR,RMSALL
LIBR: .FCTR LB:[1,1]CEISRSX/LB
@LB:[1,1]RMS11S
        .END
```

After compiling TEST.C, taskbuild the files using either of the following commands on VMS:

$ **mcr tkb @test**

or

$ **mcr tkb**
TKB> **@TEST**

### 1.5.5 Task Builder Command-Line Elements

When you invoke the Task Builder, certain files must be present, and you can use various qualifiers. The following sections describe the files that you need for task building and the qualifiers that you can specify.

### 1.5.5.1 Creating CMD and ODL Files for Task Building

Before you link a PDP-11 C object file, you may want to produce a command (CMD) file and an overlay description file (ODL).

The following is an example of a CMD file that the Task Builder uses to produce a TSK file:

```
SY:XBL201/CP=SY:XBL201/MP
//
```

For more information on CMD files, see the *RSX-11M/M-PLUS and Micro /RSX Task Builder Manual* or the *RSTS/E Task Builder Reference Manual* .

The following is an example of an ODL file that is used by the CMD file for an RSX system:

```
        .ROOT USER
USER: .FCTR SY:XBL201-LIBR
LIBR: .FCTR LB:[1,1]CEISRSX/LB
        .END
```

For information about overlaying, see the Overlay Capability and Overlay Loading Methods chapters in the *RSX-11M /M-PLUS and Micro /RSX Task Builder Manual* , or the appropriate sections about overlaying in the *RSTS/E Task Builder Reference Manual* .

### 1.5.5.2 Command-Line Elements in CMD Files

The elements that you specify on the first line in the CMD file are as follows:

```
task-file/qualifier,map-file/qualifier,infiles-list/qualifier
```

**task-file**

The file specification of the task-image output file. This file specification may be omitted if no task-image file is desired. If a specification is entered, only a file name is required; a file type value of TSK (EXE under VAX-11/RSX) is assumed if no file type is specified. Therefore, the following two commands are equivalent. Note, however, that no map file is created in either case.

```
TKB> FILE1/FP=FILE1
TKB> FILE1.TSK/FP=FILE1
```

**map-file**
The file specification of the map output file. This file specification may be omitted if no task-image map file is desired. If a specification is entered, only a file name is required; a file type value of MAP is assumed if no file type is specified. On RSX systems, the map file is automatically spooled to the line printer. On some operating systems, the map file is automatically deleted after it is printed.

**infiles-list**
The list of input files that contains compiled PDP-11 C object modules. (This list may also contain compiled or assembled libraries and modules that were written in a language other than C, 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.

### 1.5.5.3 Task Builder Qualifiers

You can use command qualifiers to modify the Task Builder's output, as well as to include the On-Line Debugging Tool (ODT). Task building output consists of an image file and an optional map file.

The following list summarizes some of the most commonly used command qualifiers that you can specify in the CMD file. A brief description of each qualifier follows this list. For a complete list of task-building command qualifiers, see the sections about link qualifiers and TKB qualifiers in the *RSX-11M/M-PLUS and Micro /RSX Task Builder Manual* , or the section about task builder qualifiers in the *RSTS/E Task Builder Reference Manual* .

**Task-Image Output File Qualifiers**
You can use the following qualifiers for the task-image output file:

**/FP**
Specifies that the task uses the Floating-Point Processor (FP11) or floating-point microcode option (KEF11A).

**/DA**
Specifies that the system debugging aid ODT is to be included in the task.

**/CP**
Specifies that the task be checkpointable; must use for tasks using standard I/O or memory management (calloc, free, and so on).

**/ID**
Specifies that the task use I- and D-space. You can build an I- and D-space task on RSX-11M-PLUS (Version 4.3 or higher), Micro /RSX (Version 4.3 or higher), and RSTS/E (Version 10.0 or higher).

The PDP-11 C compiler generates code that will run in I- and D-space.

**/MU**
Specifies that multiple versions of the task may be run simultaneously. The read-only portions of the task are shared.

**Map File Qualifiers**
You can use the following qualifiers for the map file:

**/CR**
Specifies that a global cross-reference listing is to be appended to the map file.

**/SP**
Specifies that the map file is to be spooled to the line printer.

**Input-File Qualifiers**
You can use the following qualifiers for input files:

**/LB**
Specifies that the input file is to be a library file. (See Section 1.5.8.)

**/MP**
Specifies that the input file is an overlay description file. (See

Section 1.5.9.)

### 1.5.6 Task Builder Error Messages

If the Task Builder detects any errors while linking object modules, it displays messages indicating the cause and severity of the error. If any fatal error conditions occur (that is, errors with a severity of

*

FATAL

*

), the Task Builder does not produce an image file.

Some common errors that occur during linking are as follows:

.

The input file has a file type other than OBJ, and no file type was specified on the command line.

If you do not specify a file type, the Task Builder searches for a file that has a file type of OBJ by default. If the file is not an object file and you do not identify it with the appropriate file type, the Task Builder signals an error message and does not produce an image file.

.

You tried to link a nonexistent module.

The Task Builder signals an error message if you misspell a module name or if the compilation contains fatal diagnostics.

.

A reference to a symbol name remains unresolved.

An error occurs when you omit required module or library names from the command line and the Task Builder cannot locate the definition for a specified global symbol reference. In the following example on RSTS/E, a main program module, OCEAN.OBJ, calls the subprogram modules REEF.OBJ, SHELLS.OBJ,

and SEAWD.OBJ, and the following LINK command is executed:

$ **link/cc ocean, reef, shells, lb:cfpure/lb**

Because SEAWD is not included in the link, the Task Builder signals the following error message:

TKB -- *DIAG* -1 undefined symbols segment OCEAN
SEAWD

.

The task has grown over the 32K limit.

The error would be as follows:

Segment OCEAN has addr overflow:allocation deleted

If an error occurs when you link modules, you can often correct the error by reentering the command string and specifying the correct modules or libraries. If an error indicates that a program module cannot be located, you may be linking the program with the wrong PDP-11 C library.

### 1.5.7 Storage Considerations

Most storage for objects with the **auto** storage class specifier is allocated on the stack in PDP-11 C. Therefore, when linking, you should carefully consider how much automatic storage your program needs at any time. Since C is a stack language, many PDP-11 C programs require additional stack space beyond the default provided by TKB or the RT-11 Linker. If you do not allow for this, insufficient stack space will cause your program to behave unpredictably.

As your program executes, it uses stack space for the following:

.

Automatic variables within subroutines

.

Parameters passed to subroutines

.

Subroutine return addresses and return values

.

Registers saved by subroutines (up to 54 bytes)

.

Run-time library storage

Determine the amount of storage you need to allocate by summing the space used for each item that uses stack space. Include items from both the main program and each subroutine. Each automatic variable, parameter, return address, and return value requires the following space on the stack:

. char, short, pointer-2 bytes

. long, float-4 bytes

. double-8 bytes

. arrays, structures-multiples of the variables involved

As with variables, you need to calculate the size of the parameters, return addresses and values, and registers for each subroutine as well as for the main program.

To set the size of the stack when using the RSX Task Builder on RSX or RSTS systems, use the STACK option to set the number of words of STACK used. The following Task Builder command file allocates 1024 bytes of space to the stack:

```
SY:TEST/CP=SY:TEST/MP
/
STACK=512
//
```

On RT-11 systems, the stack is located above address 476. To set the stack size when using the RT-11 Linker on RT-11 or RSTS systems, use the /M and /B qualifiers to set the address of where the stack begins and to link the code above the stack. The following Linker commands allocate about 1024 bytes of space to the stack:

```
r link
test/m:2500/b:2500,test=test,sy:ceisrt
```

### 1.5.8 Library Usage

Libraries consist of a collection of object modules. When the

Task Builder or Linker encounters a library specification, it searches the library for definitions of any of the currently undefined global symbols. The modules containing these definitions are included in the task image being built.

Run-time libraries (RTL) contain functions and macros to perform input, output, and various task related to specific operating environments. For proper support, you must link your program to the run-time library developed for your operating system. Section 1.5.8.1 explains how to select and specify the run-time library.

Disk and resident libraries are available for use with some operating systems. Disk Libraries are stored in files on disk. The Task Builder can make a disk library a physical part of a task image. From disk libraries, the Task Builder copies object modules into the task image of each task that references those modules.

Resident Libraries are located in main memory and are shareable; that is, a single copy of each library is used by all tasks that refer to it. The Task Builder can make a resident library a logical part of a task image but not a physical part; that is, the Task Builder can link the library to a task image but cannot copy the library to a task image.

Section 1.5.8.2 has more information about system disk and resident libraries. User disk and user resident libraries are described in Section 1.5.8.3.

PDP-11 C provides a run-time library that can be installed as a supervisor-mode library on some systems. When a user task is linked to this library, a large part of the PDP-11 C run-time library resides in supervisor mode, thereby increasing the amount of user mode intruction space available for your program. Section 1.5.8.4 has more information about using the PDP-11 C supervisor-mode library. Refer to the *RSX-11M/M-PLUS and Micro /RSX Task Builder Manual* and the *RSTS/E Task Builder Reference Manual* for more information about supervisor-mode libraries.

### 1.5.8.1 PDP-11 C Run-Time System Object Libraries

You must link your program with the correct PDP-11 C run-time library (RTL), so that the proper run-time support is included. Each supported target operating system has two run-time libraries associated with it. The FPU run-time libraries support floating-point instructions, and the EIS run-time libraries support EIS instructions.

When you compile your programs with /ENVIRONMENT=NOFPU, you may link them to either the FPU or EIS run-time library. However, if the target machine has floating-point hardware, it is suggested that you link to the FPU library.

When you compile your programs with /ENVIRONMENT=FPU, you should link to the FPU run-time library. Though linking to the EIS library will work in some cases, certain RTL routines might encounter problems. The release notes indicate some, but not all, of the problems that you could encounter.

The following table shows the different library names and the instruction set they require for each supported operating system.

---

**Instruction Set**

**RSX Systems**

**RSTS/E Systems**

**RT-11 Systems**

---

FPU (floating-point) CFPURSX CFPURE CFPURT
EIS CEISRSX CEISRE CEISRT

---

For programs that use standard I/O, refer to the /CP taskbuilder switch and the input/output support package sections in the *PDP-11 C Run-Time Library Reference Manual* for additional information.

### 1.5.8.2 Using System Libraries

Each system has a system disk library. Consult with your system manager to determine which system resident libraries are available on your system. You can create your own user disk and resident libraries.

Each RSX and RSTS/E system has a system disk library called LB:SYSLIB.OLB. In addition, each RSX system has available to it three system resident libraries. RSTS/E systems have one system resident library available that are pertinent to PDP-11 C.

The system disk library is as follows:
LB:[1,1]SYSLIB.OLB

The Task Builder automatically searches the system disk library to see if any undefined global references remain after all the input files have been processed. If the definition of one of these undefined global symbols is found, the appropriate object module is included in the task being built.

Consult your system manager to determine which of the following system resident libraries are available on your system.

---

**Library System Description**

---

FCSRES RSX only A shared library of commonly used FCS-11 input/output (I/O) routines
FCSFSL RSX only A supervisor-mode File Control Services (FCS) library
RMSRES RSX and

| | |
|---|---|
| RSTS/E | A shared library of RMS-11 I/O routines can be built in supervisor mode on RSX-11M-PLUS systems |

---

These system resident libraries are linked to a task by using the Task Builder option, as follows:

For FCSRES:
LIBR = FCSRES:RO

For FCSFSL:
RESSUP=FCSFSL/SV

For RMSRES on a RSX system:
LIBR = RMSRES:RO

or
RESSUP=LB:[3,54]RMSRES/SV:0

For RMSRES on a RSTS/E system:
LIBR = RMSRES:RO

or
RESSUP=RMS$:RMSRES/SV:0

### 1.5.8.3 Creating User Libraries

Using the Librarian Utility Program (LBR), you can construct your own PDP-11 C or assembly language disk libraries. You then access these libraries by using the library qualifier, /LB after the library name. Consult the *RSX-11M/M-PLUS Utilities Manual* and the *RSTS/E Programmer's Utilities Manual* for further information on the LBR.

For example, if MATRIXLIB.OLB is a disk library containing matrix manipulation routines and PROG is the object file of a compiled PDP-11 C program that calls the matrix routines, you could enter the following command line for the Task

Builder:
$ **tkb prog/fp=prog,matrixlib/lb**

You can construct resident libraries using the taskbuilder.
For more information, see the *RSX-11M/M-PLUS and Micro /RSX Task Builder Manual* .

### 1.5.8.4 Using the supervisor-mode Library

When a task uses supervisor-mode libraries the virtual address space available for the task is increased because the supervisor-mode library resides in a different address space. Refer to the *RSX-11M/M-PLUS and Micro /RSX Task Builder Manual* and the *RSTS/E Task Builder Reference Manual* for more information about supervisor-mode libraries.

PDP-11 C provides a run-time library that can be installed as a supervisor-mode library on RSX-11 M-PLUS, Micro /RSX and RSTS/E systems that support the FPU processor. PDP-11 C does not provide a supervisor-mode library for the RT-11 operating system.

When you link a user task to the run-time supervisor-mode library, the user mode instruction space available is substantially increased. This allows you to write larger programs without using overlays.

Even more user mode instruction space can be made available when the run-time supervisor-mode library is used together with the system resident libraries, RMSRES for RSTS/E and RSX systems, and FCSFSL and FCSRES for RSX systems. (See Section 1.5.8.2 for more information about system resident libraries.)

Since the run-time supervisor-mode library is Position Independent Code (PIC), it does not have to reside in APR0 when other supervisor-mode libraries are linked with the task.

The supervisor-mode library contains a subset of the CFPURSX.OLB or CRPURE.OLB PDP-11 C run-time libraries.The files for the supervisor-mode library include the following:

.
For RSX and Micro /RSX systems:
CCSMRX.TSK
CCSMRX.STB
.
For RSTS/E systems:

CCSMRE.TSK
CCSMRE.STB
CCSMRE.LIB

You must install the library before you can use it. Use one of the following formats to install the library.

For RSX and (Micro /RSX:
Using DCL:
$ **install/task_name:ccsmrx lb:[1,1]ccsmrx.tsk**

Using MCR:
> **ins lb:[1,1]ccsmrx/ron=yes**

For RSTS/E:
$ **install/library/read_only lb:ccsmre**

To link to the library, you must reference the module CSMSUP.OBJ. The reference to CSMSUP.OBJ must occur before any reference to the PDP11-C run-time library. This can be done by extracting CSMSUP.OBJ as an object or by referencing CSMSUP in the taskbuilder .ODL file.

To extract CSMSUP.OBJ as an object, use one of the following commands:

For RSX or Micro /RSX:
LBR CSMSUP.OBJ=LB:[1,1]CFPURSX/EX:CSMSUP

For RSTS/E:
LBR CSMSUP.OBJ=LB:CFPURE/EX:CSMSUP

To reference CSMSUP in the taskbuilder .ODL file, use one of the following commands:

For RSX or Micro /RSX:
label .FCTR directory : yourtask -LB:[1,1]CFPURSX/LB:CSMSUP- label1

For RSTS/E:
label .FCTR directory : yourtask -LB:CFPURE/LB:CSMSUP- label1 \

For example, in the following taskbuilder .ODL file the CSMSUP.OBJ is referenced before the reference to the RSX PDP-11 C run-time library CFPURSX. For an RSTS/E system, reference the PDP-11 C run-time library CFPURE.
CPCSM.ODL

```
.root user
  user: .fctr sy:yourtask-csmsup-user1
  user1:.fctr rmsrot-libr,rmsall
  libr: .fctr lb:[1,1]cfpursx/lb:$prmxf-lb:[1,1]cfpursx/lb
  @lb:[1,1]rms11s
  .end
```

The task is linked with the resident library by using the RESSUP taskbuilder option as shown in the following example. Note that the size of the stack is set to allow data to be passed on the stack. The following example is written for an RSX system, and uses CCSMRX and CFPURSX. For a RSTS/E system, use CCSMRE and CFPURE.
CPCSM.CMD

```
yourtask/cp/id,yourtask/cr/ma/-sp,yourtask=yourtask/mp
stack=3000
ressup=lb:[1,1]CCSMRX/SV:0
```

Programs which use RMS or FCS to support PDP-11 C standard I/O can increase the available user-mode instruction space by linking the task with both the PDP-11 C supervisor-mode library and either RMSRES, FCSRES or FCSFSL.

The following taskbuilder command file and ODL file show how a program can be linked using the RMS-11 I/O routines library, RMSRES, and the supervisor-mode library, CCSMRX.
CMD File:

```
 hello/fp/cp=hello/mp
 ressup=lb:[3,54]rmsres/sv:0
```

```
 ressup=lb:[1,1]ccsmrx/sv
 //
ODL File:
   .root user-rmsrot,rmsall
 user: .fctr sy:hello-csm-io-libr
 csm: .fctr lb:[1,1]cfpursx/lb:csmsup
 io: .fctr lb:[1,1]cfpursx/lb:$prmxf
 libr: .fctr lb:[1,1]cfpursx/lb
 @lb:[1,1]rmsslx
   .end
```

The following taskbuilder command file and ODL file show how a program can be linked using RMSRES and the supervisor-mode library CCSMRE.

```
CMD File:
 hello/fp/cp=hello/mp
 ressup=lb:rms$:rmsres/sv:0
 ressup=lb:ccsmre/sv
 //
ODL File:
   .root user-rmsrot,rmsall
 user: .fctr sy:hello-csm-io-libr
 csm: .fctr lb:[1,1]cfpure/lb:csmsup
 io: .fctr lb:[1,1]cfpure/lb:$prmxf
 libr: .fctr lb:[1,1]cfpure/lb
 @lb:[1,1]rmsslx
   .end
```

The following taskbuilder command file and ODL file show how a program can be linked using the File Control Services library, FCSFSL, with the supervisor-mode library CCSMRX. To use the FCS-11 I/O routines library, FCSRES, with the supervisor-mode library, substitute FCSRES for FCSFSL in the SUPLIB command line in the command file.

```
CMD File:
 hello/fp/cp=hello/mp
 suplib=fcsfsl/sv:0
 ressup=lb:ccsmrx/sv
 //
ODL File:
   .root user
 user: .fctr sy:hello-csm-io-libr
 csm: .fctr lb:[1,1]cfpursx/lb:csmsup
```

```
io: .fctr lb:[1,1]cfpursx/lb:$prcxf
libr: .fctr lb:[1,1]cfpursx/lb
  .end
```

### 1.5.9 Overlays

The overlay facility provided by the Task Builder and RT-11 Linker allows large programs to be executed in relatively small areas of main memory. An overlaid program is essentially a program that has been broken down into parts, or overlays, that are loaded into memory automatically during program execution. Please refer to Section 8.4 for more detailed information.

Additional information on overlays can be found in the following books:

- For the RSX environment-Overlay Capability and Overlay Loading Methods in the *RSX-11M/M-PLUS and Micro /RSX Task Builder Manual*
- For the RSTS environment- *RSTS/E Task Builder Reference Manual*
- For the RT-11 environment- *RT-11 System Utilities Manual*

### 1.6 Running a PDP-11 C Program

After you link your program, you can use the RUN command to execute it. The RUN command has the following format:

RUN file-spec

**file-spec**
Specifies the file you want to run.

The following example executes the image SAMPLE.TSK:

$ **run sample**

See Section 2.9.2 for information on passing arguments to a main function.

During execution, an image can generate a fatal error called an exception condition. When an exception condition occurs, the system displays an error message. Run-time errors can also be issued by the operating system or by certain utilities.

When an error occurs during the execution of a program, the program is terminated and the operating system condition handler displays one or more messages on the user-terminal device. On RSX and RSTE/E systems, the message is followed by a register display.

For example, if a reserved instruction condition occurs, a run-time message followed by a register dump similar to the following RSX register dump appears:

```
Task = "TT43" terminal
   Reserved inst execution
R0 = 001751
R1 = 100102
R3 = 177777
R4 = 014716
R5 = 176026
SP = 001200
PC = 004002
PS = 170010
```

In the previous example:

**R0-R5**
Are the contents of each register.

**SP**
Is the contents of the stack pointer.

**PC**
Is the value of the program counter. This value represents the location in the program image at which the error occurred. The location is relative to the virtual memory address that the Task Builder assigned to the code program section of the module indicated by module name.

**PS**
Is the contents of the Processor Status Word (PSW).

### 1.7 Debugging a PDP-11 C Program

You can use the On-Line Debugging Tool (ODT), a user-interactive debugging aid. On RSX and RSTS/E systems, you can use the /DA qualifier to specify that ODT be included in the task when linking.

For more information about ODT, see the *RSX-11M/M-PLUS and Micro /RSX Debugging Reference Manual* .

[next] [previous] [contents]

---

## 2. Program Structure

This chapter introduces the basic features of PDP-11 C to the experienced programmer. The text provides detailed examples and short tutorials, as well as pointers to other chapters in this guide. PDP-11 C background material, and the following components of program structure are detailed:

- Function definitions
- Function declarations
- Function prototypes
- Function parameters and arguments
- Program identifiers
- Blocks
- Comments
- PDP-11 C language keywords
- Lexical Continuation
- Trigraphs

## 2.1 C Programming Language Background

The C language is a general-purpose programming language that is manageable due to its small size, flexible due to its ample supply of operators, and powerful due to its utilization of modern control flow and data structures. The C language was originally designed and implemented on a UNIX® system on the PDP-11. The designers of the language comment on its functionality in the following passage:

> ``The [C] language **. . .** is not tied to any one operating system or machine; and although it has been called a ' system programming language ' because it is useful for writing operating systems, it has been used equally well to write major programs in many different domains."

**1**

Like assembly language, C was not designed to accommodate the needs of any particular application. The C language manipulates and stores data with regard to the similarities of modern machine architecture. However, C is not as complex as assembler language and is not machine dependent. C is highly portable. A program is portable if you can compile and run its source program using several different compilers on several different machines.

There is an ANSI standard for the C language that promotes the consistency of functionality between C implementations on different systems. There needs to be consistency if C is to be portable across systems; this is one of the most desirable features of the language. So, not only should C source programs be portable, the language features themselves should produce the same effects on all systems when you compile and run programs.

The C language was developed in a UNIX system environment and eventually was used to rewrite most of that operating system, so many standard methods of operation in C are related to UNIX. For instance, UNIX systems access files by a numeric file descriptor, so many C implementations provide functions to access files by file descriptor.

Some standard C constructs include preprocessor directives

and a run-time library of functions and macros. In many implementations, a preprocessor completes the tasks designated in the preprocessor directives located in the source code before any action is taken by the compiler. Because the C language has no means to input and output information, a run-time library usually provides this service.

## 2.2 The PDP-11 C Programming Language

The PDP-11 C programming language is a highly reliable product that is highly compatible with the ANSI C Standard. PDP-11 C is an optimizing C language processor for Digital's major operating systems on the PDP-11. Because C can be used to program system applications, PDP-11 C is an alternative to MACRO-11. This allows users, using a high-level language, to write code for inclusion into read-only memory (ROM), resident libraries, device drivers, and other low-level system routines.

PDP-11 C runs native on the supported host systems and produces PDP-11 objects compatible with the RSX Task Builder and the RT-11 Linker. The standard libraries are provided in object form and are portable across systems, except for the library routines that provide direct access to operating system functions. For example, Standard I/O (stdio) is operating system dependent.

Within the VMS environment, PDP-11 C is a cross compiler, running as a native VMS image and producing PDP-11 object code. If you want to build and run your task in the VMS environment, VAX-11 RSX or CP/RSX must be installed on your system. If you do not want to link or run on VMS, you must use either DECnet or physical media to transport generated objects to the target system. Libraries are provided with the VMS kit to support RSX-11M, RSX-11S, RSX-11M-PLUS, Micro /RSX, RSTS/E, RT-11, VAX-11 RSX, and VAX CoProcessor/RSX.

In the RSX, RSTS/E, and RT-11 host environments, the compiler generates native PDP-11 object code. These compilers can generate object code for all target systems. The libraries for the RSX-11M, RSX-11S, RSX-11M-PLUS, Micro /RSX, RSTS/E, RT-11, VAX-11 RSX, and VAX CoProcessor/RSX target systems are supplied with the RSX, RSTS/E, and RT-11 compilers.

The PDP-11 C programming language incorporates the features that are fundamental to the C language and that exist in most C compilers. However, PDP-11 C also provides features, unique to PDP-11 C, that work directly and efficiently with PDP-11 operating system environments.

## 2.3 Writing a Program

The first program presented here is a simple one that adds two numbers and stores the total in a variable. Example 2-1 shows how to code such a program.

Key to Example 2-1:

**1 Comments.** The text contained between the characters (/ * ) and ( * /) are comments. You cannot place comments within comments (that is, they cannot be nested), but you can place comments anywhere white space is allowed. *White space* is an area within the source code where blank spaces, tabs, or blank lines separate code. In later chapters, permitted white space is defined for PDP-11 C constructs.

**2 User-Defined Functions.** PDP-11 C programs are comprised of user-defined functions that cannot be nested. A user-defined function named main is defined. In PDP-11 C, execution of a program must begin by calling a function named main.

PDP-11 C functions have methods of exchanging information using parameters and arguments. In the function definition in Example 2-1, the lack of parameters is designated by the keyword **void** within the parentheses. The function main in this example does not receive information through parameters, and there are no function calls.

To specify parameters in a function definition, you list the parameter identifiers within the parentheses and separate them with commas ( , ). You must declare the types of parameters either within the parentheses or in a declaration list before the body of the function. If you call a function from within function main (you normally do not call the main function from another part of your

program), the function name is followed by a list of arguments delimited by parentheses and separated by commas.

The function performs its task as determined by the statements found in the body, and may or may not return a value to the calling expression. The body of any function is delimited by braces ({ }). They are analogous to the **DO-END** of PL/I, or the **BEGIN-END** of Pascal. The body may contain one or more **return** statements. A **return** statement specifies what, if anything, is returned to the expression that called the function. Depending on the set-up of the function, you can omit the **return** statement, and its return value will remain undefined. If a function does not return a value, you can declare the function to be of type **void** . For more information concerning functions, refer to Section 2.7.2.

**3 Variable Declarations.** The variable *total* is declared and defined within the function main. You must declare all variables before referencing them within the program. Declarations end with a semicolon ( ; ). When you declare a variable, you specify its data type. Data types specify the amount of storage required and how to interpret the stored object. The variable *total* is of type **int** (integer). PDP-11 C interprets variables of type **int** as signed objects which require 16 bits (2 bytes or 1 word) of memory. For more information concerning data types, refer to Chapter 5.

When you define a variable, you specify its storage class which affects its location and lifetime. Variables declared within a function have a default storage class of **auto** (automatic). Variables of this storage class receive storage space when the function is activated, and storage is freed when control of the calling function resumes. See Chapter 6 for descriptions of other types of storage classes.

You specify PDP-11 C storage classes by placing the storage class keyword either before or after the data type keyword in the variable declaration. Note, however, that placing the storage class keyword anywhere other than before the data type keyword in the variable declaration is considered ``obsolescent'' by the ANSI C Standard. Keywords are the reserved words used to identify data types (such as **int** , **double** ), storage classes (such as **auto** , **static** ), statements (such as **if** , **goto** ), and operators

(such as **sizeof** ). Keywords are predefined identifiers and cannot be redeclared. You cannot use these words to identify variables and functions in your programs. You must express keywords in lowercase letters. For a list of the PDP-11 C keywords, refer to Section 2.11.

PDP-11 C is a case-sensitive language. You can declare variables in any mixture of upper- or lowercase letters. The case of the references must match the case of the variable declaration. For example, if you declare *total* as a variable, you must reference *total* . If you attempt to reference *Total* , an error occurs; the compiler does not recognize the variable name due to the initial capital letter.

**4 Statements.** The sum of 2 + 2 is stored in variable *total* . This is accomplished using a valid PDP-11 C statement. You can use any valid expression as a statement by ending it with a semicolon ( ; ). Identifier *total* is a declared variable; the equal sign ( = ) and the plus sign ( + ) are valid PDP-11 C operators; and the numbers being added are valid constants. For more information concerning the various PDP-11 C statements, refer to Chapter 3. For more information concerning the PDP-11 C operators, refer to Chapter 4.

## 2.4 Producing Input/Output

The C language includes no facilities to administer I/O. However, all implementations, including PDP-11 C, have methods that allow the programs and users to communicate. The lack of communication in Example 2-1 is inconvenient; there is no way to know if the program assigns the correct value to variable *total* . You can use a PDP-11 C Run-Time Library (RTL) function to output the value of variable *total* to the terminal.

All C compilers conforming to the ANSI Standard are accompanied by a Run-Time Library of functions and macros to perform input, output, and various tasks related to specific operating environments. The PDP-11 C Run-Time Library (RTL) provides many of the functions and macros that are included with other implementations of the C language. These functions work directly and efficiently with the host operating system environment.

PDP-11 C RTL functions are segments of object code that are accessed when external references within your program are resolved.

Before you can execute any of the example programs in this manual, you must define the libraries that the Task Builder or Linker must search to resolve references to PDP-11 C RTL functions. For general information concerning libraries, refer to Chapter 1.

A header file is a file that contains a set of definitions or declarations of related functions, types, and macros. The default file type for a header file is .H. Appendix B briefly describes each header file provided with this implementation of PDP-11 C.

For more information concerning macros, refer to Chapter 7.

For more information on the various methods of accessing PDP-11 C RTL functions, refer to the *PDP-11 C Run-Time Library Reference Manual* .

Example 2-2 shows that by using the PDP-11 C RTL function **printf** , a PDP-11 C program can print a message to the terminal.

Key to Example 2-2:

**1** The header file *stdio.h* is supplied by PDP-11 C. It defines the arguments to and the type of value returned by the Standard Library I/O functions such as **printf** .

**2** The PDP-11 C RTL function **printf** writes to the standard output file (the terminal screen). The first call to the RTL function **printf** passes a string as the argument. The second call to **printf** passes a string with special formatting characters and a variable as arguments. Within the formatting string, the percentage sign ( % ) is replaced by the value of *total* and the letter ``d'' forces the value of *total* to be expressed as a decimal number. The period ( . ) prints immediately after the value of *total* .

The output for Example 2-2 follows:

```
Here is the answer: 4.
```

If you want to print the value of *total* on a separate line, then the newline character (\n) must be added to the string. Example 2-3 shows how to output on two lines.

The output from Example 2-3 follows:

```
Here is the answer . . .
4.
```

Now that a program producing output has been presented, it is necessary to compile, link, and execute the program to see the results. Compiling a program translates the source code to object code; linking a program organizes storage and resolves external references (for example, references to PDP-11 C RTL functions); and running a program executes the image.

A file is distinguished by a file name and a file type. Choose the file name so that the file is easily identifiable to the user. The maximum number of characters allowed in the file name is dependent on the operating system:

.

RSTS/E and RT-11: maximum of six-character names plus the file type which can have a maximum of 3 characters.

.

RSX: maximum of nine-character names plus the file type which can have a maximum of 3 characters.

.

VMS: maximum of 39-character names plus the file type which can have a maximum of 39 characters.

Choose the file type to reflect the function of the file. The file

type C is the default for the PDP-11 C compiler. If the file name ADD is given to the PDP-11 C compiler, the compiler will look for the file ADD.C.
After you create and name your program, the program can be compiled, linked, and executed on VMS (using VAX-11 RSX for linking and running) as follows:

```
$ pdpcc add.c
$ mcr tkb add/cp/fp=add,lb:[1,1]cfpursx/lb
$ run add
```

Here is the answer **. . .**

```
4.
$
```

The file type OBJ is the default assigned to the object file. EXE, TSK, and SAV are the default file types assigned to image files for VMS, RSX and RSTS/E, and RT-11 and RSTS/E systems, respectively.
Use the CFPURSX library on RSX machines with FPU. Refer to Section 1.5.8.1 for information on which library to link to in other situations.

## 2.5 Controlling Program Flow

There will be occasions when you must execute one or more PDP-11 C statements given a certain condition. There will be other occasions when you must execute one or more PDP-11 C statements repeatedly, within the body of a loop, until you meet a certain condition. There are several statements in PDP-11 C that accomplish these tasks. These statements are the **if** statement, the **switch** statement, the **do** statement, and the **for** statement. For information concerning the **while** statement, another statement that loops until meeting a condition, refer to Chapter 3.

### 2.5.1 Testing for a Condition (if Statement)

When executing one or more PDP-11 C statements given a certain condition, you can use the **if** statement. Example 2-4 shows a program using the **if** statement.

Key to Example 2-4:

**1** The PDP-11 C RTL function **getchar** retrieves a character from the standard input device (the keyboard). The program pauses, waiting for the user to enter a character and to press the Return key. The function **getchar** retrieves one character and ignores any others that are entered.

**2** If the letter that the user enters is either ' a ' or ' A ' , then a message stating that the choice is correct is displayed. If any other letter is entered, then a message stating that the choice is incorrect is displayed. The equality operator (= =) compares the variable *ch* with the constants ' a ' and ' A ' . The logical OR operator ( *k* ) presents the condition to test. If there is more than one statement to be executed upon condition, then you must enclose the statements within braces ({ }). A statement or statements enclosed within braces is called a **block** or **compound statement** . The concept of blocks is important when determining the scope of variables. For more information concerning blocks, refer to Section 2.12.

The interaction between the user and the program in Example 2-4 might be as follows:

$ **run example4**
Guess which letter I'm thinking of!
**b**
You're wrong.
You'll have to try again!

### 2.5.2 Testing for Multiple Conditions (switch Statement)

The **switch** statement can perform the same task as the **if** statement does in Example 2-4, but **switch** is useful when many conditions must be tested. Example 2-5 is an example that uses the **switch** statement.

Key to Example 2-5:

**1** When using the macro **tolower** , you must include the header file *ctype.h* in the compilation process. The file *ctype.h* is located in the directory containing supplied header files. (See Table 7-1.)

In PDP-11 C, the preprocessor directives are processed by an early phase of the compiler, not by a separate program as the name preprocessor implies. Directives, unlike other PDP-11 C lines of source code, begin with a number sign ( # ). The number sign must be the leftmost nonwhite-space character on the preprocessor directive line. A preprocessor directive ends with the first new line character that follows # . Do not end preprocessor directives with a semicolon.

The header file *ctype.h* is not the only module that contains macros and definitions used by the RTL functions; there are several ways to include definitions in the program stream. For more information concerning the PDP-11 C RTL and the header files, refer to the *PDP-11 C Run-Time Library Reference Manual* .

**2** The compiler replaces the reference to the **tolower** macro with a line of PDP-11 C source code that, when the program is run, translates the value of the variable *ch* to a lowercase letter. To see the macro definition of **tolower** , print the file CTYPE.H (see Table 7-1 for the location of supplied header files on your system). For more information concerning the possible side effects of macros, refer to Chapter 7.

The output for Example 2-5 is as follows:

$ **run example5**

Guess which letter I'm thinking of!
**A**
You're right!
The **switch** statement executes one or more of a series of cases based on the value of the expression in parentheses. If the value of variable *ch* is ' a ' , then the statements following the label case ' a ' : are executed. In Example 2-5, the **tolower** macro translated all alphabetic answers to lowercase letters, so there is no need to test for uppercase letter ' A ' . When a case label is matched with the value of variable *ch* , all the statements following are executed until the compiler encounters a **break** statement (which terminates the immediately enclosing statement), a **return** statement (which terminates the enclosing function), or the end of the **switch** statement. The statements following the **default** label are executed if the value of the variable does not match any of the other case labels. For more information concerning **switch** statements, refer to Chapter 3.

### 2.5.3 Loops

In the previous examples, you could only guess once during the execution of the program. To guess another letter, you had to execute the program again. If you want to execute a segment of code repeatedly until a condition is met, you may use a loop. Some loops execute a block of statements, known as the loop body, a specified number of times. Some loops test for a condition first and then execute the body of the loop if the condition is true. Some loops execute the loop body and then test for a condition, which guarantees at least one execution of the body. In PDP-11 C, this last loop is called the **do** statement. Example 2-6 shows how to use the **do** statement to alter the letter-guessing program to repeat a segment of code until the correct answer is supplied.

Key to Example 2-6:

**1** The case label tests to see if the value of the character is a newline character (\n). The newline character is entered when you press the Return key. If it is the newline character, the character is ignored and a new character is taken from the terminal.

**2** The **while** expression at the end of the **do** statement uses the not equal to operator (!=) and translates as follows:

``while the variable ch is not equal to ' a ' ."

Sample output for Example 2-6 follows:

```
$ run example6
Guess which letter I'm thinking of!
Keep guessing until you get it!
B
You're wrong.
You'll have to try again!
A
You're right!
```

You can alternately use the **for** statement to specify the number of times to execute the loop body; in the previous examples, you can use **for** to limit the number of guesses that the user may attempt. Example 2-7 illustrates this use of the **for** statement.

Key to Example 2-7:

**1** The **for** statement controls how many times the body of the loop is executed. The first expression inside the parentheses following the keyword **for** initializes variable *i* (being used as the loop incrementor) to the value 1. The second expression establishes an upper bound; the value of *i* is not to exceed 3. The third expression establishes the increment or decrement value of *i* that will be executed after every execution of the loop body. The double plus signs (++) represent the increment operator; they increase the value of a variable by 1. The loop body is executed, and each time the value of *i* increases by 1 until the value of *i* is greater than 3.

**2** The double minus signs (- -) represent the decrement operator. The decrement operator is used in this example to subtract one from the value of *i* so that newline characters are not counted as the guess of a letter.

Sample output for Example 2-7 follows:

```
$ run example7
Guess which letter I'm thinking of!
You have three guesses. Make them count!
B
You're wrong.
You'll have to try again!
C
You're wrong.
You'll have to try again!
```

**U**
You're wrong.
Sorry, you ran out of guesses!

## 2.6 Values, Addresses, and Pointers

In PDP-11 C, every variable has two types of values: a memory location and a stored object. An **lvalue** is the variable's address in memory, and an **rvalue** is the stored object. Consider the following example:

```
put_it_here = take_this_object;
```

This assignment statement is not very different from statements in other programming languages, but think about the differences between locations in memory and objects stored in memory. This assignment takes the rvalue of *take_this_object* and places it in memory at the lvalue of *put_it_here* .

Consider the following PDP-11 C declarations and assignment statement:

```
int x = 2, y;
/* put_it_here = take_this_object; */
        y = x;
```

The two distinct variables have different memory locations (lvalues), but, after the assignment statement, they contain objects of the equivalent value 2.

A variable's rvalue can be many things, such as an integer, a real number, a character string, or a data structure. One variable value can be the address of another variable. When one variable points to another, the first one's rvalue will be the second's lvalue.

A declaration of a variable whose rvalue is a pointer to another variable is as follows:

```
int *pointr;
```

The indirection notation (

```
*
```

) specifies that the variable is a pointer, which in this example points to an object of data type **int** . Pointers are declared as pointing to an object of a particular data type.

You can assign the address of a variable to the pointer as follows:

```
int *pointr; /* Declarations */
int x = 10, y = 0;
pointr = &x; /* Assignment */
```

The rvalue of the variable *pointr* is the lvalue of variable *x* . In this example, the ampersand (&), which is the ``address of'' operator, translates to the following: ``take the lvalue of this variable instead of its rvalue.'' In previous examples, the rvalue of the variable on the right side of the equal sign ( = ) was taken.
Figure 2-1 shows the difference between rvalues and lvalues as illustrated in the last example.

The value of the variable *pointr* contains the address of variable *x* . Remember that the location of variables in memory and the order in which the compiler allocates them is unpredictable and left to the discretion of the compiler. After you assign an address to the pointer, you will want to use it. For example, if you want to assign the rvalue of *x* to a variable y, you can use the pointer in a PDP-11 C statement as follows:

```
y = *pointr;
```

The asterisk (

```
*
```

) is the PDP-11 C indirection operator; the object of the variable being pointed to by *pointr* is assigned to y. The indirection operator translates as follows: ``the rvalue of this variable points to some other variable, so go to that location and access the stored object.'' Figure 2-2 shows the status of the variables after you execute the last code example.

For more information concerning pointers, refer to Chapter 5.

## 2.7 Function Definitions

You may declare or define functions you wish to call or use in a PDP-11 C program. You should always declare user-defined functions before you call them. The following sections explain the rules for defining functions, but you may wish to refer to the discussion of declarations and definitions in Chapter 5 before continuing here.

In a function definition, you specify the PDP-11 C statements that execute whenever you call the function. You also specify the parameters (if any) of the function. The parameters of a function provide a means to pass data to the function. See Section 2.9 for a detailed discussion of parameters and arguments.

Example 2-8 presents two sample function definitions.

Key to Example 2-8:

**1** Program execution begins with function main. A left brace ( { ) signifies the beginning of the function body; a right brace ( } ) signifies the end of the body. The function body is any set of valid PDP-11 C statements or declarations. Often, the body includes one or more **return** statements, as shown here. A **return** statement can specify an expression whose value is returned to the calling function. If the expression is omitted, the returned value is undefined in the calling function. If the **return** statement is not included, the function terminates when the right brace is encountered, and its return value is undefined. For more information concerning the **return** statement, refer to Chapter 3.

**2** This statement begins a new function identifier, **lower** , that returns an integer; function lower accepts the single integer parameter c_up.

For more information concerning the PDP-11 C operators used in the previous example, refer to Chapter 4.

### 2.7.1 Main Function and Function Identifiers

The execution of a PDP-11 C program must begin at the function whose identifier is main. In Example 2-8, the main function physically precedes the function lower, but the two

function definitions can appear in the reverse order.
Function names have compile-time scope rules that are different from those that apply to other identifiers. Any valid function identifier followed by a left parenthesis is declared implicitly as the name of a function whose storage class is external and whose return value is of the data type **int** . A function prototype declares a user-defined function before it is called. For more information concerning scope and storage classes, refer to Chapter 6.

### 2.7.2 Parameter List Declarations

There are two methods to declare function parameters. The preferred method of declaring parameter data types is shown in the following code example:

```
int lower( int c_up )
{
   .
   .
   .
```

For instance, your function definition may appear as follows:

```
int function_name( int lower, int upper, int temp, char x, float y )
{
   .
   .
   .
```

When you use the function prototype format in a function definition, you must supply both an identifier and a data type specification for each parameter. If you do not, PDP-11 C will generate an error message.
In a function prototype definition, you must use the keyword **void** to specify an empty parameter list.
An example of the use of the **void** keyword follows:

```
char function_name( void )
{ return 'a'; }
```

The following example shows a second method of declaring function parameters. This method does not allow parameter type-checking and is considered obsolete.

```
lower( c_up )
int c_up;
{
   .
   .
   .
```

To make your code concise, you may list the data types of the function parameters within the parameter list. If you use this method, your function definition also serves as a function prototype. See Section 2.8.1 for more information concerning the effect of function prototypes.

### 2.7.3 Function-Return Data Types

By default, all PDP-11 C functions return objects of data type **int** . In Example 2-8, function lower returns an integer to the function main, using the **return** statement.

If you define a function that returns anything other than an integer, you need to specify the function-return data type in the function definition. The following example shows the definition of a function returning a character:

```
char letter( int param1, char param2, int *param3)
{
   .
   .
   .
   return param2;
}
```

If a function does not return a value, or if you do not call the function within an expression that requires a value, you can define the function as type **void** . The presence of the **void** keyword in a function declaration causes an error to be generated under the following conditions:

- If the function returns a value
- If you call the **void** function in an expression that requires a return value

The following example shows how to use the **void** keyword to specify a function without a return value:

```
void message( char *s )
{
   printf("%s\n",s);
   printf("Stop making sense!");
   return;
}
```

### 2.7.4 Variable-Length Parameter Lists

If you decide to define a function with a variable-length parameter list, you can use an ellipsis ( **. . .** ) in a function

prototype declaration to designate the variable-length portion of the parameter list, as follows:

```
function_name( int lower, int upper, int temp, char x, float y, ... )
{
   .
   .
   .
}
```

Within the function body, use the **va_start** , **va_arg** , and **va_end** macros defined in the *stdarg.h* header file to access the argument list passed to the function. These macros provide a portable means of accessing variable-length argument lists. For more information concerning these macros, refer to the *PDP-11 C Run-Time Library Reference Manual* .

When using ellipses for variable-length argument lists, you must have at least one argument preceding the ellipses. The following definition is legal:

```
function_name( double lower, ... )
{
   .
   .
   .
}
```

The following definition is not legal:

```
function_name( ... )
{
   .
   .
   .
}
```

Example 2-9 shows a variable argument construct:

The output for Example 2-9 follows:

```
3 4 string1
3.000000 string2 4
```

---

**Note**

---

If you use function prototypes, you should use ellipses ( **. . .** ) within parameter lists so that the

compiler does not typecheck the trailing parameters. See Section 2.8.1 for more information concerning function prototypes.

---

## 2.8 Function Declarations

You may call a function without declaring it if the function's return value is an integer. If the return value is anything else, the function may have to be declared. Example 2-10 illustrates when you need to declare a function.

Key to Example 2-10:

**1** Since the location of the function definition is after the main function in the source code, and since function lower has a return type of **char** , you have to declare the function before calling it.

In a function declaration, you can use the keyword **void** to specify an empty argument list, as follows:

```
int main()
{
   char function_name( void );
        .
        .
        .
}
char function_name( void )
{ }
```

If the function does not return a value, you can use the **void** keyword in the declaration, as follows:

```
int main()
{
   void function_name( );
        .
        .
        .
}
void function_name( )
{ }
```

If you specify argument data types or **void** in the parameter list of a function declaration as shown in the following example, PDP-11 C treats the function declaration as a function prototype for the scope of the declaration:

```
int main()
{
   char function_name( int x, char y );
```

```
        .
        .
        .
}
```

Since the declaration is within the scope of main, PDP-11 C uses the function declaration as a function prototype only within main. See Section 2.8.1 for more information concerning function prototypes.

### 2.8.1 Function Prototypes

A function prototype is a function declaration that specifies the data types of its arguments in the identifier list. PDP-11 C uses the prototype to ensure that any function definition, and all declarations and calls within the scope of the prototype, contain the correct number of arguments or parameters, and that each argument or parameter is of the correct data type.

In each compilation unit in your program, determine where to place the corresponding function prototype. The position of the prototype determines the prototype's scope; the scope of the function prototype is the same as the scope of any other declaration. PDP-11 C checks all function definitions, declarations, and calls from the position of the prototype to the end of its scope. Misplacing the prototype so that a function definition, declaration, or call occurs outside the scope of the prototype may cause an undefined function error or cause the definition of distinct function prototype declarations.

Corresponding function prototype declarations are identical to the declarative part of a function definition that specifies data types in the identifier list. All function declarations not followed by a defining block end with a semicolon ( ; ). The following code example is a prototype that corresponds with either of the previous sample function definitions:

```
char function_name( int lower, int *upper, char (*func)(), double y );
```

When declaring a function prototype that is not followed by a defining block, you do not need to use the same parameter identifiers as in the function definition. If you choose, you do not need to specify any identifiers in the prototype declaration. The scope of the identifiers within function prototypes exists only within the identifier list; you are free to use those identifiers outside of the prototype.

You can use the following function:

**Function:**

```
char function_name( int lower, int *upper, char (*func)(), double y )
{ }
```

For any of the following prototypes:

**Prototypes:**

```
char function_name( int lower, int *upper, char (*func)(), double y );
char function_name( int a, int *b, char (*c)(), double d );
char function_name( int, int *, char (*)(), double );
```

You can specify variable-length argument lists in function prototypes by using ellipses. You must have at least one argument in the list preceding ellipses. The following example illustrates the specification of a variable-length argument list:

```
char function_name( int lower, ... );
```

You cannot omit data type specifications in a function prototype. Also, you cannot have a variable-length argument list that is not preceded by at least one argument. The following prototypes are not legal and their use generates appropriate error messages:

```
char function_name( lower, *upper, char (*func)(), float y );
char function_name( , , char (*func)(), float y );
char function_name( ... );
```

Use function prototypes to ensure that all corresponding function definitions, declarations, and calls within the scope of the prototype conform to the number and type of parameters specified in the prototype. A function prototype is considered in scope only if a function prototype declaration is specified within a block enclosing the function call or at the outermost level of the source file. If a prototype is in scope, the automatic widening of **float** arguments to **double** is not performed. However, the automatic widening of **char** argument to **int** is performed. If the number of arguments in a function definition, declaration, or call does not match the prototype, the statement generates the appropriate error message.

If the data type of an argument in a function call does not match the prototype, PDP-11 C attempts to perform conversions. If the mismatched argument is assignment compatible with the prototype parameter, PDP-11 C converts the argument to the data type specified in the prototype, according to the parameter and argument conversion rules (see Section 2.9).

If the mismatched argument is not assignment compatible with the prototype parameter, the action generates the

appropriate error message and the results are undefined.
The syntax of the function prototype is designed such that PDP-11 C can provide effective compile-time error detection on the number and types of parameters to the function. To use prototype checking for PDP-11 C Run-Time Library function calls, you can include the header files appropriate for the RTL functions used in your program. You place the **#include** preprocessor directives at the top of any applicable compilation units.
For more information concerning the RTL prototype include modules, refer to the *PDP-11 C Run-Time Library Reference Manual* . For more information concerning preprocessor directives, refer to Chapter 7. For more information concerning compilation units and scope, refer to Chapter 8.

## 2.9 Using Parameters and Arguments

PDP-11 C functions can exchange information by means of parameters and arguments. (In this manual, the term **parameter** denotes the variable within parentheses named in a function definition; the term **argument** denotes an expression that is part of a function call.) In Example 2-8, function lower has the single parameter c_up. When this function is called from the function main, argument c is evaluated and passed to function lower.

The following rules apply to parameters and arguments of PDP-11 C functions:

- The number of arguments in a function call must be the same as the number of parameters in the function definition, except if the definition includes the ellipsis. A function may or may not have arguments.
- In PDP-11 C, the maximum number of arguments (and corresponding parameters) is 255 for a single function.
- Arguments are separated by commas. However, the comma is not an operator in this context, and the arguments may be evaluated by the compiler in any order. Do not expect function calls or expressions in the argument list to be evaluated in any particular order.
- In PDP-11 C, arguments are passed by value; that is, when a function is called, the parameter receives a copy of the argument's value, not its address. This rule applies to all scalar variables, structures, and unions passed as arguments. If a function modifies the value of its argument, those arguments will be unchanged in the calling function.
- Because arguments can be addresses or pointers, a function can use addresses to modify the values of variables defined in or within the scope of the calling function.
- The types of evaluated arguments must match the types

of their corresponding parameters when a function prototype is in scope. In the presence of the ellipsis, any undeclared parameters are also subject to the following type conversions. When a function is called and a function prototype is not in scope, PDP-11 C does not compare the types of the arguments with those of the corresponding parameters so it does not generally convert the arguments to the types of the parameters. Instead, all of the expressions in the argument list are converted according to the following conventions:

- Any arguments of type **float** are converted to **double** .
- Any arguments of types **char** or **short** are converted to **int** .
- Any arguments of type **unsigned char** are converted to **unsigned int** .
- Any function name appearing as an argument is converted to the address of the named function. You must declare the corresponding parameter as a pointer to a function, which evaluates to a value of the same data type as the function.
- Any array name appearing as an argument is converted to the address of the first element of the array. You must declare the corresponding parameter either as an array of the given type or as a pointer to the given type. Since character-string constants are declared implicitly as arrays of characters, this rule also applies to the use of string constants as arguments.

No other default conversions are performed on arguments. If you know that a particular argument must be converted to match the type of the corresponding parameter, use the cast operator. For more information concerning the cast operator, refer to Chapter 4.

.

If you declare variables in the parameter declaration section that do not exist in the parameter list, the appropriate error message will be issued.

.

If you do not declare parameter types, they are implicitly declared to be of type **int** .

.

The passing of parameters is affected by the function's

linkage. For more information, see the **#pragma linkage** section in Chapter 7.

### 2.9.1 Function and Array Identifiers as Arguments

You can use a function identifier without parentheses and arguments. In this case, the function identifier evaluates to the address of the function. This method of referencing is useful when passing a function identifier in an argument list. You can pass the address of one function to another as one of the arguments.

If you wish to pass the address of a function in an argument list, the function must either be declared or defined, even if the return value of the function is an integer. Example 2-11 shows when you must declare user-defined functions and how to pass functions as arguments.

Key to Example 2-11:

**1** You can pass function x in an argument list, since its definition is located before the function main.

**2** You must declare function y before you pass the function in an argument list, since its function definition is located after the main function.

**3** When you pass functions as arguments, do not include the parentheses. Similarly, when you specify arrays, do not include subscripts.

**4** When declaring parameters which represent functions, declare them as pointers to functions. For convenience, declarations of parameters, which are functions or arrays, can be declared as ordinary function or array declarators; the compiler automatically converts them to pointers.

PDP-11 C treats array parameters in the same way.

For more information about pointers, addresses, and dereferencing, see Chapter 5.

### 2.9.2 Passing Arguments to the Function Main

The function main in a PDP-11 C program can accept arguments from the command line from which it was invoked. The syntax for such a main function is as follows:

```
int main(int argc, char *argv [])
```

In this syntax, parameter *argc* is the count of arguments present in the command line that invoked the program. Parameter *argv* is an array of character strings of the

arguments.
In the main function definition, the parameters are optional. However, you can access only the parameters that you define. You can define function main in either of the following ways:
int main(void)
int main(int argc, char *argv[])
On RSX systems, you can pass a command line by using RUN/COMMAND: (DCL) or RUN/CMD= (MCR); otherwise you must install the program as described below. On the RSX and RSTS/E operating systems, arguments may be passed from the command line if the program is installed. On RSX systems, install your program using the command INSTALL. The following example shows how to install the program, ECHO.C (only three characters may be used for the name):
$ **install echo/task=...ech**
On RSTS/E systems, you install ECHO.C as a CCL as in the following example:
$ **define/command echo**
On RT-11, you need only run the program with the arguments to main following the program name.
Example 2-12 shows ECHO.C, which displays the command-line arguments that were used to invoke it.

Sample output for Example 2-12 follows:
$ **ech Long "Day's" "Journey into Night"**
program: ech
argument 1: long
argument 2: Day's
argument 3: Journey into Night
At run time, PDP-11 C converts most arguments on the command line to lowercase. PDP-11 C internally parses and modifies the altered command line to make argument access on PDP-11 C compatible with C programs developed on other systems.
All alphabetic arguments in the command line are delimited by spaces or tabs. Arguments with embedded spaces or tabs must be enclosed in quotation marks ( " " ). Uppercase characters in arguments are converted to lowercase, but arguments within quotation marks are left unchanged.

## 2.10 Identifiers

Identifiers consist of up to 31 letters, digits, and underscore characters ( _ ). When compiling using the /NOSTANDARD qualifier, the dollar sign ( $ ) may also be used in identifiers. When using global names, all underscore characters ( _ ) are converted to periods ( . ). If you create identifiers with a length of more than 31 characters, the compiler ignores all characters after the 31st character. If the identifier will be seen by the RT-11 Linker or the RSX Task Builder, as in a declaration with **[extern]** or **#module** , the first 6 characters must be unique after conversion of all letters to uppercase and must obey the external environment's rules.

The first character must not be a digit, and to avoid conflict with names used by PDP-11 C, should not be an underscore character. PDP-11 C uses a preceding underscore to identify implementation-specific macros, keywords, constants, and functions.

Upper- and lowercase letters specify different variable identifiers. For example, the compiler interprets abc, aBc, and ABC as different variable names.

The dollar sign and underscore characters within identifiers for global symbols are reserved for use by Digital. Identifiers that contain dollar signs are not portable and are not accepted when using the /NOSTANDARD compiler switch.

Digital recommends that you use the following conventions if practical:

.

Avoid using underscores as the first character of your identifiers.

.

Use uppercase letters in identifiers if they are constants that are given values by the **#define** directive.

.

In all instances of a global name, use identical spellings and case. All names that become part of the Task Builder or RT-11 Linker's global symbol table are represented there in uppercase. Consider the following examples:

```
int globalvalue c$errn = 0;
globalvalue C$ERRN;
```

The compiler will consider these to denote different global

names; however, uppercase forms for both are passed to the RT-11 Linker or Task Builder, potentially causing errors when the program is linked or executed. For more information concerning the **globalvalue** specifier, refer to Chapter 6.

.

Use lowercase or mixed case letters for all other identifiers.

## 2.11 Keywords

Keywords are predefined identifiers. They cannot be redeclared. They identify data types, storage classes, and certain statements in PDP-11 C. Note that many conventional words in PDP-11 C programs are not keywords and can be redeclared. The notable examples are the names of functions, including main and the functions found in libraries that accompany the PDP-11 C compiler. Keywords must be expressed in lowercase letters. Table 2-1 lists the PDP-11 C keywords.

To maintain VAX C compatibility, do not redefine VAX C or C++ keywords. The VAX C keywords are shown in Table 2-2.

## 2.12 Blocks

A block is a compound statement surrounded by braces ({ }). You can use a block when the grammar of PDP-11 C requires a single statement. The common cases are the bodies of functions and **if** , **for** , **do** , **switch** , and **while** statements. Note that this definition of a block may conflict with its definition in other languages. In PDP-11 C, the terms block and compound statement are equivalent.

A block may also contain declarations. If it does, any declarations of **auto** , **register** , or **static** variables are local to the block. Example 2-13 presents nested blocks and the differences in the scope of declared variables.

Key to Example 2-13:

**1** In the outer block of the function main, variable *i* used in the **if** statement is an integer. The default storage class for this variable is **auto** .

**2** Within the block in the **if** statement, variable *i* is a single-precision floating-point value with the default storage class **auto** .

Sample output for Example 2-13 follows:

```
Inner-block variable i:30000001023.999997
Outer-block variable i:1
```

If initialization is specified for any **auto** or **register** variables in a nested block, it is performed each time control reaches the block normally. Such initializations are not performed if a **goto** statement transfers control into the middle of the block or if the block is the body of a **switch** statement. For more information concerning data types, refer to Chapter 5. For more information concerning scope and storage classes, refer to Chapter 6.

### 2.13 Comments

Comments, delimited by the character pairs (/*) and (*/), can be placed anywhere that white space can appear. The text of a comment can contain any characters except the close-comment delimiter (*/). Comments cannot be nested.

## 2.14 Lexical Continuation

You can lexically continue a line in your program by placing a backslash (\) as the last character in the line. You can specify lexical continuation at any point, except within a trigraph. This feature is useful for specifying long preprocessor directive lines (such as **#define** ) and for long string literals. For information on string literal concatenation, see Section 2.15.

The following is an example of lexical continuation:

```
#define RESET() \
        ( \
        a = 0, \
        b = 0, \
        c = 0, \
        d = 0, \
        )
```

### 2.15 String Literal Concatenation

You can concatenate string literals by placing the two string literal tokens adjacent to each other. You can place any number of space and tab characters or comments between the two string literals. You can also use string literal concatenation across separate lines.

This feature is useful for defining long string literals. You can concatenate any number of string literals, restricted only by memory and address space limitations of the target environment. A terminating null byte is placed at the end of the resulting concatenated string, but not at the end of the individual string literals before concatenation.

The following is an example of a string literal before concatenation:

```
"This literal con"
        "catenates."
```

After concatenation, the string literal becomes the following:

"This literal concatenates."

[next] [previous] [contents]

## 2.16 Trigraphs

Trigraphs are 3-character sequences used to represent specific characters that are not supported by certain terminals, printers, and display devices.

**Note**

Most Digital terminals, printers, and display devices support all characters required for programming in C. Generally, use of trigraphs are required only with devices that are not produced by Digital or when using alternate character sets (see Section 7.7.1.)

All trigraph sequences begin with two question marks (??) followed by a universally supported character that most closely resembles the unsupported character. Table 2-3 shows the defined trigraph sequences and the characters into which each trigraph sequence is translated. Trigraph sequences are translated in all contexts, including sequences within string literals and character constants. Only the trigraph sequences shown are translated; two question marks (??) followed by any other character are not translated.

[next] [previous] [contents]

---

## 3. Statements

This chapter describes the statements in the PDP-11 C programming language. Statements are executed in the sequence in which they appear in a program, except as indicated.

### 3.1 The Labeled Statement

Labels are identifiers used to flag a location in a program and to be the target of a **goto** statement.

The syntax of a label is as follows:

identifier:

Any statement can be preceded by a label. The scope of the label is the current function body. Labels have their own name space and, therefore, do not interfere with variable names. Labels are used only as the targets of **goto** statements.

### 3.2 Compound Statement

A compound statement or block is a group of two or more statements enclosed within braces ({ }). You can use a block wherever the grammar of PDP-11 C requires a single statement. It allows more than one statement to appear where a single statement is required by the language. The common cases are bodies of functions and **if** , **for** , **do** , **switch** , and **while** statements. The following code is an example of a block:

```
if (a == 2)
{
   int x = 5;
   z = 1;
   if (y < x)
        funct(y, z);
   else
        funct(x, z);
}
```

The block contains optional declarations followed by an optional list of statements, all enclosed in braces. If you include declarations, the variables they declare are local to the block, and for the rest of the block they supersede any previous declaration of variables of the same name. Inside blocks, you can initialize variables whose declarations include the **auto** , **register** , **static** , or **globaldef** storage class specifiers.

A block is entered either normally when control flows into it, or when a **goto** (or **switch** ) statement transfers control to a label in the block itself. Automatic storage for all blocks within a function is allocated at function entry. If a block is entered through a **goto** (or in a **switch** ) statement, initialization of variables defined within that block will not occur. For more information concerning storage classes, refer to Chapter 6.

All function definitions are compound statements. The compound statement following the parameter declarations in a function definition is called the function body.

### 3.3 The Null Statement

Use null statements to provide null operations in situations where the grammar of the language requires a statement, but the program requires no work to be done.
The syntax of the null statement follows:

```
;
```

You need to use the null statement with the **if** , **while** , **do** , and **for** statements in cases where the grammar requires a statement body but the program requires no functional operation. The most common use of this statement is in loop operations, where all the loop activity is performed by the test portion of the loop. For example, the following statement finds the first element of an array known to have a value of 0:

```
for(i=0; array[i] != 0; i++)
    ;
```

Refer to Section 3.2 and Section 3.6 for more information concerning the statements mentioned here.

[next] [previous] [contents]

---

### 3.4 The Expression Statement

You can use any valid expression as a statement by terminating it with a semicolon. The following is an example of an expression used as a statement:

```
i++;
```

This statement increments the value of the variable *i* .

For more information concerning the valid PDP-11 C expressions, refer to Chapter 4.

### 3.5 Selection Statements

A selection statement selects from a set of statements depending on the value of a controlling expression. The following sections describe the **if** statement and the **switch** statement.

#### 3.5.1 The if Conditional Statement

An **if** statement executes a statement depending on the evaluation of an expression and can optionally be written with an **else** clause.

The syntax of the **if** statement follows:

```
if ( expression )
statement
else
statement
```

The **else** clause and following **statement** are optional.

An example of the **if** statement follows:

```
if (i < 1)
   funct(i);
else
   {
       i = x++;
       funct(i);
   }
```

If the evaluated expression within parentheses is true (in the example, if variable *i* is less than 1), then the statement following the evaluated expression executes; the statement following the keyword **else** does not execute. If the evaluated expression is false, then the statement following the keyword **else** executes.

All logical operators define a true result to be nonzero. Therefore, any scalar expression in the conditional statement can be a logical expression with predictable results (true or false; nonzero or zero).

An ambiguity occurs when you omit an **else** clause from a nested **if** sequence. This is resolved by matching the **else** clause with the most recent **if** statement that does not have an **else** clause. For example, in the following example, the **else** matches with the inner **if** :

```
if (x > y)
```

```
    if (x > z)
        z = 1;
    else
        z = 0;
```

However, you can control the matching by using braces in the following way:

```
if (x > y)
{
    if (x > z)
        z = 1;
}
else
    z = 0;
```

### 3.5.2 The switch Statement

The **switch** statement executes one of a series of cases, based on the value of the expression.

The syntax of the **switch** statement follows:

switch ( expression )
statement

The usual arithmetic conversions are performed on the expression, but the result must be an integer type. For more information concerning data types, refer to Chapter 5. The statement is typically a compound statement, within which one or more **case** labels prefix the statements that execute if the expression matches the **case** .

The syntax for a **case** label and expression follows:

case constant-expression :

The constant expression must also be of type **int** . No two **case** labels can specify the same value.

Only one statement in the compound statement can have the following label:

default :

The syntax for a **default** statement follows:

default : statement

The **case** and **default** labels can occur in any order. Note that each case flows into the next unless explicit action is taken, such as a **break** statement. When the **switch** statement is executed, the following sequence takes place:

1. The **switch** expression is evaluated and compared with the constant expressions in the **case** labels.
2. If the expression's value matches a **case** label, the statements following that label are executed. If the

list of statements ends with the **break** statement, the **break** terminates the **switch** statement; otherwise, execution falls through to the next set of statements. (See Example 3-1.) The **switch** statement can also be terminated by a **return** or **goto** statement; if the **switch** is inside a loop, it can be terminated by a **continue** statement. For more information concerning interrupting statements, refer to Section 3.7.

3. If the expression's value does not match any **case** label but there is a **default** case, the **default** case is executed. It need not be the last case listed. If a **break** statement does not end the **default** case and it is not the last case, the next case encountered is executed.
4. If the expression's value does not match any **case** label and there is no **default** , the body of the **switch** statement is not executed.

In general, the **break** statement should be used to ensure that a **switch** statement executes as expected. Example 3-1 uses the **switch** statement to count blanks, tabs, and newlines entered from the terminal.

Key to Example 3-1.

**1** A series of **case** labels is used to increment the counters.

**2** The **break** statement causes control to go back to the **while** loop every time a counter increments. The program automatically passes control to the **while** loop if none of the case statements is selected.

Sample execution and output for Example 3-1 follows:

$ **run example.exe**
**Every good boy.**
**The quick brown fox.**
**Line with 2**

---

---

**Tab**

---

---

---

---

**Tab**

---

---

**tabs.**

At this point, enter the end-of-file character, Ctrl/Z. The program prints the following:

```
Blanks Tabs Newlines
      7 2 3
```

If you omit the **break** statements, the program prints the following:

```
Blanks Tabs Newlines
     12 2 5
```

Without the **break** statements, each case drops through to the next case. The number shown for tabs happens to be right, because the tabs case is first in the **switch** statement and is executed only if ch = = ' \t ' . Notice that the number shown for newlines is the correct number plus the number of tabs, and the number shown for blanks is the total of all three cases. Any statements that appear within a **switch** statement before the first **case** label or default label are ineffective. Consider the following example:

```
switch (ch)
   {
       int x = 1; /* Ineffective initialization */
       printf("%d", x); /* This first printf won't be executed */
       case 'a' :
       { int x = 5; /* Proper initialization */
          printf("%d", x);
          break;
       }
       case 'b' :
          .
          .
          .
   }
```

In the previous example, if the variable *ch* equals ' a ' , then the program prints the value 5. The initialization outside of the case label is ineffective, and the printf preceding case ' a ' cannot execute because it cannot be reached.

### 3.6 Iteration Statements (Looping)

An iteration statement causes a compound statement called the **loop body** to be executed until an expression evaluates to false. In PDP-11 C, the **for** and **while** statements evaluate an expression and then execute the body of the loop. Some loops execute the loop body and then evaluate the expression, which guarantees at least one execution of the body; in PDP-11 C, the **do** statement executes this loop. The following sections describe the **for** , **while** , and **do** statements.

#### 3.6.1 The while Statement

The **while** statement evaluates an expression and executes a statement (the loop body) zero or more times, until the expression evaluates to false.

The syntax of a **while** statement follows:

while ( expression )
statement

An example of the **while** loop follows:

```
x = 0;
while (x < 10)
   {
        array[x] = x;
        x++;
   }
```

This statement tests the value of the variable x; if variable x is less than 10, it assigns x to the *x* th element of the array and then increments the variable x. If the expression in parentheses evaluates to false, the loop body does not execute, and control passes to the statement following the **while** loop.

#### 3.6.2 The for Statement

The **for** statement evaluates three expressions and executes a statement (the loop body) until the second expression evaluates to false. The **for** statement is particularly useful for executing a loop body a specified number of times.

The syntax for the **for** statement is as follows:

for ( expression-1 ; expression-2 ; expression-3 )
statement

The **for** statement executes the loop body zero or more times. It uses three expressions as shown. Semicolons ( ; ) are used

to separate the expressions; notice that a semicolon does not follow the last expression. A **for** statement executes the following steps:

1. Expression-1 is evaluated only once, before the first iteration of the loop. It usually specifies the initial values for variables.
2. Expression-2 is a logical expression that determines whether or not to terminate the loop. Expression-2 is evaluated before each iteration. If the expression evaluates to false, the **for** loop body does not execute and control passes to the statement following the **for** loop. If the expression evaluates to nonzero, the body of the loop is executed.
3. Expression-3 is evaluated after each iteration. It usually specifies increments or decrements for the variables initialized by expression-1.
4. Iterations of the **for** statement continue until expression-2 produces a false (zero) value, or until some statement such as **break** or **goto** causes control to be transferred elsewhere.

An example of the **for** loop follows:

```
for (x=0; x<10; x++)
    array[x]=x;
```

This statement initializes the variable *x* to 0. It then tests if the value of *x* is less than 10, and if the expression evaluates to nonzero, assigns the value of *x* to the *x* th element in the array. It then increments the variable *x* .

The **for** statement is equivalent to the following code:

```
expression-1;
while ( expression-2 )
{
statement
expression-3;
}
```

Any of the three expressions in a loop can be omitted. If expression-2 is omitted, the test condition is always true; that is, the **while** in the expansion becomes **while** (x), where x is not equal to zero. If either expression-1 or expression-3 is omitted from the **for** statement, that expression is effectively ignored.

The following statement illustrates a loop that will be infinite unless the statement body executes a **break** , **return** , or **goto** .

```
for (;;) statement
```

### 3.6.3 The do Statement

The **do** statement executes a statement (the loop body) one or more times, until the expression in the **while** clause evaluates to false.

The syntax for the **do** statement follows:

```
do
statement
while ( expression ) ;
```

An example of the **do** statement follows:

```
x=0;
  do
    {
     array[x]=x;
     x++;
    }
  while(x<10);
```

The statement is executed at least once, and the expression is evaluated after each subsequent execution of the loop body. If the expression is true, the statement is executed again.

### 3.7 Jump Statements

This section describes the statements you use to break to another statement. These statements are primarily used to exit **switch** statements and loops.

#### 3.7.1 The goto Statement

The **goto** statement transfers control unconditionally to a labeled statement, where the label identifier must be located in the scope of the function containing the **goto** statement.
The syntax of the **goto** statement follows:
goto identifier;
An example of the **goto** statement follows:

```
#include <stdio.h>
int main ()
{
      int i;
      for (i=0; i<4; i++)
      {
            printf("My number is %d.\n", i);
            if (i == 2)
               goto even_number;
            else
               printf("I have an odd number.\n");
      }
      even_number:
            printf("I have an even number.\n");
}
```

Sample output follows:

```
$ run goto.exe
My number is 0.
I have an odd number.
My number is 1.
I have an odd number.
My number is 2.
I have an even number.
```

Be careful when branching into a block using the **goto** statement. When a **goto** statement branches into a block, initialization of automatic variables declared in that block (and any enclosing blocks between the **goto** statement and block containing the label) will not be performed.

### 3.7.2 The continue Statement

The **continue** statement passes control to the end of the immediately enclosing **while** , **do** , or **for** statement.
The syntax for the **continue** statement follows:

```
continue;
```

The **continue** statement is equivalent to the **goto** label statement, shown here, for each of the looping statements in the syntax examples that follow:

```
while( . . . ) do for( . . . ; . . . ; . . . )
   { { {
   . . .
   . . .
   . . .
   goto label; goto label; goto label;
   . . .
   . . .
   . . .
   label: label: label:
   ; ; ;
   } } }
                              while( . . . );
```

In the preceding syntax examples, a **continue** statement passes control to a location referred to by label. The **continue** statement is intended only for loops, not for **switch** statements. A **continue** inside a **switch** statement that is inside a loop causes continued execution of the enclosing loop after exiting from the body of the **switch** statement.
An example of the **continue** statement follows:

```
#include <stdio.h>
int main ()
{
      char c;
      while ((c = getchar()) != EOF)
            if (c == '\t') /*Skips over tabs*/
                 continue;
            else putchar(c);
}
```

Sample output follows:

```
$ run continue.exe
Skip over any tabs
```

**Tab**

**Tab**

**that I type.**

Skip over any tabs that I type.
Enter the end-of-file character, Ctrl/Z to exit the program.

### 3.7.3 The break Statement

The **break** statement terminates the immediately enclosing **while** , **do** , **for** , or **switch** statement. Control passes to the statement following the loop or switch body.
The syntax for the **break** statement is as follows:

```
break;
```

An example of the **break** statement follows:

```
#include <stdio.h>
int main ()
{
	int c;
	while (c = getchar())
	{
		if (c == '\n')
			break;
		putchar(c);
	}
}
```

Sample output follows:

```
$ run break.exe
The program will terminate when I press return.
The program will terminate when I press return.
$
```

### 3.7.4 The return Statement

The **return** statement causes a return from a function, with or without a return value.
The syntax of the **return** statement follows:
return [expression];
An example of the **return** statement follows:

```
#include <stdio.h>
int main ()
{
	int x = 0;
	int add_one(int i);
	printf("%d\n", x);
	x = add_one(x);
	printf("%d\n", x);
}
int add_one(int i)
{
	i = i + 1;
	return i;
}
```

Sample output follows:

```
$ run return.exe
0
1
$
```

The compiler evaluates the expression (if you specify one) and returns the value to the calling function. If necessary, the compiler converts the value to the declared type of the containing function's return value. If there is no specified return value, the value is undefined.
You can declare a function without a **return** statement to be of type **void** . You cannot have a **return** statement with an expression in a function whose return type is **void** . For more information concerning the **void** data type and function return values, refer to Chapter 2.
The value returned by the main function is passed to the operating system when the program exits.

## 4. Expressions and Operators

An **expression** is any series of symbols that PDP-11 C uses to produce a value. The simplest expressions are constants and variable names that yield a value directly. Other expressions combine operators and subexpressions to produce values.

In some instances, the compiler makes conversions so that the data types of the operands are compatible. This chapter refers to these rules as the **usual arithmetic conversions** . See Section 4.9.1 for more information concerning these rules.

This chapter discusses the following topics:

- lvalues and rvalues
- Primary expressions and operators
- PDP-11 C operators
- Unary expressions and operators
- Binary expressions and operators
- The conditional expression and operator
- Assignment expressions and operators
- The comma expression and operator
- Data type conversions

### 4.1 Addresses (lvalues) and Objects (rvalues) of Variables

A variable identifier is one of the primary PDP-11 C expressions. (See Section 4.3 for more information concerning primary expressions.) This type of expression yields a single value. However, when using the variable identifier with other operators, the expression evaluates to the variable's location in memory. The address of the variable is the variable's lvalue. The object stored at that address is the variable's rvalue. For example, PDP-11 C uses both the lvalue and the rvalue of variables to evaluate the following expression:

x = y;

The contents of variable *y* are taken and assigned to variable *x* . The expression on the right side evaluates to the variable's rvalue while the expression on the left side evaluates to the variable's lvalue in the performance of assignment.

The following syntax defines those PDP-11 C expressions that either have or produce lvalues:

lvalue ::=

identifier
primary [ expression ]
lvalue . identifier
primary -> identifier
*

expression
( lvalue )

These expressions represent, respectively:

- Identifiers of scalar variables, structures, and unions
- References to scalar array elements
- Pointers to structure and union members, except for references to fields that are not lvalues
- References to pointers (also called dereferenced pointers; an asterisk (

*

) followed by an address-valued expression)

.

Any of the above expressions, enclosed in parentheses

All lvalue expressions represent a single location in a computer's memory. Chapter 2 shows the difference between lvalues and rvalues.

### 4.2 Overview of the PDP-11 C Operators

You can use the simpler variable identifiers and constants in conjunction with PDP-11 C operators to create more complex expressions. Table 4-1 presents the set of PDP-11 C operators.

The operators fall into the following categories:

- Unary operators take a single operand.
- Binary operators take two operands and perform a variety of arithmetic and logical operations.
- The conditional operator is the only ternary operator. It takes three operands and evaluates either the second or third expression, depending on the evaluation of the first expression.
- Assignment operators assign a value to a variable, optionally performing an additional operation before the assignment takes place.
- The comma operator guarantees left-to-right evaluation of comma-separated expressions.
- Primary operators usually modify or qualify identifiers (see Section 4.3 for more information).

Table 4-2 presents the precedence by which the compiler evaluates operations. Those operators with the highest precedence appear at the top of the table; those with the lowest appear at the bottom. Operators of equal precedence appear in the same row.

Consider the following expression:

A+B*C

The identifiers B and C are multiplied first because the multiplication operator (

*

) has a higher precedence than the addition operator ( + ). The associative rule applies to each row of operators. Consider the following expression:

A/B/C

This expression is evaluated as follows because the division operator evaluates from left to right.

(A/B)/C

### 4.3 Primary Expressions and Operators

Simple expressions are called **primary expressions** ; they denote values. Primary expressions include previously declared identifiers, constants (including strings), array references, function calls, and structure or union references. The syntax descriptions of the primary expressions are as follows:

primary ::=
    identifier
    constant
    string-literal
    ( expression )

The simplest forms are identifiers, such as variable names and string or arithmetic constants. Other forms are expressions (delimited by parentheses), function calls, array references, lvalues and rvalues, and structure and union references.

#### 4.3.1 Parenthetical Expressions

An expression within parentheses has the same type and value as the same expression without parentheses. As in declarations, any expression can be delimited by parentheses to change the grouping of the operators in a larger expression.

#### 4.3.2 Function Calls

A function call is a primary expression followed by parentheses. The parentheses may contain a list of arguments (separated by commas) or may be empty. An undeclared function is assumed to be a function returning **int** . If you declare an identifier as a ``function returning **. . .** '', but use the identifier in a context other than a function call, it converts to ``the address of function returning **. . .** ''. When you pass an argument that is an array or function, specify the identifier in the argument list. The compiler passes the address of the array or function to the called routine. This means that the corresponding parameters in the called function must be declared as pointers.

The following is an example of a function declaration and a function call:

```
int f1(); /* Function declaration */
    .
    .
    .
f1(); /* Function call */
```

Consider the following declaration:

```
double atof();
```

The previous example declares a function returning **double** . You can then use the identifier **atof** in a function call as follows:

```
result = atof(c);
```

You can use the identifier **atof** in other contexts without the parentheses as follows:

```
dispatch(atof);
```

The identifier **atof** converts to the address of that function, and the address is passed to the function dispatch.

Functions can also be called by means of a pointer to a function. Consider the following pointer declaration and assignment:

```
double (*pfd)( );
    .
    .
    .
pfd = atof;
```

To call the function, you can specify the following form:

```
result = (*pfd)(c);
```

PDP-11 C also accepts a pointer to a function as shown in the following example:

```
result = pfd (c);
```

### 4.3.3 Array References

Use the bracket operators ([ ]) to refer to elements of arrays. In an array defined as having three dimensions, you can refer to a specific element within the array, as in the following example:

```
int *x; /* Pointer to integer */
int sample_array[10][5][2]; /* Array declaration */
int i = 10;
sample_array[9][4][1] = i; /* Assign value to element */
```

This example assigns a value of 10 to element sample_ array[9][4][1].

If an array reference is not fully qualified, it refers to the address of the first element in the dimension that is not

specified.
Consider the following different assignments:

```
x = sample_array[9][4]; /* Assigns the address of */
                        /* sample_array[9][4][0] to x. */
*x = 10; /* Assigns a value of ten to */
                        /* the element sample_array[9][4][0] */
                        /* which x now points to. */
```

A reference to an array name with no bracket operators is often used to pass the array's address to a function, as in the following statement:

```
funct(sample_array);
```

You can also use the bracket operators to perform general pointer arithmetic as follows:

```
addr[intexp]
```

In this example, *addr* is the address of some previously declared object (pointer-valued) and *intexp* is an integer-valued expression. The result of the expression is scaled or multiplied by the size in bytes of the addressed object. If *intexp* is a positive integer, the result is a subsequent object of this size; if *intexp* is 0, the result is the same object; if *intexp* is negative, the result is a previous object. The expressions

*

(addr + intexp) and addr[intexp] are equivalent because both expressions reference the same memory location;

*

(addr + intexp) points to the same element as addr[intexp].

### 4.3.4 Structure and Union References

A member of a structure or union can be referenced with either of two operators: the dot ( . ) or the arrow (->).
A primary expression followed by a period followed by an identifier refers to a member of a structure or union and is itself a primary expression. The first expression must be an lvalue naming a structure or union. The identifier must name a member of that structure or union. The result is a reference (if the member is a scalar) to the named member of the structure or union. The name of the desired member must be preceded by a period-separated list of the names of all higher level members. For more information concerning structures and unions, refer to Chapter 5.
A primary expression followed by an arrow (specified with a

hyphen ( - ) and a greater-than symbol (>)) followed by an identifier refers to a member of a structure or union. The first expression must be a pointer to a structure or a union. The identifier following the arrow operator must name a declared member of that structure or union. The result is a reference to the named member.

### 4.4 Unary Operators

You can form expressions by combining a unary operator with a single operand. All unary operators are of equal precedence and group from right to left. They perform the following operations:

- Negate a variable arithmetically ( - ) or logically ( ! )
- Increment (++) and decrement (- -) variables
- Find addresses ( & ) and dereference pointers ( * )
- Calculate a one's complement ( ~ )
- Force the conversion of data from one type to another (the cast operator)
- Calculate the sizes of specific variables or of types ( **sizeof** )
- Force integral promotions ( + )

#### 4.4.1 Negating Arithmetic and Logical Expressions

Consider the syntax of the following expression:

- expression

This is the arithmetic negative of expression. The compiler performs the arithmetic conversions. The negative of an **unsigned short int** is computed by subtracting its value from $2^{16}$. The negative of an **unsigned long int** is computed by subtracting its value from $2^{32}$.

Consider the following expression:

!expression
The result is the logical (Boolean) negative of the expression. If the result of the expression is 0, the negated result is 1; if the result of the expression is not 0, the negated result is 0. The type of the result is **int** . The expression can be a pointer (or another address-valued expression) or an expression of any arithmetic type.

### 4.4.2 Incrementing and Decrementing Variables

Consider the syntax of the following expression:
++lvalue
The object that the lvalue refers to in the expression is incremented before its value is used. After evaluating this expression, the result is the incremented rvalue, not the corresponding lvalue. For this reason, expressions that use the increment and decrement operators in this manner cannot appear by themselves on the left side of an assignment expression where an lvalue is needed.
Consider the syntax of the following expression:
lvalue++
The object that the lvalue refers to in the expression increments after its value is used. The expression evaluates to the value of the object *before* the increment, not the incremented variable's lvalue.
If the operand is a pointer, the address is incremented by the length of the addressed object, not by the value 1. If declared as an integer or floating point, the variable increases by the value 1.
If the lvalue points to another variable:
- -lvalue
lvalue- -
then these expressions decrement not by 1, but by the size of the addressed object. The data type of the variable determines the amount of the increment or decrement. If declared as a pointer, the variable increments or decrements by the size of the addressed object's data type. For example, if declared as a pointer to integer, the variable increments or decrements by the value 2. For example:

```
int *ip;
char *cp;
ip--; /* decremented by 2 */
--cp; /* decremented by 1 */
```

When using the increment and decrement operators, do not

depend on the order of evaluation of expressions. Consider the following ambiguous expression:

```
k = x[j] + j++;
```

Is the value of variable j in x[j] evaluated before or after the increment occurs? Do not assume which expressions the compiler will evaluate first. To avoid ambiguity, increment the variable in a separate statement.

### 4.4.3 Computing Addresses and Dereferencing Pointers (& * )

Consider the syntax of the following expression:

& identifier

The expression results in the lvalue (address) of the identifier. The ampersand operator ( & ) may not be applied to **register** variables or to bit fields in structures or unions.

When an expression evaluates to an address, as in the following example, the address is used to indirectly access the object to which the address refers:

* pointer

An expression using the indirection operator ( * ) evaluates to the object pointed to by a pointer or by an address-valued expression.

### 4.4.4 Calculating a One's Complement ( ~ )

Consider the syntax of the following expression:

~ expression

The result is the one's complement of the evaluated expression; it converts each 1-bit into a 0-bit and vice versa. The expression must be integral (an integer or character). The compiler performs necessary arithmetic conversions.

### 4.4.5 Forcing Conversions to a Specific Type (Cast Operator)

The cast operator forces the conversion of its operand to a void type, qualified scalar type, or unqualified scalar type. You can also cast to a **typedef** if it represents a scalar type.

Structures and unions may not appear in a cast operator; however, pointers to structures or unions may. The operator consists of a data type name, in parentheses, preceding the operand expression, as follows:

(type-name) expression

The resulting value of the expression converts to the named data type, as if the expression were assigned to a variable of that type. If the operand is a variable or constant, its value converts to the named type. The variable's contents do not change. The type name has the following formal syntax:

type-name ::= type-specifier abstract-declarator

abstract-declarator ::=

empty

( abstract-declarator )

*

abstract-declarator

abstract-declarator ( )

abstract-declarator [ constant-expression ]

Abstract declarators may include the brackets and parentheses that indicate arrays and function calls. However, cast operations cannot force the conversion of any expression to an array, function, structure, or union. The brackets and parentheses are used in operations such as the following example, which casts identifier P1 to ``pointer to array of **int** ."

(int (*)[]) P1

This kind of cast operation does not change the contents of P1; it only causes the compiler to treat the value of P1 as a pointer to such an array.

### 4.4.6 Calculating Sizes of Variables and Data Types (sizeof)

Consider the syntax of the following expressions:

sizeof expression

sizeof ( type-name )

The result is the size, in bytes, of the operand. In the first case, the result of **sizeof** is the size determined by the type of the expression. In the second case, the result is the size, in bytes, of an object of the named type. The syntax of type-name is the same as that for the cast operator. While you may take the size of unions and structures, you cannot cast them. For example:

```
int i;
char a[10];
i = sizeof a; /*value 10*/
i = sizeof (short int); /*value 2*/
```

See Section 4.4.5 for more information concerning the cast operator.

### 4.5 Binary Operators

The binary operators are categorized as follows:

- Additive operators: addition ( + ) and subtraction ( - )
- Multiplication operators: multiplication ( * ), modulo ( % ), and division ( / )
- Equality operators: equality ( = = ) and inequality (!=)
- Relational operators: less than ( < ), less than or equal to (<=), greater than ( > ), and greater than or equal to (>=)
- Bitwise operators: AND ( & ), OR ( | ), and XOR ( ^ )
- Logical operators: AND (&&) and OR ( *k* )
- Shift operators: left shift (<<) and right shift (>>)

The following sections describe these binary operators.

#### 4.5.1 Additive Operators (+ -)

The additive operators ( + ) and (-) perform addition and subtraction. Their operands are converted, if necessary, following the usual arithmetic conversions described in Section 4.9.1.

You can increment an array pointer by adding an integral variable to the address of an array element. The compiler calculates the size of one array element, multiplies that by the integer to obtain the offset value, and then adds the offset value to the address of the designated element. For example:

```
int arr[10];
int *p = arr;
p = p + 3; /* Increments by 2*3 */
```

You may subtract a value of any integral type from a pointer or address; in that case, the same conversions apply as for addition.

If you subtract two addresses of objects of the same type, the

result converts (divides by the length of the object) to an **int** representing the number of objects separating the addressed objects. The result of this conversion is unpredictable unless the two objects are in the same array.

---

**Note**

---

The size of objects of type **int** vary among implementations of the C language. In PDP-11 C, the data types **int** and **short** are of the same size, 16 bits.

---

### 4.5.2 Multiplication Operators ( * / %)

The multiplication operators ( * ), ( / ), and ( % ) perform arithmetic conversions, if necessary. The binary operator ( * ) performs multiplication. The binary operator ( / ) performs division. When integers are divided, truncation is toward zero.

The binary modulo operator ( % ) divides the first operand by the second and yields the remainder. Both operands must be integral. The sign of the result is the same as the sign of the quotient. If variable *b* is not 0, the following statement is always true:

```
a = (a/b)*b + a%b;
```

### 4.5.3 Equality Operators (= = !=)

The equality operators equal to (= =) and not equal to (!=) perform the necessary arithmetic conversions on their two operands. These operators produce a result of type **int** . In the following statement, the result is the value 1 if both relational

expressions have the same truth value , and 0 if they do not.
a<b == c<d
Two pointers or addresses are equal if they identify the same storage location. You can compare a pointer or address with an integer, but the result is not portable unless the integer is 0. A null pointer is considered equal to 0.
Although different symbols are used for assignment and equality, ( = ) and (= =) respectively, PDP-11 C allows either operator in some contexts, so you must be careful not to confuse them. For example, consider the following:
if (x=1) statement-1;
else statement-2;
In the previous example, statement-1 always executes, since the result of assignment x=1 delimited by parentheses is equivalent to the value of *x* , which is equal to 1, true.

---

**Note**

---

The following example shows a common error in programming comparisons:
int x;
if (x=1) /* Common error in programming comparisons */
if (1==x) /* This syntax does the comparison */
if (1=x) /* This syntax causes a compiler error */
To avoid this error, use the syntax if (1= =x) for comparisons because omitting the second equal sign causes a compiler error.

---

### 4.5.4 Relational Operators (< > <= >=)

The relational operators compare two operands and produce a result of type **int** . The result is the value 0 if the relation is false, and 1 if it is true. The operators are less than ( < ), greater than ( > ), less than or equal to (<=), and greater than or equal to (>=). The compiler performs necessary arithmetic conversions.
If you compare two pointers or addresses, the result depends on the relative locations of the two addressed objects. Pointers to objects at lower addresses are less than pointers to objects at higher addresses. If two addresses indicate elements in the

same array, the address of an element with a lower subscript is less than the address of an element with a higher subscript. The relational operators group from left to right. However, note that in the following example, the first statement compares the variable *c* with 0 or 1 (possible results of a<b); it does not mean ``if *b* is between *a* and *c* **. . .** ". The second statement shows the proper way to perform this test.

```
if (a<b<c)...
if (a<b && b<c)...
```

### 4.5.5 Bitwise Operators (& | ^)

The bitwise operators may be used only with integral operands: with variables of types **char** and with **int** of all sizes. The compiler performs the necessary arithmetic conversions. The result of the expression is the bitwise AND ( & ), XOR-exclusive OR ( ^ ), or OR ( | ) of the two operands. The compiler always evaluates all operands. Figure 4-1 shows the effects of Boolean algebra when using the bitwise operators.

In Boolean algebra, PDP-11 C compares values bit by bit. If you are using the bitwise AND, and are comparing a bit value 1 and a bit value 0, the result is 0. When using the bitwise AND, both compared bits must be 1, for the result to be 1. When using the bitwise OR, either bit value can be 1 for the result to be 1. When using the bitwise EXCLUSIVE-OR, either value, but not both, must be 1 for the result to be 1.

### 4.5.6 Logical Operators (&& | | )

The logical operators are AND (&&) and OR ( | | ). These operators guarantee left-to-right evaluation. The result of the expression (of type **int** ) is either 0 (false) or 1 (true). If the compiler can make an evaluation by examining only the left operand, it does not evaluate the right operand. Consider the following expression:

```
E1 && E2
```

The result is 1 if both its operands are nonzero, or 0 if one operand is 0. If expression E1 is 0, expression E2 is not evaluated. Similarly, the following expression is 1 if either operand is nonzero, and 0 otherwise. If expression E1 is nonzero, expression E2 is not evaluated.

```
E1 || E2
```

The operands of logical operators need not have the same type, but each must be one of the fundamental types or must

be a pointer or other address-valued expression.

### 4.5.7 Shift Operators (<< >>)

The shift operators (<<) and (>>) take two operands, both of which must be integral. The compiler performs necessary arithmetic conversions on both operands if they are not integers. The right operand is then converted to **int** , and the type of the result is the type of the left operand. Consider the following expression:

E1 << E2

The result is the value of expression E1 shifted to the left by E2 bits. The compiler clears vacated bits. Consider the following expression:

E1 >> E2

The result is the value of expression E1 shifted to the right by E2 bits. The compiler clears vacated bits if E1 is **unsigned** ; otherwise, the bits are filled with a copy of E1's sign bit. The result of the shift operation is undefined if the right operand (E2 in the previous example) is negative or if the value of E2 is greater than 16.

Figure 4-2 illustrates the shift operators.

### 4.6 Conditional Operator (?:)

The conditional operator (?:) takes three operands. It tests the result of the first operand and then evaluates one of the other two operands based on the result of the first. For example, consider the following:

E1 ? E2 : E3

If expression E1 is nonzero (true), then E2 is evaluated. If E1 is 0 (false), E3 is evaluated. Conditional expressions group from right to left. The compiler makes conversions in the following order:

1. If possible, the arithmetic conversions are performed on expressions E2 and E3, so that they will result in the same type.
2. If expressions E2 and E3 are address expressions indicating objects of the same type, the result has that type.
3. One of the E2 and E3 operands may be an address expression, and the other, the constant 0. The result has the type of the addressed object.

### 4.7 Assignment Expressions and Operators

PDP-11 C has several assignment operators. An assignment is not only an operation but is also an expression. Assignments result in the value of the target variable after the assignment. They can be used as subexpressions in larger expressions. The set of assignment operators consists of the equal sign ( = ) alone and in combination with binary operators. An assignment expression has two operands (an lvalue and an expression separated by one of these operators). Consider the following assignment expression:

E1 += E2;

This is equivalent to the following expression:

E1 = E1 + E2;

The expression E1 is evaluated once and must result in an lvalue. The type of the assignment expression is the type of E1, and the result is the value of E1 after the operation. You must delimit some expressions in parentheses if the expressions possibly contain other operators of a lower precedence. Consider the following expression:

a *= b + 1;

This is the same as the following expression:

a = a * (b + 1);

In the following simple assignment expression, the value of expression E2 replaces the previous object of E1.

E1 = E2

The following expression adds 100 to the contents of a_number[1].

a_number[1] += 100;

The result of this expression is the result of the addition and has the same type as a_number[1].

If both assignment operands are arithmetic, the right operand is converted to the type of the left before the assignment (see Section 4.9.1).

You can use the assignment operator ( = ) to assign values to structure and union members. You can assign one structure value to another as long as you define the structures to be the same type. With all other assignment operators, all right operands and all left operands must either be pointers or evaluate to arithmetic values. If the operator is (-=) or (+=), the left operand may be a pointer, and the right operand

(which must be integral) is converted in the same manner as the right operand in the binary plus ( + ) and minus (-) operations.
Using a cast, you can assign an address to an integer, an integer to a pointer, and the address of an object of one type to a pointer of another type. Such assignments are simple copy operations, with no conversions. This usage may cause addressing exceptions when you use the resulting pointers. However, if the constant 0 is assigned to a pointer, the result is a null pointer. The equality operators distinguish a null pointer from a pointer that points to any object.

---

**Note**

---

Assigning an integer to a pointer, or a pointer to an integer, is nonportable and not recommended.

---

### 4.8 Comma Expression and Operator ( , )

When two expressions are separated by the comma operator, they evaluate from left to right, and the compiler discards the result of the left expression. If you separate many expressions with commas, the compiler discards all but the result of the rightmost expression, yet the side effect of the other expressions remains.

The following example shows the use of the comma operator in both the initialization and incrementation segments of a **for** loop. Using the comma operator, multiple operations may be executed as one.

During the initialization, the variable *x* is assigned the value 0, the variable *y* is assigned the value 1, and the variable *z* is assigned the value 0. Each time the loop executes, the expression given as the incrementation expression is executed. Using the comma operator, this expression increments *x* , adds 2 to the variable *y* , and adds 10 to the value of *z* .

```
#include <stdio.h>
int main ()
{
int x,y,z;
     for (x=0, y=1, z=0; x < 3; x++, y+=2, z+=10)
     {
          printf("x: %d y: %d z: %d \n", x, y, z);
     }
}
```

The output is as follows:

```
x: 0 y: 1 z: 0
x: 1 y: 3 z: 10
x: 2 y: 5 z: 20
```

The type and value of the result of a comma expression are the type and value of the rightmost operand. The operator evaluates operands from left to right.

You must delimit comma expressions with parentheses if they appear where commas have some other meaning, as in argument and initializing lists. Consider the following expression:

```
f(a, (t=3,t+2), c)
```

This example calls the function *f* with the arguments a, 5, and c. In addition, variable *t* is assigned the value 3.

## 4.9 Data Type Conversions

PDP-11 C performs data type conversions in four situations:

- When two or more operands of different types appear in an expression (including an assignment).
- When arguments other than long integers, addresses, or double-precision floating-point numbers are passed to a function.
- When arguments that do not conform exactly to the parameters declared in a function prototype are passed to a function.
- When the data type of an operand is deliberately converted by the cast operator. See Section 4.4.5 for more information on the cast operator.

### 4.9.1 Converting Operands

The following rules (referred to as the *usual arithmetic conversions* ) govern the conversion of operands in arithmetic expressions. Although they do not specify explicit conversions at the machine-language level, the rules govern in the following order:

1. First, if either operand has type **long double** , it will convert the other operand to type **long double** .
2. Otherwise, if either operand has type **double** , it will convert the other operand to type **double** .
3. Otherwise, if either operand has type **float** , it will convert the other operand to type **float** .
4. Otherwise, the integral promotions are performed on both operands. Then the following rules are applied:
   a. If either operand has type **unsigned long int** , the other operand is converted to **unsigned long int** .
   b. Otherwise, if one operand has type **long int** and the other has type **unsigned int** , the operand of type **unsigned int** is converted to **long int** .
   c. Otherwise, if either operand has type **long int** , the other operand is converted to **long int** .
   d. Otherwise, if either operand has type **unsigned int** ,

the other operand is converted to **unsigned int** .
e. Otherwise, both operands have type **int** .

---

**Note**

---

The values of floating operands and of the results of floating expressions may be represented in greater precision and range than required by the type; the types are not changed thereby.

---

The arithmetic conversions are performed on all arithmetic operands. Some operators, such as the shift operators (>>) and (<<), require integers as operands. If one operand is of type **float** or **double** , you cannot meet this requirement.
PDP-11 C attempts to perform arithmetic in single precision. If an operand of type **float** appears in an expression, it is treated as a single-precision object unless the expression also involves an object of type **double** , in which case the usual arithmetic conversion applies.
When an operand of type **double** is converted to **float** (for example, by an assignment), the compiler rounds the operand before truncating it to **float** .
The compiler may convert a **float** or **double** value operand to an integer by assignment to an integral variable. In PDP-11 C, the truncation of the **float** or **double** value is always toward zero.
Conversions also take place between the various kinds of integers. In PDP-11 C, variables of type **char** are bytes treated as signed integers. When a longer integer is converted to a shorter integer or to **char** , it is truncated on the left; excess bits are discarded. For example:

```
int i;
char c;
i = 0xFF41;
c = i;
```

This code assigns hex 41 ( ' A ' ) to variable *c* . The compiler converts shorter signed integers to longer ones by sign extension.
Whenever the compiler combines an unsigned integer and a signed integer, the signed integer converts to **unsigned**

and the result is **unsigned** . All conversions from signed to unsigned perform an intermediate conversion to **int** . For example, the compiler converts a **char** operand to an unsigned version by first converting it to a signed **int** and then by truncating it to form the unsigned version. All conversions from unsigned to signed (such as conversions done with the cast operator) involve an intermediate conversion to **unsigned int** .

You can also add integers to pointers, in which case the integer is scaled (multiplied) by a factor that depends on the type of the object to which the pointer points. See Section 4.5.1 for more information concerning scaling pointers.

### 4.9.2 Converting Function Arguments

The data types of function arguments are assumed to match the types of the formal parameters unless a function prototype declaration is present. In the presence of a function prototype, all arguments in the function invocation are compared for assignment compatibility to all parameters declared in the function prototype declaration. If the type of the argument does not match the type of the parameter but is assignment compatible, PDP-11 C converts the argument to the type of the parameter (see Section 4.9.1). If an argument in the function invocation is not assignment compatible to a parameter declared in the function prototype declaration, PDP-11 C generates an error message.

Unless a function prototype is present, all arguments of type **float** convert to **double** ; all variables of type **char** convert to **int** ; all variables of type **unsigned char** convert to **unsigned int** ; and an array or function name converts to the address of the named array or function. The compiler performs no other conversions automatically, and any mismatches after these conversions are programming errors.

Use the cast operator to pass arguments to parameters of different types. See Section 4.4.5 for more information on the cast operator. For more information concerning the manipulation of argument lists, refer to Chapter 2.

[next] [previous] [contents]

---

## 5. Data Types and Declarations

The values of both constants and variables have **data types** .
This chapter discusses the following topics with respect to
data types:

- Constants
- Variables
- Integers
- Characters
- Floating-point values
- Pointers
- Enumerated types
- Arrays
- Structures and unions
- The **void** keyword
- The **typedef** keyword
- Interpreting variable declarations

## 5.1 Constants

You can represent data in PDP-11 C using constants. A constant is a primary expression with a defined value that does not change. You may represent a constant in a literal form, which contains the explicit numbers, letters, and operators that comprise the constant, or you may define a symbol to represent the constant value. (For more information concerning symbolic representation of constants, refer to the section on token definitions in Chapter 7.)

Constants have data types, as does all data in PDP-11 C. The data type determines the amount of storage needed and determines how to interpret the stored object or constant value. The compiler determines the data type of constants by the way in which their values are represented in the source code.

## 5.2 Variables

You can also represent data in PDP-11 C using variables whose values can change throughout the execution of the program. All variables used in a program must be declared. When you declare a variable, you specify the data type of the stored object. An **object** , in PDP-11 C, is a value requiring storage. Declarations determine the size of storage allocated, whereas definitions force the allocation of storage. See Section 5.2.1 for more information concerning data types of variables.

Unlike constants, variables can be declared and defined. Most variable declarations are also definitions because storage is allocated at that point in the program. To declare a variable, specify the data type. To define a variable, assign the proper storage class to the variable and place the variable declaration within the program structure. If you initialize a variable in the declaration, the variable is defined. For more information concerning variable definitions, scope, and storage allocation, refer to Chapter 6.

### 5.2.1 Classification of Variables

There are two kinds of variables: **scalar** and **aggregate** . Scalar variables have objects that can be manipulated arithmetically in their entirety. These objects are single characters, individual numbers, and pointers. Aggregate variables are data structures (arrays, structures, and unions) that are comprised of distinct elements (members) that you can declare to be of either a scalar or aggregate data type.

#### 5.2.1.1 Data Type Keywords

To declare or define variables, you need to know the PDP-11 C keywords associated with each data type. Table 5-1 lists the PDP-11 C data type keywords according to classification.

In the sections that follow, the keywords and operators used to declare variables of given data types are listed in the section header for ease of reference.

PDP-11 C also supports the type qualifiers **const** and

**volatile** . For information concerning these type qualifiers, refer to Chapter 6.

### 5.2.1.2 Format of a Variable Declaration

A variable declaration can be composed of the following items:

.

Data type specifiers, such as a data type or data type qualifier keyword, one structure, union, or **enum** tag, and if necessary, a **typedef** name.
Any of these gives the data type of the declared object.

.

An optional storage class keyword.
A storage class keyword affects the lifetime of a variable and determines how it is stored. If you omit the storage class keyword, there is a default storage class that depends upon the location of the declaration within the program. The positions of the storage class keywords and the data type keywords are interchangeable.

.

Declarators, which list the identifiers of the declared objects and which may contain operators that declare a pointer, function, or array of objects of the declared type.

.

Initializers for each declared object or aggregate element giving the initial value of a scalar variable or the initial values of structure members or array elements.
An initializer consists of an equal sign ( = ) followed by either a single expression or a comma-list of one or more expressions in braces.

For example, the following declaration both declares and defines the integer variable, *var_number* , which has an initial value of 10.

```
int var_number = 10;
```

The keyword **int** specifies the amount of storage needed on a PDP-11 system for an integer. The identifier *var_number* follows. The equality operator ( = ) initializes the variable with the literal constant 10; for the initialization to take place, storage is allocated and the variable is defined. Declarations must end in a semicolon ( ; ).

The variable declaration in the previous example was not difficult to interpret, but even experienced C programmers have difficulty interpreting complex variable declarations. See Section 5.13 for more information concerning the

interpretation of PDP-11 C variable declarations.

### 5.3 Integers (int, long, short, char, signed, unsigned)

Integer variables are declared with the keywords **int** , **long** , **short** , **char** , **signed** , and **unsigned** . The following is an example of an integer declaration:

```
int x;
```

Character variables are declared with the keyword **char** . An example of a character declaration with the initialization of a character variable is as follows:

```
char ch = 'a';
```

Table 5-2 specifies the sizes and ranges of integers.

In PDP-11 C, values of the **int** data type require 16 bits of storage. (This is different from VAX C where the **int** size is 32 bits.) Therefore, note that values of the **int** and **short** data types require an identical amount of storage.

The following sections describe the constants that you can assign to the integer variables.

#### 5.3.1 Integer Constants

There are three types of integer constants: decimal, hexadecimal, and octal. These integer constants consist of the following:

- **Decimals** : 0 to 9
  An integer constant is assumed to be decimal unless it begins with 0, 0x, or 0X.
- **Hexadecimals** : 0 to 9, a to f, A to F
  Use the prefix 0x or 0X to specify hexadecimal numbers.
- **Octals** : 0 to 7
  Use prefix 0 to specify octal numbers.

To specify an unsigned constant, use the suffix u or U.

To specify a **long** integer constant (4 bytes, 1 longword), use the suffix l or L, or specify a constant value which is too large for an **int** . Integer constants that exceed a longword are treated as programming errors.

Integer constants must not include a decimal point; constants with a decimal point are floating point constants.

Character constants, such as ' a ' and ' $ ' , are also valid

integer constants. Their integer values in PDP-11 C are the values of the corresponding ASCII codes.
Some examples of valid integer constants could include:

```
133L /* Long decimal integer */
1234U /* Unsigned integer constant */
0x17A /* Hexadecimal integer */
056 /* Octal integer */
'a' /* Decimal 97 */
'$' /* Decimal 36 */
```

Examples of invalid integer constants include:

```
143. /* Includes a decimal point; *
                        * Is a floating constant */
4444444444 /* Out of range for int */
77af /* Hexadecimal constants must be *
                        * prefixed with "0x" */
```

### 5.3.2 Character Constants

A character constant is an integer value, requiring 16 bits (1 word) of memory, that is enclosed in apostrophes. Character constants can be a single ASCII character, as in the following example:

```
char ch = 'a'; /* Lowercase letter 'a' is a constant *
                        * assigned to ch. */
```

The character constant ' a ' has the ASCII value of 97. If the value is that of a single character constant, the compiler stores the character in the low order byte and pads the remaining byte with a NUL character ( ' \0 ' ).
Character constants do not have to be single characters, as shown in the following example:

```
int two_bytes = 'ab'; /* This constant contains 2 characters */
printf("%c\n", two_bytes);
printf("%.2s", &two_bytes); /* String with maximum 2 characters */
```

Sample output from the program follows:

```
$ run example
a
ab
$
```

If you print variable *two_bytes* as a character, the **printf** function prints only the character located in the low order byte of the integer allocation. To print both of the characters in the word allocated to the variable, you have to print the variable as a string and pass the address of the integer variable as an argument. If you print the integer variable

as a string, be sure to specify a precision of at most 2 since you can never be sure if the next byte in the string is a terminating NUL character.
The apostrophe ( ' ) and quotation mark ( " ) are significantly different punctuation marks in PDP-11 C, indicating a character constant and a string constant, respectively.
One context in which the difference is important is in an argument list. If you specify a function argument as a string, and wish to pass a character constant, you must enclose the character in quotation marks, not apostrophes, even if the string is only 1 to 2 characters in length. See Section 5.8 for more information concerning character-string constants.

### 5.3.3 Escape Sequences

In PDP-11 C, escape sequences are character strings that represent a single printing or nonprinting character. The term escape sequences does *not* designate a string beginning with the ASCII character ESC, as in VT100 escape sequences.
Table 5-3 presents the escape sequences that specify the nonprinting characters, the apostrophe, and the backslash ( \ ).

An escape sequence, such as ' \n ' , denotes a single character.
The form ' \ddd ' is used to specify any byte value (usually an ASCII code), where the digits ddd are one to three octal digits. The octal digits are limited to 0 to 7. A common use is to specify the ASCII NUL character, as follows:

```
'\0'
```

Similarly, the form ' \xddd ' is used to specify any byte value (usually an ASCII code), where the digits ddd are used to specify one or more hexadecimal digits.
The following are examples of valid escape sequences of the form ' \ddd ' and ' \xddd ' . Both of these escape sequences are used to specify an a-umlaut (ä) on a VT2xx terminal in octal and hexadecimal digits, respectively.

```
'\344'
'\xe4'
```

If the character following the backslash in an escape sequence is illegal, the backslash is ignored; that is, the value of the character constant is the same as if the backslash were not present.

## 5.5 Pointers

Pointers in PDP-11 C are variables that contain 16-bit addresses of other objects. A pointer is declared with the asterisk notation and the data type of the object to which it points. For example:

int *px;

Identifier *px* is declared as a pointer to a variable of type **int** . The expression

*

*px* yields the object to which *px* points, therefore

*

*px* is an **int** .

Static and extern pointers are initialized to NULL unless initialized otherwise. A NULL pointer is a pointer variable that has been assigned the integer constant 0. An **auto** pointer that is not initialized will initially contain an undefined value.

An attempt to access data by means of a NULL or an uninitialized pointer may result in a hardware error or other, undefined behavior.

The valid pointer operators are assignments of pointers of the same type, adding or subtracting a pointer and an integer, subtracting or comparing two pointers to members of the same array, and assigning or comparing to zero.

For example, if *p* is a pointer to some element of an array, then *p*++ increments *p* to point to the next element. *p*+=*i* increments *p* to point *i* elements beyond where it currently points.

The unary asterisk (

*

) is also the indirection operator in PDP-11 C. The unary asterisk operates as follows:

x = *px;

This statement assigns the value of the object pointed to by *px* to variable *x* . Since the asterisk can be used in any sort of declarator, you can have pointers to scalars, to functions, to

other pointers, to structures, and so forth.
The ampersand ( & ) operator is used to take the address of an object. For example, consider the following:

```
px = &x;
```

This statement assigns the address of variable *x* to pointer *px* . After an assignment such as this, a reference to

```
*px
```

yields the value of *x* .
It is illegal to apply the ampersand operator to bit-fields or **register** variables. In both cases an error will be issued.

### 5.6 Enumerated Types (enum)

An enumerated type is a user-defined data type that is not derived from other fundamental types. Each listed enumerator is associated with an incremented integer constant starting with zero, unless the enumerators are explicitly assigned. The following example illustrates the declaration of a variable and an enumeration type or tag:

```
enum shades
   {
       out, verydim, dim, prettybright, bright
   } light;
```

This declaration defines the variable *light* to be of an enumerated type *shades* . The variable can assume any of the enumerated values.

The tag *shades* becomes the enumeration tag of the new type; *out* , *verydim* , **. . .** , *bright* are the enumeration constants with values 0 to 4. These enumerators are the constant values of the type *shades* and can be used wherever integer constants are valid.

If the tag has already been declared, you can use the tag as a reference to that enumerated type, as in the following declaration:

```
enum shades light1;
```

The variable *light1* is an object of the enumerated data type, *shades* .

An **enum** tag can have the same spelling as other identifiers in the same program, including variable identifiers and member names in structures and unions, but excluding other tag identifiers. However, **enum** constant names may not be the same as variables, functions, and **typedef** names. They can be the same as labels and tags. PDP-11 C allows forward reference to **enum** tags that have not yet been declared in the source code, but are declared further on in the program.

Internally, each enumerator is associated with an integer constant; the compiler gives the first enumerator the value 0 by default, and the remaining enumerators are incremented by the value 1, as they are read from left to right. Any enumerator can be set to a specific integer constant value. The enumerators to the right of such a construct (unless they are also set to specific values) then receive values that are 1

greater than the previous value. For example, consider the following:

```
enum spectrum
   {
        red, yellow=4, green, blue, indigo, violet
   } color2;
```

This declaration gives *red* , *yellow* , *green* , *blue* , **. . .** , the values 0,4,5,6, **. . .** .

Examining the value of a variable like *color2* displays an integer, not a string such as red or yellow. Although they are stored internally as integers, regard enumerated data types as distinct from the fundamental types.

### 5.7 Arrays ([ ])

Arrays are declared using the square bracket notation ([ ]), as in the following declaration of a 10-element array of integers called *table_one* :

```
int table_one[10];
```

The type specifier **int** gives the data type of the elements. The elements of an array can be of any scalar or aggregate data type. The identifier *table_one* specifies the name of the array. The constant expression gives the number of elements in a single dimension. Array subscripts in PDP-11 C begin with the integer 0 (not 1); they must be integral. In the previous example, the first element is *table_one[0]* and the last element is *table_one[9]* . Unpredictable results may occur if you specify a subscript larger than or equal to the declared dimension bound; you would then be accessing objects outside the memory allocated to the array. PDP-11 C, like many other C implementations, does not perform automatic bounds checking.

PDP-11 C supports multidimensional arrays: arrays declared as an array of arrays. Consider the following:

```
int table_one[10][2];
```

Here, variable *table_one* is a two-dimensional array containing 20 integers. You can use PDP-11 C operators in forming expressions with specific elements of an array, as follows:

```
++table_one[0][0]; /* Increment first element */
```

In C, arrays are stored in row-major order. The element *table_one[0][0]* immediately precedes *table_one[0][1]* , which in turn immediately precedes *table_one[1][0]* .

When you declare an array, either single- or multidimensional, the integer constant is optional in the first pair of brackets. Omission of the constant expression is useful in the following cases:

.

If the array is external, its storage is allocated by a remote definition. Therefore, the constant expression can be omitted for convenience when the array name is declared, as in the following example:

```
extern int array1[];
void first_function(void)
```

```
{
   .
   .
   .
}
```

In a separate compilation:

```
int array1[10];
void second_function(void)
{
   .
   .
   .
}
```

For more information concerning external data declarations, refer to Chapter 6.

.

If the declaration of the array includes initializers, the size of the array can be omitted.

```
char array_one[] = "Shemps"
char array_two[] = { 'S', 'h', 'e', 'm', 'p', 's', '\0' };
```

The two definitions initialize variables with identical elements. These arrays have seven elements: six characters and the NUL character (\0), which terminates all character strings. PDP-11 C determines the size of the array from the number of characters in the initializing character-string constant or initialization list.

.

If the array is used as a function parameter, it is defined in the calling function. The declaration of the parameter in the called function can omit the constant expression. The address of the beginning of the array is passed and subscripted references in the called function can modify elements of the array.

The following example shows how a character array is used in this manner:

```
#include <stdio.h>
int adder()
int main(void)
{
                              /* Initialize array */
   static char arg_str[] = "Thomas";
   int sum;
```

```
    sum = adder(arg_str); /* Pass address of array */
    printf("The sum is %d\n",sum);
}
/* Function adds ASCII values of letters in array */
int adder(char param_string [])
{
    int i, sum=0; /* Incrementor and sum */
                          /* Loop until NUL char */
    for (i=0; param_string[i] != '\0'; i++)
        sum += param_string[i];
    return sum;
}
```

When the function adder is called, parameter *param_string* receives the address of the first character of argument *arg_str* , which can then be manipulated in adder. The declaration of *param_string* serves to give the type of the parameter (in this case, effectively pointer to array of **char** ) not to reserve storage for the array. Note that the function adder relies on a NUL-terminated string.

### 5.7.1 Initialization of Arrays

When initializing array elements, separate the values with a comma and delimit the comma-list with braces ({ }). The rules for specifying a comma-list are as follows:

.

If the initializer for an array begins with a left brace ( { ), then the following comma-list provides initial values for the array elements. The list of initializers can end with a comma, which is ignored. The number of initializers cannot be greater than the number of elements.

.

If the initializer for a subarray does not begin with a left brace, then only enough elements are taken from the initializer list to supply values to the array's elements. In this case, there can be more initializers than there are elements, and any remaining values in the list are left to initialize the next aggregate.

Initialize a single-dimension array as follows:

```
int ex_array[5] = { 1, 22, 333, 4444, 55555 };
```

Initialize a multidimensional array as follows:

```
int ex_array[2][5] =
   {
```

```
        { 1, 22, 333, 4444, 55555 },
        { 5, 4, 3, 2, 1 }
    };
```

The element *ex_array[0][0]* has a value of 1, *ex_array[0][1]* has a value of 22, **. . .** , *ex_array[1][0]* has a value of 5, *ex_array[1][1]* has a value of 4, **. . .** , and so forth.

Another method of initializing the same array is as follows:

```
int ex_array[2][5] = { 1, 22, 333, 4444, 55555, 5, 4, 3, 2, 1 };
```

PDP-11 C initializes the elements in row-major order. The leftmost brace determines the row number of a multidimensional array. Elements in row 0 are initialized before elements in row 1.

You may omit elements in an initialization, as follows:

```
int ex_array[2][5] =
    {
        { 1, 22, 333, 4444 }
    };
```

The element *ex_array[0][0]* has the value 1, *ex_array[0][1]* has the value 22, *ex_array[0][2]* has the value 333, and *ex_array[0][3]* has the value 4444. Because *ex_array* is an aggregate type, the last element in the first row is initialized to 0. All the elements in the second row that were not specified in the initialization are initialized to 0.

---

**Note**

---

You cannot initialize array elements without initializing all preceding elements. The following initialization is not valid:

```
example[3] = { 1 , , 3 };
```

In the previous example, you have to initialize the first and second element before initializing the third.

---

As a special case, a character array may be initialized by a string literal; successive characters of the string initialize members of the array. The trailing null is placed when the array is declared without bounds, or when there is room for it.

For example,

```
    char a[] = "abc";
```

```
    char b[3] = "abc";
    char c[4] = "abc";
```

is identical to:

```
    char a[] = {'a','b','c','\0'};
    char b[] = {'a','b','c'};
    char c[] = {'a','b','c','\0'};
```

The array **a** contains the null because it was declared without bounds. The array **b** does not contain the trailing null because there was no room for it. The array **c** contains the null because there is room for it.

## 5.8 Character-String Variables and Constants (char *, char[ ])

PDP-11 C treats character strings as arrays; they are treated as the address in memory of the first character in the string. There are several ways to declare character-string variables. You can declare a character string by designating a pointer to the first character of that string, as in the following:

```
char *ex_string = "thomasina";
```

This expression copies an address, not a string, to variable *ex_string* . The object to which *ex_string* points, a character-string constant, ends with the NUL character ( ' \0 ' ).

You can declare character-string variables as you would declare an array. For example:

```
char string_one[] = "thomasina";
char string_2[10] = "thomasina";
```

See Section 5.7.1 for more information concerning declaration and initialization of character-string variables.

To copy one string to another, use the **strcpy** or the **strncpy** PDP-11 C Run-Time Library (RTL) functions, as follows:

```
#include <stdio.h>
#include <string.h>
int main(void)
{
   char ex_string[26];
                                    /* Copy string into array */
   strcpy(ex_string, "Character-string constant");
   printf("%s\n", ex_string);
        .
        .
        .
}
```

For more information concerning the PDP-11 C RTL functions for copying strings, refer to the *PDP-11 C Run-Time Library Reference Manual* .

A character-string constant is a series of characters enclosed in quotation marks ( " " ). Consider the following:

```
"This is a string constant *** "
```

It has data type of an array of **char** . The string is initialized with the given characters. The compiler terminates the string with a NUL character ( ' \0 ' ). There is no formal limit to the length of a string constant. The actual limit to a string constant's length in PDP-11 C is 65,535 characters. This limit is subject to further PDP-11 hardware-specific constraints at the time the object file is created. All strings, even when written identically, are distinct objects.
The apostrophe ( ' ) and quotation mark ( " ) are significantly different punctuation marks in PDP-11 C. See Section 5.3.2 for more information.
The following rules apply to the characters used in character-string constants:

.
All characters, including the escape sequences, can be used in strings.

.
A quotation mark within a string must be preceded by a backslash ( \ ).

.
A backslash followed immediately by a newline is ignored, allowing long strings to be continued in the first column of the next line.

### 5.9 Structures and Unions (struct, union)

Structures and unions share the following characteristics:

- Their members can be variables of any type, including other structures and unions or arrays. A member can also consist of a specified number of bits, called a bit-field.
- The only operators that are valid with structures and unions are the simple assignment ( = ), **sizeof** , dot ( . ), and arrow (->) operators. In particular, structures and unions may not appear as operands of the equality (= =), inequality (!=), or cast operator.
- They can be assigned to other structures and unions with the assignment operator ( = ). The two structures or unions in the assignment must have the same type.
- They can be passed to and returned by functions. The argument must have the same type as the function parameter. A structure or union is passed by value, just like a scalar variable; that is, the entire structure or union is copied into the corresponding parameter.

The difference between structures and unions lies in the way their members are stored.

- The members of a structure all begin at different offsets from the base of the structure. The offset of a particular member corresponds to the order of its declaration; the first member is at offset 0. Each successive member of a structure begins at the next nonbit-field byte or word boundary depending on the alignment requirement of the type of the member. An unnamed bit-field of width zero causes the next member (generally another bit-field) to be aligned on the required boundary. This alignment of structure members is a PDP-11 C convention and is also followed by all other PDP-11 languages. Other C implementations may align members differently.
- In a union, every member begins at offset 0 from the address of the union. The size of the union in memory is

the size of its largest member. When the single storage space allocated to the union contains a smaller member, the extra space between the end of the smaller member and the end of the allocated memory remains unaltered. You can initialize only the member of a union that appears first in the list of union members.

### 5.9.1 Declaring a Structure or Union

Structures and unions are declared with the **struct** or **union** keywords. You can follow the keywords **struct** or **union** by a tag, which gives a name to the structure or union type in much the same way that an **enum** tag gives a name to the enumerated type. You can then use the tag with the **struct** or **union** keywords to declare variables of that type without specifying individual member declarations again.

Two structures, two unions, or enumerators cannot have the same tag, but the tags can be the same as the identifiers used for variables and function names and member names. The compiler distinguishes them by context. The scope of a tag is the same as the scope of the declaration in which it appears.

The tag is followed by braces ({ }) that enclose a list of member declarations. Each declaration in the list gives the data type and name of one or more members. The names of structure or union members can be the same as other variables, function names, or members in other structures or unions. The compiler distinguishes them by context. In addition, the scope of the member name is the same as the scope of the declaration in which it appears.

The list of member declarations can be followed by declarators which declare structure or union objects.

Structure or union declarations can take one of five forms, as follows:

1. If a declaration includes only a tag and a list of member declarations, then the list of member declarations defines the tag to be a data type by which other objects can be declared. For example:

```
struct person
   {
      char first[20];
      char middle[3];
      char last[30];
   };
```

2. When a declaration includes a tag, a list of member

declarations, and a list of identifiers, the identifiers become objects of the structure type and the tag is considered to be a shorthand notation, or mnemonic, for the structure type. Consider the following example:

```
struct person
   {
      char first[20];
      char middle[3];
      char last[30];
   } george, mary ;
```

3. If the tag is omitted, the structure or union definition applies only to the variable identifiers that follow in the declaration. Consider the following example:

```
struct
   {
      char first[20];
      char middle[3];
      char last[30];
   } george, mary;
```

4. The fourth form uses the tag to refer to a structure or union defined in another declaration. The definition is then applied to the variable identifiers that follow the tag name in the declaration.

```
struct person george,mary;
```

5. The fifth form uses only the **struct** or **union** keyword and the tag to override other identical tags in scope, and to reserve the tag for a later definition within a new scope. A definition within a new scope overrides any previous tag definition appearing in an outer scope. This use of declaring tags is called vacuous structure tag declaration. The declaration does not require the size of the structure as determined by the structure member list. Using such declarations, you can eliminate ambiguity when forward referencing tag identifiers. The following example illustrates such a case:

```
struct ambiguous {...};
{
   struct ambiguous; /* Vacuous structure tag declaration. */
                     /* Ignore previous tag currently in scope. */
   struct inner
   {
      struct ambiguous *pointer; /* Declare a structure pointer by */
        . /* forward referencing. */
```

```
            .
            .
        };
        struct ambiguous /* Complete the definition of "ambiguous" */
        {...}; /* at this scope. */
    }
```

In the example, the pointer to the structure defined using tag *ambiguous* points to the second declaration of *ambiguous* , not to the first.

Structures and unions can contain other structures and unions. For example:

```
struct person
  {
      char first[20];
      char middle[3];
      char last[30];
      struct
      {
          int day;
          int month;
          int year;
      } birth_date;
  } george, mary;
```

### 5.9.2 Referencing Members of Structures or Unions

A reference to a member of a structure must be a fully qualified or a pointer-qualified reference. For example, the fully qualified references to the members last and year from the example in the previous section are as follows:

```
strcpy(george.last, "Harrison");
george.birth_date.year = 1944;
```

A member name denotes the member's data type and its offset from the base of the structure. There are no restrictions on the reuse (as a member name) or redeclaration of a particular name, except that the same name cannot be used for more than one member in the same structure.

In PDP-11 C, and in other modern C compilers, a structure or union reference must be completely qualified; that is, you must prefix a member name in a reference either with a pointer qualifier (pointer-name ->) or with the name of the structure or union and the names of all intervening members. For example, consider the following structure declaration:

```
int main()
```

```
{
   struct
        {
           struct { int a1,a2,a3; } mema;
           struct { int a1,a2,a3; } memb;
        } *pointer, structure;
   pointer = &structure;
   structure.mema.a1 = 1; /* Unambiguous */
   pointer->memb.a1 = 2;
   structure.a1 = 3; /* Ambiguous: which "a1"? */
   pointer->a1 = 4;
}
```

Member *a1* must be uniquely qualified as being a member of structure *mema* or structure *memb* . In fact, structure members that are themselves structures must be given variable identifiers (mema and memb) to make it possible to construct fully qualified references.

A member name is unique if it conforms to either of the following requirements:

.
It is used only once.

.
If it is used more than once (in different structures), every use denotes a member of the same data type and at the same offset from the base of its structure.

If you use member names that refer to different structures than those in which they were declared (a programming practice not recommended), the compiler issues diagnostic messages. The following checks apply to the use of member names for references to structures and unions in which they are not declared:

.
If a member name is unique, you can use it in a reference to a structure of which it is not a member, since the address and size of the referenced data can be determined without ambiguity. However, the compiler issues a nonfatal warning message. This usage is maintained for compatibility with other C implementations.

.
If a member name is not unique (ambiguous), its use in such a reference causes a fatal error message.

### 5.9.3 Initialization of Structures and Unions

In structure and union declarations, initializers follow the structure or union variables, not the members. Separate initializing values with commas; delimit them with braces ({ }). See Section 5.7.1 for more information concerning comma-lists.

An example of the initialization of two structure variables follows:

```
struct
   {
        int i;
        float c;
   } a = { 1, 3.0e10 }, b = { 2, 1.5e5 };
```

The initialization of a union assigns the initializing value to the *first* member in the list of unions. You cannot assign an initializer to any other member of the union but the first. In the following example, you can only initialize **i** .

```
union
   {
      int i;
      float f;
   } u = { 7 };
```

The compiler assigns structure initializers in increasing member order. If there are fewer initializers than members, the structure is padded with zeros. For more information concerning storage classes, refer to Chapter 6.

---

**Note**

---

There is no way to specify iterations of an initializer or to initialize a member in the middle of a structure without also initializing the previous members.

---

Example 5-1 shows these initialization rules applied to an array of structures.

Key to Example 5-1:

**1** You must delimit the initialization of each of the array rows with braces.

**2** You must delimit a structure initialization with braces.

**3** You must delimit an array initialization with braces.
This program writes the following output to **stdout** :

```
row/col ch i c
-----------------------------------
[0][0]: a 1 3.000000e+10
[0][1]: b 2 4.000000e+10
[0][2]: c 3 5.000000e+10
[1][0]: 0 0.000000e+00
[1][1]: 0 0.000000e+00
[1][2]: 0 0.000000e+00
```

### 5.9.4 Variant Structures and Unions

Variant structure and union declarations allow you to reference members of nested aggregates without having to reference intermediate structure or union identifiers.

---

**Note**

---

PDP-11 C recognizes and implements variant structures and unions for compatibility with VAX C, but they are not in the ANSI standard. When compiling PDP-11 C programs, the default is /NOSTANDARD, which allows the keywords **variant_struct** and **variant_union** to be recognized. If you specify /STANDARD or /STANDARD=ANSI, these keywords will not be available.

---

When you nest a variant structure or union declaration within another structure or union declaration, the enclosed variant aggregate ceases to exist as a separate aggregate, and PDP-11 C propagates its members to the enclosing aggregate.
You declare variant structures and unions using the keywords **variant_struct** and **variant_union** . The format of these declarations is the same as regular structures or unions except for the following:

.

Variant aggregates must be nested within other valid structure or union declarations.

.

You cannot use a tag in a variant aggregate declaration.

.

You must provide a variable identifier in the variant aggregate declaration.

To illustrate the use of variant aggregates, consider the following code example, which does not use variant aggregates:

```
/* The numbers to the right of the code represent the byte offset *
  * from the enclosing structure or union declaration. */
struct TAG_1
{
   int a; /* 0-byte enclosing_struct offset */
   char *b; /* 2-byte enclosing_struct offset */
   union TAG_2 /* 4-byte enclosing_struct offset */
   {
        int c; /* 0-byte nested_union offset */
        struct TAG_3 /* 0-byte nested_union offset */
        {
           int d; /* 0-byte nested_struct offset */
           int e; /* 2-byte nested_struct offset */
        } nested_struct;
   } nested_union;
} enclosing_struct;
```

If you want to access nested member *d* , then you need to specify all of the intermediate aggregate identifiers, as follows:

```
enclosing_struct.nested_union.nested_struct.d
```

If you attempted to access member *d* without specifying the intermediate identifiers, then you would be accessing the incorrect offset from the incorrect structure. For instance, if you specified the following:

```
enclosing_struct.d
```

PDP-11 C uses the address of the original structure (enclosing_struct), and adds to it the assigned offset value for member *d* (0 bytes), even though PDP-11 C calculated the offset value for *d* according to the nested structure (nested_struct). Consequently, PDP-11 C accesses member *a* (0 byte offset from enclosing_struct) instead of member *d* .

The following code example illustrates the same code using variant aggregates:

```
/* The numbers to the right of the code present the byte offset *
  * from enclosing_struct. */
struct TAG_1
{
```

```
   int a; /* 0-byte enclosing_struct offset */
   char *b; /* 2-byte enclosing_struct offset */
   variant_union
   {
        int c; /* 4-byte enclosing_struct offset */
        variant_struct
        {
           int d; /* 4-byte enclosing_struct offset */
           int e; /* 6-byte enclosing_struct offset */
        } nested_struct;
   } nested_union;
} enclosing_struct;
```

The members of variant aggregates nested_union and nested_struct are propagated to the immediately enclosing aggregate (enclosing_struct). The variant aggregates cease to exist as individual aggregates.

Since variant aggregates nested_union and nested_struct do not exist as individual aggregates, you cannot use tags in their declarations and you cannot use their identifiers (nested_union, nested_struct) in any reference to their members. However, you are free to use the identifiers in other declarations and definitions within your program.

If you need to access member *d* , you use the following notation:

enclosing_struct.d

If you use the following notation, unpredictable results occur:

enclosing_struct.nested_union.nested_struct.d

If you use regular structure or union declarations within a variant aggregate declaration, PDP-11 C propagates the structure or union to the enclosing aggregate, but the members remain a part of the nested aggregate. For instance, if the nested structure in the last example was of type **struct** , the following offsets would be in effect:

```
struct TAG_1
{
   int a; /* 0-byte enclosing_struct offset */
   char *b; /* 2-byte enclosing_struct offset */
   variant_union
   {
        int c; /* 4-byte enclosing_struct offset */
        struct TAG_2 /* 4-byte enclosing-struct offset */
        {
           int d; /* 0-byte nested_struct offset */
```

```
            int e; /* 2-byte nested_struct offset */
        } nested_struct;
    } nested_union;
} enclosing_struct;
```

### 5.9.5 Bit-Fields

A structure member may consist of a specified number of bits, called a bit-field, which may be named or unnamed. A colon is used to separate the member's declarator (if any) from a constant-expression that gives the field width in bits. No field may be longer than 16 bits (1 word) in PDP-11 C.

If no field name precedes the field-width expression, it indicates an unnamed field of the specified width. Since bit-field structure members are not aligned on byte or word boundaries, this form can create unnamed gaps in the structure's storage. As a special case, an unnamed field of width zero causes the next member (generally another bit-field) to be aligned on the next word boundary.

Bit-fields must be of data types **int** , **unsigned int** , **unsigned** , **signed int** , or **signed** . Bit-fields can also have a type that is a qualified or unqualified version of **int** , **unsigned int** , or **signed int** . The use of other data types is an error. In PDP-11 C, bit-fields of type **int** are unsigned. This is incompatible with VAX C, in which bit-fields of type **int** are signed.

The following restrictions apply to the use of fields:

.
You cannot declare arrays of bit-fields.

.
The address-of operator (&) cannot be applied to bit-fields, and consequently there cannot be pointers to bit-fields.

.
You cannot use the **sizeof** operator on bit-fields

Constructs of all data types except bit-fields are aligned on the next byte or word boundary. Sequences of bit-fields are packed as tightly as possible. In PDP-11 C, fields are assigned from low bit offset to high bit offset. If necessary, a bit-field will cross word boundaries (for example, it will wrap to the next word).

Figure 5-1 illustrates the alignments resulting from the following code:

```
static struct
    {
```

```
        char c; /* offset 0 */
        short int i; /* offset 2 */
        unsigned fld1 : 3; /* offset 4, bit 0 */
        unsigned fld2 : 4; /* offset 4, bit 3 */
        unsigned : 0;
        unsigned fld3 : 4; /* offset 6, bit 0 */
    } a = { 'A', 1024, 06, 012, 014 } ;
```

In Figure 5-1, member *a.i* is aligned on the second word because the **int** type requires word alignment. Notice that fields *a.fld1* and *a.fld2* are packed as tightly as possible in the low-order byte of the third word. The unnamed, zero-length field causes *a.fld3* to be aligned on the next word boundary.

### 5.10 Aggregates

The variables used in the examples in Section 2.6 were single objects that could be manipulated, in their entirety, in an arithmetic expression. These types of variables are called **scalar** variables. The PDP-11 C data structures-arrays, structures, and unions-are called **aggregates** . Aggregates are comprised of segments called **members** , or in the case of arrays, they are called **elements** . Members are sections of the structure that you can declare to be of a scalar or an aggregate data type.

#### 5.10.1 Arrays and Character Strings

An array is an aggregate whose elements are of the same type. Elements of an array can be any one of the scalar or aggregate data types.

In PDP-11 C, character strings are represented internally as arrays of type **char** . You may declare and initialize a character string using the indirection notation ( *

), as an array of a specified number of members, or as an array of an unspecified number of members, as follows:

```
char *str = "Hello";
char string[6] = "Hello";
char strng[] = "Hello";
```

Character strings end with the NUL character (\0). In the previous example, the NUL character is appended to ``Hello'' making the string 6 characters in length. For more information concerning string-handling functions, refer to the *PDP-11 C Run-Time Library Reference Manual* . Example 5-2 shows the use of character strings and arrays.

The output for Example 5-2 follows:

```
$ run example8
Guess which letter I'm thinking of!
B
You're wrong.
You'll have to try again!
```

### 5.10.2 Structures and Unions

Structures and unions are aggregates whose members can be of different types. Structures and unions are declared using the keywords **struct** and **union** , respectively, an optional tag name, and a list of member declarations delimited by braces ({ }). A member of a structure or a union is a declared segment of the data structure. The syntax for declaring a member is the same as for declaring any variable. The structure or union tag is a name that can be used when declaring structure or union variables of the same type elsewhere in the program. Members of structures and unions may be referenced as follows:

```
int main(void)
{
   struct foo_tag /* optional tag is foo_tag */
        {
           char letter_1;
           char letter_2;
           int number;
        } characters = {'a', 'b', 59}; /* initialize variable */
   characters.letter_1 = characters.letter_2;
}
```

You may reference members using the structure or union variable name, directly followed by a period ( . ), directly followed by the member name. As in the previous example, structures are initialized using a variable name and an assignment operator ( = ) immediately following the declaration of the members. The values of the members are delimited by braces and separated by commas ( , ). The address of the first member of a structure begins, in memory, at the base of the data structure, which is referred to as **offset zero** . The address of the second begins after the first, and so on.

Unions are declared in the same way as structures, but all members in a union begin at offset zero. This means that all members of a union share the same memory. Only the first member of a union may be initialized. The size of the union in memory is as large as its largest member. When the single storage space allocated to the union contains a smaller member, the extra space between the end of the smaller member and the end of the allocated memory remains unaltered. Example 5-3 illustrates the nature of unions.

The output for Example 5-3 follows:

```
$ run example9.sav
Lincoln
Jackson
M
Mackson
```

The RTL function **strcpy** copies the second string argument into the first array argument. To use the RTL function **strcpy** , you must include the header file *string.h* as shown in Example 5-3. When assigning values to smaller union members, the compiler does not fill the remaining space with NUL characters ( ' \0 ' ); whatever was in memory at the time remains. For more information concerning structures and unions, refer to Chapter 5.

Example 5-4 shows a structure definition and its usage.

Key to Example 5-4:

**1** In the example, the structure declaration with the tag storage has four members. The first three members are of type **char** . The last member is of type **int** .

**2** The variable **letter** is declared using the tag storage and individual members of the structure are initialized. The equal sign initializes the members of the structure variable with constants. The constants are separated by a comma and are delimited by braces. The number of initializing constants cannot exceed the number of members. However, as in this example, you may omit constants; the compiler pads the uninitialized member (in the example, member num_guesses) with zeros. You cannot initialize a member in the middle of any aggregate without initializing the previous members.

**3** **Return** finishes program execution.

Sample interaction for Example 5-4 follows:

```
$ run example10
Guess which letter I'm thinking of!
You've 3 guesses. Make them count!
B
You're wrong.
You'll have to try again!
C
You're wrong.
You'll have to try again!
```

**U**

You're wrong.
Sorry, you've run out of guesses!
After executing these program examples, you are well on your way to programming in PDP-11 C.

### 5.11 The void Keyword

The **void** keyword is a special data type specifier that you use in function definitions and declarations for the following purposes:

.
To specify a function that does not return a value

.
To specify a function prototype which declares a function with no arguments

For instance, the following example shows how to use **void** to specify a function that does not return a value:

```
void message( )
{
   printf("Stop making sense!");
   return;
}
```

The following example shows how to use **void** to specify a function prototype definition that takes no arguments:

```
char function_name( void )
```

For more information concerning the **void** data type and function prototypes, refer to Chapter 2.

[next] [previous] [contents]

---

### 5.12 The typedef Keyword

The keyword **typedef** is used to define an abbreviated name, or synonym, for a type definition. In such a declaration, the identifiers name types instead of variables. For example:

typedef char CH, *CP, STRING[10], CF(void);

In the scope of this declaration, CH is a synonym for character, CP for pointer to character, STRING for 10-element array of characters, and CF for function returning a character. Each of the type definitions can be used in that scope to declare variables, as in:

CF c; /* "c": Function returning a character */
STRING s; /* "s": 10-character string */

### 5.13 Interpreting Declarations

The PDP-11 C programming language syntax for declaring objects is unlike the declaration syntax of other languages. Because the exact meaning of a complicated PDP-11 C declaration is not always immediately apparent, even to an experienced C programmer, this section gives guidelines for interpreting and constructing PDP-11 C declarations.

PDP-11 C uses the same set of operators and symbols for declarators as for identifiers in an expression. For example, the following example declares integer *x* and pointer *px* .

```
int x;
int *px;
```

Declarator

```
*
```

*px* has the same form as that used to yield an integer in an expression, such as the following:

```
x = *px;
```

In the case of simple declarators, this symmetry makes it fairly easy to determine the type of an expression or the meaning of a declarator. Expression

```
*
```

*px* results in the integer object to which *px* points.

More complicated declarators can be more difficult to interpret without some additional guidelines. The important one to remember is that the symbols used in declarators are PDP-11 C operators, subject to the usual rules of precedence and grouping (associative nature). In order of precedence, the operators used in declarators are:

1. The primary-expression operators (( )) for `` function returning **. . .** " and ([ ]) for ``array of **. . .** ", where the ellipsis indicates the type specified in the declaration. These operators group from left to right.
2. The unary asterisk (

```
*
```

), for indirection or ``pointer to **. . .** ", which groups from right to left.

Consider the following, for example:
int *x[];
Even this brief declaration may be confusing. Does it declare an array of pointers to integers, or a pointer to an array of integers? Since the brackets are of higher precedence, it follows that:
1.
    *

    x[ ] is an integer.
2. x[ ] is a pointer to an integer.
3. x is an array of pointers to integers.
Most complicated declarators and expressions can be interpreted fairly quickly by such a sequential breakdown.
Note that the asterisk was removed before the brackets because it is of lower precedence.
Also note that this interpretation process has the desirable property of enumerating all the possible usage constructs involving a declarator and giving the semantic interpretation.
When constructing or interpreting declarations or expressions, use the following scheme

**1**
for translating operators to English and vice versa:
.
    ``
     *

     " == ``pointer to"
.
    ``( )" == ``function returning"
.
    ``[ ]" == ``array of"
For a more interesting example, consider the following:
char *x()[];
The breakdown is:
1.
    *

    x( )[ ] is **char** .
2. x( )[ ] is (pointer to) **char** .

3. x( ) is (array of) (pointer to) **char** .
4. x is (function returning) (array of) (pointer to) **char** .

In step 3, the brackets operator is removed first because primary-expression operators have equal precedence and group from left to right. That is, ``( )[ ]" means ``function returning array of," not ``array of function returning."

As a general rule, when breaking down a declaration this way, remove the operators with the lowest precedence first. Then, if operators are of equal precedence and group from left to right, remove the rightmost operator first; if they group from right to left, remove the leftmost operator first.

In the previous example, the declaration shown is semantically invalid; PDP-11 C allows functions returning addresses of arrays, but not functions returning arrays. Perhaps the intention of the programmer was a function returning the address of an array of pointers to characters. The declaration can be made valid by starting at the bottom of a breakdown and working back to a valid declaration:

1. x is (function returning) (pointer to) (array of) (pointer to) **char** .
2. x( ) is (pointer to) (array of) (pointer to) **char** .
3. *

   x( ) is (array of) (pointer to) **char** .
4. (*

   x( ))[ ] is (pointer to) **char** .
5. *

   (*

   x( ))[ ] is **char** .
6. char *

   (*

x( ))[ ]; is the final declaration.

In the final declaration, the first asterisk (since it groups right to left) applies to **char** .

Parentheses, in addition to the function parameter-list operator (( )), are used in declarations to change the binding of operators. For example, the outer parentheses introduced in step 4 prevent the brackets from binding to the inner set of parentheses.

As a last case, consider the following:

char (* (*x()) []) ();

This means:

1. (
   *

   (
   *

   x( )) [ ]) ( ) is **char** .
2.
   *

   (
   *

   x( )) [ ] is (function returning) **char** .
3. (
   *

   x( )) [ ] is (pointer to) (function returning) **char** .
4.
   *

   x( ) is (array of) (pointer to) (function returning) **char** .
5. x( ) is (pointer to) (array of) (pointer to) (function returning) **char** .
6. The identifier x is a (function returning) a (pointer to) an (array of) (pointers to) (functions returning) characters.

Spaces were used in the example to separate the declarator into its component parts. Since spaces, tabs, and newlines are ignored by the parser, they should be used in actual declarations for clarity.

## 6. Scope, Storage Classes, and Allocation

The PDP-11 C language defines a number of storage-class keywords that specify the location of storage and the lifetime of the storage allocation. Storage-class qualifiers are keywords you can use with the storage-class and data type keywords that restrict access to and determine the lifetime of variables. The order of the storage-class keyword, the storage-class qualifier, the data type qualifier, and the data type keyword within the variable declaration does not matter. Each declaration, by virtue of its position in the program source code, has a default storage class, but you may override the default by specifying a storage-class specifier or a storage-class qualifier.

This chapter describes the following:

- Scope of an identifier
- Location of storage
- Lifetime of storage allocation
- Internal storage class
- Static storage class
- Global storage class
- Data type qualifiers
- **globalvalue** specifier
- Explicit psect control
- Storage-class qualifiers

### 6.1 The Scope of an Identifier

The scope of an identifier is the portion of the program in which the identifier has meaning. An identifier has meaning if it is recognized by the compiler, or at the time of task building, by the Task Builder on RSX and RSTS/E systems, or by the Linker on RT-11 systems. The following sections explain the rules to follow for your program identifiers to have meaning to both the compiler and the Task Builder or Linker, in all desired portions of your program.

All tags are subject to the same scope rules as other identifiers. A member of a structure or union may have the same name as a member of another structure or union; the scope of the member names can exist concurrently. However, when referencing one of the members in a section of the program where the scopes of both members are concurrent, take care to specify to which structure or union the member belongs. For more information concerning the scope of structure and union members, refer to Chapter 5.

#### 6.1.1 The Compilation and Linking Process

To understand scope, you must understand how PDP-11 C uses functions, compilation units, object files, object modules, and programs.

When you write PDP-11 C source programs, you can use several methods to compile a program. You can compile a single source file, or a group of source files, into a single **object file** . The group of source files compiled to create a single object file is called the **compilation unit** . When documentation to other implementations refers to the source file, the PDP-11 equivalent is the compilation unit, not necessarily a single source file. The single, resultant object file has a file extension of OBJ by default.

The Task Builder or Linker accepts the object file as input and then resolves all external references, such as references to PDP-11 C Run-Time Library (RTL) functions. Internally, segments of object code, such as the object file and the RTL object code, are known to the Task Builder or Linker as **object modules** . The object module has the same name (without an extension) as the object file, by default. For information on how to override the default module name,

refer to Chapter 7.

The second way to build programs is to compile several compilation units into separate object files. The Task Builder or Linker can take more than one object file as input; then, the Task Builder or Linker resolves references between these individual modules as well as to external references. For more information concerning compiling and linking, refer to Chapter 1.

### 6.1.2 Position of the Declaration

In determining the scope of a function or variable identifier, you must consider the position of a declaration within the program. A declaration often determines the size of a storage allocation, whereas a definition initiates the allocation of storage. Since declarations often are definitions, this section refers to definitions and declarations as declarations. You may wish to review Chapter 5 before reading the rest of this section.

The location of a declaration establishes the scope of an identifier. If a declaration is located inside of a block that is delimited by braces ( { } ), the compiler recognizes the identifier from the point of the declaration to the end of the block. If a declaration is located outside of all functions, the compiler recognizes the identifier from the point of the declaration to the end of the compilation unit.

You can specify a storage-class specifier or qualifier within an identifier's declaration. A storage-class specifier indicates a storage class, but a qualifier modifies access to that storage. The order of the storage-class specifier, storage-class qualifier, and the data type keyword within the declaration does not matter. Consider the following example:

```
auto int x; /* And, equivalently ... */
int auto x;
```

You can declare identifiers with no storage class; the compiler recognizes these identifiers from the point of the declaration to the end of the enclosing block or function body. You can declare identifiers that are static; if the declaration is outside all function bodies, the compiler recognizes these identifiers from the point of the declaration to the end of the compilation unit.

You can also declare identifiers that are of the storage class global. If the declaration is outside all function bodies, the compiler recognizes these identifiers from the point of the

declaration to the end of the compilation unit. The global storage class differs from the static storage class in that the Task Builder and Linker can recognize a global variable. The global storage class establishes a scope that can span object modules.
Table 6-1 lists the storage classes, the storage-class specifiers used to establish scope and the section in this manual that discusses each storage class in more detail.

You can use the data type qualifiers ( **const** and **volatile** ) or the storage-class qualifier ( **readonly** and **noshare** ) to restrict access to data or to specify storage requirements.

---

**Note**

---

The storage-class qualifier **readonly** and **noshare** are provided for compatibility with VAX C, but offer no functionality.

---

See Section 6.9 for more information concerning the data type qualifiers. See Section 6.10 for more information concerning the storage-class qualifiers.

### 6.1.3 Lexical Scope and Link-Time Scope

In using the storage-class specifiers and qualifiers, as well as positioning the definitions and declarations of your identifiers, keep the following two goals in mind:

- Compile the program so that the compiler recognizes all identifiers in the compilation unit, thus avoiding error messages.
- Link the program so that the Task Builder or RT-11 Linker resolves all references to global data definitions, thus avoiding error messages.

You must make a distinction between the following types of scope:

Lexical scope The region of a compilation unit within which

an identifier is known to the compiler. When this guide uses the term scope, lexical scope is implied.

Link-time scope The regions of an entire program within which a global identifier is known to the Linker. Only the identifiers in the global storage class have a significant link-time scope.

Table 6-2 lists the PDP-11 C storage-class specifiers and shows both the link-time scope and lexical scope implied by each specifier when used inside and outside of functions.

In Table 6-2, (none) signifies the absence of a storage-class specifier from the declaration. The compiler treats a (none) inside a function or block as an identifier declared with the **auto** keyword. The compiler treats a (none) outside all functions as a global definition, a (none) storage-class specifier of the global storage class.

### 6.1.4 Program Example

Example 6-1 illustrates how the placement of variable identifiers determines the scope of these identifiers.

The following list specifies the variable identifiers in the previous example, and from which functions they can be accessed without compile-time errors:

---

**Identifier Scope**

---

EXT_1 This variable is declared outside all functions in Compilation Unit 1. This declaration is a reference to the definition of the same variable in the Compilation Unit 2. In Compilation Unit 1, you can access EXT_1 in the function f2 (from the point of the declaration to the end of the compilation unit). EXT_1 will have link-time scope.

In Compilation Unit 2, the definition of this variable is outside all functions; you can access EXT_1 in the functions f3, f4, and f5 (from the point of the declaration to the end of the compilation unit).

EXT_2 This variable is defined outside all functions in Compilation Unit 1. You can access EXT_1 in the functions f1 and f2 (from the point of the declaration to the end of the compilation unit).
In Compilation Unit 2, the declaration of this variable is located inside the function f3; you can access EXT_1 from the location of this declaration to the end of function f3. EXT_2 will have link-time scope.

STAT There are two variables with the same name but with different permanent storage locations. These are two different variables. This is because they do not have link-time scope.
In Compilation Unit 1, the variable is defined outside all functions. You can access STAT, in Compilation Unit 1, in the functions f1 and f2 (from the point of the declaration to the end of the compilation unit).
In Compilation Unit 2, the separate variable is defined inside the function f5; you can access STAT from this declaration to the end of the function f5.

---

Another way to determine scope is to consider the placement of the declaration as a matter of privacy. In Compilation Unit 2, identifier EXT_2 is made private to function f3 by placing the declaration inside the function body. If you want to keep a variable private to Compilation Unit 1, declare the variable using the storage-class specifier **static** . Using the storage-class specifiers **auto** and **register** assures privacy to the function, since these specifiers cannot be used outside a function body, and storage is deallocated at the end of execution of the containing function body. There is no way to access a variable declared with **auto** or **register** in another function or compilation unit.

### 6.2 Storage Allocation

When you define a variable, the storage class determines its location and lifetime. The lifetime of a variable is the length of time for which storage is allocated. Storage for a variable can be allocated in the following locations:

- On the run-time stack
- In a machine register
- In a program section (psect)

Variables that are placed on the stack or in a register are temporary. For example, the variables of storage class **auto** and **register** are temporary. Their lifetimes are limited to the execution of a single block or function. All declarations with no storage class are also definitions; the compiler generates code to establish storage at this point in the program.

Program sections, or **psects** , are used for permanent variables; the lifetime of the storage associated with the identifiers extends through the course of the entire program. A psect represents an area of memory that has a name, a size, and a series of attributes that describe the intended or permitted usage of that portion of memory. For example, the compiler places variables of the static and global storage classes in psects; you have some control as to which psects contain which identifiers (see Section 6.8).

Table 6-3 shows the location and lifetime of a variable when you use each of the storage-class keywords:

In Table 6-3, the notation **extern** signifies identifiers of the global storage class. A single definition must exist for each identifier having the global storage class; other declarations, which use the **extern** specifier, may exist that refer to that definition. This notation is used throughout this chapter. See Section 6.5 for more information concerning the global storage class.

### 6.3 Internal Storage Class

Internal storage class refers to a storage class that permits identifiers declared outside a function body to be recognized only from the declaration to the end of the immediately enclosing block. You can assign the internal storage class to identifiers using the **auto** and **register** storage-class specifiers. The following sections describe these specifiers.

#### 6.3.1 Defining a Variable for Automatic Storage Allocation (auto)

Use the **auto** storage-class specifier to define a variable whose storage is allocated automatically upon entry into the function containing the block in which the variable is declared and is automatically deallocated upon exit from the function. The code generated by the compiler contains instructions to allocate and deallocate the storage by using machine registers and the run-time stack. You can have more than one **auto** variable with the same name as long as you declare them in separate blocks or functions. You cannot use **auto** outside a function.

If you explicitly initialize an **auto** variable, the program code initializes the variable to that value each time the declaring block is entered normally. This initialization cannot occur if control passes into a block by some other means, such as a **goto** statement or if the block is the body of a **switch** statement. For more information concerning the **switch** and **goto** statements, refer to Chapter 3.

Within a function, **auto** is the default storage class. That is, any variable (other than a function name) declared within a function without a storage-class specifier is given the **auto** storage class. Functions are of the extern storage class by default.

---

**Note**

---

The compiler can assign **auto** variables to machine registers, if possible. Otherwise, they are placed on the

run-time stack.

---

Example 6-2 shows how to reinitialize two **auto** variables with the same name.

Key to Example 6-2:

**1** This definition of variable *x* extends through the entire function.

**2** This definition of variable *x* is limited to the **for** statement and supersedes the value of variable *x* in the surrounding function.

The output for Example 6-2 follows:

```
$ run example.exe
main: 2
for loop: 3
main: 2
```

In this program, the variable *x* is defined twice within the main function, but the two variables do not conflict. While the **for** loop is executing, the variable *x* declared inside the block supersedes the variable *x* declared outside the block.

### 6.3.2 Defining a Variable for Placement in a Machine Register (register)

Variables declared with the **register** storage class are similar to **auto** variables. You can use the **register** internal storage class only inside functions, blocks, and function parameter declarations.

---

**Note**

---

The **register** storage-class specifier is the *only* specifier that you can use in a parameter declaration.

---

A **register** variable differs from a variable of storage class **auto** in the way that compiler-generated program code allocates storage. The **register** storage-class keyword suggests that the compiler flag the variable for placement in a machine register. This does not guarantee that the program

code will place the variable in a register. The compiler checks the following conditions to determine whether or not a variable is flagged to be placed in a register:

- If the variable is not used, the optimizer may remove it entirely.
- If the program contains too many register candidates, not all of them are assigned to registers.

For more information, see the On-Line Release Notes.

### 6.4 Static Storage Class

The static storage class allows you to create permanent storage for a variable using the **static** storage-class specifier in the variable declaration. If declared inside of a block, its scope begins at the declaration and spans the remainder of the block. If declared outside of all functions, its scope is limited to the rest of the compilation unit; you can not access a variable of the static storage class from another compilation unit. If a **static** identifier with the same name is declared in another module, the Task Builder or Linker knows nothing of the other variable; the other variable has a separate allocation.

If no initialization is present in the declaration of a variable of the static storage class, the Task Builder or RT-11 Linker initializes the variable to 0. However, unlike **auto** variables, the compiler-generated program code does not reinitialize storage for a **static** variable every time control reenters a function containing the definition of a **static** variable. For example, if you exit a function when a **static** integer variable has the value of 4, the variable retains that value even if control reenters the defining function.

A function can also be defined with the **static** storage class. A **static** function is not known to the Task Builder or Linker and can be referenced only from within its defining module. For more information concerning the possible combinations of specifiers and qualifiers and the effects of the storage-class qualifiers on program section attributes, refer to Chapter 7.

### 6.5 Global Storage Class

You can declare identifiers of the global storage class in the following manner:

- A definition not using another storage-class keyword, located outside all function bodies, declares a global variable whose scope extends from the point of the definition to the end of the compilation unit.
- A declaration using the **extern** keyword, usually located in another compilation unit, is a reference to the original definition. This declaration extends the lexical scope of the variable into the second compilation unit. If this declaration is inside a block, it extends the lexical scope from the point of the declaration to the end of the block. If this declaration is outside a block, it extends the lexical scope from the point of the declaration to the end of the compilation unit.
- The global storage class is the default storage class for variables having file scope. You can use more than one **extern** declaration to reference the global definition.

Use the following rules when deciding whether or not to use the **extern** specifier:

- If the variable is defined before it is referenced, and the definition is in the same compilation unit, you do not need to declare the variable with the **extern** specifier.
- If the variable is defined after it is referenced, you need to first declare it with the **extern** specifier.
- If the variable is defined in a separate compilation unit, you must always declare it with the **extern** specifier.

Consider the following example:

```
double D = 2.37;
int main(void)
{
   extern int A;
   printf("a:\t%d\n", A);
```

```
    printf("d:\t%g\n", D);
}
int A = 5;
```

The *main* function in this program references two global variables, *A* and *D* . Since the variable *D* is defined before it is referenced, it does not have to be declared in the *main* function. Since the variable *A* is referenced before it is defined, it must be declared with the **extern** storage-class specifier.

In many implementations of the C language, you cannot use the **extern** specifier in a declaration that does not refer to a global definition elsewhere in the program. Whenever the compiler encounters the first declaration of an identifier of the global storage class in a PDP-11 C program, it creates a global symbol to represent the location of that variable. Therefore, in PDP-11 C, you can use the **extern** specifier in a declaration that does not refer to a global definition elsewhere in the program. However, this is not good programming practice and your programs may not be portable to other systems.

### 6.5.1 Global Names on PDP-11 Systems

All global names input to the RSX Task Builder or RT-11 Linker must be 6 characters or less and must be of the Radix-50 character set. Although the PDP-11 C compiler does not place any restrictions on the names of global variables in a source program, these names will be translated by the compiler. When creating the output files, the PDP-11 C compiler translates all global symbols to Radix-50 using these rules:

- Lowercase characters translate to uppercase characters
- Underscores translate to periods ( . )
- Global symbols truncate to 6 characters
- Dollar signs ( $ ) remain the same

The compiler will issue a warning if more than one global name maps to the same Radix-50 translation. The user should be aware that different global names in different compilation unit may map to the same Radix-50 name without warning.

### 6.5.2 Global Definitions

The following rules apply when using global definitions in PDP-11 C:

- Definitions of a global identifier may occur not more than once in a compilation unit, or the compiler will return an error.
- The same global variable cannot be defined in two modules that will be linked together or the Linker will return an error.
- All variables declared with the extern storage-class specifier must be defined in a module that will be linked in the final program, or the Linker will return an error.

---

**Note**

---

The global definition rules listed in this section are different than VAX C.

---

Linking the following modules would produce two Linker errors. The first error would be a multiple definition of the global variable *A* . The second error would be the missing declaration of the global variable *B* . Either program compiled alone would not produce any errors.

```
x.c y.c
int A; int A;
extern int B; extern int B;
```

[next] [previous] [contents]

### 6.6 Defining Global Definitions (globaldef) and References (globalref)

For compatibility with VAX C, PDP-11 C supports the storage-class specifiers **globaldef** and **globalref** when compiled using the /NOSTANDARD qualifier. PDP-11 C implements variables of the global storage class using link-time global names and not psects. Therefore, PDP-11 C has no need for **globaldef** and **globalref** storage-class specifiers.

### 6.7 Defining Global Values (globalvalue)

To define a global value, you use the **globalvalue** specifier. A global value is declared outside any functions. You can use the **globalvalue** specifier only with variables of type **enum** , **int** , or with pointer variables. All global values have the same name restrictions as the variables with the global storage class.

Global values are useful because they allow many programmers in the same environment to refer to values by identifier, without regard to the actual value associated with the identifier. The actual values can change, as dictated by general system requirements, without requiring changes in all the programs that refer to them. If you make changes to the global value, you have to recompile only the defining compilation unit (unless it is defined in an object library), not all the compilation units in the program that refer to those definitions.

---

**Note**

---

The **globalvalue** specifier is provided for compatibility with VAX C. Use the /NOSTANDARD switch to enable access to this specifier.

---

A variable declared with **globalvalue** does not require storage. Instead, the RSX Task Builder or RT-11 Linker resolves all references to the value. If an initializer appears with **globalvalue** , the name defines a global symbol for the given initial value. If no initializer appears, the **globalvalue** construct is considered a reference to some previously defined global value.

Predefined global values serve many purposes in system programming, such as defining status values. It is customary in system programming to avoid explicit references to such values as those returned by system services, and to instead use the global names for those values. Example 6-3 shows

how to use the **globalvalue** storage-class specifier.

In Example 6-3, FAIL is defined in the first module: the value is placed into the program stream. In the second module, FAIL is declared so that its values may be accessed. As it does for global variables, the RSX Task Builder or RT-11 Linker recognizes the global symbol as uppercase letters. Express global symbols as not more than 6 Radix-50 characters.

### 6.8 Explicit psect Control

A program section (psect) refers to an area of memory that has a name, size, and a series of attributes that describe the intended or permitted usage of that permanent storage. When the compiler allocates storage for objects of static or global storage class, storage is allocated into one of two psects: static_ro or static_rw. If the object declaration contains the **const** qualifier, storage is allocated in the current static_ro psect. If the object declaration does not contain the **const** qualifier, storage is allocated in the current static_rw psect. PDP-11 C allows the programmer to control the name and attributes of psects. For more information, see Section 7.7.2.

By modifying the attributes of the static_ro and static_rw psects, the user can control the final link-time allocation of the objects.

For example, the following program will allocate the variables *a* and *b* to psect P2, variable *d* to psect P3, and variables *c* and *e* to psect P1.

```
#pragma psect static_ro P1
#pragma psect static_rw P2
static int a;
int b;
static const int c;
#pragma psect static_rw P3
static int d;
static const int e;
```

The two sections that follow give two typical examples of how to use explicit psect control.

#### 6.8.1 Reducing Storage Requirements in Overlaid Tasks

The C language requires that objects of static and global storage classes maintain values throughout program execution. Therefore, the compiler must allocate permanent, unique storage for variables of static and global classes by assigning the following default attributes for static_ro and static_rw psects: sav, gbl, and con.

Allocating permanent, unique storage, adversely affects overlaid tasks. All storage for static and global variables is allocated into the root of the task even for variables declared in modules placed in an overlay.

Many user programs require that static variables only maintain values while the module in which they are declared is active. Thus, the storage could be allocated in the same overlay as the module that declares it. This can be done using the **#pragma psect** directive and by specifying the following attributes: lcl, rel, and nosav. See Section 7.7.2 for more information.

### 6.8.2 Data Sharing Using psects

The most common method for sharing data between two modules is by using variables of global storage class. This requires that variables be defined exactly once in one module and declared using the **extern** qualifier in all other modules. Further, the names of global variables are subject to translation by the PDP-11 C compiler.

An alternate method of sharing data can be accomplished by using explicit psect control. If several modules declare the static_rw psect with the same name and attributes gbl and ovr, they will be declaring the same area of storage. Linking the following two modules will assign the same storage location to *A* and *C* and the same storage location to *B* and *D* .

x.c

```
#pragma psect static_rw SHARE, gbl, ovr
static int A;
static int B;
```

y.c

```
#pragma psect static_rw SHARE, gbl, ovr
static int C;
static int D;
```

## 6.9 Data Type Qualifiers

Data type qualifiers affect the allocation or access of data storage. The data type qualifiers include the **const** and the **volatile** qualifiers. Each is described in detail in the following sections.

### 6.9.1 The const Qualifier

The **const** data type qualifier restricts access to stored data. If you declare an object to be of type **const** , you cannot modify that object.

The following rules apply to the use of the **const** data type qualifier:

- You can specify **const** with any of the other data type keywords in a declaration.
- If you specify **const** when declaring an aggregate, all the aggregate members are treated as objects of type **const** .
- You can specify **const** with **volatile** or with any of the storage-class specifiers or qualifiers.
- The address of a **const** object can be assigned to a pointer to a non- **const** object, but if you use that pointer to alter the value of the object, the result is undefined.

The following example declares the variable *x* to be a constant integer:

```
int const x;
```

When declaring pointers, depending upon the placement of the **const** qualifier in the declaration, PDP-11 C either interprets the pointer or the object to which it points as the constant variable. For instance, the following example declares the variable *y* to be a constant pointer to an integer because the **const** qualifier appears after the asterisk:

```
int * const y;
```

In the following example, the variable *z* is declared as a pointer to a constant integer because the asterisk appears after the **const** qualifier:

```
int const * z;
```

If a variable has static or global storage class and is declared

with the **const** qualifier, it will be placed in a default read only static psect. Using **const** on automatic variables does not affect their storage allocation.

### 6.9.2 The volatile Qualifier

The **volatile** data type qualifier prevents an object from being stored in a machine register, forcing it to be allocated in memory. This data type qualifier is useful for declaring data that is to be accessed asynchronously. A device driver application often uses **volatile** data storage.
The following rules apply to the use of the **volatile** qualifier:

- You can specify **volatile** with any of the other data type keywords in a declaration.
- If you specify **volatile** when declaring an aggregate, all the aggregate members are treated as objects of type **volatile** .
- You can specify **volatile** with **const** or with any of the storage-class specifiers or qualifiers.
- The address of an object of some other type can be assigned to a **volatile** pointer, but the rules of the **volatile** data type qualifier must be followed if you refer to the object using that pointer.

[next] [previous] [contents]

### 6.10 Storage-Class Specifiers

Storage-class specifiers are provided for compatibility with VAX C, but do not have any functionality. The storage-class specifiers include **noshare** , **readonly** , and **_align** .
The PDP-11 C compiler can accept a storage-class specifier and a storage-class qualifier in any order; usually, the qualifier is placed after the specifier in the source code.
For example:

```
extern noshare int x;
   /* Or, equivalently... */
int noshare extern x;
```

**Note**

These storage-class specifiers are provided for compatibility with VAX C. Use the /NOSTANDARD switch to enable access to these specifiers.

## 7. Preprocessor Directives

Preprocessor directives are lines in the source file that direct the compiler to alter its normal processing of PDP-11 C source code. PDP-11 C preprocessor directives, except **#pragma** and **#module** , are defined formally by the ANSI C Language Standard. Therefore, ANSI preprocessor directives do not vary from one compiler to another.

If you plan to port programs to and from other C implementations, you should take care in choosing which preprocessor directives to use within your programs. See Section 7.2 for more information concerning conditional compilation. For a complete discussion of portability concerns, refer to the appendix on compatibility concerns in the *PDP-11 C Run-Time Library Reference Manual* .

This chapter discusses the following preprocessor operations and directives:

- Token replacements (including preprocessor macro substitution)-( **#define** , **#undef** )
- Controls under which conditional segments of code are to be compiled or not-( **#if** , **#ifdef** , **#ifndef** , **#else** , **#elif** , **#endif** , and the **defined** operator)
- A diagnostic message that includes the specified sequence of preprocessing tokens-( **#error** )
- Include source text from an external file-( **#include** )
- A new line number and file name specification for diagnostics- ( **#line** )
- A Task Builder or RT-11 Linker module-title specification- ( **#module** )
- Perform a specific PDP-11 C task, as described later in this chapter-( **#pragma** )

This chapter also discusses the predefined macros defined by the ANSI C Language Standard, as well as macros that are provided for compatibility with VAX C macros.

Preprocessor directives are independent of the usual scope rules; they remain in effect from their occurrence until the end of the compilation unit, or until overridden by another preprocessor directive. For more information concerning compilation units, refer to Chapter 1.

[next] [previous] [contents]

---

### 7.1 Token Definitions (#define, #undef)

The **# define** directive specifies a token string and an identifier with optional arguments. The token string is substituted for every subsequent occurrence of that identifier in the program text, unless it occurs inside a character constant, a comment, or a quoted string. You use the **#undef** directive to cancel a definition for a token.

The syntax of the **# define** directive follows:

#define identifier token-string

#define identifier(identifier, **. . .** ) token-string

If you omit the token string, every subsequent occurrence of that identifier in the program text is deleted from the text to be processed by the compiler.

After a token string is substituted in the source file, the compiler rescans the source line from the beginning of the substituted text to determine whether the previously inserted text contains identifiers defined by other **# define** directives. If so, the identifiers are replaced by their currently specified token strings. Example 7-1 illustrates nested **#define** directives.

---

**Note**

---

/DEFINE and /UNDEFINE perform the same functions from the command line as **#define** and **#undef** . For more information, refer to Chapter 1.

---

Compile Example 7-1 with the following command:

$ **cc/list/show=intermediate example**

The following listing results:

```
1 /* Show multiple substitutions and listing format */
2
3 #define AUTHOR james + LAST
4
5 int main()
6 {
```

```
 7 int writer,james,michener,joyce;
 8
 9 #define LAST michener
10 writer = AUTHOR;
            1 writer = james + LAST ;
            2 writer = james + michener ;
11 #undef LAST
12 #define LAST joyce
13 writer = AUTHOR;
            1 writer = james + LAST ;
            2 writer = james + joyce ;
14
        }
 1
```

On the first pass, the compiler replaces the identifier AUTHOR with the token string james + LAST. On the second pass, the compiler replaces the identifier LAST with its currently defined token string value. At line 9, the token string value for LAST is the identifier michener, so michener is substituted at line 10. At line 12, the token string value for LAST is redefined to be the identifier joyce, so joyce is substituted at line 13. The following line is the final text that the compiler processes:

```
writer = james + joyce;
```

Comments within the definition line can be continued without the backslash/newline.

### 7.1.1 Object-Like Macros

The first form of the **#define** directive defines a simple substitution, usually of a constant for a mnemonic identifier. The identifier can be up to 31 characters. A common use of the directive is to define a replacement for an identifier as follows:

```
# define len (5 + 4)
total = 5 * len + 45
```

The substitution text in the preceding example is delimited with parentheses to avoid ambiguities when the text is substituted in the program. If the parentheses were omitted, then the expression that results from the substitution would not be evaluated as expected. For example:

```
# define len 5 + 4
total = 5 * len + 45
```

will be substituted with:

total = 5 * 5 + 4 + 45
Thus, since the precedence of the
*
operator is higher than
that of the + operator (refer to Table 4-2), the variable *total* is assigned the value 74 rather than 90.

### 7.1.2 Canceling Definitions (#undef)

The following directive cancels a previous definition of the identifier by **#define** :
#undef identifier

### 7.1.3 Function-Like Macros

Macros are text substitutions that include a list of parameters. A macro substitution looks like a function call. If you call a function, control passes from the program to the function object code at run time; if you reference a macro, source code is inserted into the program at compile time. The parameters are replaced by the corresponding arguments and the text is inserted into the program stream. The syntax of a macro definition follows:
#define name([parm1[,parm2,...]]) [token-string]
In the previous syntax definition, *name* , *parm1* , *parm2* , and so forth are identifiers, and token-string is arbitrary text. No space is allowed between *name* and the left parenthesis.
After the macro definition, all macro references in the source code with the following form are replaced by the token string from the directive.
name([arg1[,arg2,...]])
Any formal parameters that appear in the token string are replaced by the corresponding arguments from the reference. For example, argument *arg1* replaces parameter *parm1* , and so forth.
As shown in the syntax of the macro definition, the token string is optional. If the token string is omitted from the macro definition, every subsequent occurrence of the macro reference (including actual arguments) is deleted from the text to be processed by the compiler.
The token string in the macro definition, as well as actual arguments in a macro reference, may contain other macro references. If a macro definition either directly or transitively references itself, the recursive reference is not substituted.

The following is an example of macro substitution:

```
#define COMPLAIN(message) \
    (fprintf \
        (stderr, \
         "%s at line %d in file %s", \
         message, \
         __LINE__, \
         __FILE__))
/* ... */
if (i > LIMIT)
    COMPLAIN ("Variable i exceeds LIMIT");
```

The **#define** preprocessing statement above defines the COMPLAIN macro. The subsequent reference to the COMPLAIN macro is replaced with the following:

```
if (i > LIMIT)
    (fprintf
        (stderr,
         "%s at line %d in file %s",
         "Variable i exceeds LIMIT",
         __LINE__,
         __FILE__));
```

Preprocessor directive and macro reference syntax is independent of the PDP-11 C language. The following list gives the rules for specifying macro definitions:

.

The macro name and the formal parameters are identifiers and are specified according to the rules for identifiers in the PDP-11 C language.

.

Spaces, tabs, and comments may be used freely within a **# define** directive. In particular, they may appear anywhere that the delta symbol ( ¡ ) appears in the following example:

```
# ¡ define ¡ name( ¡ parm1 ¡ ,<
MATH_CHAR>(uppercase_delta)parm2 ¡ ) ¡ \
¡ token-string ¡
```

.

White space cannot appear between the name and the left parenthesis that introduces the parameter list. White space may appear inside the token string and in the parameter list. Also, at least one space, tab, or comment must separate *name* from **define** . Comments may appear within the token string, but they do not become

part of the macro definition.

The following list gives the rules for specifying macro references:

.

Comments and white space characters (spaces, horizontal and vertical tabs, carriage returns, newlines, and form feeds) may be used freely within a macro reference. In particular, they may appear anywhere that the delta symbol ( ¡ ) appears in the following example:
¡ name ¡ ( ¡ arg1 ¡ ,
¡ arg2 ¡ )

.

Arguments consist of arbitrary text. Syntactically, they are not restricted to PDP-11 C expressions. They may contain embedded comments and white space. Comments are ignored, but white space is preserved during the substitution.

.

The number of arguments in the reference must match the number of parameters in the macro definition, but individual arguments may be null.

.

Commas separate arguments except where they occur inside string literals or character constants, comments, or parentheses. You must balance parentheses within arguments.

You must be careful when specifying macro arguments that use the increment (++), decrement (- -), and assignment (such as +=) operators or other arguments that may cause side effects. Function calls are another source of possible side effects. For example, you can define a macro called **upcase** as follows:

#define upcase(c) ((c) >= 'a' &&(c) <= 'z' ? (c) &0X5F: (c))

If the argument p++ is given to this macro, the effect within the program stream may not be as desired. At run time, these expressions may not be evaluated in left-to-right order. For this reason, specifying macro arguments that may cause side effects is not good programming practice. Even if you are aware of possible side effects, the token strings within macro definitions may be changed, which changes the side effects without warning.

### 7.1.3.1 Stringizing Preprocessing Operator ( # )

Unlike some previous implementations of C, the ANSI C Language Standard does not allow the substitution of macro arguments within string literals or character constants. Instead, the # stringizing preprocessing operator (possibly in combination with string literal concatenation; see Section 2.15) is used for a similar function. The number sign ( # ) operator may be specified before a parameter in the replacement text to enclose the actual argument within quotations, as follows:

```
#define DISPLAY_SHUTDOWN(min) \
    puts ("System shutting down in " # min "minutes")
/* ... */
DISPLAY_SHUTDOWN (5);
```

The **# min** above is replaced with ``5" during macro substitution.

For example,

DISPLAY_SHUTDOWN (5)

is replaced with

puts ("System shutting down in " "5" "minutes")

and after string literal concatenation becomes:

puts ("System shutting down in 5 minutes")

---

**Note**

---

The number sign ( # ) operator can be used only for macros with arguments.

---

### 7.1.3.2 Token Concatenation Preprocessing Operator (##)

The ## token concatenation preprocessing operator can be used to concatenate two preprocessing tokens in macro replacement text into a single token. This feature is useful in forming token spellings based on actual arguments in macro substitutions. After actual argument substitution and before rescanning for nested macro invocations, the preprocessing tokens occurring to the left and right of the ## operator are concatenated to form a single token as follows:

```
#define INITIALIZE_LIST(list_name) \
    ((list_name ## _head = NULL), (list_name ## _tail = NULL))
/* ... */
```

```
INITIALIZE_LIST (students);
INITIALIZE_LIST (instructors);
```

The previous two invocations of the INITIALIZE_LIST macro will be expanded as follows:

```
(students_head = NULL, students_tail = NULL);
(instructors_head = NULL, instructors_tail = NULL);
```

### 7.1.4 Listing Substituted Lines

The /SHOW command line qualifier has two optional values that enable the listing of all lines that have been modified by macro substitutions. The values are EXPANSION and INTERMEDIATE.

Consider the following qualifiers:

```
/LIST/SHOW=EXPANSION
```

The listing produced by the compiler with the previous qualifiers shows both the original line and the final form of the substituted line. Substituted lines are flagged in the margin with numbers designating the nesting level of substitution.

Consider the following qualifiers:

```
/LIST/SHOW=INTERMEDIATE
```

The compiler lists all intermediate substitutions with one substitution per line.

Without one of these two qualifiers or /SHOW=ALL, the compiler lists only the original form of an error-free line. When a message is cited against a line, the final form of the substituted line is always shown.

Example 7-1 in Section 7.1 shows the effect of the /SHOW=INTERMEDIATE qualifier. For more information concerning the format of PDP-11 C compiler listings, refer to Chapter 1.

### 7.2 Conditional Compilation (#if, #ifdef, #ifndef, #else, #elif, #endif)

Six directives are available to control conditional compilation. They delimit blocks of statements that are compiled if a certain condition is true. You can nest these directives. The beginning of the block of statements is marked by one of three directives: **#if** , **#ifdef** , or **#ifndef** . Optionally, an alternative block of statements can be set aside with the **#else** or the **#elif** directives. The end of the block is marked by an **#endif** directive.

If the condition checked by **#if** , **#ifdef** , or **#ifndef** is true, then PDP-11 C ignores all lines between an **#else** or **#elif** and an **#endif** directive.

If the condition is false, then the lines between the **#if** , **#ifdef** , or **#ifndef** and an **#else** , or **#elif** or **#endif** directive are ignored. The compiler flags ignored lines with the letter X in the compiler listing margin.

The **#if** directive has the following form:

#if constant-expression

This directive checks whether the constant expression is nonzero (true). The operands must be integer constants. The increment (++), decrement (- -), **sizeof** , pointer ( * ), address (&), and cast operators are not allowed in the constant expression.

The constant expression in an **#if** directive is subject to text replacement and can contain references to identifiers defined in previous **#define** directives. The replacement occurs before the expression is evaluated.

If an identifier used in the expression is not currently defined and is not an operand of the defined operator, the compiler issues an informational message and treats the identifier as though it were the constant zero.

The **#ifdef** directive has the following form:

#ifdef identifier

This directive checks whether the identifier is currently defined by a **#define** directive.

The **#ifndef** directive has the following form:

#ifndef identifier
This directive checks to see if the identifier is not defined or if it has been undefined by the **#undef** directive.
The **#else** directive has the following form:
#else
This directive delimits alternative source lines to be compiled if the condition tested for in the corresponding **#if** , **#ifdef** , **#ifndef** , or **#elif** directive is false. An **#else** directive is optional.
The **#elif** directive has the following form:
#elif constant-expression
The **#elif** line performs a task similar to the combined use of the **#else** and **#if** statements in PDP-11 C. This directive delimits alternative source lines to be compiled if the condition in the corresponding **#if** , **#ifdef** , **#ifndef** , or previous **#elif** directive is false *and* if the additional constant expression presented in the **#elif** directive is true. An **#elif** directive is optional.
The **#endif** directive has the following form:
#endif
This directive ends the scope of the most recent **#if** , **#ifdef** , or **#ifndef** directives.
The number of **#endif** statements must correspond exactly to the number of **#if** , **#ifdef** , or **#ifndef** statements. The **#endif** statement must occur in the same source file as the corresponding **#if** , **#ifdef** , or **#ifndef** statement. You must not specify an **#endif** statement to correspond with an **#elif** statement.

### 7.2.1 The defined Operator

If you need to check to see if many tokens are defined, you may use the preprocessing **defined** operator in a single use of the **#if** directive. In this way, you can check for token definitions in one concise line without having to use many **#ifdef** or **#ifndef** directives.
For example, the following three **#ifdef** ... **#endif** sequences check three tokens:

```
# ifdef token1
printf( "Oh, Mary!\n" )
# endif
# ifndef token2
printf( "Oh, Mary!\n" )
# endif
```

```
# ifdef token3
printf( "Oh, Mary!\n" )
# endif
```

You can use the **defined** operator in a single use of the **#if** preprocessor directive, as follows:

```
# if defined (token1) || !defined (token2) || defined (token3)
printf( "Oh, Mary!\n" )
# endif
```

You can only use the **defined** operator in the evaluated expression of an **#if** or **#elif** preprocessor directive.

### 7.3 The #error Directive

The **#error** directive has the following form:

#error tokens

This directive produces a diagnostic message that includes the specified sequence of preprocessing tokens. For example:

```
#if ARRAY_SIZE != 5
#error "ARRAY_SIZE" is assumed to be 5, but is not
#endif
```

The following message would be displayed:

```
%PDP11C-W-LEX_USER_ERROR, User declared error: "ARRAY_SIZE" is assumed
to be 5, but is not
```

### 7.4 File Inclusion (#include)

The **#include** directive inserts external text into the token stream delivered to the compiler. Often, global definitions for use with PDP-11 C functions and macros are included in the program stream with the **#include** directive. PDP-11 C supports nesting of **#include** files to at least eight levels. In a given compilation, PDP-11 C may support higher levels of **#include** file nesting depending on available resources.

**Note**

Unlike VAX C, PDP-11 C does not support text modules and text libraries with the **#include** directive.

#### 7.4.1 Inclusion Using Angle Brackets ( <> )

The first form of the directive follows:

#include <file-spec>

This form of file inclusion delimits the file specification with angle brackets ( <> ). It is generally used with header files supplied with PDP-11 C.

The identifier file-spec is a valid file specification or a logical name. The compiler first translates the specified file name to see if it is a valid file specification. If the specification is not a valid file specification, an error occurs.

For the bracketed form, the order of search follows:

1. The directories specified in the /INCLUDE_DIRECTORY qualifier (if any).
2. The directory or search list of directories specified in the logical name PDP11C$INCLUDE on VMS, RSX-11M-PLUS, Micro /RSX, and RSTS/E systems (if any).
3. The directory specified in the logical name CLB on RSX-11M/M-PLUS, Micro /RSX, RSTS/E, and RT-11 systems (if any).
4. The directory or search list of directories specified by LB:[1,1] (on VMS and RSX-11M/M-PLUS systems),

CC$: (on RSTS/E systems), and SY: (on RT-11 systems).
PDP-11 C uses the first occurrence of the specified file that it finds according to the search order for the bracketed form.
If the specified file cannot be found in any of the previously described locations, an error is reported.
You *cannot* define PDP11C$INCLUDE to be a rooted directory or subdirectory of the following form:
DBA0:[dir-name.]
When defining PDP11C$INCLUDE, use complete directory specifications.
For more information concerning search lists, refer to the DCL command DEFINE in the *VMS DCL Dictionary* .
Table 7-1 lists the logical names for the PDP-11 C host environments and their correspondence to VAX C logical names (if any).

### 7.4.2 Inclusion Using Quotation Marks ( " " )

The second form of the **#include** preprocessor directive follows:
#include "file-spec"
This form of file inclusion delimits the file specification with quotation marks ( " " ). It is generally used with user-defined header files.
For the quoted form, the order of search follows:
1. The directory containing the top-level source file
2. The directories specified in the /INCLUDE_DIRECTORY qualifier (if any)
3. The directory or search list of directories (if any) specified in the logical name C$INCLUDE on VMS, RSX-11M-PLUS, and Micro /RSX systems
4. The current default directory (DK: on RT-11)
5. If all the previous searches fail, the search order for the bracketed form is used as shown in Section 7.4.1.

PDP-11 C uses the first occurrence of the specified file that it finds according to the search order for the quoted form.
If the specified file cannot be found in any of the previously described locations, an error is reported.
Note that the compiler *first* searches the directory containing the compiled source file for the included file, *not* the current default directory. With PDP-11 C, the source file is the first top-level source file, the .C file.
For example, given the current directory, DBA0:[CURRENT], and the following CC command line, the compiler first

searches DBA0:[OTHERDIR] for any included files delimited by quotation marks, even though the current RMS default is the directory, DBA0:[CURRENT]:

$ **cc dba0:[otherdir]example.c**

In VMS and RSX-11M-PLUS environments, you have the flexibility of defining C$INCLUDE to be any valid directory or list of directories you choose before each compilation of your program. At the DCL or PDP-11 C command level, you may use the /INCLUDE_DIRECTORY qualifier to provide an additional search level for include files.

As with the PDP11C$INCLUDE, do not define C$INCLUDE to be a rooted directory or subdirectory. Use complete directory specifications when defining C$INCLUDE.

For more information concerning search lists, refer to the DCL command DEFINE in the *VMS DCL Dictionary* . For a correspondence of logical names used by PDP-11 C on each host system and by VAX C, refer to Table 7-1.

---

**Note**

---

If you include a file from LB:[1,1] by using angle brackets and the included file contains a second **#include** line that delimits the file specification with quotation marks, the compiler first searches the directory containing the top-level source file for the specified file, not LB:[1,1].

---

### 7.4.3 Token Substitution in #include Directives

PDP-11 C allows macro substitution within the **#include** preprocessor directive.

For instance, if you want to include a file name, you can use the following two directives:

```
# define token1 "file.ext"
# include token1
```

If you use defined tokens in **#include** directives, the tokens must evaluate to one of the two following acceptable **#include** file specifications, or PDP-11 C generates an error message:

<file-spec>

"file-spec"

[next] [previous] [contents]

### 7.5 Specification of Line Numbers (#line, #)

The PDP-11 C compiler keeps track of information about relative line numbers in each file involved in the compilation. It uses the number when it delivers diagnostic messages to the terminal and listing, and when it expands the _ _LINE_ _ and _ _FILE_ _ macros. The compiler increments the line counter for the subsequent lines from the line number specified by the **#line** directive. The directive can also specify a new file specification for the program source file. The **#line** directive will not change the line numbers in the left margin of your compilation listing, only the line numbers given in messages (for example, error messages) and in the expansion of the _ _LINE_ _ and _ _FILE_ _ predefine macros.

The formats of the **#line** directive follow:

**#line** constant identifier
**#line** constant string
# constant identifier
# constant string

The compiler gives the line following a **#line** directive the number specified by the parameter constant. The second parameter can be specified as either a PDP-11 C identifier or a string literal. It supplies a valid PDP-11 file specification. The character string must not exceed 255 characters.

**Note**

Omission of the **#line** keyword is provided for compatibility with VAX C and is not defined by the ANSI standard.

[next] [previous] [contents]

### 7.6 Specification of Module Name and Identification (#module)

The **#module** directive is provided as an alternate syntax of the **#pragma module** directive for compatibility with VAX C. For more information, refer to Section 7.7.3.

**Note**

The **#module** directive is provided for compatibility with VAX C. To enable access to this directive, compile using the /NOSTANDARD switch.

## 7.7 Implementation-Specific Preprocessor Directive (#pragma)

This section describes the implementation-specific preprocessor directives, or pragmas, that are available in the PDP-11 C compiler. The **#pragma** directive is a standard method for implementing features that vary from one C compiler to the next.

The following rules apply to the use of PDP-11 C pragmas:

- No pragmas have any effect between different compilation units of the same compilation.
- Unless otherwise noted, the use of upper- and lowercase alphabetic characters is significant.
- The preprocessing tokens following the **#pragma** keyword, up to the terminating newline, are subject to macro replacement unless in single or double quotes.
- Using pragmas that PDP-11 C does not recognize results in an informational message.

### 7.7.1 #pragma charset

The charset pragmas specify the source, message, list, and execution character sets respectively. The charset pragmas that you can specify in PDP-11 C are as follows:

# **pragma** *charset*

*2*
*6*
*4*

source
message
list
execution

*3*
*7*

<charset_name>

The source, message, list, and execution character sets are initially set to iso-latin-1. You can change any of these character sets from the initial iso-latin-1 default. Once you have changed a character set from iso-latin-1, you may specify the same new character set any number of times in the compilation unit.

For the message, list, and execution character sets, you cannot specify a second character set change for the same pragma in the same compilation unit. For example, if you change the list character set to french, you can specify french any number of times for the list character set in this compilation unit, but you cannot specify german for the list character set in the same compilation unit. This restriction does not apply to the source character set. You can change the source character set to a new character set any number of times in the same compilation unit.

The source charset specifies the character set of the source file. If you issue the source charset pragma in a source file that is cited in the command line (but not in an included file), the specified character set becomes the new default and current character set.

Source files that you specify in the command line are presumed to be in the default source character set unless a source charset pragma is encountered. Files that you include with the **#include** directive are presumed to be in the character set of the including file unless a source charset pragma is encountered. When the end of an included file is reached, the source character set reverts to that of the including file.

PDP-11 C processes source files internally in the iso-latin-1 character set. Compilation time increases when the source character set is other than iso-latin-1.

The charset message pragma specifies the character set of the user terminal, if interactive, or the log file, if batch. This pragma should be the first item in the source file. Any messages that are displayed before this pragma is encountered will be displayed in the iso-latin-1 character set.

The list charset pragma specifies the character set of the device on which the listing file is to be displayed.

The execution charset pragma specifies the character set

of the environment in which the compiled user program will execute. String literals and character set constants are translated to the execution character set. You must specify this pragma before the first string literal or character constant, or an error is signaled.
Each of the source, message, list, and execution character sets may be specified independently of each other. Alternatively, all four character sets may be set to the same value in a single directive by not specifying a source, message, list, or execution keyword in the **#pragma** charset directive.
The following is an example of the **#pragma** charset directive. In this example, the french_canadian character set is specified for the device on which the listing file is to be displayed:

```
#pragma charset list french_canadian
```

The following example shows how to set the source, message, list, and execution character sets to the finnish character set in a single directive:

```
#pragma charset finnish
```

PDP-11 C supports the character sets shown in Table 7-2.

When using the #pragma charset source directive, use trigraphs to represent those characters that are not available in the specified source character set. For example:

```
#pragma charset source british
/* Note effect of British source - trigraphs required */
??=pragma charset list iso_latin_1
int printf();
main ()
{
    printf("#\n"); /* Script-L will print */
    printf("??=\n); /* '#' will print */
}
```

Another example is:

```
#pragma charset source italian
/* Note effect of Italian source - trigraphs required */
??=pragma charset list iso_latin_1
int printf();
main ()
??<
    printf("??=??/n"); /* '#' will print */
??>
```

Digital recommends that you do not specify Swiss as a

source character set, because the "_" character has a Swiss replacement for which there is no corresponding trigraph. However, using the Swiss character set as a message, execution, or listing character set poses no problems.

### 7.7.2 #pragma psect

The psect pragmas specify the program sections where generated code and data are allocated. The psect pragmas that you can specify in PDP-11 C are as follows:

#pragma psect const [<psect_name>[,<attributes>,...]]
#pragma psect static_ro [<psect_name>[,<attributes>,...]]
#pragma psect static_rw [<psect_name>[,<attributes>,...]]
#pragma psect code_i [<psect_name>[,<attributes>,...]]
#pragma psect code_d [<psect_name>[,<attributes>,...]]

The psect_name has the form of any other C identifier; however, the compiler will translate this identifier to a 6-character, Radix-50 name using the same rules as used for global storage variables. For more information on global storage variables, see Section 6.5.1.

The psect name is optionally followed by a list of psect attributes. Valid attributes are: ro, rw, i, d, lcl, gbl, rel, abs, con, ovr, sav, nosav. These attributes are identical to those which follow the MACRO-11 .PSECT directive, except that they must be specified in lowercase. For more information, see the *PDP-11 C Run-Time Library Reference Manual* . Table 7-3 lists the types of code or data associated with each psect type.

The scope of the psect pragmas ranges from just after the psect pragma until the next psect pragma of the same type, or the end of the compilation unit, whichever comes first.

If you specify a psect pragma with no psect name and attributes, the default PDP-11 psect of the type you specified is assumed.

When you specify a psect for the first time, the default psect attributes are assumed for any unspecified attributes. When you subsequently specify a psect, you can specify only the same attributes or leave them unspecified. Once you establish attributes for a psect, you cannot change them. The attributes of the PDP-11 C default psects cannot be changed.

In addition, note the following:

.

The pragma psect const can only be issued once for each

compilation unit.

.

The pragma psect code_i and code_d can only be issued outside a function body.

.

For more information on the pragma psect static_ro and pragma static_rw, see Section 6.8.

### 7.7.3 #pragma module

When you compile source files to create an object file, the compiler assigns to the object file the last file name (from left to right) of those specified in the compilation unit. Files separated in the command line with the ``+" concatenation operator form a compilation unit. By default, this same name (truncated to 6 characters) is used as the module title that is carried internally to the object file and that appears in compiler and object-librarian listings and load maps. By default, the compiler also gives the module a V1.0 version identification.

For example, the following command line will create an object file named MYPROGRAM.OBJ, which is internally identified as MYPROG V1.0:

$ **cc myheader.h + myprogram.c**

To change the internal module title and version, use the **#pragma module** directive or the **#module** directive.

The syntax of the **#pragma module** directive follows:

#[pragma] module

*8*
*<*
*:*

identifier
string

*9*
*=*
*;*

*2*
*4*

[,] identifier
[,] string

*3*
*5*

The first identifier or string in the **#pragma module** directive refers to the module title and contains up to 6 Radix-50 characters not including a space. The optional second identifier or string refers to the version and is a string of up to 6 Radix-50 characters. Radix-50 characters are the uppercase letters A through Z, the digits 0 through 9, space (``"), period ( . ), and dollar sign ( $ ). Lowercase letters are converted to uppercase.

---

**Note**

---

The **#module** directive (without the pragma keyword) is provided for compatibility with VAX C. Use of the **#module** directive (without the pragma keyword) causes a warning if /NOSTANDARD=ANSI is specified on the command line.

---

You may specify this directive only once for each compilation unit.

### 7.7.4 #pragma list

The list pragmas enable or disable the listing and control the running title and subtitle fields at the head of every page in the listing. The list pragmas that you can specify in PDP-11 C are as follows:

#pragma list on
#pragma list off
#pragma list title "string"
#pragma list subtitle "string"

The list on and list off pragmas enable or disable the listing respectively. PDP-11 C implements a listing-enabled counter similar to that of MACRO-11. Initially, the counter is 0. A **#pragma list off** directive decrements the counter. A **#pragma list on** directive increments the counter. The

listing is enabled whenever the value of the counter is greater than or equal to 0; otherwise the listing is disabled.
You must specify the /LIST qualifier for the list pragmas to have an effect. If you do not specify the /LIST qualifier or if you specify the /NOLIST qualifier, the list pragmas have no effect. The list pragmas also have no effect on the listing of machine code if you specify the /SHOW=MACHINE qualifier.
The list title pragma specifies a title that appears at the top of every page of the listing. You may specify this pragma only once for each compilation unit. You may specify up to 44 arbitrary characters.
The list subtitle pragma specifies a subtitle to appear at the top of every page of the listing. You may specify this pragma any number of times for each compilation unit. You may specify up to 44 arbitrary characters. The use of upper- and lowercase for alphabetic characters is significant.

### 7.7.5 #pragma linkage

The linkage pragmas are used to define the exact calling mechanism for functions. The pragma defines the function's linkage so that later in the compilation unit when the function is either defined or referenced, the function will be called with the previously defined linkage.
The syntax for the linkage pragma follows:
#pragma linkage linkage-specifier [function [,function]...]
The linkage specifier can be one of six specifiers: c, pascal, fortran, rsx_ast, rsx_sst, rsx_csm. The linkage specifier is optionally followed by a list of function names.
If function names follow the linkage specifiers, those functions will be given that linkage. If no function names follow the linkage specifier, the #pragma sets the default linkage for all functions that follow. That is, all functions whose linkage has not been explicitly specified using another **#pragma linkage** will take on that linkage. This default linkage remains in effect for the rest of the compilation unit or until another **#pragma linkage** occurs without function specifiers. If no linkage is specified, the function will be called with the C linkage.
In the following example, funct1 and funct2 are assigned the Pascal linkage, funct3 is assigned the FORTRAN linkage, and funct4 is assigned the C linkage.

```
#pragma linkage fortran /*Assigns fortran linkage
                                    to any function not
```

```
                                        specifically assigned
                                        a linkage until another
                                        general linkage is defined.*/
#pragma linkage pascal funct1,funct2 /*Assigns pascal linkage
   int funct1(); to funct1 and funct2.*/
   int funct2();
   int funct3();
#pragma linkage c /*Assigns c linkage to
   int funct4(); any function not specifically
                                          assigned a linkage from
                                          this point on.*/
```

Note that you should not specify a linkage without function names in a header file or you may inadvertently redefine your calling mechanism for the rest of your compilation unit. For more specific information on the effect of the linkages pragma, see the chapter on using PDP-11 C with other PDP-11 languages in the *PDP-11 C Run-Time Library Reference Manual* .

### 7.7.6 #pragma [no]standard

Use **#pragma nostandard** to tell PDP-11 C to ignore the current setting of the command line qualifier /STANDARD=ANSI until further notice. It has no effect if the qualifier was not specified.

The **#pragma nostandard** directive has the following format:

```
#pragma [no]standard
```

The **nostandard** and **standard** pragmas are used together to define regions of source code where portability diagnostics are never to be issued. The following example demonstrates the use of these pragmas:

```
#pragma nostandard
globalvalue int MAXERR = 10;
#pragma standard
```

In this example, **nostandard** prevents the issuance of a diagnostic against the **globalvalue** storage class qualifier, which is not defined by the ANSI C language standard.

If the compiler detects more occurrences of the **nostandard** pragma than it does the **standard** pragma, the following informational message is issued:

```
LEX_MISPRAGMASTAND, Mismatched #pragma standard preprocessor directive (s)
```

When this message appears, check that each **nostandard** pragma has a matching **standard** pragma, both in the main

source file and in any included files.

### 7.8 Predefined Macros

The following sections describe the predefined macros defined by the ANSI C Language Standard and the PDP-11 C predefined macros that you can use in your programs.

#### 7.8.1 PDP-11 C Predefined Macros

The PDP-11 C compiler defines the following preprocessor substitutions; these symbols are defined as if the following text fragment were included by the compiler before every compilation unit. These macros have two leading underscores, which conforms to the ANSI C Language Standard.

```
#define __pdp11 1
#define __pdp11c 1
#define __dec_c 1
#define __vms_host 1 /* Only on VAX/VMS hosts */
#define __rsx_host 1 /* Only on RSX hosts */
#define __rsts_host 1 /* Only on RSTS/E hosts */
#define __rt11_host 1 /* Only on RT-11 hosts */
#define __PDP11 1
#define __PDP11C 1
#define __DEC_C 1
#define __VMS_HOST 1 /* Only on VAX/VMS hosts */
#define __RSX_HOST 1 /* Only on RSX hosts */
#define __RSTS_HOST 1 /* Only on RSTS/E hosts */
#define __RT11_HOST 1 /* Only on RT-11 hosts */
```

You can use these definitions to separate portable and nonportable code in any of your PDP-11 C programs. The symbols can be used by a PDP-11 C programmer to conditionally compile PDP-11 C programs used on more than one operating system to take advantage of system-specific features. See Section 7.2 for more information concerning the use of the preprocessor conditional compilation directives.

#### 7.8.2 Digital Extension Macros

The CC$gfloat and PDP11 macros are Digital extensions. Because these two macro names do not begin with two leading underscores, they are not ANSI conformant and are *not* defined when the /STANDARD=ANSI qualifier is

specified (even after a **#pragma nostandard** directive). The CC$gfloat macro is defined for compatibility with VAX C:
#define CC$gfloat 0
Under VAX C, the CC$gfloat macro expands to 1 if you specify the /G_FLOAT qualifier; otherwise, the CC$gfloat macro is 0. The CC$gfloat macro enables VAX C programmers to conditionally compile sections of code that depend on the representation of double objects. Because PDP-11 systems and PDP-11 C do not support the G-float format, PDP-11 C defines CC$gfloat as 0, indicating to a VAX C program that is ported to PDP-11 C that the G-float format is not being used for double objects.
The PDP11 macro is defined for compatibility with other C language processors on PDP-11 systems:
#define PDP11 1

### 7.8.3 The _ _DATE_ _ Macro

The _ _DATE_ _ macro evaluates to a string specifying the date on which the compilation started. The string presents the date in the following format:
Mmm-dd-yyyy
The first d is a space if dd is less than 10.
The following is an example of how to use the _ _DATE_ _ macro:
printf("%s",__DATE__);

### 7.8.4 The _ _TIME_ _ Macro

The _ _TIME_ _ macro evaluates to a string specifying the time when the compilation started. The string presents the time in the following format:
hh:mm:ss
The following is an example of how to use the _ _TIME_ _ macro:
printf("%s", __TIME__);

### 7.8.5 The _ _FILE_ _ Macro

The _ _FILE_ _ macro evaluates to a string specifying the file specification of the current source file. The string presents file in the following format:
disk:[directory]filename.extension;n
The following is an example of how to use the _ _FILE_ _ macro:
printf("file %s", __FILE__);

The expansion of the _ _FILE_ _ macro can be altered with the **#line** directive (see Section 7.5).

### 7.8.6 The _ _LINE_ _ Macro

The _ _LINE_ _ macro evaluates to an integer specifying the number of the line in the source file containing the macro reference. The number presents the line in the following format:

n

The following is an example of how to use the _ _LINE_ _ macro:

```
printf("At line %d in file %s", __LINE__, __FILE__);
```

The expansion of the _ _LINE_ _ macro can be altered with the **#line** directive (see Section 7.5).

### 7.8.7 The _ _STDC_ _ Macro

The _ _STDC_ _ macro evaluates to the decimal constant 1. The following is an example of how to use the _ _STDC_ _ macro:

```
#ifdef __PDP11C
#define PASTE(a,b) a##b
#elif __STDC__
#define PASTE(a,b) a##b
#else
#error cannot define the PASTE macro in this environment
#endif
```

The _ _STDC_ _ macro is defined only if /STANDARD=ANSI is specified on the command line. The _ _STDC_ _ macro can be used to determine at compile time if the compilation environment supports the ANSI C Language Standard.

### 7.8.8 The _ _RAD50 and _ _RAD50L Macros

PDP-11 C provides two macros for specification of radix-50 constant values. The _ _RAD50 macro takes a one- to three-character string literal argument and converts it to a short-word radix-50 value. The _ _RAD50L macro takes a one- to six-character string literal argument and converts it to a long-word radix-50 value. In both cases, the radix-50 value is shown in the listing represented as an octal constant if /LIST is selected and either /SHOW=EXPANSION or /SHOW=INTERMEDIATE is selected on the command line. The Example 7-2 illustrates using both of these macros.

Key to Example 7-2:

**1** The _ _RAD50 macro expansion is shown in the listing as a short-word octal constant representing the argument in radix-50.

**2** The _ _RAD50L macro expansion is shown in the listing as a long-word octal constant representing the argument in radix-50.

Note that the _ _RAD50 and _ _RAD50L macros are PDP-11 C extensions and may not be portable to other C environments.

[next] [previous] [contents]

---

## 8. PDP-11 C Implementation Notes

### 8.1 Use of Memory Management Functions

The PDP-11 C Runtime system maintains a list of free space which can be allocated by calls to the **malloc** , **calloc** , or **realloc** functions. On programs linked with the RSX taskbuilder on RSX or RSTS/E systems, this space is initially the space between the end of the program code and the end of the task's window 0. On programs linked with the RT-11 Linker on RT-11 or RSTS/E, the initial free space is obtained by doing a .SETTOP #-2 during initialization of the job which obtains all of the free space possible for the job.

When memory is returned by use of the **realloc** or **free** functions, the returned memory is linked to the head of a circularly linked list of free memory.

When memory is requested, the free list is searched for a block of memory large enough to accommodate the request. When the first such area is found, that block of free space is reduced by the amount of memory requested, the requested memory is allocated, and a pointer to it returned.

If no single block of free space large enough to accommodate the request is found after searching the entire free list, a consolidation operation takes place. During this consolidation operation, any adjacent blocks of free memory are merged into a single block. Also the free list is re-ordered from low memory to high memory. After the consolidation operation, the free list is searched again to see if the request can now be accommodated.

For programs linked with the RT-11 Linker, if the request still cannot be accommodated after consolidation of free space, the function returns indicating that the request cannot be fulfilled.

For programs linked with the RSX taskbuilder, an attempt to extend the task is made, and the space obtained is added to the free list. The amount of the task extension will be the amount of memory requested, rounded up to the next highest 256 word increment. If this task extension request fails, the maximum available task extension will be performed.

After extending the task, the free list is again searched for a block large enough to accommodate the request. If this fails, a second consolidation operation is performed, and the list is searched again. Finally, if this fails, the function returns

indicating that the request cannot be fulfilled.
Programs linked with the RSX taskbuilder can increase the size of the initial area of free space at taskbuild time by using the EXTTSK taskbuilder option. At installation time the area can be increased by the use of the /INC qualifier to the RSX MCR INS command or the /EXTENSION qualifier to the RSX DCL command INSTALL. At run-time, the area can be increased by the use of the /INC qualifier to the RSX MCR RUN command or the /EXTENSION qualifier to the RSX DCL RUN command.
By knowing how much your task will grow, you can pre-extend the initial allocation of free space using one of the above commands and save some or all task extensions from being done. Alternately, you could extend the task at run-time to its maximum possible size by invoking the **malloc** function with a size of 65535U. Although this returns a value of zero, it does extend the task to the maximum size.
On RSX, in order for a task extension to be done, the task must be checkpointable. One way to do this is by linking the task using the /CP taskbuilder option. Alternatively, you could either use the /CHECKPOINT qualifier to the RSX DCL INSTALL or RUN commands, or the /CKP qualifier to the RSX MCR INS or RUN command. If the task is not checkpointable, only the free space initially available to the task will be available.
The RSTS/E taskbuilder accepts the /CP switch but ignores it.

### 8.1.1 Providing Alternative Space for Memory Management

PDP-11 C programs which use memory resident overlays, or are linked using the /PR:n switch cannot use memory management functions. Programs that mix PDP-11 C routines with routines written in other languages that use similar methods for memory management, could have problems when each language tries to manage memory in the same place.
The above problems can be overcome by providing the PDP-11 C RTL with a fixed area of memory to be used. The space provided must reside in the root of overlaid programs. The size of this space is fixed and cannot be changed at program run-time. Tasks that provide memory in this manner do not need to be checkpointable, and the /CP switch is not required when taskbuilding on RSX systems.

To provide this alternative space for memory management, declare an array of the desired size. The size of the space provided should be a multiple of 4.
Fill in the global symbol C$MEMU with the starting address of the array. The location following C$MEMU should be filled in with the starting address of the array, plus the size of the array.
The following example provides an area of 4096 bytes for memory management:

```
static char memspace[4096];
const char *C$MEMU[2] =
        {
        memspace,
        memspace+sizeof(memspace)
        };
```

## 8.2 Compilation Performance and Capacity on PDP-11 Host Systems

The following sections describe how data caching, as well as placement and size of the work file, effect compilation performance.

### 8.2.1 Data Caching

On PDP-11 hosts, PDP-11 C uses a disk-based work file with one or two levels of data caching. The first level of caching, or primary cache, is done in mapped memory (within the 32K-word virtual address space of the PDP-11 C compiler task). The primary cache is part of the PDP-11 C compiler task image; it is always present in memory when PDP-11 C is running and is not present in memory when PDP-11 C is not in memory. PDP-11 C uses a primary cache on all PDP-11 host systems.

The second level of caching, or secondary cache, is done in unmapped memory (beyond the 32K-word virtual address space of the PDP-11 C compiler task). This feature is optional and is not available on host systems that do not support the I/D space feature. If selected through the /MEMORY command line qualifier, PDP-11 C attempts to obtain additional, physical memory from the host operating system. If available, this additional memory is used as a secondary cache. Whenever data overflows the primary cache, a region of the secondary cache is mapped and the data is stored in the secondary cache. Similarly, when data cannot be found in the primary cache, a region of the secondary cache is mapped and searched for the desired data. While a secondary cache access is somewhat slower than a primary cache access, it is significantly faster than a disk access.

The larger the value specified with the /MEMORY qualifier, the greater the performance and capacity of PDP-11 C. Only when PDP-11 C processes sufficient data that it overflows the primary and secondary caches does it begin to use the disk file, and even then the caches continue to be used to maximize performance. If the secondary cache obtained from the host operating system is larger than the disk file specified or defaulted with the /WORK_FILE_SIZE qualifier,

the disk file is not used at all. Note that while each 4K-word region specified with the /MEMORY qualifier is equivalent to 15 disk blocks specified with the /WORK_FILE_SIZE qualifier, the requested number of secondary cache regions may not be available, resulting in a smaller or no secondary cache. Also note that the additional memory used by PDP-11 C is unavailable to other tasks, applications, and other simultaneous invocations of PDP-11 C while the invocation of PDP-11 C that obtained the extended memory is running.

### 8.2.2 PDP-11 C Work File

Performance can also be enhanced by placing the PDP-11 C work file on a fast, random access device. For instance, on RT-11 host systems, performance can be made comparable to that obtained through the extended memory feature on RSX and RSTS/E systems by using the VM virtual device for the PDP-11 C work file. On all PDP-11 host systems, PDP-11 C attempts to open the file on device WF:. If this fails, PDP-11 C then opens the work file in a host-specific location. Assigning a fast, random access device to WF: can significantly improve performance.

Performance of PDP-11 C can be impaired when large values are specified with the /WORK_FILE_SIZE qualifier. When a large value is specified, PDP-11 C must use additional virtual memory to extend its bitmap of used/unused disk blocks. Disk blocks are managed in sets of eight. Thus, a 1-word bitmap can keep track of 128 blocks. PDP-11 C maintains a minimum bitmap of 64 words, which is sufficient for up to 8192 blocks. If the value specified with the /WORK_FILE_SIZE qualifier is between 8193 and 40960, a block buffer is removed from the primary cache and is used to extend the bitmap, thereby decreasing the primary cache hit rate and negatively impacting performance. Furthermore, if the value is specified between 40961 and 65535, two 1-block buffers are required to extend the bitmap, further impairing performance. Therefore, values greater then 8192 should only be specified with very large compilations that require it, or when performance is not a consideration. Naturally, the number of blocks specified must be available on the work file device; these blocks remain unavailable to other tasks and applications while PDP-11 C is running.

[next] [previous] [contents]

---

### 8.3 PDP-11 C Run-Time Psects

This section describes the psects used by the PDP-11 Run-Time Library. Table 8-1 lists each of the run-time library psects and their use.

### 8.4 Overlaying Tasks

Some PDP-11 C tasks that work when not overlaid may fail during program startup when they are overlaid. This can happen because of the way task startup information is included into the root of the task.

The PDP-11 C OTS work area which is located in the psect $$C includes a vector of initialization functions which may be required by standard library functions used in the task. Programs which use only some functions do not need to include all the overhead of initialization, nor do they need space for functions which the task does not use.

Modules which require startup routines cause the linker to pull the initialization function only as far toward the root as is necessary to resolve names. In a non-overlaid program, all routines are properly included in the root. However, in an overlaid program, the initialization may not make it into the root. When the program runs, it is likely to fail.

For example, a task might consist of a root and two segments (A and B). If, in this example, **malloc** is only referenced in segment A and **printf** is only referenced in segment B and the root segment requires no initialization for either memory management or I/O functions, then the initialization will fail. Each segment indicates the initialization it needs, but this will not make it into the root initialization psect $$C.

Therefore, when PDP-11 C code is used in overlaid tasks, you must explicitly build a segment into the root that references all modules which contribute to the $$C psect.

To determine which modules are needed, examine the task map and look at the contents of the $$C psect (in any segment).

The following program uses a simple overlay structure. The root segment calls the branch, which calls **assert** .

ROOT.C:

```
extern void func1 ();
int main () {
    func1 ();
}
```

BRANCH.C:

```
#include <assert.h>
void func1 () {
```

```
    int i = 1;
    assert (i == 1);
}
TEST.CMD
test/cp,test/-sp=test/mp
TEST.ODL
        .root root-libr-tree
libr: .fctr lb:[1,1]cfpursx/lb
tree: .fctr *(branch-libr)
        .end
```

When this program is taskbuilt and run, it will fail during initialization (before main( ) is called). The following example shows fragments of TEST.MAP:

```
TEST.MAP:
TEST.EXE;1 Overlay description:
Base Top Length
---- --- ------
000000 002673 002674 01468. ROOT
002674 043175 040302 16578. BRANCH
*** Root segment: ROOT
Memory allocation synopsis:
Section Title Ident File
------- ----- ----- ----
. BLK.:(RW,I,LCL,REL,CON) 001260 000000 00000.
    :
$$C :(RW,D,GBL,REL,OVR,SAV) 001670 000076 00062.
                            001670 000000 00000. VEXTA 06.07 CFPURSX.OLB;9
                            001670 000076 00062. C$INIT V01.09 CFPURSX.OLB;9
    :
*** Segment: BRANCH
Memory allocation synopsis:
Section Title Ident File
------- ----- ----- ----
. BLK.:(RW,I,LCL,REL,CON) 002674 000202 00130.
    :
$$C :(RW,D,GBL,REL,OVR,SAV) 001670 000076 00062.
                            001670 000074 00060. C$SIGD V01.03 CFPURSX.OLB;9
                            001670 000060 00048. C$EXID V01.04 CFPURSX.OLB;9
                            001670 000070 00056. C$SIOD V01.09 CFPURSX.OLB;9
                            001670 000066 00054. C$MLLD V01.03 CFPURSX.OLB;9
    :
```

As you can see, $$C in BRANCH is made up of contributions from modules C$SIGD, C$EXID, C$SIOD, and C$MLLD

which are not mentioned in $$C of ROOT. To fix the problem, explicitly include them into the root:

MODIFIED TEST.ODL:

```
        .root root-init-libr-tree
libr: .fctr lb:[1,1]cfpursx/lb
init: .fctr lb:[1,1]cfpursx/lb:c$exid:c$mlld:c$sigd:c$siod
tree: .fctr *(branch-libr)
        .end
```

When the task is linked in this way, it will work correctly.

### 8.5 RT-11 User Service Routine (USR) Load Area

Under RT-11, if the USR is not resident, the PDP-11 C RTL will attempt to set the USR to swap at the location of the root C$STDI and C$OTSI psects. During program startup, the PDP-11 C RTL checks to see if the size of these psects is large enough to accommodate USR. If it is, location 46 in the job is set to the address of the C$OTSI psect. By doing this, USR will not take up any additional address space.

It is unlikely that a job that uses the PDP-11 C memory management routines will have less than 2K words of space in the C$STDI and C$OTSI psects. However, if less than 2K words of space is present and USR is not resident, the memory management initialization routine will set USR to swap at the high 2K of memory obtained by doing a .SETTOP #-2.

[next] [previous] [contents]

---

### 8.6 Event Flags

Under RSX, PDP-11 C uses event flag 24 when performing Standard Library I/O functions using the RSX native I/O or FCS I/O Packages.

### 8.7 Argument Passing Using Linkages

Linkages are used in PDP-11 C to define the exact internal calling mechanism used for function calls. A function may be assigned a linkage using the **#pragma linkage** directive as shown in Chapter 7. PDP-11 C supports the following linkages:

- PDP-11 C
- PDP-11 FORTRAN-77
- PDP-11 Pascal
- RSX AST
- RSX CSM
- RSX SST

For more information on the internal calling mechanisms, including stack and register usage of the six linkages, refer to the *PDP-11 C Run-Time Library Reference Manual* .

### 8.8 Defining Your Own Locales

PDP-11 C offers several pre-defined locales such as Danish, French, English, and C. In addition to these pre-defined locales, PDP-11 C allows users to define locales specific to their needs, and include these locales for use with PDP-11 C RTL functions such as **setlocale** and **localeconv** , as well as the character testing and mapping functions.

The information needed to define a locale is stored in a number of tables. The following sections describe the contents of these tables. After the tables have been created, they must be placed into the appropriate psects where they can be found by the **setlocale** and **localeconv** functions. The header file, <defloc.h>, is provided which defines several macros to assist in this. Please note that you are not required to use these macros, but they will make defining a locale easier.

The names of the locale macros in the <defloc.h> header file are:

.

To define collating locale:

```
DEFINE_LC_COLL( locale-name,
                4-char-gbl-nam,
                _order_table,
                _upcase_table,
                _downcase_table)
```

.

To define character-testing locale:

```
DEFINE_LC_CTYPE(locale-name,
                4-char-gbl-nam,
                _tab_table)
```

.

To define monetary formatting data locale:

```
DEFINE_LC_MONETARY( locale-name,
                    4-char-gbl-nam,
                    &MFT_TABLE)
```

.

To define non-monetary formatting data locale:

```
DEFINE_LC_NUMERIC( locale-name,
                   4-char-gbl-nam,
                   &MFT_TABLE)
```

.

To define the time formatting data locale:

```
DEFINE_LC_TIME( locale-name,
                4-char-gbl-nam,
                &TIM_STR_TABLE)
```

The format of the macro parameters found in the <defloc.h> header file are:

locale-name The pointer to the locale name, or string literal of the locale name (such as "C"). This is the string name that is used as an argument to the **setlocale** function to identify the locale.

4-char-gbl-nam A maximum of 4 characters which will serve as the global entry points into the appropriate locale psects. The macros will append a different two-character number onto each global name created to form the complete global name entry for each item placed into the appropriate PDP-11 C psects.

table,.. The address of the user-defined locale table.

An example of a macro is:

```
DEFINE_LC_NUMERIC( "user locale name",user,&MFT_TABLE)
```

This macro will place the address of the locale name and table address into the required PDP-11 C psect. It will also create global entry-point names to point to the placed items. The first names created will be USER00::, and USER01::. The global name USER00 will point to the locale name address found in the non-monetary formatting time psect used by PDP-11 C. The global name USER01 will point to the user's table address found within the non-monetary formatting time psect used by PDP-11 C . Thus, these symbols give the user direct access to the required psect fields used by PDP-11 C.

Because global names are created, the four-character names must be different for each macro call issued by the user. Otherwise, duplicate global symbol names will be created causing compilation or link time errors.

The format of each macro is defined in more detail within <defloc.h> header file. You should read the comments placed above each locale macro definition before attempting to use the macro in your C modules.

Although locales can have information in five categories, your defined locale need not provide information for all five categories. For example, if you only want a different collation

table, just provide that table; the other tables will remain the same.
Refer to Example 8-1 for some ideas on how to define your own locale tables.

Key to Example 8-1.

**1** Defines three structures for future use.

**2** The character set testing table contains one entry for each member of the character set. At the offset in the table equal to a particular character's value, an entry is made which determines whether that character tests TRUE or FALSE for the various character testing functions. This table is set when the setlocale category LC_CTYPE or LC_ALL is specified.
The following values are defined by <defloc.h>:

_U Character is uppercase
_L Character is lowercase
_D Character is a digit
_S Character is whitespace
_P Character is punctuation
_C Character is a control character
_X Character is a hexadecimal digit
_V Character is a printing character
These values can be *or* 'd together. In fact, the <defloc.h> header file defines the logical *or* for several of these:

_XD _X or _D
_XU _X or _U
_XL _X or _L
_SC _S or _C
For example, if the character 'a' has the value of 97 and it should test TRUE for the **isalnum** , **isalpha** , **isgraph** , **islower** , **isprint** , and **isxdigit** functions, then the 97th entry of the table would have the value _XL.

**3** The character set collation table contains one entry for each member of the character set. At the offset in the table equal to a particular character's value, an entry is made which determines the position of that character in the collation sequence. This table is set when the setlocale category LC_COLLATE or LC_ALL is specified.
For example, if the character 'a' has the value of 97

and you want it to be the first character in the collation sequence, the 97th entry of the table would have the value 1.

**4** The character set mapping tables contain one entry for each member of the character set. At the offset in the table equal to a particular character's value, an entry is made which determines the mapping of that character for the various character mapping functions. These tables are set when the setlocale category LC_CTYPE or LC_ALL is specified. There is an uppercase table and a lowercase table.
For example, if the character 'a' has the value of 97 and it should return 97 for the **tolower** function and 65 for the **toupper** function, the 97th entry of the lowercase table would have the value 97, and the 97th entry in the uppercase table would have the value 65.

**5** The time table defines the values returned by the **strftime** function. The lc_time_strings contain:

- 7-character string constants corresponding to the abbreviated names to be used for the seven days of the week
- 7-character string constants corresponding to the full names to be used for the seven days of the week
- 12-character string constants corresponding to the abbreviated names to be used for the twelve months of the year
- 12-character string constants corresponding to the full names to be used for the twelve months of the year
- 2-character string constants corresponding to identify AM and PM
- 24-character string constants corresponding to the 24 time zones beginning with GMT and proceeding west.

**6** The non-monetary formatting data table defines the values returned by the **localeconv** function. The lc_ nmformat structure is filled in with the desired contents

of the lconv structure to be returned by **localeconv** for this locale. This table is set when the setlocale category LC_NUMERIC or LC_ALL is specified.

**7** The monetary formatting data table defines the values returned by the **localeconv** function. The lc_mformat structure is filled in with the desired contents of the lconv structure to be returned by **localeconv** for this locale. This table is set when the setlocale category LC_MONETARY or LC_ALL is specified.

### 8.9 Excluding printf Format Support Code

The RTL support code for the **printf** family of functions ( **printf** , **fprintf** , **sprintf** , **vfprintf** , **vprintf** , **vsprintf** ) is sizable. Often, a program does not need all of the formatting flexibility provided by these functions.

This implementation of PDP-11 C allows you to exclude the support code for some of the conversion specifiers for formatted output from a task. Excluding the support code could save up to 3000 bytes of space. (See *PDP-11 C Run-Time Library Reference Manual* for more information about conversion specifications for PDP-11 C standard output.

By default, support for all formats is provided. Support for the c, s, and n formats cannot be excluded. Support for the d, i, o, p, u, x, X, f, e, E, g, and G formats can be optionally excluded.

To exclude the support routines for a particular format, include in one module of the program a **globalvalue** statement which defines the global symbol for that format with the value 0. See Section 6.7 for more information on **globalvalue** . Table 8-2 lists the symbols for each format.

For example, for a program that does not have f, e, E, g or G format output, the following two **globalvalue** statements should appear in the program:

```
globalvalue $PFLOA = 0;
globalvalue $PFLOE = 0;
```

The following program simply prints a constant literal string. It excludes all of the unnecessary support:

```
#include <stdio.h>
globalvalue $PULON = 0;
globalvalue $PLONG = 0;
globalvalue $POLON = 0;
globalvalue $PHLON = 0;
globalvalue $PFLOA = 0;
globalvalue $PFLOE = 0;
main ()
{
    printf ("hello, world\n");
}
```

If an attempt is made to use an excluded format, no

characters for that value will be printed. If the value specified in the **globalvalue** statement is not 0, the behavior is undefined.

If you link to the supervisor mode PDP-11 C Run-time library, support for all formats is always included. The support resides in the supervisor mode library.

[next] [previous] [contents]

---

## A. PDP-11 C Compiler Messages

This appendix lists the PDP-11 C compiler diagnostic messages.

## A.1 Introduction

For each message, the appendix gives the mnemonic, the message text, an explanation of the message, and suggested actions to be taken to avoid the message.

Some messages substitute information from the program in the message text. In this appendix, the portion of the text to be substituted is shown as "

****

" or

****

. If quotes appear around the asterisks, quotes appear in the substituted message.

There are four types of compiler messages: informational, warning, error, and fatal. Each type affects the compiler in a different way as follows:

- Informational-does not affect compiler; compiler still produces object, macro, and listing files (if selected).
- Warning-compiler still produces object, macro, and listing files; check your code to ensure accuracy.
- Error-compiler continues through current phase and produces object and listing files (if selected); no macro or object files are produced; if 30 or more error messages are issued, the current phase is aborted and a listing file (if selected) is generated; the default error limit of 30 can be changed with the /ERROR_LIMIT qualifier.
- Fatal-compiler aborts; no object or macro files are produced; only if possible, a listing file is produced.

You can suppress the warning and informational messages with the /[NO]WARNINGS qualifier on the PDPCC command line. You may want to do this so that the compiler broadcasts only the most severe messages to the terminal. For more information concerning the /[NO]WARNINGS qualifier, refer to Chapter 1.

The messages used by the PDP-11 C compiler are in a separate file. The compiler searches for the message file in the following locations and in following order:

.

**V AX/VMS:**

1. PDP11C$MESSAGES (logical),
2. PDP11C$MESSAGE_DIRECTORY:PDP11C$ENGLISH_ MSG.MSG,
3. SYS$COMMON:[SYSMSG]PDP11C$ENGLISH_ MSG.MSG.

.

**RSX-11M-PLUS:**

1. SY:PDP11C.MSG,
2. PDP11C$MESSAGES (logical),
3. PDP11C$MESSAGE_DIRECTORY:PDP11C.MSG,
4. LB:[1,2]PDP11C.MSG.

.

**RSX-11M:**

1. SY:PDP11C.MSG,
2. LB:[1,2]PDP11C.MSG.

.

**RSTS/E:**

1. SY:PDP11C.MSG,
2. CC:PDP11C.MSG,
3. CC$:PDP11C.MSG.

.

**RT-11/XM:**

1. DK:PDP11C.MSG,
2. CC:PDP11C.MSG,
3. SY:PDP11C.MSG.

[next] [previous] [contents]

---

## A.2 Compiler Messages

**ALC_TEMPOVERFLOW, Your program requires more temporaries than the compiler can handle**
**Fatal:** The expressions you used in your program require more temporaries than the compiler can handle.
**User Action:** Rewrite your program to use less complicated expressions.

**CD_CANT_OPEN_TERMINAL, Cannot open console I/O device**
**Fatal:** The console device (for example, terminal or batch log file) could not be opened for output.
**User Action:** Determine the error with the console device.

**CD_UNSUPVERS, Unsupported operating system version**
**Fatal:** PDP-11 C V1.0 supports V AX/VMS V5.0 and later.
**User Action:** Upgrade your V AX/VMS system to V AX/VMS V5.0 or later.

**CLP_AMBIG_QUAL, Ambiguous qualifier or keyword name**
**Error:** Too few characters were used to truncate a keyword or qualifier name to make it unique.
**User Action:** Reenter the command; specify at least enough characters of the keyword or qualifier name to make it unique.

**CLP_BAD_DELEM, Missing comma, or plus before file name**
**Error:** You have entered the wrong command line syntax.
**User Action:** Correct the syntax and reenter.

**CLP_BAD_OPFILE_ATTRIB, File has bad attributes**
**Error:** The specified command file to open had bad attributes.
**User Action:** Check the attributes of the file and reenter.

**CLP_CLFILE_ERROR, Unexpected file close error**

**Error:** During a file close operation, an error was encountered while closing the specified file.
**User Action:** Check the characteristic for the specified file to see why the file could not be closed and reenter.

**CLP_FINPUT_ERROR, Unexpected file input error**

**Error:** Input from the specified file could not be obtained.
**User Action:** Check the file attribute and protection for the specified file and reenter.

**CLP_FINPUT_LINE_LONG, File input line is too long**

**Error:** The record line length of the input file exceeded the maximum number of characters.
**User Action:** Shorten the record line length in the input file or verify that the file record attributes are correct.

**CLP_INCONSIST, CLP internal inconsistency in module; submit SPR**

**Error:** The Command Line Processor has detected an internal inconsistency.
**User Action:** Gather as much information as you can about the conditions in effect when the error occurred and submit a Software Performance Report (SPR).

**CLP_INPUT_ERROR, Unexpected input error**

**Error:** Input from the terminal could not be obtained.
**User Action:** Check the terminal attributes and reenter.

**CLP_INPUT_LINE_LONG, Input line is too long**

**Error:** The command line length exceeded the maximum number of characters.
**User Action:** Shorten the line length.

**CLP_INV_FILENAME, Invalid file name**

**Error:** The file name string for a file specification contains illegal characters or is too long.
**User Action:** Check for a programming error. Verify that the file name string is a valid file name.

**CLP_INV_INPUT, Invalid input**

**Error:** The specified user input was not a valid literal string or character string.
**User Action:** Check the user input for illegal characters and reenter.

**CLP_INV_INTEGER, Invalid integer value**

**Error:** The specified user input was not a legal integer value.
**User Action:** Check the specified user integer for characters which are not 0 to 9.

**CLP_INV_NOKWD, NO prefix on keyword name not allowed**

**Error:** An attempt was made to negate a qualifier keyword that cannot be negated.
**User Action:** Remove the negate prefix.

**CLP_INV_NOQUAL, NO prefix on qualifier name not allowed**

**Error:** An attempt was made to negate a qualifier that cannot be negated.
**User Action:** Remove the negate prefix.

**CLP_INV_NOQUAL_VAL, Value not allowed on NO qualifier**

**Error:** An attempt was made to give value to a negated qualifier.
**User Action:** Remove the value reference.

**CLP_INV_QUAL, Invalid qualifier name**

**Error:** The user-specified qualifier name was not recognizable.

**User Action:** Check for invalid string and reenter.

**CLP_INV_QUAL_VAL, Value not allowed for qualifier**

**Error:** An attempt was made to give a value to a qualifier that does not take a value.

**User Action:** Remove the value reference.

**CLP_MAX_OPFILE, CLP maximum number of open files exceeded**

**Error:** The maximum number of nested indirect command files has been exceeded.

**User Action:** Remove the reference to the indirect command file that causes the maximum nesting limit to be exceeded and reenter.

**CLP_MISS_GRPVAL, Missing keyword, or qualifier value when value is required**

**Error:** A qualifier was specified with group values, and no keyword list or value list was supplied.

**User Action:** Supply the appropriate value reference.

**CLP_MISS_PAREN, INVALID DELIMITER, Please supply ending parenthesis**

**Error:** A group value list was specified which contains an invalid value separator or does not contain an ending parenthesis.

**User Action:** Correct the group value syntax and reenter.

**CLP_MISS_VALUE, Invalid switch keyword or qualifier value when value is required**

**Error:** A qualifier was specified that requires a user keyword or value, and no value was supplied.

**User Action:** Supply the appropriate value reference.

**CLP_MODE_INCONSIST, Unexpected CLP file mode; submit SPR**

**Error:** The Command Line Processor has detected an internal file mode inconsistency.

**User Action:** Gather as much information as you can about the conditions in effect when the error occurred and submit an SPR.

**CLP_MODQUAL_INCONSIST, Only two string values allowed for the module qualifier**

**Error:** The user specified more than two string values for the module qualifier.

**User Action:** Supply the appropriate number of values and reenter.

**CLP_NO_GRPVAL, Qualifier does not allow group values**

**Error:** A qualifier was specified that does not allow a group list of values but contains a group list of values.

**User Action:** Supply the appropriate value reference type.

**CLP_NO_OPFILE, File not found**

**Error:** The user-specified file was not found.

**User Action:** Specify a file that exists and reenter.

**CLP_NOCL_QUOTE, No closing " on literal**

**Error:** You did not use matching quotation marks for a literal string.

**User Action:** Reenter the command using the correct quotation syntax.

**CLP_OPFILE_ERROR, Unexpected file open error**

**Error:** An error was encountered while opening the specified file.

**User Action:** Check the attributes for the specified file and reenter.

**CLP_QUAL_VAL_REQ, Value required for qualifier**

**Error:** A qualifier was specified that requires a user value and no value was supplied.
**User Action:** Supply the appropriate value reference.

**CLP_UNK_KWD, Unknown keyword name**
**Error:** The user-specified keyword was not recognizable.
**User Action:** Remove the unknown keyword reference and reenter.

**CLP_UNK_QUAL, Unknown qualifier name**
**Error:** The user-specified qualifier was not recognizable.
**User Action:** Remove the unknown qualifier reference and reenter.

**INCONSISTENCY, Internal inconsistency or stack overflow**
**Fatal:** An internal error or stack overflow has occurred within PDP-11 C.
**User Action:** Examine your program for complex expressions or declarations and simplify as necessary; if this does not resolve the problem, submit SPR.

**LEX_BADSTRINGSIZE, The # preprocessing operator must be followed by a parameter**
**Warning:** The # preprocessor operator was encountered in a macro definition but was not followed by a parameter.
**User Action:** Remove the # operator or specify a parameter after the # operator.

**LEX_CLOSE_FAILED, Error closing source file**
**Fatal:** An unexpected error occurred when closing a source file.
**User Action:** Determine the cause of the error and correct.

**LEX_CMT_UNCLOSED, Unterminated comment**
**Warning:** The end of file was reached before encountering the
*

/ comment terminating delimiter.
**User Action:** Terminate the comment.

**LEX_CONSTTOOLONG, Numeric constant is too long; truncated to** "
*****

**1**
*****

"
**Warning:** Too many digits were encountered in a numeric constant.
**User Action:** Reduce the number of digits in the numeric constant.

**LEX_DEFTOOLONG, Text in a #define preprocessor directive is too long; directive ignored**
**Warning:** The length of the token-string in the **#define** directive exceeded the implementation's limit.
**User Action:** Simplify the directive.

**LEX_DUPPARAMETER, Duplicate parameter ignored**
**Warning:** The identifier for a macro parameter was encountered more than once in the formal parameter list.
**User Action:** Remove or change the duplicate parameter identifier.

**LEX_EXECHARSETDEF, The execution file character set cannot be defined twice in a compilation unit; directive ignored**
**Warning:** You cannot specify the **#pragma charset execution** directive more than once in a compilation unit.

**User Action:** Remove the redundant directive.

**LEX_EXECHARSETREF, The execution file character set has already been used; directive ignored**
**Warning:** You have specified a **#pragma charset execution** directive after a string literal or character constant has already been processed.
**User Action:** Place the directive before any string literals or character constants.

**LEX_EXPECTEDEOL, End of line expected**
**Warning:** Unexpected text was encountered in a preprocessing directive.
**User Action:** Remove or place the extraneous text in a comment.

**LEX_EXTRAMODULE, Redundant #module preprocessor directive ignored**
**Warning:** You specified more than one **#module** directive in a single compilation; the excess directive or directives were ignored.
**User Action:** Make sure that only one **#module** directive exists in the source file, and that it is placed before any PDP-11 C source code.

**LEX_EXTRATITLE, Redundant #pragma list title preprocessor directive ignored**
**Warning:** You cannot specify the **#pragma list title** directive more than once in a compilation unit.
**User Action:** Remove the redundant directive.

**LEX_FLOAT_E_NODIGITS, Illegal floating point constant**
**Warning:** No digits were specified to the right of the E in a floating-point constant.
**User Action:** Specify an exponent value to the right of the E in the floating-point constant.

**LEX_IFEVALDIVZ, Division by zero while evaluating #if or #elif expression; ``true'' expression assumed**
**Warning:** The specified expression contains a division by zero.

**User Action:** Modify the expression to avoid a division by zero.

**LEX_IFEVALSTACK, Stack overflow while evaluating #if or #elif expression; ``true'' expression assumed**
**Explanation:** The specified expression is too complex.
**User Action:** Simplify the expression using fewer levels of parentheses, and so on.

**LEX_IFSYNTAX, Syntax error in #if or #elif expression; ``true'' expression assumed**
**Warning:** A preprocessor token was encountered in a context where it was not expected.
**User Action:** Remove or correct the preprocessor token.

**LEX_ILL89_OCT, The digits 8 and 9 are not octal digits**
**Warning:** The digit 8 or 9 was encountered in an octal constant.
**User Action:** Use only the digits 0 to 7 in the octal constant.

**LEX_ILL_BSC, Illegal backslash sequence in string or character constant**
**Warning:** An unrecognized escape sequence was encountered in a string literal or character constant.
**User Action:** Reference the list of recognized escape sequences in Table 5-3 and use only recognized escape sequences. Alternatively, use an octal or hexadecimal escape sequence.

**LEX_ILLDBLCON, Illegal double constant**
**Warning:** The specified floating-point value cannot be represented as a double precision floating-point constant.
**User Action:** Correct the floating-point constant.

**LEX_ILLDECCON, Illegal decimal constant**
**Warning:** The specified decimal value cannot be represented as an integer constant of the specified or defaulted type.
**User Action:** Specify a different type suffix or correct the constant.

**LEX_ILLFLTCON, Illegal float constant**
**Warning:** The specified floating-point value cannot be represented as a single precision floating-point constant.
**User Action:** Correct the floating-point constant.

**LEX_ILL_HEX, Illegal hexadecimal sequence in string or character constant**

**Warning:** A nonhexadecimal digit was encountered in a hexadecimal escape sequence in a string literal or character constant.

**User Action:** Use only hexadecimal digits in a hexadecimal escape sequence.

**LEX_ILLHEXCON, Illegal hexadecimal constant**

**Warning:** The specified hexadecimal value cannot be represented in an integer constant of the specified or defaulted type.

**User Action:** Specify a different type suffix or correct the constant.

**LEX_ILLINCLDIR, Illegal device/directory specification with /INCLUDE_DIRECTORY qualifier**

**Explanation:** An illegal directory specification was encountered in a /INCLUDE_DIRECTORY qualifier.

**User Action:** Specify a legal directory specification with the /INCLUDE_DIRECTORY qualifier.

**LEX_ILLNUMCONST, Illegal numeric constant; trailing characters ignored**

**Warning:** Additional characters were encountered at the end of a numeric constant.

**User Action:** Remove the additional characters or separate the numeric constant from the next token with a space.

**LEX_ILLOCTCON, Illegal octal constant**

**Warning:** The specified hexadecimal octal cannot be represented in an integer constant of the specified or defaulted type.

**User Action:** Specify a different type suffix or correct the constant.

**LEX_INVALIDIF, Invalid constant or operator in #if or #elif expression; ``true'' expression assumed**

**Warning:** You used an invalid construction in an **#if** or **#elif** expression, which is assumed to be true.

**User Action:** Correct the expression.

**LEX_INVDEFNAME, Missing or invalid name in #define preprocessor directive; directive ignored**
**Warning:** The indicated directive was missing a required name. For example:

```
# define
```

The entire directive was ignored.
**User Action:** Correct or remove the directive.

**LEX_INVFILESPEC, Missing or invalid file specification in #include preprocessing directive; directive ignored**
**Error:** A **#include** preprocessor directive was encountered with a form other than one of the following:

```
#include <FILESPEC>
#include "filespec"
#include macro_id
```

The specification macro_id is a macro that expands to one of the preceding two forms.
**User Action:** Specify the **#include** directive using one of the forms shown above.

**LEX_INVHEXCHAR, Invalid hexadecimal character value; high-order bits truncated**
**Warning:** An escape character specified in hexadecimal exceeded the limit of a 1-byte character.
**User Action:** Correct the hexadecimal constant to represent a valid escape character.

**LEX_INVLINEFILE, Invalid file specification in #line preprocessor directive; directive ignored**
**Warning:** The file specification was syntactically invalid, and the directive was ignored.
**User Action:** Correct the directive.

**LEX_INVLINELINE, Missing or invalid line number in #line preprocessor directive; directive ignored**
**Warning:** The line number was missing or was syntactically invalid, and the directive was ignored.
**User Action:** Correct the directive.

**LEX_INVLISTTITLE, Missing or invalid title specification in #pragma title/subtitle preprocessor directive; directive ignored**

**Warning:** A preprocessor token other than a string literal was encountered in a **#pragma list title** or **#pragma list subtitle** preprocessor directive.

**User Action:** Specify the listing title or subtitle as a string literal.

**LEX_INVMODIDENT, Missing or invalid ident specification in # [pragma] module preprocessor directive; directive ignored**

**Warning:** The ident specification in the directive either was not a valid identifier or was not a valid character-string constant.

**User Action:** Correct the directive.

**LEX_INVMODTITLE, Missing or invalid title specification in # [pragma] module preprocessor directive; directive ignored**

**Warning:** The required title in the directive either was missing or was not a valid identifier.

**User Action:** Correct the directive.

**LEX_INVOCTALCHAR, Invalid octal character value; high-order bits truncated**

**Warning:** The octal value in an escape sequence was too large, as in ' \477 ' . Its high-order bits were truncated.

**User Action:** Correct the value.

**LEX_INVPPKEYWORD, Missing or invalid keyword in preprocessor directive; directive ignored**

**Warning:** You wrote a directive with no keyword. For example:

```
# ABC
```

The directive is ignored.

**User Action:** Correct or remove the directive.

**LEX_IOBADATTR, Illegal file attributes**

**Error:** PDP-11 C does not support the attributes of the specified file.

**User Action:** On VMS and RSX host systems, convert

the file to sequential organization, variable length records, carriage return carriage control format, and maximum record length no greater than 510. On RSTS/E host systems, convert the file to RSTS/E native format, and maximum record length no greater than 510.

**LEX_IOEXISTS, File exists**

**Error:** PDP-11 C has attempted to open a new file that should not already exist.
**User Action:** Delete or rename the file.

**LEX_IOFNF, Error opening source file - file not found**

**Error:** The specified source file could not be found.
**User Action:** Check the spelling of the filespec, the assignment of the C$INCLUDE and PDP11C$INCLUDE logical names, and the directories specified with the /INCLUDE_DIRECTORY qualifier.

**LEX_IOLINETOOLONG, Line too long**

**Error:** PDP-11 C does not support source file input with lines greater than 510 characters.
**User Action:** Shorten the line length of lines that exceed 510 characters; use the ``\'' lexical continuation operator if necessary.

**LEX_IONOROOM, No room on device**

**Error:** PDP-11 C attempted to open a new file on a device that had insufficient room available.
**User Action:** Delete or purge files on the device to make additional room or specify another device.

**LEX_IOUNEXPECTED, Unexpected I/O error**

**Error:** PDP-11 C encountered an unexpected I/O error on the specified file.
**User Action:** Determine the cause of the error and correct.

**LEX_IOUNEXPEOF, Unexpected end of file**

**Error:** PDP-11 C encountered the end of an input file in a context where this was not expected.
**User Action:** Determine the cause of the error and correct.

**LEX_LISCHARSETDEF, The listing file character set cannot be defined twice in a compilation unit; directive ignored**

**Warning:** You cannot specify the **#pragma charset list** directive more than once in a compilation unit.
**User Action:** Remove the redundant directive.

**LEX_LISCHARSETREF, The listing file character set has already been used; directive ignored**
**Warning:** This message should not occur.
**User Action:** Submit an SPR.

**LEX_MACNORPAREN, Missing ')' in macro invocation; ')' assumed**
**Warning:** A functionlike macro invocation was encountered without a closing right parenthesis.
**User Action:** Complete the macro invocation with a closing right parenthesis.

**LEX_MACSYNTAX, Syntax error in macro definition; directive ignored**
**Warning:** The syntax of the parameter list in a macro definition was invalid. (You must enclose the parameter list in parentheses and delimit individual parameters with commas.)
**User Action:** Correct the syntax.

**LEX_MACUNEXPEOF, Unexpected end-of-file encountered in a macro reference; macro not substituted**
**Error:** The end-of-file was encountered during a macro reference; the reference was deleted.
**User Action:** Check whether you have misplaced the closing parenthesis in the macro argument list.

**LEX_MAXMACNEST, Maximum text replacement nesting level exceeded; macro invocation not substituted**
**Error:** You have specified a macro reference that causes substitutions to a depth greater than the implementation limit of 100.
**User Action:** Simplify the macro definitions.

**LEX_MESCHARSETDEF, The message character set cannot be defined twice in a compilation unit**
**Warning:** You cannot specify the **#pragma charset message** directive more than once in a compilation unit.
**User Action:** Remove the redundant directive.

**LEX_MESCHARSETREF, The message character set has already been used; directive ignored**
**Warning:** You have specified a **#pragma charset message** directive after a message has already been processed (possibly during command line parsing).
**User Action:** Correct any command line errors to avoid command line messages.

**LEX_MISPARENS, Mismatched parentheses in #if or #elif expression; ``true'' expression assumed**
**Warning:** The expression in a **#if** or **#elif** preprocessor directive contained unbalanced parentheses.
**User Action:** Make sure that you balanced the parentheses in the expression.

**LEX_MISSENDIF, Missing #endif preprocessor directive(s)**
**Error:** The compiler did not encounter an **#endif** line for the most recent **#if** , **#ifdef** , or **#ifndef** .
**User Action:** Be sure that all directives are properly structured, and, if appropriate, add the missing **#endif** preprocessor directives.

**LEX_MODULENOTANSI, The #module directive is not in conformance with ANSI C; use #pragma module**
**Warning:** The **#module** directive is provided for VAX C compatibility and is not a portable construct.
**User Action:** Use the **#pragma module** directive for identical processing or specify /NOSTANDARD on the command line.

**LEX_NAMETOOLONG, Identifier name exceeds 31 characters; truncated to** "
********

"
**Warning:** PDP-11 C identifiers are limited to a length of 31 recognized characters.

**User Action:** Shorten the indicated identifier.

**LEX_NOLINEKWNOTANSI, Omitting the "line" keyword from the #line directive is not in conformance with ANSI C; use #line**
**Warning:** The implicit **#line** directive ( # followed by a line number) is provided for compatibility with VAX C and is not a portable construct.
**User Action:** Use the **#line** directive for identical processing or specify /NOSTANDARD on the command line.

**LEX_NONTERMCHAR, Nonterminated character constant**
**Warning:** The compiler encountered the end of the source line before the end of a character constant. The compiler assumed the indicated value.
**User Action:** Correct the constant.

**LEX_NOTRAD50, The specified value cannot be represented in a RADIX-50 long word; directive ignored** "
********

"
**Warning:** The specified value must be composed of 1-to-6 alphanumeric characters, the dollar sign (``$"), or the period (``.").
**User Action:** Correct the value.

**LEX_NOWIDELIT, A wide character string literal is not allowed in this context**
**Warning:** A wide-character string literal was encountered in a context where a normal string literal is expected.
**User Action:** Remove the L prefix from the string literal.

**LEX_NULCHARCON, Character constant contains no characters, ' \0 ' assumed**
**Warning:** The compiler detected a single apostrophe ( ' ) at the end of the source line.

**User Action:** Check whether the apostrophe is extraneous; otherwise correct the constant.

**LEX_NULHEXCON, Hexadecimal constant contains no digits; 0x0 assumed**

**Warning:** A hexadecimal escape constant was encountered that did not include any hexadecimal digits.
**User Action:** Correct the hexadecimal constant.

**LEX_PASTEATEND, The ## operator may not occur at the end of a macro definition**

**Warning:** The ANSI C Language Standard stipulates that the ## token-paste operator may not occur at the end of a macro definition.
**User Action:** Remove the ## token-paste operator from the end of the macro replacement text.

**LEX_PASTEUPFRONT, The ## operator may not occur at the beginning of a macro definition**

**Warning:** The ANSI C Language Standard stipulates that the ## token-paste operator may not occur at the beginning of a macro definition.

**User Action:** Remove the ## token-paste operator from the beginning of the macro replacement text.

**LEX_READ_FAILED, Error reading source file**

**Fatal:** An unexpected error occurred while reading a source input file.

**User Action:** Determine the cause of the error and correct.

**LEX_REDEFINE, Macro redefinition with different replacement text than a previous definition**

**Warning:** A previously defined macro was redefined in a subsequent **#define** preprocessor directive with a different value.

**User Action:** Use a different macro identifier or undefine the macro with the **#undef** preprocessor directive before redefining.

**LEX_REOPEN_FAILED, Error reopening source file**

**Fatal:** A source input file that had previously been successfully opened, read, and closed, could not be reopened by PDP-11 C.

**User Action:** Determine cause of the problem and correct.

**LEX_REPOVERFLOW, Length of macro expansion exceeds maximum buffer capacity; macro invocation not substituted**

**Error:** The length of the replacement text for a macro reference or the length of the text plus the rest of the line exceeded the implementation's limit.

**User Action:** Shorten the replacement text or use multiple substitutions to achieve the desired result.

**LEX_RESERVED,** "
********

" **is a reserved identifier;**
**directive ignored**
**Warning:** You have specified a reserved identifier name in a **#define** or **#undef** preprocessor directive. Such reserved names may not be redefined or undefined. They are as follows:

.
_ _DATE_ _
.
_ _FILE_ _
.
defined
.
_ _TIME_ _
.
_ _LINE_ _
.
_ _RAD50
.
_ _RAD50L
.
_ _STDC_ _

**User Action:** Choose a different spelling for the identifier.

**LEX_STR_UNCLOSED, Unterminated string literal**
**Warning:** End-of-line was encountered before the end of a string literal.
**User Action:** Terminate the string literal with a closing quote ( " ) character or continue the string literal using lexical continuation.

**LEX_TOOFEWMACARGS, Argument list contains too few arguments; missing arguments assumed to be null**
**Warning:** You wrote a reference to the indicated macro with fewer arguments than were specified in its definition.
**User Action:** Make sure that the number of arguments in the macro reference is the same as the number of

parameters in the definition.

**LEX_TOOMANYCHAR, Character constant contains too many characters; long int constant assumed and high-order bits truncated**
**Warning:** Too many characters were specified in a character constant to fit within a **long int** .
**User Action:** Specify no more than 4 characters.

**LEX_TOOMANYCHARINT, Character constant contains too many characters for an int constant; long int constant assumed**
**Informational:** Too many characters were specified in a character constant to fit within an **int** .
**User Action:** Specify no more than 2 characters.

**LEX_TOOMANYMACARGS, Argument list contains too many arguments; excess arguments ignored**
**Warning:** You wrote a reference to the indicated macro with more arguments than were specified in its definition.
**User Action:** Make sure that the number of arguments in the macro reference is the same as the number of parameters in the definition.

**LEX_TOOMANYMACPARM, Parameter list for macro contains too many parameters; excess parameters ignored**
**Warning:** The number of macro parameters in a **#define** preprocessor directive exceeded the implementation limit of 64.
**User Action:** Rewrite the macro definition so that it uses 64 or fewer parameters.

**LEX_UNDEFIFMAC, Identifier is not currently a macro; constant zero assumed**

**Informational:** The identifier in a constant expression in an **#if** or **#elif** preprocessor directive was not currently defined as a macro. The expression was evaluated as if the identifier were a constant zero.

**User Action:** Define the identifier as a macro or remove the reference to it.

**LEX_UNEXPELIF, Unexpected #elif preprocessor directive encountered; directive ignored**

**Warning:** The **#elif** preprocessor directive occurred out of place and was ignored.

**User Action:** Check the logic of all directives in the program to be sure that it is valid.

**LEX_UNEXPELSE, Unexpected #else preprocessor directive encountered; directive ignored**

**Warning:** The **#else** preprocessor directive occurred out of place and was ignored.

**User Action:** Check the logic of all directives in the program to be sure that it is valid.

**LEX_UNEXPENDIF, Unexpected #endif preprocessor directive encountered; directive ignored**

**Warning:** The **#endif** preprocessor directive occurred out of place and was ignored.

**User Action:** Check the logic of all directives in the program to be sure that it is valid.

**LEX_UNIMPLEMENTED, This feature is not implemented in this configuration of PDP-11 C**

**Warning:** Refers to a **#pragma x** where **x** is not supported under PDP-11 C.

**User Action:** Remove the line in your code that refers to the unsupported **#pragma** .

**LEX_UNKNOWN_CHAR, Unrecognized character**

**Warning:** The line contained either an entirely

meaningless character or one that appears out of its proper context; for example, a number sign ( # ) that was not the first character on a line.
**User Action:** Move or remove the character.

**LEX_UNRECPRAGMA, Unrecognized pragma; directive ignored**
**Informational:** You have specified a **#pragma** preprocessor directive that is not recognized by PDP-11 C.
**User Action:** Correct the syntactic or semantic error that rendered the directive unrecognizable. Common errors include misspelled parameters and ambiguous abbreviations.

**LEX_USER_ERROR, User declared error:** "
********

"
**Warning:** A **#error** directive was encountered.
**User Action:** Determine the conditions that cause the **#error** directive to be processed and correct or remove the **#error** directive.

**LEX_WCHARCONST, PDP-11 C supports minimal ANSI conformance for wide character constants; subsequent messages for this constant may be misleading**
**Informational:** A wide-character constant was encountered. PDP-11 C implements only minimal ANSI conformance for wide-character constants. Subsequent messages may be misleading.
**User Action:** Use a regular character constant.

**LEX_WCHARLITCONCAT, A wide character string literal cannot be concatenated with a regular string literal**
**Warning:** A wide-character string literal was encountered immediately after a regular string literal, or vice versa.
**User Action:** Within a string literal concatenation, use only wide-character string literals or regular string

literals, but not both.

**MIO_FLOATOVERFLOW, Float overflow; value not representable as a float; try double**

**Error:** The specified value could not be represented as a float.

**User Action:** Try to place the number into a double.

**MIO_STACKOVERFLOW, Stack overflow during machine-independent optimization; simplify expression**

**Fatal:** Machine-independent optimization has detected an internal stack overflow while optimizing an expression.

**User Action:** Simplify the expression; if the problem persists, submit an SPR.

**MRF_CLOSE, Unable to close the PDP-11 C message file**
**Error:** The compiler cannot close the message file.
**User Action:** Submit an SPR.

**MRF_FORMAT, Format error in the PDP-11 C message file**
**Error:** The compiler cannot understand the message file.
**User Action:** Ensure that the message file is in its proper location. If the file exists, but may have become corrupted by a disk failure, etc., then re-installing PDP-11 C should fix the problem. If the problem persists, submit an SPR.

**MRF_INTERN, Internal error accessing message file, please SPR**
**Error:** The compiler cannot find the message file.
**User Action:** Submit an SPR.

**MRF_OPEN, Unable to open the PDP-11 C message file**
**Error:** The compiler cannot find the message file.
**User Action:** Ensure that the message file is in its proper location.

**MRF_READ, Cannot read the PDP-11 C message file**
**Error:** The compiler cannot understand the message file.
**User Action:** Submit an SPR.

**MRF_SYNCH, PDP-11 C message file is incompatible with compiler image**
**Error:** The message file is not the same version as the compiler.
**User Action:** Ensure that the message file is in its proper location.

**OGN_FILE_EXISTS, Listing file already exists**
**Fatal:** The specified output listing file already exists. This error will only occur on a VMS or RSX system when an explicit version number is specified in the output file

specification.
**User Action:** Remove or change the explicit version number in the output file specification or delete the existing file.

**OGN_MAC_FILE_EXISTS, Macro file already exists**
**Fatal:** The specified output macro file already exists. This error will only occur on a VMS or RSX system when an explicit version number is specified in the output file specification.
**User Action:** Remove or change the explicit version number in the output file specification or delete the existing file.

**OGN_MAC_NO_ROOM, No room for macro file on device**
**Fatal:** Occurred because either the user directory was full and the output macro file could not be created, or the required disk space could not be allocated when writing to the file.
**User Action:** Delete existing files to provide room for new ones.

**OGN_MAC_UNEXPECTED_IO, Unexpected I/O error on macro file**
**Fatal:** An unexpected error occurred during creation of the output macro file.
**User Action:** Ensure the output file specification is valid, and the access exists to the output directory. See system manager.

**OGN_NO_MACRO_PRODUCED, No macro file produced**
**Informational:** Any error-level message prevents creation of the output macro file.
**User Action:** Correct all error-level messages.

**OGN_NO_OBJ_PRODUCED, No object file produced**
**Informational:** Any error-level message prevents creation of the output object file.
**User Action:** Correct all error-level messages.

**OGN_NO_ROOM_FOR_FILE, No room for listing file on device**

**Fatal:** Occurred because either the user directory was full and the output listing file could not be created, or the required disk space could not be allocated when writing to the file.

**User Action:** Delete existing files to provide room for new ones.

**OGN_OBJ_FILE_EXISTS, Object file already exists**

**Fatal:** The specified output object file already exists. This error will only occur on a VMS or RSX system when an explicit version number is specified in the output file specification.

**User Action:** Remove or change the explicit version number in the output file specification or delete the existing file.

**OGN_OBJ_NO_ROOM, No room for object file on device**

**Fatal:** Occurred because either the user directory was full and the output object file could not be created, or the required disk space could not be allocated when writing to the file.

**User Action:** Delete existing files to provide room for new ones.

**OGN_OBJ_UNEXPECTED_IO, Unexpected I/O error on object file**

**Fatal:** An unexpected error occurred during creation of the output object file.

**User Action:** Ensure the output file specification is valid, and the access exists to the output directory. See system manager.

**OGN_UNEXPECTED_IO, Unexpected I/O error on listing file**

**Fatal:** An unexpected error occurred during creation of the output listing file.

**User Action:** Ensure the output file specification is valid, and the access exists to the output directory. See system manager.

**OVL_ASYNCH, Asynchronous overlays not supported**

**Fatal:** The PDP-11 C compiler save image on your RT-11 system may be corrupted.
**User Action:** Reinstall the CC.SAV compiler save image.

**OVL_BIGROOT, Image root is too large**

**Fatal:** The PDP-11 C compiler save image on your RT-11 system may be corrupted or the device may have gone off-line.
**User Action:** Ensure that the device is on-line or reinstall the CC.SAV compiler save image as appropriate.

**OVL_HEADER, Error reading image header**

**Fatal:** The PDP-11 C compiler save image on your RT-11 system may be corrupted or the device may have gone off-line.
**User Action:** Ensure that the device is on-line or reinstall the CC.SAV compiler save image as appropriate.

**OVL_LABEL, Error reading save label**

**Fatal:** The PDP-11 C compiler save image on your RT-11 system may be corrupted or the device may have gone off-line.
**User Action:** Ensure that the device is on-line or reinstall the CC.SAV compiler save image as appropriate.

**OVL_NOMEM, Insufficient memory**

**Fatal:** There is not sufficient extended memory (i.e., memory obtained via the virtual .SETTOP programmed request) available on your RT-11 system to load the PDP-11 C compiler image.
**User Action:** PDP-11 C uses 4K words (8K bytes) of low-core memory and 28K words (56K bytes) of extended memory, or a total of 32K words (64K bytes). Make sure that sufficient memory is available to PDP-11 C.

**OVL_READ, Overlay read error**

**Fatal:** The PDP-11 C compiler save image on your RT-

11 system may be corrupted or the device may have gone off-line.
**User Action:** Ensure that the device is on-line or reinstall the CC.SAV compiler save image as appropriate.

**OVL_ROOT, Error loading image root**

**Fatal:** The PDP-11 C compiler save image on your RT-11 system may be corrupted or the device may have gone off-line.
**User Action:** Ensure that the device is on-line or reinstall the CC.SAV compiler save image as appropriate.

**OVL_ROOT2, Error loading latter part of image root**

**Fatal:** The PDP-11 C compiler save image on your RT-11 system may be corrupted or the device may have gone off-line.
**User Action:** Ensure that the device is on-line or reinstall the CC.SAV compiler save image as appropriate.

**OVL_SPAN, Image overlay cannot span address**

**Fatal:** The PDP-11 C compiler save image on your RT-11 system may be corrupted.
**User Action:** Reinstall the CC.SAV compiler save image.

**OVL_SHORT, Short word count loading image root**

**Fatal:** The PDP-11 C compiler save image on your RT-11 system may be corrupted or the device may have gone off-line.
**User Action:** Ensure that the device is on-line or reinstall the CC.SAV compiler save image as appropriate.

**OVL_SHORT2, Short word count loading latter part of image root**

**Fatal:** The PDP-11 C compiler save image on your RT-11 system may be corrupted or the device may have gone off-line.
**User Action:** Ensure that the device is on-line or reinstall the CC.SAV compiler save image as appropriate.

**OVL_VIRTOV, Virtual overlay error**

**Fatal:** An error occurred while processing a virtual overlay in the RT-11 hosted PDP-11 C compiler. The PDP-11 C compiler save image on your RT-11 system may be corrupted or the device may have gone off-line.
**User Action:** Ensure that the device is on-line or reinstall the CC.SAV compiler save image as appropriate.

**SYN_ARGEXTRA, Too many arguments**

**Warning:** The number of actual arguments is more than the number specified in the function's prototype or declaration.

**User Action:** Ensure that the number of arguments matches the number specified in the function's prototype or declaration.

**SYN_ARGINCOMPAT, Argument incompatible for assignment**

**Warning:** The actual argument passed to a function has a type incompatibility with the type specified in the function's prototype or declaration.
**User Action:** Correct the type of the argument, perhaps using a cast.

**SYN_ARG_LIST_TOO_LONG, Function reference specifies an argument list whose length exceeds the PDP-11 architecture limit**

**Error:** The size of your argument list in the function call exceeded 255 arguments.
**User Action:** Rewrite the function definition and function call using fewer arguments.

**SYN_ARGMISSING, Missing arguments**

**Warning:** The number of actual arguments is less than the number specified in the function's prototype or declaration.
**User Action:** Ensure that the number of arguments matches the arguments specified in the function's prototype or declaration.

**SYN_ARGSCALDEF, Can't perform default promotion on argument number** "

********

**" as it is not a scalar**

**Error:** The default promotion rules (used when a function has no prototype in scope) do not allow for passing nonscalar (for example, structure or union) types.
**User Action:** If the function allows nonscalar arguments, ensure that this call is preceded by its prototype; otherwise, correct the types of the actual arguments.

**SYN_ASNMLVALREQ, The left operand of an assignment operator must be an lvalue**

**Error:** The left operand of your assignment operator was not an lvalue.

**User Action:** Rewrite your expression so that you enable a location where the operand can be assigned. The rvalue (right-hand side) will be loaded into a temporary register and then placed into the storage of the lvalue (left-hand side).

**SYN_BADPSECT, The program section (psect) specified by this statement has conflicting 'nowrite' attributes with another definition of the same program section**

**Warning:** The psect shown was previously defined as read-only, and cannot be re-defined as read-write.

**User Action:** Make the psect definitions agree.

**SYN_BITWINTREQ, Both operands of a bitwise operator must have integral type**
**Error:** Both operands of your bitwise operator did not have integral type.
**User Action:** Change both your operands to have integral type.

**SYN_CASECONST, The " case " clause requires a constant**
**Error:** You specified a value in a **case** label that was not a constant.
**User Action:** Replace the **case** value with a valid constant expression.

**SYN_CASEDUP, Duplicate " case " clause value**
**Error:** The same label for a **case** statement appeared twice.
**User Action:** Rewrite to eliminate identical **case** clauses.

**SYN_CASTTYPE, A cast must be either a cast to void or a cast between scalar types**
**Error:** A cast cannot involve casting to a struct, union, or other nonscalar type.
**User Action:** Change the type of the cast.

**SYN_COND1SCALREQ, The first operand of a " ?: " operator must be a scalar**
**Error:** Your first operand of a ``?:" operator was not a scalar.
**User Action:** Change your first operand of a ``?:" operator to a scalar.

**SYN_CONFLICTDECL, This declaration of "**
********

"
**conflicts with a previous declaration of the same name**
**Warning:** Identical declarations conflict with each other.

**User Action:** Change one of the declarations so that they are not identical to each other.

**SYN_CONPSECTATTR, This psect has attributes conflicting with those previously specified**
**Warning:** The psect shown has attributes conflicting with those previously specified.
**User Action:** Make the psect attributes agree.

**SYN_DEFDUP, A " switch " statement may have only one " default " clause**
**Error:** Your **switch** statement has more than one **default** clause.
**User Action:** Rewrite your **switch** statement to have only one **default** clause.

**SYN_DUPDEFINITION, Duplicate definition of "**
********
"
**Warning:** The named definition appeared more than once in the program.
The two definitions are essentially the same. Both definitions specify the same data types and organizations, but there may be differences in the values, initializers, or array bounds. If the name is a function, there may be a difference in the number or types of parameters or in the contents of the function body.
**User Action:** The purpose of this message is to call a possible programming error to your attention.

**SYN_DUPGLOBALNAME , Duplicate global name**
"
********
"
**Warning:** This declaration of a global object conflicts with another global object declared previously. Since PDP-11 C truncates global names after the first 6 characters and converts them to uppercase, you need to ensure that the first 6 characters of your global names are unique.
**User Action:** Remove duplicate global name references.

**SYN_DUPLABEL, Duplicate label: "**

********

"

**Error:** You specified duplicates of the indicated label in the same function. (Label identifiers must be unique within a function definition.)
**User Action:** Rewrite the labels (and **goto** statements that refer to them) to eliminate the duplicates.

**SYN_DUPMAINFUNC, Duplicate main function**

**Error:** You defined two or more main functions in a single compilation unit.
A main function is a function with the name ``main''. If the compilation unit contains more than one main function, the compiler recognizes only the first as the main function.
**User Action:** Make sure that there is only one main function defined in the compilation unit.

**SYN_DUPMEMBER, Duplicate declaration of member**
"
********

"

**Warning:** You declared two members with the same name in the same structure.
**User Action:** Rename one of the members or remove one of the member declarations.

**SYN_DUPPARAMETER, Duplicate parameter** "
********

"

**Warning:** The stated function parameter occurred more than once in the function's formal parameter list. For example:

```
funct(a,b,c,a) { }
```

All occurrences of the parameter after the first are ignored.
**User Action:** Remove or change the duplicate parameter identifier.

**SYN_ENUMOVERFLOW, Overflow detected in evaluating enumerated item** "
********

"

**Warning:** The value of your enumerated item exceeds 32767.
**User Action:** Define your enumerated item value to be within the accepted boundaries.

**SYN_EXTRAFORMALS, Extraneous formal parameter(s) ignored in declaration of** "
********

"

**Warning:** You included a function's formal parameters

in a function declaration or definition.
For example, the following function declaration is not allowed because it names the function's parameters:

```
int funct(a,b,c);
```

The parameters a, b, and c are ignored.
Similarly, the following example defines a function returning a pointer to a function returning an integer. The names of the parameters of the function returning an integer are not allowed.

```
(*f(p1,p2))(q1,q2)
int p1, p2;
{ . . . }
```

The compiler ignores the parameters q1 and q2.
**User Action:** Check the syntax of the function declaration and, if appropriate, remove the extraneous identifiers.

**SYN_FATALSYNTAX , Fatal syntax error**
**Fatal:** The compiler could not continue due to syntax errors.
**User Action:** Correct the error in the indicated line, or errors, or both reported in previous compiler messages.

**SYN_FUNCNOTDEF, Static function** "
********
" **is not defined in this compilation**
**Error:** You did not define the **static** function within the compilation that references it.
**User Action:** Define the **static** function in the compilation that references it.

**SYN_FUNCOBJ, Function return type must be void or a completed object type**
**Error:** A function cannot return an incomplete object type.
**User Action:** Ensure that the return type is fully specified before the function declaration.

**SYN_IFSCALREQ, The controlling expression of an if statement must have scalar type**

**Error:** Occurs when an **if** statement has an incorrect type (such as **struct** as the controlling expression.
**User Action:** Make sure that the controlling expression of an **if** statement has scalar type.

**SYN_ILLCONDEXPR, The second and third operands of the " ?: " operator are of incompatible type**
**Error:** You specified an invalid combination of operands in a conditional expression.
This can occur if the operands are pointers to objects of a different size or type, or if the operands are different structures.
**User Action:** Make sure that both operands are of compatible sizes and data types.

**SYN_ILLFUNCCALL, Functions with RSX AST or RSX SST linkage can not be invoked directly**
**Error:** You invoked a function declared with RSX AST or RSX SST linkage. Functions with these linkages can not be invoked directly. They may be declared, have their address taken, and be passed as arguments to other routines. These functions gain control through the AST or SST respectively.
**User Action:** Do not invoke the function with RSX AST or RSX SST linkage.

**SYN_ILLFUNCRET, Functions with RSX AST or RSX SST linkage must be of type void**
**Error:** You declared a function with RSX AST or RSX SST linkage to be other than type void.
**User Action:** Declare the function with RSX AST or RSX SST linkage to be of type void.

**SYN_ILLFUNCPARAM, Illegal parameter for a function with RSX AST or RSX SST linkage**
**Error:** You have declared a function with RSX AST or RSX SST linkage, which has an illegal parameter. A parameter to a function with RSX AST or RSX SST linkage must be of word size. In addition, functions with RSX AST linkage must have at least four parameters, and functions with RSX SST linkage must have at least two parameters. If any of the above conditions are not met, you will get this error.

**User Action:** Correct the parameter.

**SYN_ILLFUNCTYPE, Function-valued expression not found**

**Error:** A call with a function designator must have type function or a pointer to function.
**User Action:** Rewrite your expression to include either type function or a pointer to function.

**SYN_ILLTYPEASN, Incompatible types for assignment**

**Error:** Your assignment contains incompatible types.
**User Action:** Rewrite your assignment keeping in mind that the rules for type compatibility in assignment also apply to argument compatibility between actual argument expressions and their corresponding argument types in a function prototype.

**SYN_ILLTYPEINIT, Incompatible types for initialization. Initializer ignored**

**Warning:** Initializing values must be of compatible type.
**User Action:** Check the type of the object being declared and the initializer and ensure that they have the same type.

**SYN_INCOBJTYPE, Type of object "**
********
**" must be complete**

**Error: Auto** , **register** , and **globaldef** objects must have a complete type.
**User Action:** Ensure that the type is completed before the object is declared.

**SYN_INCOMPATRET, Type of returned expression is incompatible with function's type**

**Error:** The result of **return** must be of a compatible type of the declared function.
**User Action:** Ensure that your **return** has a type compatible with the declared function.

**SYN_INCSTRUCTARG, A function argument may not have incomplete type**

**Error:** Occurs when a structure or union argument has incomplete type.

**User Action:** Make sure that all function arguments have object type.

**SYN_INTVALERROR, Integer value not used where required**

**Warning:** You used a noninteger value as an initializer for an **enum** constant, or to specify the size of a bit field. You must specify these values as integer constants.

**User Action:** Specify an integer constant.

**SYN_INVALINIT, The initialization of "**

********

**" is not valid**

**Warning:** The indicated object cannot be initialized as specified. Some objects may not be initialized at all, such as functions, unions, and **extern** or **globalref** objects. In other cases, the initializer may not be appropriate, for example, a static pointer cannot be initialized with the address of an automatic variable. This and any subsequent initializers for the same object have been ignored.

**User Action:** Eliminate or correct the initializer, or correct the type or storage class of the target object, or initialize the object with an explicit assignment.

**SYN_INVARRAYBOUND, The declaration of "**

********

"

**Error:** In a declaration of an array, you omitted a required dimension bound value or specified an invalid value for a bound.

For multidimensional arrays, you must specify bounds for dimensions other than the first. You also must specify a bound for the first (or only) dimension if this declaration is a definition. Valid bound values are integer constant

expressions greater than 0.
**User Action:** Make sure that all required bounds are present and valid.

**SYN_INVARRAYDECL, "**
********

**" is an improperly**
**declared array**
**Error:** You improperly declared an array, such as an array of functions.
**User Action:** Make sure that the syntax of the declarator correctly describes the object. (The declared object may not be what you want.)

**SYN_INVARROW, The "->" operator may only be applied to a pointer object**
**Error:** You used the ``->" operator with something other than a pointer type.
**User Action:** Check your code for use of the ``->" operator applied to something other than pointer type.

**SYN_INVBREAK, Invalid use of the "break" statement**
**Error:** You used **break** outside a loop or **switch** statement.
**User Action:** Remove the **break** statement or check that any braces in recent loops or **switch** statements are properly balanced.

**SYN_INVCASEEXPR, The value in a "case" clause must have integral type**
**Error:** You used values within the **case** clause that did not have integral type.
**User Action:** Check your use of values within the **case** clause; the values must be of integral type.

**SYN_INVCASENEST, The " case " clause must appear within a " switch " statement**
**Error:** The **case** clause did not appear within a **switch** statement.
**User Action:** Rewrite your statement to include the **case** clause within the **switch** statement.

**SYN_INVCODEIPSECT, Code-i psect must be declared at file scope**
**Warning:** Code-i psect is declared inside a block.
**User Action:** Remove code-i psect pragma from the block.

**SYN_INVCONSTPSECT, Constant psect can not be declared more than once**
**Warning:** There may be only one declaration of the **const** psect for each compilation unit.
**User Action:** Remove the extraneous psect definition.

**SYN_INVCONTINUE, Invalid use of the "continue" statement**
**Error:** You used the **continue** statement outside the body of a **for** , **while** , or **do** statement.
**User Action:** Remove the **continue** statement or check that any braces in recent loops are properly balanced.

**SYN_INVDEFNEST, The " default " clause must appear within a " switch " statement**
**Error:** The **default** clause did not appear within a **switch** statement.
**User Action:** Rewrite your statement to include the **default** clause within the **switch** statement.

**SYN_INVDEREF, Address-valued expression not found**

**Error:** Attempted to dereference a nonpointer object.
**User Action:** Remove the dereference operator; ensure that the correct operand is being used.

**SYN_INVDOTLVAL, The left side of a "." operator must be an lvalue**

**Error:** The ``." operator requires a left operand, which is a name for storage.
**User Action:** Ensure that the left operand is an lvalue; if it is a pointer to storage, then ``->" should be used.

**SYN_INVEQ, Invalid operand of an equality operator**

**Error:** You used an operand that is not compatible with the equality operator.
**User Action:** Correct the operand.

**SYN_INVFIELDSIZE, The declaration of** "
********

"

**specifies an invalid field size; size of 16 bits assumed**

**Warning:** The indicated field declaration was invalid because it specified too large a size.
**User Action:** Correct the declaration to specify either a single, smaller field or several contiguous fields.

**SYN_INVFIELDTYPE, The declaration of** "
********

" **specifies an invalid field data type; type** " **unsigned** " **assumed**

**Warning:** You declared a field with an invalid data type. Fields must be declared (and manipulated) as integers or enumerated types.
**User Action:** Correct the declaration to specify a valid data type.

**SYN_INVFUNCCLASS, The declaration of an**

**identifier** "

********

" **for a function that has a block scope shall have no explicit storage-class other than extern**

**Warning:** You declared a function with the wrong storage class.

**User Action:** Change the storage class of your function to external.

**SYN_INVFUNCDECL, "**
********

**" is an improperly declared function**
**Error:** You improperly declared a function. For example, you may have omitted the parameter list or a semicolon between the function and a previous declaration.
**User Action:** Correct the syntax of the declaration.

**SYN_INVGLOBALNAME, Invalid global name "**
********

**"**
**Warning:** The ASCII name specified cannot be converted into a valid Radix-50 name.
**User Action:** Rename the global name.

**SYN_INVLINKAGE, Linkage for function "**
********

**"**
**must be specified before it is either defined or referenced**
**Warning:** Function linkage is specified after it is referenced or defined.
**User Action:** Define the linkage before any references to the function.

**SYN_INVMEMNAME, The right operand of " . " or " -> "**
**is not a declared name in this structure or union**
**Error:** The right operand of your ``." or ``->" was not properly declared.
**User Action:** Declare the right operand in the expression.

**SYN_INVOBJDEREF, The operand of "**
*****

" **(dereference) must be a pointer to an object**
**Error:** The expression to which ``*" is applied is not a pointer to an object.
**User Action:** Ensure that the operand is of the proper type.

**SYN_INVPARELLIPSIS, The use of ellipsis in a function prototype conflicts with the function definition**
**Error:** Both function prototype and function definition must specify a variable parameter list.
**User Action:** Make the function prototype consistent with the function definition.

**SYN_INVPRAGMA, Invalid pragma definition**
**Warning:** Syntax error detected in a **pragma** statement.
**User Action:** Correct the syntax to one of the forms shown in Section 7.7.

**SYN_INVPROTODEF, The parameter list for a function prototype definition must associate identifier for each type in the parameter list**
**Error:** The function definition uses the prototype format but does not contain an identifier for each type in the parameter list.
**User Action:** Place an identifier name in the appropriate type declaration.

**SYN_INVPSECTNAME, Invalid psect name specified**
**Warning:** A psect name must consist of 6 or fewer Radix-50 characters.
**User Action:** Ensure that the name meets the requirements.

**SYN_INVPTRMATH, Invalid pointer arithmetic**
**Error:** You attempted to perform an invalid arithmetic operation on a pointer or pointers. The only valid arithmetic operations allowed with pointers are addition

and subtraction.
For addition, the only allowable forms are as follows:

pointer + integer
pointer += integer

For subtraction, the only allowable forms are as follows:

pointer - integer
pointer -= integer
pointer - pointer

In the last form, both pointers must point to objects of compatible type.
**User Action:** Make sure that the expression conforms to one of the previous forms listed. If necessary, cast one or both operands to a compatible type.

**SYN_INVPTRSUBTYPE, Pointer subtraction must be between compatible pointer types**
**Warning:** You used incompatible pointer types in a pointer subtraction.
**User Action:** Check to make sure your pointers are of compatible subtracting type. For subtraction, the only allowable forms are as follows:

pointer - integer
pointer -= integer
pointer - pointer

**SYN_INVREL, Invalid operand of a relational operator**
**Error:** You used an operand that was not compatible with the relational operator you used.
**User Action:** Correct the operand.

**SYN_INVSTORCLASS, The " auto " storage class is invalid for the declaration of "**

********

"

**Warning:** You made one of the following programming errors:

- You specified a storage class that is invalid in the context in which the declaration appears; for example, specifying **auto** in a declaration located outside of a function.
- You specified a storage class that is incompatible with another storage class specifier; for example, specifying both **static** and **extern** .
- You specified a storage class that is incompatible with the data type of the indicated declarator; for example, specifying **globalvalue** for an array.

In all cases, the compiler ignores the storage class specifier.
**User Action:** Correct the declaration.

**SYN_INVSUBSCRIPT, Invalid subscript; "[ ]" must be applied to an array or a pointer to an object, and an integer**
**Error:** You specified a subscript in reference to a bit-field.
**User Action:** Correct the syntax. If the structure containing the bit-field is an array, you must specify the subscripts with the qualifier rather than with the member name.

**SYN_INVSUTYPE, The left operand of " . " or " -> " must be a (pointer to) a structure or union**
**Error:** The left operand of ``." or ``->" was not a (pointer to) a structure or union.
**User Action:** Correct the operand.

**SYN_INVSWITCHEXPR, The control expression in a "switch" statement must have integral type**
**Error:** The expression is not of integral type.
**User Action:** Ensure that the expression is of integral type.

**SYN_INVTAGUSE, Invalid use of** "
********

" **tag**
**Error:** You used a previously defined tag name in a declaration of a different type. For example:

```
enum color {red, green, blue};
struct color *cp;
```

You may only use a given tag with one of the types **enum** , **struct** , or **union** . Any identifiers declared with the mismatched type will be undefined.
**User Action:** Either make sure that each use of the tag name specifies the same type, or use different tag names with each type.

**SYN_INVUADDR, The operand of "&" must be an lvalue or function, and may not be a register or bitfield**
**Warning:** The ``&" (address-of) operator must be applied to an object that has storage associated with it or to a function name.
**User Action:** If ``&" has been applied to a register value, the **register** keyword can be removed from the declaration; otherwise, ensure that the specified object has storage.

**SYN_INVVARIANT, Invalid declaration of variant aggregate** "
********

"
**Warning:** You attempted an invalid variant structure or union declaration such as an array of variants, a pointer to a variant, or a list of variant names.
**User Action:** Either remove the variant keywords from the declaration or make sure that the keywords are used

in a valid structure or union declaration.

**SYN_INVVOIDUSE, " void " is only valid in a parameter list when it appears alone; its use is ignored**
**Warning: void** has been used in a function prototype parameter list but is not the only item in the list.
**User Action:** Either eliminate **void** or eliminate the extra parameter types in the parameter list.

**SYN_ITERSCALREQ, The controlling expression of an iteration statement must have scalar type**
**Error:** Occurs when **for** , **while** , or **do** statements have an incorrect type (such as **struct** ) as the controlling expression.
**User Action:** Use a scalar as the controlling expression when writing an iteration statement.

**SYN_LMUL_ARITH, The left operand of a " * " or " / " operator must have arithmetic type**
**Error:** You did not specify arithmetic type for the left operand of your operator.
**User Action:** Correct the operand.

**SYN_LOGSCALREQ, Both operands of a logical operator must have integral type**
**Error:** You declared the operands in your logical operator as having a type other than integral.
**User Action:** Correct the operand.

**SYN_LREM_INT, The left operand of a " % " operator must have integral type**
**Error:** You declared the left operand as having a type other than integral.
**User Action:** Correct the operand.

**SYN_MAIN02PARAMS, The "main" routine should have 0 or 2 parameters**

**Warning:** The ``main" routine should either specify no parameters or exactly two, like this:

```
int main ()
int main (int argc, char *argv [])
```

**User Action:** Correct the declaration of ``main."

**SYN_MAINRETTYPE, The "main" routine should specify a return type of "int"**

**Warning:** The ``main" routine should be of type int, as in either of these declarations:

```
int main ()
int main (int argc, char *argv [])
```

**User Action:** Correct the declaration of ``main."

**SYN_MISPARAMNUMBER, The number of parameters declared does not match the number declared in a previous function prototype**

**Error:** A function prototype for this function, which appeared earlier in the source file, contains a different number of parameters than this declaration.

**User Action:** Determine which declarator is correct and modify the other declarator to match it.

**SYN_MISPARAMTYPE, The type of parameter number 1 does not match the type declared in a previous function prototype**

**Warning:** The type of a parameter in a function definition does not match the type specified for that parameter in the previous prototype.

**User Action:** Determine which type is correct for that parameter and correct either the function definition or the prototype.

**SYN_NODECL, Your program doesn't declare any data or routines**

**Informational:** A compilation unit should define at least one data item or routine. This might not happen, for

example, if the module were commented out.
**User Action:** Ensure that the module contains a definition.

**SYN_NOTFUNC, Function-valued expression not found**
**Error:** You used an expression in the context of a function call, but the expression does not evaluate to a function.
**User Action:** Make sure that the expression properly evaluates to a function; also make sure that you properly dereference any pointer to a function.

**SYN_NOTPARAMETER,** "
********

" **is not listed in the function's formal parameter list; its declaration is ignored**
**Warning:** You declared the specified identifier as a function parameter, but the identifier does not appear in the parameter list. For example:

```
f(a) int a,b; { . . . }
```

The identifier b does not appear in the formal parameter list of function f. Its declaration is not portable and is probably a coding error. The compiler treats b as if it were declared inside the function definition; in this case, b becomes an automatic variable.
**User Action:** Correct the declaration or the parameter list.

**SYN_PARSTK_OVRFLW, Parse stack overflow**
**Fatal:** The source code in your program was too complex, containing statements nested too deeply.
**User Action:** Simplify the program.

**SYN_REDEFPROTO,** "
********

" **conflicts with either the function definition or with a function prototype that appears earlier in the file**

**Warning:** The prototype conflicts with a previous declaration of this function, either in number, type of arguments, or in the return type of the function.
**User Action:** Determine what attribute does not match and what the correct attribute should be. Correct the invalid definition.

**SYN_RMUL_ARITH, The right operand of a "*" or "/" operator must have arithmetic type**
**Error:** You declared the right operand as having a type other than arithmetic.
**User Action:** Correct the operand.

**SYN_RREM_INT, The right operand of a "%" operator must have integral type**
**Error:** You declared the right operand as having a type other than integral.
**User Action:** Correct the operand.

**SYN_SHIFTINTREQ, Both operands of a shift operator must have integral type**
**Error:** You did not declare both operands of the shift operator to have integral type.
**User Action:** Rewrite both operands to have integral type.

**SYN_SIZEOFOBJ, The "sizeof" operator must be applied to a complete object type**
**Warning:** An incomplete object has no defined size.
**User Action:** Complete the type before using "sizeof."

**SYN_SYNTAXERROR ,** "
********

" **found** "
********

" **when expecting** "
********

"
**Error:** Syntax error detected.
**User Action:** Check your syntax.

**SYN_TENTDEFINC, Tentative definition of** "
********

" **has internal linkage, but its type is incomplete**
**Error:** The return type of a static function must be fully specified before the function can be used.
**User Action:** Ensure that the type is completed.

**SYN_TOOMANYINITS, The initializer list for** "
********

"
**specifies too many initializers; excess initializers ignored**
**Warning:** You specified too many initializers for the indicated variable. (If the indicated item is an array or structure, it may be only partially initialized.)
**User Action:** Make sure that all braces near the initializer sublists are balanced; if the item being initialized

is or contains an array, make sure that you accounted for all dimensions.

**SYN_TYPECONFLICT,** "
********
" **conflicts with a previous data type in this declaration; previous data type ignored**
**Warning:** You specified more than one data type specifier in this declaration, and the indicated specifier conflicted with a previous one.
**User Action:** Check for a missing semicolon in the previous declaration; otherwise, make sure that all specifiers are compatible.

**SYN_UCOMPINTREQ, The operand of a unary complement operator must have integral type**
**Error:** You declared the operand as having a type other than integral.
**User Action:** Correct the operand.

**SYN_UIDSCALREQ, The operand of unary** " **++** " **or** " " **must be a scalar**
**Error:** You declared the operand as having a type other than scalar.
**User Action:** Correct the operand.

**SYN_UMINARIREQ, The operand of a unary minus operator must have arithmetic type**
**Error:** You declared the operand as having a type other than arithmetic.
**User Action:** Correct the operand.

**SYN_UNDECLFUN, Function** "
********
" **not declared; assumed of type "extern int ( )"**
**Informational:** You did not declare a function to be a specific type. The default type that will be assumed will be **extern int** .

**User Action:** Check to make sure that **extern int** is the function type you need; if not, redeclare the function to the necessary type.

**SYN_UNDECLNAME, Identifier** "
********
" **is not declared within the scope of this usage**
**Error:** You referenced a variable that was never properly declared.
**User Action:** Check that the identifier is declared, and that its case and spelling are consistent in all uses.

**SYN_UNDEFLABEL, Label** "
********
" **referenced but not defined in this function**
**Error:** You wrote `` **goto** label-name" for an undefined label. The scope of a label name is restricted to the function in which it is used as a label; **goto** statements cannot branch to labels inside other functions.
**User Action:** Check the spelling of the label name or make other corrections as appropriate.

**SYN_UNDEFSTRUCT, Structure or union member** "
********
" **has a type that is not fully defined at this point in the compilation**
**Error:** A member of either your structure or union has an incomplete data type.
**User Action:** Correct the type of your structure or union member.

**SYN_UNOTSCALREQ, The operand of a unary not operator must have scalar type**
**Error:** You declared the operand as having a type other than scalar.
**User Action:** Correct the operand.

**SYN_UPLSCALREQ, The operand of a unary plus operator must have scalar type**
**Error:** You declared the operand as having a type other than scalar.
**User Action:** Correct the operand.

**SYN_VARNOTMEMBER, A variant aggregate must be a member of a struct or union**
**Error:** You attempted to specify a **variant_struct** or a **variant_union** outside of an aggregate declaration.
**User Action:** If you intend to use the structure or union as declared, and if the structure or union is the outermost aggregate in a group of nested aggregates, replace the variant keywords with **struct** or **union** . If you intend to use the structure or union as a variant aggregate, and if the structure or union is otherwise properly declared, nest the declaration within a valid structure or union declaration. If you use the **variant_struct** or **variant_union** keywords in declarations other than structure or union declarations, remove the variant keywords.

**SYN_VOIDNOTFUNC,** "
********
" **is not declared to be a function; only functions may be declared** " **void** "
**Error:** You declared an object other than a function to be **void** .
**User Action:** Check the syntax of the declarator.

**TOO_MANY_ERRORS, Encountered more than** "
********

"

**errors, compilation terminated**
**Fatal:** More errors were encountered than the installation-defined default error limit, or the limit specified with the /ERROR_LIMIT qualifier.
**User Action:** Correct the errors or use the /ERROR_ LIMIT qualifier to increase the error limit or the /NOERROR_LIMIT qualifier to eliminate the error limit.

**WF_DSOVERFLOW, Data set overflow in work file; increase the work file size with the /WORK_FILE_ SIZE qualifier**
**Fatal:** The capacity of one of the PDP-11 C data sets has been exceeded. The PDP-11 C work file is internally divided into a number of data sets. The amount of storage consumed by data sets varies dynamically according to need. The maximum capacity of a data set is determined when PDP-11 C starts up and is based upon how the data set is internally defined and upon the size of the work file. For each 1K blocks of work file size, PDP-11 C doubles the capacity of its data sets at the expense of less efficient data packing in the work file.
**User Action:** Increase the value specified with the /WORK_FILE_SIZE qualifier in 1K-block increments until the error no longer occurs.

**WF_FILEORDEV, File or device error on work file**
**Fatal:** An error occurred while opening, reading, or writing the PDP-11 C work file.
**User Action:** Determine the cause of the error and correct.

**WF_INSUFFICIENTWF, Work file too small; increase the work file size with the /WORK_FILE_SIZE qualifier**
**Fatal:** PDP-11 C has run out of work file storage on a PDP-11 host.
**User Action:** Increase the work file size with the /WORK_FILE_SIZE qualifier; increase the amount of extended (unmapped) memory available to PDP-11 C with the /MEMORY qualifier; or simplify the compilation unit.

**WF_INSUFFICIENT_MEMORY, Insufficient memory**

**Fatal:** Memory requirements exceeded available resources.

**User Action:** On V AX/VMS host systems, decrease the size of the compilation unit or increase system quotas. On PDP-11 host systems, this error occurs when parsing complex command lines and indirect command line files; simplify the command lines or indirect command line files.

**WF_NOROOM, No room on device for work file**

**Fatal:** There was no room to open the PDP-11 C work file on the work file device.

**User Action:** Purge or delete files on the work file device to make room for the PDP-11 C work file.

**WF_TOOMUCHMEM, The value specified with the /MEMORY qualifier is too large; specify a value of 511 or smaller**

**Fatal:** The number of 8K-byte extended memory regions specified with the /MEMORY qualifier exceeds the 4M-byte physical memory limit of the PDP-11.

**User Action:** Specify 511 or a smaller value.

**WF_UNEXPECTED, Unexpected I/O error on work file**

**Fatal:** An unexpected error occurred while opening, reading, or writing the PDP-11 C work file.

**User Action:** Determine the cause of the error and correct.

## B. PDP-11 C Header Files

This appendix lists the library header files for PDP-11 C. File location is system dependent:

- On RSX and VMS, the header files are in the directory LB:[1,1].
- On RSTE/E, the header files are in the directory CC$:.
- On RT-11, the header files are in either the SY: or CLB: directory.

In general, each header file declares functions, types, or macros used in the area of the Run-Time Library (RTL) indicated in the ``Description'' column in each of the tables in this appendix. You can print or type individual files, or you can issue the following command to print all files with their file names appearing at the top of each page:

$ **print lb:[1,1]*.h**

Table B-1 describes each of the Standard Library header files.

Table B-2 describes each of the File Control Services (FCS) Extension Library header files.

Table B-3 describes each of the Record Management Services (RMS) Extension Library header files.

Table B-4 describes each of the system interface header files.

[next] [previous] [contents]

---

## C. PDP-11 C Internationalization

This appendix addresses the two major areas of PDP-11 C internationalization: compiler and run-time internationalization.

## C.1 Compiler Internationalization

In PDP-11 C, you can assign specific compiler character sets to four areas: source files, the message environment, listing files, and the execution environment. To specify character sets for character constants and strings, use the following charset pragmas, respectively:

#pragma charset source <charset_name>
#pragma charset message <charset_name>
#pragma charset list <charset_name>
#pragma charset execution <charset_name>

PDP-11 C uses the the ISO-Latin-1 character set by default for the source files, messages, listing files, and execution environment. You can change to any of the character sets listed in Table 7-2, using the guidelines found in Section 7.7.1.

The following is an example of the **#pragma charset** directive. In this example, the swiss character set is specified for the device on which the source file is to be displayed.

#pragma charset source swiss

When writing source files to be displayed on non-Digital devices (terminals, printers, and display devices), the use of trigraphs may be required. Trigraphs are 3-character sequences that represent specific characters that may not exist on some terminals. All occurrences of trigraph sequences (listed in Table 2-3) are replaced with the corresponding single character. See Section 2.16 for more information on trigraphs.

As an example, the following source line:

printf("Eh???/n");

becomes (after replacing the trigraph sequence):

printf("Eh?\n");

### C.2 Run-Time Internationalization

You can use the **#pragma charset** to specify the character set for character constants and character strings in your source program. In addition, PDP-11 C provides support for run-time internationalization in the locale.h header file that defines a structure and two functions used in supporting alternate character sets and other international support. The two functions that are provided are **setlocale** and **localeconv** .

### C.2.1 Set Locale Function (setlocale)

The **setlocale** function is used to specify alternate character sets, collating sequences, and various formats (for example, money and time). The **setlocale** function takes two arguments:

.
The first argument specifies the category of the locale that you want to change.

.
The second argument specifies the locale you want to set the category to.

The category argument which names the program's entire locale is LC_ALL. The other values for category name only a portion of the program's locale (LC_COLLATE, LC_CTYPE, LC_NUMERIC, LC_MONETARY, LC_TIME).

---

**Note**

---

For more information on the **setlocale** function, refer to the *PDP-11 C Run-Time Library Reference Manual* .

---

In the following example, the **setlocale** function specifies a program's locale for all five categories:

```
#include <locale.h> /*Include locale.h
                                **header file.*/
```

```
setlocale (LC_ALL,"french,danish,,french"); /*Name the locale
                                              **for all categories.*/
```

The first argument in this example (LC_ALL) specifies that the locale of all categories will be changed. The second argument is a character string that specifies the locale for each of the five categories, separated by a comma. Omitting a category (by using two consecutive commas or by not specifying trailing arguments) yields the default locale for those categories. As you can see in the example, the following is set:

.
LC_COLLATE is set to french.

.
LC_CTYPE is set to danish.

.
LC_NUMERIC retains the default C locale.

.
LC_MONETARY is set to french.

.
LC_TIME retains the default C locale.

The second example shows an alternate way to specify the program's locale by specifying one category at a time.

```
#include <locale.h> /*Include locale.h
                                          **header file*/
setlocale (LC_ALL,""); /*Reset all categories
                                          **to standard C locale*/
setlocale (LC_COLLATE,"french"); /*Name the locale*/
setlocale (LC_CTYPE,"danish"); /*for specific*/
setlocale (LC_MONETARY,"french"); /*categories*/
```

The first argument of each invocation of **setlocale** indicates that we will change only the locale of the category indicated. The second argument specifies to which locale that category is set. Note that in the previous example, only three categories were changed. The remaining categories will keep the C locale defaults. Both programs yield the same results.

### C.2.2 Defining a Locale Structure (localeconv)

The **localeconv** function sets the components of an object of type **struct lconv** to values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the current locale. If you have not specified any locale changes using **setlocale** , the default for all the

categories is the C locale, which is ASCII.
The **localeconv** function returns a pointer to the filled-in object.
Example C-1 shows typical source code using **localeconv** .
It sets the current locale for character functions and conversion to the *french* locale. The LC_TIME category retains the default locale. Localeconv( ) is called to access the monetary formatting data, and a number is converted from a floating-point value to a monetarily formatted quantity. The data in the **lconv** structure is used to format a positive monetary value.

### C.2.3 Character Handling Functions

The function versions of the character handling functions defined in the *ctype.h* header file return the values from the selected locale based on the locale set by **setlocale** . Note that the macro versions support only the ASCII locale.
The program in Example C-2 computes the number of alphanumeric characters in the locale ``C'' (the ASCII locale) in three different ways. First, the program uses the macro **isalnum** . Then, it takes the address of the function **isalnum** and invokes the function through its address. Finally, the program undefines the macro **isalnum** and leaves only the function definition in scope.

[next] [previous] [contents]

---

**D. Language Summary**

This appendix briefly describes the following C language features:

- Data type keywords
- Precedence of operators
- Statements
- Conversion rules
- Escape sequences
- Preprocessor directives

[next] [previous] [contents]

---

## D.1 Data Type Keywords

Table D-1 shows data type keywords.

[next] [previous] [contents]

---

## D.2 Precedence of Operators

Table D-2 lists the operators from highest precedence to lowest. In the binary operator category, operators appear in descending order of precedence, line by line.

[next] [previous] [contents]

---

### D.3 Statements

**Syntax:**
[ expression ] ;
identifier : statement
{ [ declaration-list ] [ statement-list ] }
**if** ( expression ) statement [ **else** statement ]
**while** ( expression ) statement
**do** statement **while** ( expression )
**for** ([ expression ] ; [ expression ] ; [ expression ]) statement
**switch** ( expression ) statement
**case** constant-expression statement
**default:** statement
**break** ;
**continue** ;
**return** [ expression ] ;
**goto** identifier ;

[next] [previous] [contents]

## D.4 Conversion Rules

### Arithmetic Conversion

| Any operand of type: | Is converted to: |
|---|---|
| char | int |
| unsigned char | unsigned int |
| float | double |

| If operand type is: | The result and the other operands are: |
|---|---|
| double | double |
| unsigned | unsigned |

| Otherwise, both operands are: | And the result is: |
|---|---|
| int | int |

### Function Argument Conversion

| Any argument of type: | If not within the scope of a function prototype, is converted to type: |
|---|---|

**float double**
**char int**
array pointer to array
function pointer to function

[next] [previous] [contents]

## D.5 PDP-11 C Escape Sequences

| Character | Mnemonic Escape Sequence |
|---|---|
| bell BEL | \a |
| question mark ? | \? |
| newline NL | \n |
| horizontal tab HT | \t |
| vertical tab VT | \v |
| backspace BS | \b |
| carriage return CR | \r |
| form feed FF | \f |
| backslash \ | \\ |
| apostrophe ' | \ ' |
| quotes " | \ " |
| bit pattern ddd | \ddd or \xddd |

Use the form ``\ddd" to specify any byte value (usually an ASCII code), where the digits ddd are one to three octal digits. The octal digits are limited to 0 to 7.
Similarly, use the form ``\xddd" to specify any byte value (usually an ASCII code), where the digits are used to specify one or more hexadecimal digits.

## D.6 Preprocessor Directives

**Syntax:**

# **define** identifier [([ param1 , **. . .** param2 ])] token-string
# **undef** identifier
# **error** tokens
# **include** < file-spec >
# **include** " file-spec "
# **if** constant-expression
# **ifdef** identifier
# **ifndef** identifier
# **else**
# **elif** constant-expression
# **endif**
#[ **line** ] constant "string"
#[ **line** ] constant identifier
# **module** identifier identifier
# **module** identifier "string"
# **pragma** charset

*2*
*4*

source
message
list
execution

*3*
*5*

*8*
>
>
>
>
<
>
>

```
                    >
                    >
                    :

                        iso_latin_1
                        french_canadian
                        dec_mcs
                        german
                        ascii
                        italian
                        british
                        norwegian
                        danish
                        portuguese
                        dutch
                        spanish
                        finnish
                        swedish
                        french
                        swiss

                                    9
                                    >
                                    >
                                    >
                                    >
                                    =
                                    >
                                    >
                                    >
                                    >
                                    ;
```

# **pragma** psect

```
                8
                <
                :

                    const
                    static_ro
                    static_rw
                    code_i
                    code_d
```

*9*
*=*
*;*

*2*
*6*
*6*
*6*
*6*
*6*
*6*
*6*
*6*
*4*

,

*8*
>
>
>
<
>
>
>
*:*

ro
rw
i
d
lcl
gbl
rel
abs
con
ovr
sav
nosav

```
                                >
                                >
                                >
                                =
                                >
                                >
                                >
                                ;

                                   ,...

                                       3
                                       7
                                       7
                                       7
                                       7
                                       7
                                       7
                                       7
                                       7
                                       5
```

# **pragma** module identifier identifier
# **pragma** module identifier "string"
# **pragma** list

```
            8
            <
            :

               on
               off
               title "string"
               subtitle "string"

                        9
                        =
                        ;
```

# **pragma** linkage

```
            8
            >
            <
```

```
>
:

    c
    pascal
    fortran
    rsx_ast
    rsx_sst
    rsx_csm

        9
        >
        =
        >
        ;

          [identifier,...]
```

[next] [previous] [contents]

---

## Glossary

**Example 1-1: Default Compiler Listing**

```
Example 1-1 DUA0:[C]EXAMPL.C; PDP-11 C V1.2-015 Page 1 1
000001 6 February 1992 3:03 PM 2
3 1 # ifdef __PDP11C
      2 4 # pragma list title "Example 1-1"
      3 5 # pragma module EXAMPL "000001"
      4 # endif
      5
      6 /* This module is used as an example to demonstrate the
                        various listing options that are available with PDP-
                        11 C. In particular, this comment shows how line wrap
                        of long source lines appears in the listing. */
      7
      8 # include <setjmp.h>
     28 /* Use 6 character, extern names on the PDP-11 */
     29 # ifdef __PDP11C
     30 # define recovery_context RECCTX
     31 # define error_recovery ERRCVY
     32 # define process PROCES
     33 # endif
     34
     35 # define TRUE 1
     36 # define FALSE 0
     37
     38 jmp_buf recovery_context;
     39 int error_recovery (void);
     40 int process (void);
     41
     42 #ifdef __PDP11C
     43 4 # pragma list subtitle "main() - Main Entry Point"
     44 # endif
     45 int main (void)
     46 {
     47
     48 # if ERROR_RECOVERY
6 1
1: %PDPC-I-LEX_UNDEFIFMAC, Identifier is not currently a macro; constant zero
     assumed
          At line 27 in file SYS$SYSUSER:[RAITTO.C.LEX]EXAMPL.C;
7 49 X if (setjmp(recovery_context) != 0)
```

```
   50 X return error_recovery();
   51 X else
   52 X return process();
   53 # else
   54 return process();
   55 # endif
   56
   57 }
```

Message summary: Informational 1 Warning 0 Error 0 **8**
Compiler Command
---
EXAMPL/LIST

**Example 1-2: Compiler Listing Options**

```
EXAMPL Example 1-2 DUA0:[C]EXAMPL.C; PDP-11 C V1.2-015 Page 1
000001 6 February 1992 3:03 PM
     1 # ifdef __PDP11C
     2 # pragma list title "Example 1-2"
     3 # pragma module EXAMPL "000001"
     4 # endif
     5
     6 /* This module is used as an example to demonstrate the
                    various listing options that are available with PDP-
                    11 C. In particular, this comment shows how line wrap
                    of long source lines appears in the listing. */
     7
     8 # include <setjmp.h>
1 9 1 /*
   10 1 ** setjmp.h
   11 1 */
   12 1 /* used by: setjmp() & longjmp() functions */
   13 1
   14 1 # ifndef __SETJMP_H
   15 1 # define __SETJMP_H
   16 1
   17 1 # define JMPBUF_STATE_SZ 8
   18 1
   19 1 typedef int jmp_buf[ JMPBUF_STATE_SZ ];
2  1 typedef int jmp_buf[ 8 ];
   20 1
   21 1 # define setjmp c$stjp
   22 1 # define longjmp c$lgjp
   23 1 int setjmp ( jmp_buf env );
              1 int c$stjp ( jmp_buf env );
   24 1 void longjmp ( jmp_buf env, int val );
              1 void c$lgjp ( jmp_buf env, int val );
   25 1
   26 1 # endif
   27 1
   28 /* Use 6 character external names on the PDP-11 */
   29 # ifdef __PDP11C
   30 # define recovery_context RECCTX
   31 # define error_recovery ERRCVY
```

```
   32 # define process PROCES
   33 # endif
   34
   35 # define TRUE 1
   36 # define FALSE 0
   37
   38 jmp_buf recovery_context;
                1 jmp_buf RECCTX;
   39 int error_recovery (void);
                1 int ERRCVY (void);
   40 int process (void);
                1 int PROCES (void);
   41
   42 # ifdef __PDP11C
   43 # pragma list subtitle "main() - Main Entry Point"
   44 # endif
   45 int main (void)
   46 {
EXAMPL Example 1-2 DUA0:[C]EXAMPL.C; PDP-11 C V1.2-015 Page 2
000001 main() - Main Entry Point 6 February 1992 3:03 PM
   47
   48 # if ERROR_RECOVERY
                1 # if TRUE
3  2 # if 1
   49 if (setjmp(recovery_context) != 0)
                1 if (c$stjp(recovery_context) != 0)
                1 if (c$stjp(RECCTX) != 0)
   50 return error_recovery();
                1 return ERRCVY();
   51 else
   52 return process();
                1 return PROCES();
   53 # else
   54 X return process();
   55 # endif
   56
   57 }
4  .TITLE EXAMPL
                                        .IDENT /000001/
                                .PSECT $READW,RW,D,GBL,REL,CON,SAV
                        $READW:
000000 RECCTX::.BLKB 16. ; RECCTX
                                .GLOBL C$STJP,ERRCVY,PROCES,C$MAI
```

```
                                .PSECT $CODEI
5 6 7
000000 MAIN:: ; main
000000 010546 MOV R5,-(SP) ; 45
000002 016746 000000G MOV RECCTX,-(SP) ; 49 RECCTX,
000006 005746 TST -(SP)
000010 004767 000000G CALL C$STJP ; c$stjp
000014 012605 MOV (SP)+,R5
000016 005726 TST (SP)+
000020 001410 BEQ 1$
000022 005746 TST -(SP) ; 50
000024 004767 000000G CALL ERRCVY ; ERRCVY
000030 012666 000004 MOV (SP)+,4(SP)
000034 012605 MOV (SP)+,R5
EXAMPL Example 1-2 DUA0:[C]EXAMPL.C; PDP-11 C V1.2-015 Page 3
000001 main() - Main Entry Point 6 February 1992 3:03 PM
000036 000207 RETURN
000040 000407 BR 2$
000042 005746 1$: TST -(SP) ; 52
000044 004767 000000G CALL PROCES ; PROCES
000050 012605 000004 MOV (SP)+,4(SP)
000054 012605 MOV (SP)+,R5
000056 000207 RETURN
                                .END
Compiler Command
    ---------------
EXAMPL/LIST=EXAMPL_ALL/SHOW=ALL/DEFINE="ERROR_RECOVERY TRUE" 8
```

[next] [previous] [contents]

---

**Example 2-1: Simple Addition in PDP-11 C**

```
/* This program adds two numbers and places the sum in * 1
  * the variable total. */
int main(void) 2 /* The function name "main" */
       { /* Begins function body */
3 int total; /* Variable of type "int" */
                                  /* Blank lines are allowed */
4 total = 2 + 2; /* Answer placed in "total" */
     } /* Ends the function body */
```

**Example 2-2: Output of Information**

```
/* This program adds two numbers, assigns the value 4 to *
  * variable total, and then prints the answer on the *
  * terminal screen. */
#include <stdio.h> 1
int main(void)
{
    int total;
    total = 2 + 2;
                                    /* Print intro string */
2 printf("Here is the answer: ");
    printf("%d.", total); /* Print the answer */
}
```

[next] [previous] [contents]

---

**Example 2-3: Output Using the Newline Character**

```
/* This program adds two numbers, stores the sum in the *
  * variable total, and then prints the answer on two *
  * separate lines on the terminal screen. */
#include <stdio.h>
int main(void)
{
   int total;
   total = 2 + 2;
                                    /* Print intro string */
   printf("Here is the answer...\n");
                                    /* Print the answer */
   printf("%d.", total);
}
```

**Example 2-5: Conditional Execution Using the switch Statement**

```
/* This program plays the same guessing game as the *
  * previous example except that it uses the switch *
  * statement. */
#include <stdio.h>
#include <ctype.h> 1 /* Include required module */
int main(void)
{
    int ch;
    printf("Guess what letter I'm thinking of!\n");
    ch = getchar();
2 ch = tolower(ch); /* Convert "ch": lowercase */
    switch(ch) /* Examine "ch" */
        { /* Body of switch statement */
            case 'a' :
                printf("You're right!");
                break;
            default : /* Any other answer */
                printf("You're wrong.\n");
                printf("You'll have to try again!");
        }
}
```

**Example 2-6: Looping Using the do Statement**

```
/* This program plays the same guessing game as the *
  * other examples except that the user must guess until *
  * the answer is correct. This is accomplished using a *
  * do statement. */
#include <stdio.h>
#include <ctype.h>
int main(void)
{
   int ch;
   printf("Guess what letter I'm thinking of!\n");
   printf("Keep guessing until you get it!\n");
   do /* Do the following ... */
        { /* Beginning of loop body */
           ch = getchar();
           ch = tolower(ch);
           switch(ch)
              {
                 case 'a' :
                      printf("You're right!");
                      break;
                              /* Ignore RETURN (newline) ch */
              1 case '\n':
                      break;
                 default :
                      printf("You're wrong.\n");
                      printf("You'll have to try again!\n");
              } /* End of switch statement */
        } /* End of do loop body */
                              /* Condition to be tested */
2 while(ch != 'a');
}
```

**Example 2-7: Looping Using the for Statement**

```
/* This program plays the same guessing game as the *
  * previous examples except that the user is limited to *
  * three guesses. This is accomplished using a for *
  * statement. */
#include <stdio.h>
#include <ctype.h>
int main(void)
{
   int ch;
   int i; /* An incrementor for loop */
   printf("Guess what letter I'm thinking of!\n");
   printf("You have three guesses. Make them count!\n");
                                /* Do the following 3 times */
1 for (i = 1; i <= 3; i++ )
        { /* Beginning of loop body */
           ch = getchar();
           ch = tolower(ch);
           switch(ch)
              {
                 case 'a' :
                      printf("You're right!");
                      return;
                 case '\n':
                2 --i;
                      break;
                 default :
                      printf("You're wrong.\n");
                      if (i != 3)
                         printf("You'll have to try again!\n");
              } /* End of switch statement */
        } /* End of for loop body */
   printf("Sorry, you ran out of guesses!");
}
```

**Example 2-8: Case Conversion Program**

```
/* This program converts its input to lowercase. The *
  * first function passes control to the second function *
  * to convert a letter. Comments are located to the *
  * right of the code. */
#include <stdio.h> /* To use I/O definitions */
int lower (int c_up); /* Prototype for lower */
                                /* function */
int main(void)
{ 1
   FILE *infile, *outfile; /* Declare files */
   int i, c, c_out;
                                /* Open "infile" for input */
   infile = fopen("ex113.in", "r");
                                /* Open "outfile" for output */
   outfile = fopen("ex113.out", "w");
                                /* While not end of file... */
                                /* Get a char from the file */
   while ((c = getc(infile)) != EOF)
        {
           c_out = lower(c); /* Send char to "lower" */
                                /* Output the char to file */
           putc(c_out, outfile);
        }
   return; /* Optional return statement */
}
/* ------------------------------------------------ *
  * Beginning of the next function definition: *
  * ------------------------------------------------ */
                                /* Function and parameter *
                                /* type and name */
int lower (int c_up) 2
{ /* Beginning function body */
                                /* If capital, convert */
   if (c_up >= 'A' && c_up <= 'Z')
        return c_up - 'A' + 'a';
   else /* Else, return as is */
        return c_up;
} /* End of function body */
                                /* End function definition */
```

**Example 2-9: Including <stdarg.h> in a Parameter List**

```
#include <stdarg.h>
#include <stdio.h>
static void argprint (char *type, ...)
{
    va_list ap; /* Argument pointer */
    char p;
    va_start (ap, type); /* Initialize ap to point to first *
                                        * unnamed argument. Last named *
       * argument is used by va_start to *
       * get started */
    while ( (p = *type++) != '\0') {
        switch (p) { /* Each call to va_arg returns one *
                                    * arg and steps ap to the next */
        case 'i': printf ("\t%d", va_arg (ap, int)); break;
        case 'd': printf ("\t%f", va_arg (ap, double)); break;
        case 's': printf ("\t%s", va_arg (ap, char *)); break;
        default: printf ("\nOnly know how to print one of [ids]\n"); break;
        }
    }
    printf ("\n");
    va_end (ap); /* call when done */
}
int main () {
    argprint ("iis", 3, 4, "string1");
    argprint ("dsi", 3.0, "string2", 4);
}
```

**Example 2-10: Declaring Functions**

```
#include <stdio.h>
char lower(int); 1 /* The function declaration */
int main(void)
{
        .
        .
        .
   while ((c = getc(infile)) != EOF)
        {
                              /* The function call */
           c_out = lower(c);
           putc(c_out, outfile);
        }
}
char lower(int c_up) /* The function definition */
{
   .
   .
   .
}
```

**Example 2-11: Declaring Functions Passed as Arguments**

```
int x(void) { return 25; }
     1 /* Defined before it is *
                                 * used */
int z[10];
int main(void)
{
2 int y(void); /* Function declaration */
        .
        .
        .
3 funct(x, y, z); /* Passed as addresses */
        .
        .
        .
}
y(void) { return 30; } /* Function definition */
void funct(int (*f1)(), int(*f2)(), int* a) 4
                                 /* Function definition *
                                 * Declare arguments as *
                                 * pointers to functions *
                                 * returning an integer */
{
   (*f1)(); /* A call to a function */
        .
        .
        .
}
```

[next] [previous] [contents]

---

**Example 2-12: Echo Program Using Command-Line Arguments**

```
/* This program echoes the command-line arguments. */
#include <stdio.h>
int main(int argc, char *argv[])
{
   int i;
                                        /* argv[0] is program name */
   printf("program: %s\n",argv[0]);
   for (i = 1; i < argc; i++)
        printf("argument %d: %s\n", i, argv[i]);
}
```

**Example 2-13: Scope of Variable Declarations in Nested Blocks**

```
/* This program shows how variables with the same *
  * identifier can be of different data types if located *
  * in different blocks. */
#include <stdio.h>
int main(void)
{ /* Outer block of "main" */
1 int i;
    i = 1;
        .
        .
        .
    if (i == 1)
          { /* An inner block */
2 float i;
              .
              .
              .
              i = 3e10;
              printf("Inner-block variable i:%f\n",i);
          }
      printf("Outer-block variable i:%d\n",i);
}
```

**Example 3-1: Counting Blanks, Tabs, and Newlines Using the switch Statement**

```
/* This program counts blanks, tabs, and newlines in text *
  * entered from the keyboard. */
#include <stdio.h>
int main()
{
   int number_tabs = 0, number_lines = 0, number_blanks = 0;
   int ch;
   while ((ch = getchar()) != EOF)
        switch (ch)
           {
1 case '\t': ++number_tabs;
2 break;
              case '\n': ++number_lines;
                         break;
              case ' ' : ++number_blanks;
           }
   printf("Blanks\tTabs\tNewlines\n");
   printf("%6d\t%6d\t%6d\n", number_blanks,
                                 number_tabs,number_lines);
}
```

[next] [previous] [contents]

---

**Example 5-1: Initializing an Array of Structures**

```
#include <stdio.h>
int main()
{
   int l, m;
   static struct
        {
           char ch;
           int i;
           float c;
        } ar[2][3] =
1 {
2 {
3 { 'a', 1, 3e10 },
                  { 'b', 2, 4e10 },
                  { 'c', 3, 5e10 },
              }
           };
   printf("row/col\t ch\t i\t c\n");
   printf("-----------------------------------\n");
   for (l = 0; l < 2; l++)
        for (m = 0; m < 3; m++)
           {
              printf("[%d][%d]:", l, m);
              printf("\t %c \t %d \t %e \n",
                     ar[l][m].ch, ar[l][m].i, ar[l][m].c);
           }
}
```

**Example 5-2: Character String Constants and Arrays**

```
/* This program plays the same guessing games as the *
  * previous examples except that it uses character *
  * string constants and arrays. */
#include <stdio.h>
int main(void)
{
   int ch; /* Declare a character */
                                        /* Initialize messages */
   char *greeting = "Guess which letter I'm thinking of!";
   char *message1 = "You're right!";
   char *message2 = "You're wrong.";
   char *message3 = "You'll have to try again!";
   char correct[2];
   correct[0] = 'a'; /* Store correct letters */
   correct[1] = 'A';
   printf("%s\n", greeting); /* %s = char string */
   ch = getchar();
   if (ch == correct[0] || ch == correct[1])
        printf("%s", message1);
   else
        {
           printf("%s\n", message2);
           printf("%s\n", message3);
        }
}
```

[next] [previous] [contents]

---

**Example 5-3: Single Storage Allocation of Unions**

```
/* This example illustrates the storage maintenance of *
  * unions with different size members. */
#include <stdio.h>
#include <string.h>
int main(void)
{
   union /* Declare the union */
        {
           char lastname[8]; /* Array for a last name */
           char firstinit; /* Char. for first initial */
        } overlap = "Lincoln";
                                        /* Copy and print members */
   printf("%s\n", overlap.lastname);
   strcpy(overlap.lastname, "Jackson");
   printf("%s\n", overlap.lastname);
   overlap.firstinit = 'M';
   printf("%c\n", overlap.firstinit);
   printf("%s\n", overlap.lastname);
}
```

**Example 5-4: Structures**

```
/* This program plays the same guessing game as the *
  * previous examples except that it uses a structure. */
#include <stdio.h>
int main(void)
{
   int ch;
   char *greeting1 = "Guess which letter I'm thinking of!";
   char *greeting2 = "You've 3 guesses. Make them count!";
   char *message1 = "You're right!";
   char *message2 = "You're wrong.";
   char *message3 = "You'll have to try again!";
   char *message4 = "Sorry, you've run out of guesses!";
   int i;
                                  /* Store information */
1 struct storage /* Structure tag = storage */
        {
           char small_a; /* One correct letter */
           char capital_a; /* Another correct letter */
           char newline_ch; /* newline character */
           int num_guesses; /* Number of guesses */
        };
                                  /* Declare "letter" *
                                    * using tag "storage" */
2 struct storage letter = {'a', 'A', '\n'};
   letter.num_guesses = 3;
   printf("%s\n", greeting1);
   printf("%s\n", greeting2);
   for (i = 1; i <= letter.num_guesses; i++)
        {
           ch = getchar();
           if (ch == letter.small_a || ch == letter.capital_a)
              {
                 printf("%s", message1);
3 return;
              }
           else
              if (ch == letter.newline_ch)
                 --i; /*Don't count carriage return*/
              else
```

```
                {
                    printf("%s\n", message2);
                    if (i != 3)
                        printf("%s\n", message3);
                 }
        } /* End of for loop body */
        printf("%s", message4);
}
```

**Example 6-1: Scope and Externally Defined Variables**

Compilation Unit 1 Compilation Unit 2

```
----------------- -----------------
            int EXT_2; int EXT_1;
static int STAT;
f1() f3()
{ {
        . extern int EXT_2;
        . .
        . .
        . .
} }
extern int EXT_1;
f2() f4()
{ {
   . .
   . .
   . .
} .
                                    }
                                    f5()
                                    {
                                       static int STAT;
                                       .
                                       .
                                       .
                                    }
```

**Example 6-2: Reinitializing Two auto Variables**

```
/* This example prints the values of two distinct auto *
  * variables that have the same identifier. */
#include <stdio.h>
int main(void)
{
1 int i, x = 2;
    printf("main: %d\n",x);
    for (i = 0; i < 1; i++)
          {
2 int x = 3;
                printf("for loop: %d\n",x);
          }
    printf("main: %d\n", x);
}
```

**Example 6-3: Using the globalvalue Specifier**

```
/* This program illustrates references to previously defined *
  * globalvalue symbols. */
#include <stdio.h>
int test();
globalvalue FAIL = 0;
int main(void)
{
   char c;
                                        /* Get a char from stdin */
   while ( (c = getchar()) != EOF)
        test(c);
}
/* ------------------------------------------------------ *
  * The following code is contained in a separate compilation *
  * unit. *
  * ------------------------------------------------------ */
#include <stdio.h>
#include <ctype.h> /* Include proper module */
globalvalue FAIL; /* Declare global symbols */
test(param_c)
char param_c; /* Declare parameter */
{
                                        /* Test to see if alnum is true */
   if ( (isalnum(param_c)) != FAIL)
        printf("%c\n", param_c);
   return;
}
```

**Example 7-1: Nested Substitution Directives**

```
/* Show multiple substitutions and listing format */
#define AUTHOR james + LAST
int main()
{
   int writer,james,michener,joyce;
#define LAST michener
   writer = AUTHOR;
#undef LAST
#define LAST joyce
   writer = AUTHOR;
}
```

[next] [previous] [contents]

---

**Example 7-2: Using _ _RAD50 and _ _RAD50L Macros**

```
 1 struct FILE_SPEC
 2 {
 3 short device;
 4 long file_name;
 5 short file_type;
 6 };
 7
 8 struct FILE_SPEC myfile =
 9 {
10 __RAD50 ("DL1"),
1 1 0015377u ,
11 __RAD50L ("MYFILE"),
2 1 0007211252456ul ,
12 __RAD50 ("DAT"),
                1 0014474u ,
13 };
```

**Example 8-1: Setting Up Your Own Locale Tables**

```
#pragma list title "tmlc - Define a user's own strange locale."
#pragma module "tmlc", "V01.00"
/*
 * INCLUDE FILES:
 */
#include <defloc.h>
#include <stdio.h>
#include <locale.h>
/*
 * GLOBAL STORAGE or STRUCTURE DEFINITIONS:
 */
/* Non-monetary formatting table */ 1
typedef struct {
        char *decimal_point; /* "." */
        char *thousands_sep; /* "" */
        char *grouping; /* "" */
} lc_nmformat;
/* Monetary formatting table */
typedef struct {
        char *decimal_point; /* "" */
        char *thousands_sep; /* "" */
        char *grouping; /* "" */
        char *int_curr_symbol; /* "" */
        char *currency_symbol; /* "" */
        char *mon_decimal_point; /* "" */
        char *mon_thousands_sep; /* "" */
        char *mon_grouping; /* "" */
        char *positive_sign; /* "" */
        char *negative_sign; /* "" */
        char int_frac_digits; /* CHAR_MAX */
        char frac_digits; /* CHAR_MAX */
        char p_cs_precedes; /* CHAR_MAX */
        char p_sep_by_space; /* CHAR_MAX */
        char n_cs_precedes; /* CHAR_MAX */
        char n_sep_by_space; /* CHAR_MAX */
        char p_sign_posn; /* CHAR_MAX */
        char n_sign_posn; /* CHAR_MAX */
} lc_mformat;
typedef struct {
```

```
        char *abbrev_weekday_names[7];
        char *full_weekday_names[7];
        char *abbrev_month_names[12];
        char *full_month_names[12];
        char *am_pm[2];
        char *time_zones[24];
} lc_time_strings;
/*
**********************************************************************
**
** Test Strange Character Set Types:
*/
/* Define table used for support of the character-testing functions 2
   which are affected by setting the locale LC_CTYPE.
   The affected functions are found in Locale Control section of
   the ANSI C standard.
*/
#define _tab_ (_type_ + 1) /* Allow EOF as an argument in CTYPE
                                    functions. */
static const char _type_ [ ] = {
  0, /* Octal Ascii */
_C, _C, _C, _C, _C, _C, _C, _C, /* 000-007 */
_C, _SC, _SC, _SC, _SC, _SC, _C, _C, /* 010-017 \b\n\t\f\r */
_C, _C, _C, _C, _C, _C, _C, _C, /* 020-027 */
_C, _C, _C, _C, _C, _C, _C, _C, /* 030-037 */
_S, _P, _P, _V, _V, _V, _V, _P, /* 040-047 !"#$%&' */
_P, _P, _V, _V, _P, _P, _P, _V, /* 050-057 ()*+,-./ */
_XD, _XD, _XD, _XD, _XD, _XD, _XD, _XD, /* 060-067 01234567 */
_XD, _XD, _P, _P, _V, _V, _V, _P, /* 070-077 89:;<=>? */
_V, _XL, _XL, _XL, _XL, _XL, _XL, _L, /* 100-107 @ABCDEFG */
_L, _L, _L, _L, _L, _L, _L, _L, /* 110-117 HIJKLMNO */
_L, _L, _L, _L, _L, _L, _L, _L, /* 120-127 PQRSTUVW */
_L, _L, _L, _P, _V, _P, _V, _V, /* 130-137 XYZ[\]^_ */
_V, _XU, _XU, _XU, _XU, _XU, _XU, _U, /* 140-147 `abcdefg */
_U, _U, _U, _U, _U, _U, _U, _U, /* 150-157 hijklmno */
_U, _U, _U, _U, _U, _U, _U, _U, /* 160-167 pqrstuvw */
_U, _U, _U, _P, _V, _P, _V, _C, /* 170-177 xyz{|}~ */
/* Eight bit characters. */
   0, 0, 0, 0, _C, _C, _C, _C, /* 200-207 */
 _C, _C, _C, _C, _C, _C, _C, _C, /* 210-217 */
 _C, _C, _C, _C, _C, _C, _C, _C, /* 220-227 */
   0, 0, 0, _C, _C, _C, _C, _C, /* 230-237 */
   0, _P, _V, _V, 0, _V, 0, _V, /* 240-247 ¡¢£ ¥ §*/
```

```
  _V, _V, _V, _V, 0, 0, 0, 0, /* 250-257 ¨©ª« */
  _V, _V, _V, _V, 0, _V, _V, _P, /* 260-267 °±²³ µ¶·*/
    0, _V, _V, _V, 0, _V, _V, _P, /* 270-277 ¹º»¼½ ¿*/
  _L, _L, _L, _L, _L, _L, _L, _L, /* 300-307 ÀÁÂÃÅÆÇÈ*/
  _L, _L, _L, _L, _L, _L, _L, _L, /* 310-317 ÈÉÊËÌÍÎÏ*/
    0, _L, _L, _L, _L, _L, _L, _L, /* 320-327 ÑÒÓÔÕÖ×*/
  _L, _L, _L, _L, _L, _L, 0, _V, /* 330-337 ØÙÚÛÜÝ ß*/
  _U, _U, _U, _U, _U, _U, _U, _U, /* 340-347 àáâãäåæç*/
  _U, _U, _U, _U, _U, _U, _U, _U, /* 350-357 èéêëìíîï*/
    0, _U, _U, _U, _U, _U, _U, _U, /* 360-367 ñòóôõö÷*/
  _U, _U, _U, _U, _U, _U, 0, 0 /* 370-377 øùúûüý */
          };
/* Define a table used for support of the character collating 3
   functions which are affected by setting the locale portion
   for LC_COLLATE. The affected functions are found in Locale
   Control section of the ANSI C standard.
*/
/* Use Standard DEC MULT. Character Collating Sequence */
static const unsigned char _order_ [ ] = {
    0, 1, 2, 3, 4, 5, 6, 7,
    8, 9, 10, 11, 12, 13, 14, 15,
  16, 17, 18, 19, 20, 21, 22, 23,
  24, 25, 26, 27, 28, 29, 30, 31,
  32, 33, 34, 35, 36, 37, 38, 39,
  40, 41, 42, 43, 44, 45, 46, 47,
  48, 49, 50, 51, 52, 53, 54, 55,
  56, 57, 58, 59, 60, 61, 62, 63,
  64, 65, 66, 67, 68, 69, 70, 71,
  72, 73, 74, 75, 76, 77, 78, 79,
  80, 81, 82, 83, 84, 85, 86, 87,
  88, 89, 90, 91, 92, 93, 94, 95,
  96, 97, 98, 99,100,101,102,103,
104,105,105,107,108,109,110,111,
112,113,114,115,116,117,118,119,
120,121,122,123,124,125,126,127,

128,129,130,131,132,133,134,135, /* 200-207 */
136,137,138,139,140,141,142,143, /* 210-217 */
144,145,146,147,148,149,150,151, /* 220-227 */
152,153,154,155,156,157,158,159, /* 230-237 */
160,161,162,163,164,165,166,167, /* 240-247 ¡¢£ ¥ §*/
168,169,170,171,172,173,174,175, /* 250-257 ¨©ª« */
176,177,178,179,180,181,182,183, /* 260-267 °±²³ µ¶·*/
```

```
184,185,186,187,188,189,190,191, /* 270-277 ¹º»¼½ ¿*/
192,193,194,195,196,197,198,199, /* 300-307 ÀÁÂÃÅÆÇÈ*/
200,201,202,203,204,205,206,207, /* 310-317 ÈÉÊËÌÍÎÏ*/
208,209,210,211,212,213,214,215, /* 320-327 ÑÒÓÔÕÖ×*/
216,217,218,219,220,221,222,223, /* 330-337 ØÙÚÛÜÝ ß*/
224,225,226,227,228,229,230,231, /* 340-347 àáâãäåæç*/
232,233,234,235,236,237,238,239, /* 350-357 èéêëìíîï*/
240,241,242,243,244,245,246,247, /* 360-367 ñòóôõö÷*/
248,249,250,251,252,253,254,255 /* 370-377 øùúûüý */
                    };
/*
4 Strange character mapping table for toupper;
          uppercase characters are mapped to lowercase,
          lowercase characters are mapped to upper.
  */
static const unsigned char _upcase_[ ] = {
    0, 1, 2, 3, 4, 5, 6, 7,
    8, 9, 10, 11, 12, 13, 14, 15,
  16, 17, 18, 19, 20, 21, 22, 23,
  24, 25, 26, 27, 28, 29, 30, 31,
  32, 33, 34, 35, 36, 37, 38, 39,
  40, 41, 42, 43, 44, 45, 46, 47,
  48, 49, 50, 51, 52, 53, 54, 55,
  56, 57, 58, 59, 60, 61, 62, 63,
  64,
       97, 98, 99,100,101,102,103, /* A... to a ... */
104,105,106,107,108,109,110,111,
112,113,114,115,116,117,118,119,
120,121,122, /* ...Z to ...z */
                 91, 92, 93, 94, 95,
  96,
       65, 66, 67, 68, 69, 70, 71, /* abcdefg */
  72, 73, 74, 75, 76, 77, 78, 79, /* hijklmno */
  80, 81, 82, 83, 84, 85, 86, 87, /* pqrstuvw */
  88, 89, 90, /* xyz */
                 123,124,125,126,127,
128,129,130,131,132,133,134,135, /* 200-207 */
136,137,138,139,140,141,142,143, /* 210-217 */
144,145,146,147,148,149,150,151, /* 220-227 */
152,153,154,155,156,157,158,159, /* 230-237 */
160,161,162,163,164,165,166,167, /* 240-247 ¡¢£ ¥ §*/
168,169,170,171,172,173,174,175, /* 250-257 ¨©ª« */
176,177,178,179,180,181,182,183, /* 260-267 °±²³ µ¶·*/
```

```
184,185,186,187,188,189,190,191, /* 270-277 ¹º»¼½ ¿*/
192,193,194,195,196,197,198,199, /* 300-307 ÀÁÂÃÅÆÇÈ*/
200,201,202,203,204,205,205,207, /* 310-317 ÈÉÊËÌÍÎÏ*/
208,209,210,211,212,213,214,215, /* 320-327 ÑÒÓÔÕÖ×*/
216,217,218,219,220,221, /* 330-335 ØÙÚÛÜÝ */
                        222,223, /* 336-337 ß*/
192,193,194,195,196,197,198,199, /* 300-307 àáâãåæçè*/
200,201,202,203,204,205,205,207, /* 310-317 èéêëìíîï*/
208,209,210,211,212,213,214,215, /* 320-327 ñòóôõö÷*/
216,217,218,219,220,221,254,255 /* 330-335 øùúûüý */
};
/*
        Make a strange character mapping table for tolower;
        lowercase characters are mapped to upper, uppercase
        characters are mapped to lower.
 */
static const unsigned char _downcase_[ ] = {
   0, 1, 2, 3, 4, 5, 6, 7,
   8, 9, 10, 11, 12, 13, 14, 15,
  16, 17, 18, 19, 20, 21, 22, 23,
  24, 25, 26, 27, 28, 29, 30, 31,
  32, 33, 34, 35, 36, 37, 38, 39,
  40, 41, 42, 43, 44, 45, 46, 47,
  48, 49, 50, 51, 52, 53, 54, 55,
  56, 57, 58, 59, 60, 61, 62, 63,
  64,
      97, 98, 99,100,101,102,103, /* A... to a ... */
104,105,106,107,108,109,110,111,
112,113,114,115,116,117,118,119,
120,121,122, /* ...Z to ...z */
               91, 92, 93, 94, 95,
  96,
      65, 66, 67, 68, 69, 70, 71, /* abcdefg */
  72, 73, 74, 75, 76, 77, 78, 79, /* hijklmno */
  80, 81, 82, 83, 84, 85, 86, 87, /* pqrstuvw */
  88, 89, 90, /* xyz */
            123,124,125,126,127,
128,129,130,131,132,133,134,135, /* 200-207 */
136,137,138,139,140,141,142,143, /* 210-217 */
144,145,146,147,148,149,150,151, /* 220-227 */
152,153,154,155,156,157,158,159, /* 230-237 */
160,161,162,163,164,165,166,167, /* 240-247 ¡¢£ ¥ §*/
168,169,170,171,172,173,174,175, /* 250-257 ¨©ª« */
```

```
176,177,178,179,180,181,182,183, /* 260-267 °±²³ µ¶·*/
184,185,186,187,188,189,190,191, /* 270-277 ¹º»¼½ ¿*/
224,225,226,227,228,229,230,231, /* 300-307 ÀÁÂÃÄÅÆÇ*/
232,233,234,235,236,237,238,239, /* 310-317 ÈÉÊËÌÍÎÏ*/
240,241,242,243,244,245,246,247, /* 320-327 ÑÒÓÔÕÖ×*/
248,249,250,251,252,253, /* 330-335 ØÙÚÛÜÝ */
                        222,223, /* 356-357 ß*/
224,225,226,227,228,229,230,231, /* 340-347 àáâãäåæç*/
232,233,234,235,236,237,238,239, /* 350-357 èéêëìíîï*/
240,241,242,243,244,245,246,247, /* 360-367 ñòóôõö÷*/
248,249,250,251,252,253,254,255 /* 370-377 øùúûüý */
};
/* Monetary formatting data -- for strange Locale */
static lc_mformat const MFT_TM = 5
        {
                ";", /* *decimal_point */
                ",", /* *thousands_sep */
                "\3", /* *grouping */
                "TMM", /* *int_curr_symbol */
                "Mr.", /* *currency_symbol */
                ";", /* *mon_decimal_point*/
                ",", /* *mon_thousands_sep*/
                "\3", /* *mon_grouping */
                "", /* *positive_sign */
                "^", /* *negative_sign */
                2, /* int_frac_digits */
                0, /* frac_digits */
                1, /* p_cs_precedes */
                0, /* p_sep_by_space */
                1, /* n_cs_precedes */
                0, /* n_sep_by_space */
                1, /* p_sign_posn */
                1 /* n_sign_posn */
        };

static lc_nmformat const NFT_TM = 6
        {
                ";", /* *decimal_point; */
                ",", /* *thousands_sep; */
                "\3" /* *grouping; */
        };
/* Time table -- for strange Locale */ 7
static const lc_time_strings TIM_STR = {
```

```
{ /* abbreviated name weekday name table for strange locale */
                "Sun", "Mon", "Tom",
                "Wed", "Tomm", "Fri",
                "Sat"
},
{ /* Full name weekday name table for C locale */
                "Sunday", "Monday", "Tomday",
                "Wednesday", "Thomasday", "Friday",
                "Saturday"
},
{ /* Abbreviated month name table */
                "Jan", "Feb", "Mar",
                "Apr", "play", "Jun",
                "Jul", "Aug", "Sep",
                "Octy", "Nov", "Dec"
},
{ /* Full month name table */
                "January", "February", "March",
                "April", "Play-day", "June",
                "July", "August", "September",
                "Octopus", "November", "December"
},
{ /* AM, PM */
        "SAM","SPAM"
},
{ /* Time zone table std time zone names for strftime */
        "UTC","","","","TAST","TEST","TCST","TMST","TPST","","","",
        "","","","","","","","","","","","","",""
}
};
/* Define collating -- strange locale */
DEFINE_LC_COLL("tom_m",tmcl,_order_,_upcase_,_downcase_)
/* Define collating type -- strange locale */
DEFINE_LC_CTYPE("tom_m",tmty,_tab_)
/* Monetary formatting data -- strange Locale */
DEFINE_LC_MONETARY("tom_m",tmmn,&MFT_TM)
/* Non Monetary formatting data -- strange Locale */
DEFINE_LC_NUMERIC("tom_m",tmnc,&NFT_TM)
/* Time formatting data -- strange Locale */
DEFINE_LC_TIME("tom_m",tmtm,&TIM_STR)
```

**Example C-1: Sample Program Using localeconv**

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <locale.h>
#include <stdlib.h>
int main()
{
        struct lconv *formatp; /* Pointer to conversion table. */
        double frval; /* Fraction value. */
        char str[20]; /* Formats. */
        char *clocale; /* Displays the current locale. */
        char *ovalue;
        double value = 2.5;
        if (setlocale(LC_ALL,"french,french,french,french") == NULL)
                        return;
        /* Sets the first 4 categories to french: */
        /* LC_COLLATE: Used by strcoll() and strxfrm() */
        /* LC_CTYPE: Used by the character testing FUNCTIONS */
        /* LC_NUMERIC: Numeric formatting (returned by localeconv()) */
        /* LC_MONETARY: Monetary formatting (returned by localeconv()) */
        /* LC_TIME is set to the default ("C") locale */
        clocale = setlocale(LC_ALL, NULL);
        printf("The current locales are: %s\n",clocale);
                                        /* Inquires. This should return */
                                        /* "french,french,french,french,C" */
        formatp = localeconv(); /* Gets the current monetary */
                                        /* conversion format. */
        frval = modf(value,&value); /* Splits into fraction and whole */
                                        /* number, places the whole number */
                                        /* back into value. */
        strcpy(ovalue,formatp->currency_symbol);
                                        /* Copies the currency symbol to */
                                        /* output. */
        if (formatp->p_sep_by_space) /* If a space should precede the */
                strcat(ovalue," "); /* number, insert it here. */
        sprintf(str,"%g",value); /* Converts the whole number. */
        strcat(ovalue,str); /* Copies to output. */
        if (formatp->frac_digits) { /* If fractional digits are allowed */
                                        /* scale by fractional digits, and */
```

```
                                /* use frac_digits as the precision */
                                /* parameter for %*.0g conversion */
                                /* specification. */
        sprintf(str,"%*.0g",
                formatp->frac_digits, /* Precision, replaces (*). */
                frval * (formatp->frac_digits * 10)); /*Scales up. */
        strcat(ovalue,formatp->mon_decimal_point);
                                /* Copies the locale's version of */
                                /* the decimal point. */
        strcat(ovalue,str); /* Copies the fractional digits. */
    }
    printf("%s\n",ovalue); /* Returns pointer to output string. */
}
```

[next] [previous] [contents]

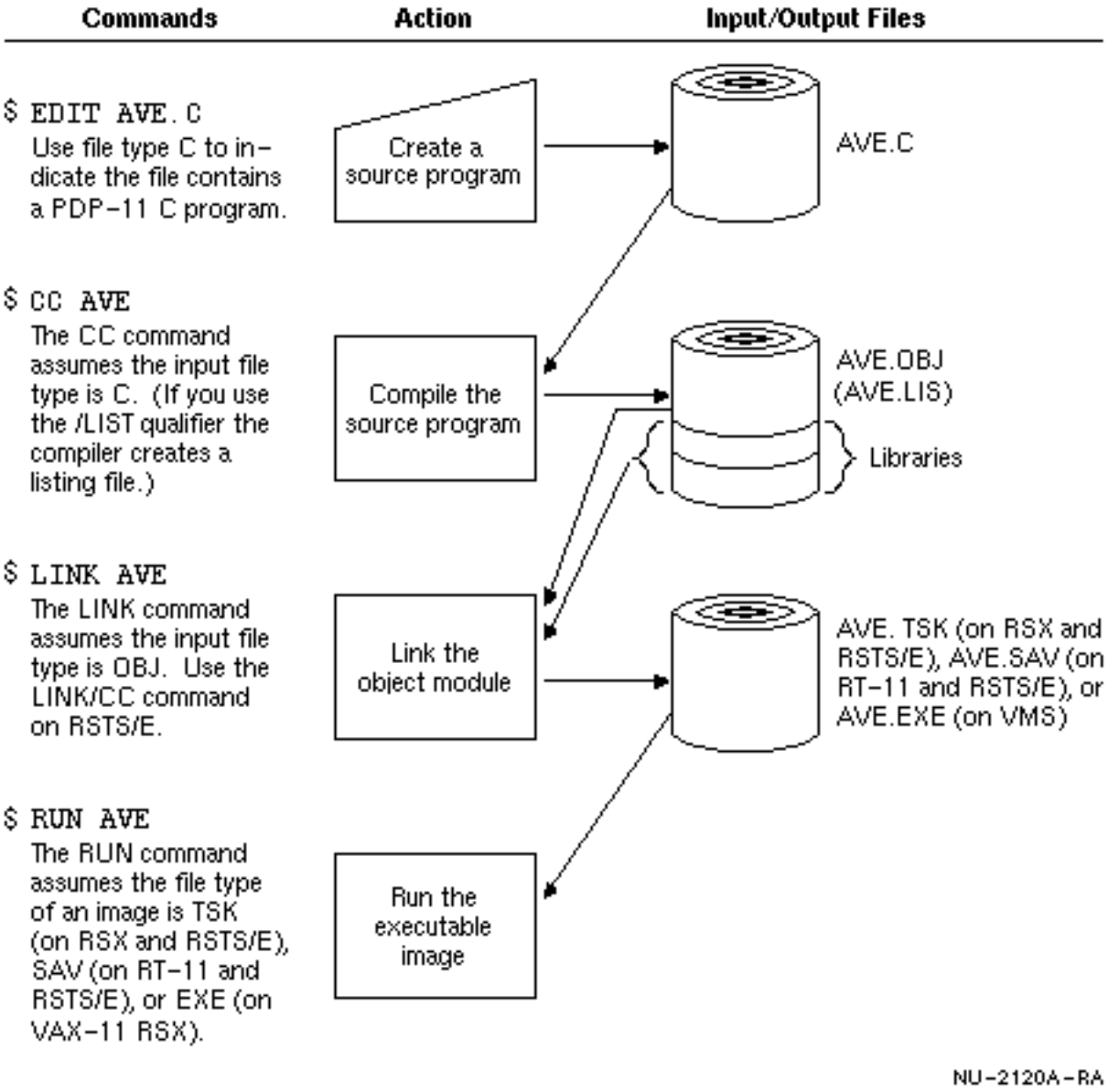
---

**Example C-2: Using the Macro and Function Versions of isalnum**

```
#include <ctype.h>
#include <locale.h>
#include <stdio.h>
int main (void)
{
   short c0, n0 = 0, n1 = 0, n2 = 0;
   setlocale(LC_CTYPE, "C");
   for ( c0 = 0; c0 < 128; c0++)
   {
     if (isalnum(c0)) n0++; /* invoke the macro version of isalnum */
     if ((&isalnum)(c0)) n1++; /* force the call the function isalnum */
#undef isalnum /* undef the macro version of isalnum */
     if (isalnum(c0)) n2++; /* invoke the function version of isalnum */
   }
   printf("The number of alphanumeric characters is %d, %d, and %d.", n0, n1, n2);
}
```

[next] [previous] [contents]

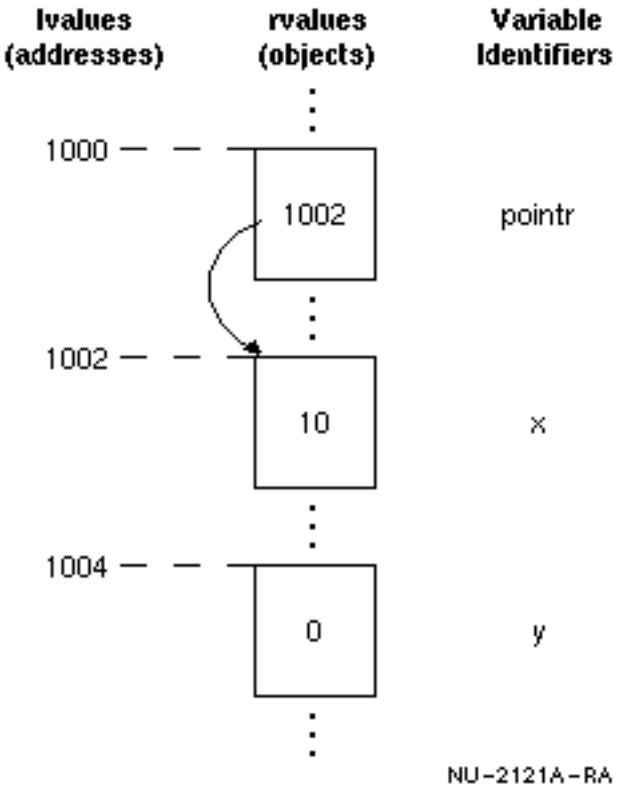
**Figure 1-1: DCL Commands for Developing Programs**



[next] [previous] [contents]

---

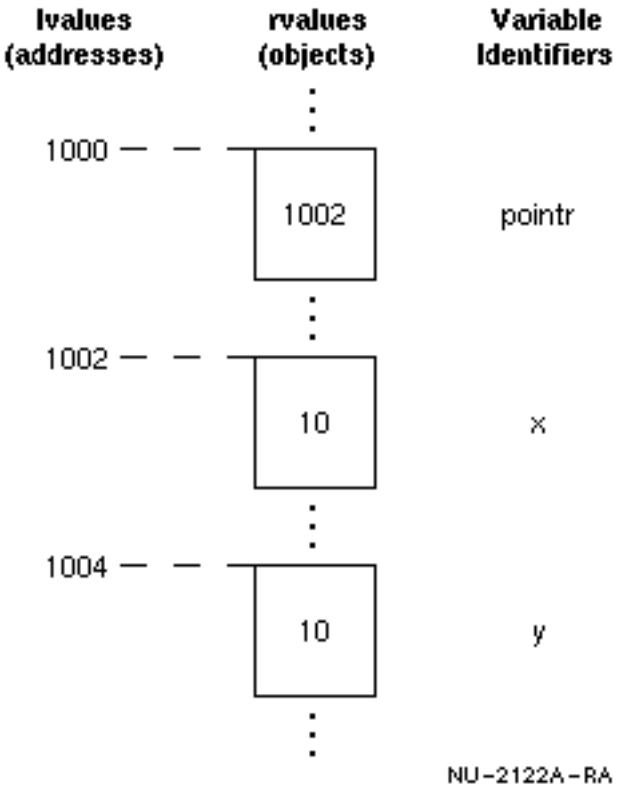
**Figure 2-1: rvalues, lvalues, and Assigning Pointers**



[next] [previous] [contents]

---

**Figure 2-2: The Indirection Operator in Assignments**



## Figure 4-1: Boolean Algebra and the Bitwise Operators

**Boolean Algebra**

AND (&)

| | 1 | 0 |
|---|---|---|
| 1 | 1 | 0 |
| 0 | 0 | 0 |

OR (|)

| | 1 | 0 |
|---|---|---|
| 1 | 1 | 1 |
| 0 | 1 | 0 |

EXCLUSIVE-OR (^)

| | 1 | 0 |
|---|---|---|
| 1 | 0 | 1 |
| 0 | 1 | 0 |

| Operator | Bitwise Operation | Decimal Value |
|---|---|---|
| AND (&) | 1 0 1 1 1 1 1 | 95 |
| | 1 1 0 0 0 0 1 | 97 |
| | 1 0 0 0 0 0 1 | 65 |
| OR (|) | 1 0 1 1 1 1 1 | 95 |
| | 1 1 0 0 0 0 1 | 97 |
| | 1 1 1 1 1 1 1 | 127 |
| X-OR (^) | 1 0 1 1 1 1 1 | 95 |
| | 1 1 0 0 0 0 1 | 97 |
| | 0 1 1 1 1 1 0 | 62 |

NU-2123A-RA

[next] [previous] [contents]

---

**Figure 4-2: Shift Operators**

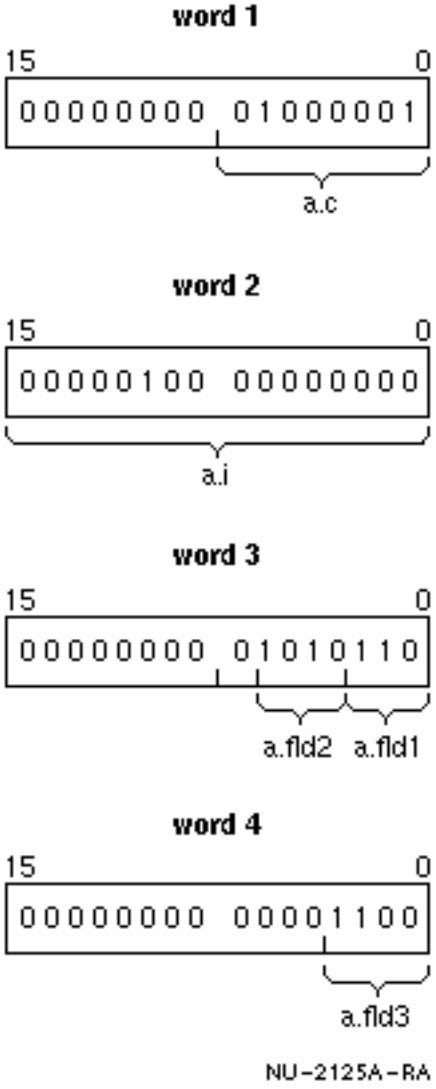
| Bits | Expression | Value |
|---|---|---|
| 1111111111111001 | -7 | -7 |
| 1111111111001000 | (-7)<<3 | -56 |
| | | |
| 0000000000000111 | 7 | 7 |
| 0000000000111000 | 7<<3 | 56 |
| | | |
| 0000000000000111 | 7 | 7 |
| 0000000000000011 | 7>>1 | 3 |
| | | |
| 1111111111111001 | 0xFFF9U | 0xFFF9U |
| 0111111111111100 | (0xFFF9U)>>1 | 0x7FFCU |

NU-2124A-RA

[next] [previous] [contents]

---

**Figure 5-1: Alignment of Structure Members**



[next] [previous] [contents]

**Table 1-1: Copying Files Among Operating Systems**

| | To media format: |
|---|---|
| **From:** | |
| **RSX Format Disk** | RSTS <NA> FIT PIP BACKUP<br>RSX COPY (DCL)<br>PIP (MCR) |
| **RT-11 Format Disk** | FLX FLX <NA><br>RT-11 <NA> COPY <NA> <NA> |
| **DOS Format Tape** | VMS COPY EXCHANGE EXCHANGE BACKUP |
| **Backup Format Tape** | |

[next] [previous] [contents]

---

**Table 2-1: PDP-11 C Keywords**

---

**Keyword Meaning**

---

**Type specifiers:**

---

**int** Integer
**long** 32-bit integer
**signed** Signed integer
**unsigned** Unsigned integer
**short** 16-bit integer
**char** 8-bit integer
**float** Single-precision, floating-point number
**double** Double-precision, floating-point number
**struct** Structure (aggregate of other types)
**union** Union (aggregate of other types)
**variant_struct**

**1**
Structure (aggregate of other types)

**variant_union**

**1**
Union (aggregate of other types)

**enum** Enumerated scalar type
**void** Function return type
**const** Type qualifier
**volatile** Type qualifier

---

**Storage-class specifiers:**

---

**auto** Allocated at function block activation
**static** Allocated at compile time

**register** Allocated at function block activation
**extern** Allocated at compile time
**globaldef**

**1**
Definition of global variable
**globalref**

**1**
Reference to global variable
**globalvalue**

**1**
Definition or declaration of global value

---

1
Type specifier or storage class qualifier provided for compatibility with VAX C. Is a keyword when compiled using the
/NOSTANDARD qualifier. Is not a keyword when compiled using the /STANDARD=ANSI qualifier

---

**Keyword Meaning**

---

**Storage-class specifiers:**

---

**typedef** Tagged set of type specifiers
**readonly**

**1**
Location may only be read
**noshare**

**1**
Is ignored by PDP-11 C

---

**Statements:**

**goto** Transfers control unconditionally
**return** Terminates a function and optionally returns a value to the caller
**continue** Causes next iteration of containing loop
**break** Terminates its corresponding **switch** or loop
**if** Executes following statement conditionally
**else** Provides an alternative for the **if** statement
**for** Iterates the next statement (zero or more times) under control of three expressions
**do** Iterates the next statement (one or more times) while a given condition is true
**while** Iterates the next statement (zero or more times) while a given expression is true
**switch** Executes one or more of the specified cases (multiway branch)
**case** Begins one case for **switch**
**default** Provides default case for **switch**

**Operator:**

**sizeof** Computes size of operand in bytes

1
Type specifier or storage class qualifier provided for compatibility with VAX C. Is a keyword when compiled using the
/NOSTANDARD qualifier. Is not a keyword when compiled using the /STANDARD=ANSI qualifier

[next] [previous] [contents]

---

**Table 2-2: VAX C Keywords**

---

**Keyword Meaning**

---

**Type specifiers:**

---

**int** Integer (On a VAX, 32 bits)
**long** 32-bit integer
**unsigned** Unsigned integer
**short** 16-bit integer
**char** 8-bit integer
**float** Single-precision floating-point number
**double** Double-precision floating-point number
**struct** Structure (aggregate of other types)
**union** Union (aggregate of other types)
**variant_struct** Structure (aggregate of other types)
**variant_union** Union (aggregate of other types)
**enum** Enumerated scalar type
**void** Function return type
**const** Type qualifier
**volatile** Type qualifier

---

**Storage-class specifiers:**

---

**auto** Allocated at every block activation
**static** Allocated at compile time
**register** Allocated at every block activation
**extern** Allocated by an external data definition (at compile time)
**globaldef** Definition of global variable
**globalref** Reference to global variable
**globalvalue** Definition or declaration of global value

---

**Keyword Meaning**

**Storage-class specifiers:**

**readonly** Allocated in read-only program section
**noshare** Assigned NOSHR program section attribute
**typedef** Tagged set of type specifiers

**Statements:**

**goto** Transfers control unconditionally
**return** Terminates a function and optionally returns a value to the caller
**continue** Causes next iteration of containing loop
**break** Terminates its corresponding **switch** or loop
**if** Executes following statement conditionally
**else** Provides an alternative for the **if** statement
**for** Iterates the next statement (zero or more times) under control of three expressions
**do** Iterates the next statement (one or more times) until a given condition is false
**while** Iterates the next statement (zero or more times) while a given expression is true
**switch** Executes one or more of the specified cases (multiway branch)
**case** Begins one case for **switch**
**default** Provides default case for **switch**
**entry** None (reserved for future use)

**Operator:**

**sizeof** Computes size of operand in bytes

[next] [previous] [contents]

### Table 2-3: Trigraph Sequences and Equivalence Characters

| Trigraph Sequence | Equivalence Character |
|---|---|
| ??= | # |
| ??( | [ |
| ??/ | \ |
| ??) | ] |
| ??' | ^ |
| ??< | { |
| ??! | \| |
| ??> | } |
| ??- | ~ |

[next] [previous] [contents]

**Table 4-1: PDP-11 C Operators**

| Operator | Example | Result |
|---|---|---|
| [ ] | a[i] | Access to array members |
| -> | ptr->memb | Access to members of structure and union objects |
| . | struct.memb | Access to members of structure and union objects |
| + [unary] | + a | Value of a |
| - [unary] | - a | Negative of a |
| * [unary] | * a | Reference to object at address a |
| & [unary] | &a | Address of a |
| ~ | ~a | One's complement of a |
| ++ [prefix] | ++a | a after increment |
| ++ [postfix] | a++ | a before increment |
| -- [prefix] | --a | a after decrement |
| -- [postfix] | a-- | a before decrement |
| sizeof | sizeof(t1) | Size in bytes of type t1 |
| sizeof | sizeof e | Size in bytes of expression e |
| (type-name) | (t1)e | Expression e, converted (cast) to type t1 |
| + | a + b | a plus b |
| - [binary] | a-b | a minus b |
| * [binary] | a * b | a times b |
| / | a / b | a divided by b |
| % | a % b | Remainder of a/b (a modulo b) |

>> a >> b a, right-shifted b bits
<< a << b a, left-shifted b bits
< a < b 1 if a < b; 0 otherwise
> a < b 1 if a > b; 0 otherwise
<= a <= b 1 if a <= b; 0 otherwise
>= a >= b 1 if a >= b; 0 otherwise

---

**Operator Example Result**

---

= = a = = b 1 if a equal to b; 0 otherwise
!= a != b 1 if a not equal to b; 0 otherwise
& [binary] a & b Bitwise AND of a and b
| a | b Bitwise OR of a and b
^ a ^ b Bitwise XOR (exclusive OR) of a and b
&& a && b Logical AND of a and b (yields 0 or 1)
*k* a *k* b Logical OR of a and b (yields 0 or 1)
! !a Logical NOT of a (yields 0 or 1)
?: a ? e1 : e2 Expression e1 if a is nonzero,
Expression e2 if a is zero
= a = b b (assigned to a)
+= a += b a plus b (assigned to a)
-= a -= b a minus b (assigned to a)
*

= a

*

= b a times b (assigned to a)
/= a /= b a divided by b (assigned to a)
%= a %= b Remainder of a/b (assigned to a)
>>= a >>= b a, right-shifted b bits (assigned to a)
<<= a <<= b a, left-shifted b bits (assigned to a)
&= a &= b Bitwise a AND b (assigned to a)
| = a | = b Bitwise a OR b (assigned to a)
^= a ^= b Bitwise a XOR b (assigned to a)
, e1,e2 e2 (e1 evaluated first)

---

[next] [previous] [contents]

---

**Table 4-2: Precedence of PDP-11 C Operators**

---

**Category Operator Associativity**

---

Primary ( ) [ ] -> . Left to right
Unary + - ! ~ ++ (type)
*

& sizeof Right to left
Binary (mult.)
*

/ % Left to right
Binary (add.) + - Left to right
Binary (shift) << >> Left to right
Binary (relat.) < <= > >= Left to right
Binary (equal.) = = != Left to right
Binary (bitand) & Left to right
Binary (bitxor) ^ Left to right
Binary (bitor) | Left to right
Binary (AND) && Left to right
Binary (OR) *k* Left to right
Conditional ?: Right to left
Assignment = += -=
*

= /= %= >>= <<= &=
^= | =

Right to left
Comma , Left to right

---

[next] [previous] [contents]

### Table 5-1: PDP-11 C Data Type Keywords

| Scalar | Aggregate | Other Type |
|---|---|---|
| **char** | **struct** | **void** |
| **double** | **union** | |
| **enum** | **variant_struct** | |
| **float** | **variant_union** | |
| **int** | | |
| **long** | | |
| **short** | | |
| **signed** | | |
| **unsigned** | | |

[next] [previous] [contents]

---

**Table 5-2: Size and Range of PDP-11 C Integers**

---

| Keyword | Size | Range |
|---|---|---|
| **long**<br>**long int**<br>**signed long**<br>**signed long int** | 32 bits | -2,147,483,648 to 2,147,483,647 |
| **unsigned long**<br>**unsigned long int** | 32 bits | 0 to 4,294,967,295 |
| **int**<br>**short**<br>**short int**<br>**signed**<br>**signed int**<br>**signed short**<br>**signed short int** | 16 bits | -32,768 to 32,767 |
| **unsigned**<br>**unsigned short**<br>**unsigned short int** | 16 bits | 0 to 65,535 |
| **char**<br>**signed char** | 8 bits | -128 to 127 |
| **unsigned char** | 8 bits | 0 to 255 |

[next] [previous] [contents]

**Table 5-3: PDP-11 C Escape Sequences**

| Character | Mnemonic | Escape Sequence |
|---|---|---|
| newline | NL | \n |
| horizontal tab | HT | \t |
| vertical tab | VT | \v |
| backspace | BS | \b |
| carriage return | CR | \r |
| form feed | FF | \f |
| backslash | \ | \\ |
| apostrophe | ' | \ ' |
| quotes | " | \ " |
| bit pattern | ddd | \ddd or \xddd |
| bell | BEL | \a |
| question mark | ? | \? |

[next] [previous] [contents]

**Table 6-1: PDP-11 C Storage Classes and Storage-Class Specifiers**

| Storage Class Specifiers | Reference Section |
|---|---|
| Internal **auto** , **register** , absence of specifier inside a block or function [1] | Section 6.3 |
| Static **static** | Section 6.4 |
| Global **extern** , absence of specifier outside of all functions | Section 6.5 |

1 Functions declared without a storage-class specifier are of the global storage class, by default.

[next] [previous] [contents]

**Table 6-2: Scope and the Storage-Class Specifiers**

| Storage Class | Inside a Function | | | Outside a Function | | |
|---|---|---|---|---|---|---|
| | Lexical Scope | Link-Time Scope | | Lexical Scope | Link-Time Scope | |
| **auto** | enclosing block | No | illegal | illegal | | |
| **register** | enclosing block | No | illegal | illegal | | |
| **static** | enclosing block | No | CU | | | |

| | | | |
|---|---|---|---|
| | | 1 | |
| | | | No |
| **extern** enclosing block | | | |
| | Yes CU | | |
| | | 1 | |
| | | | Yes |
| **globalvalue** enclosing block | | | |
| | Yes CU | | |
| | | 1 | |
| | | | Yes |
| (none) enclosing block | | | |
| | No CU | | |
| | | 1 | |
| | | | Yes |

---

1
Compilation Unit still must be declared before used.

---

[next] [previous] [contents]

**Table 6-3: Location, Lifetime, and the Storage-Class Keywords**

| Storage Class | Location | Lifetime |
|---|---|---|
| (none) | Psect, stack, or register | Temporary or permanent |
| **auto** | Stack or register | Temporary |
| **register** | Stack or register | Temporary |
| **static** | Psect | Permanent |
| **extern** | Psect | Permanent |
| **globalvalue** | No storage allocated | Permanent |

[next] [previous] [contents]

**Table 7-1: Logical Names for PDP-11 C Include Files**

| Host System | Quoted Form | Bracketed Form (Logical Name 1) | Bracketed Form (Logical Name 2) | Bracketed Form (Standard Installation Location) |
|---|---|---|---|---|
| **PDP-11 C** | | | | |
| VMS | C$INCLUDE: | PDP11C$INCLUDE: | <NA> | LB:[1,1] |
| RSX-11M-PLUS and Micro /RSX | C$INCLUDE: | PDP11C$INCLUDE: | CLB: | LB:[1,1] |
| RSX-11M | <NA> | <NA> | CLB: | LB:[1,1] |
| RSTS/E | <NA> | PDP11$INCLUDE: | CLB: | CC$: |
| RT-11 | <NA> | <NA> | CLB: | SY: |
| **VAX C** | | | | |
| VMS | C$INCLUDE: | VAXC$INCLUDE: | <NA> | SYS$LIBRARY: |

[next] [previous] [contents]

**Table 7-2: PDP-11 Character Sets**

| | | |
|---|---|---|
| iso_latin_1 | french_canadian | |
| dec_mcs | german | |
| ascii | italian | |
| british | norwegian | |
| danish | | |
| | 1 | |
| | | portuguese |
| dutch | spanish | |
| finnish | swedish | |
| french | swiss | |

1 The ``danish'' and ``norwegian'' character sets are synonymous.

**Table 7-3: Psect Types and Associated Data Types**

| Psect Type | Types of Data |
|---|---|
| const | String literals, character constants, numeric constants. The default attributes for psects of this type are con, d, lcl, nosav, ro, and rel. The default name is $CONST. |
| static_ro | Objects declared with the **const** attribute. The default attributes for psects of this type are con, d, gbl, sav, ro, and rel. The default name is $READO. |
| static_rw | Objects declared with the **static** or **extern** attribute, but not with the const attribute. The default attributes for psects of this type are con, d, gbl, sav, rw, and rel. The default name is $READW. |
| code_i | Function code. The default attributes for psects of this type are con, i, lcl, nosav, ro, and rel. The default name is $CODEI. |
| code_d | Data generated as part of the function code. The default attributes for psects of this type are con, d, lcl, nosav, ro , and rel. The default name is $CODED. |

**Table 8-1: PDP-11 RTL Psects**

| Name | Use |
|---|---|
| CC$ALL | RMS default ALL block. |
| CC$FAB | RMS default FAB block. |
| CC$DAT | RMS default DAT block. |
| CC$KEY | RMS default KEY block. |
| CC$NAM | RMS default NAM block. |
| CC$PRO | RMS default PRO block. |
| CC$RAB | RMS default RAB block. |
| CC$SUM | RMS default SUM block. |
| C$CCT0<br>C$CCT2 | Character collating table. Used for locale-specific routines to determine the collating sequence of each character set. |
| C$CMT0<br>C$CMT2 | Character mapping table. Used for locale-specific routines to determine the results of character mapping functions for each character set. |
| C$CTT0<br>C$CTT2 | Character testing table. Used for locale-specific routines to determine the results of character testing functions for each character set. |
| C$END0<br>C$END1<br>C$END2<br>C$END3 | The C$ENDx psects are used for end of task processing. The addresses of functions to be called by the PDP-11 C RTL at task-exit time are placed in the psect C$END1. For instance, the address of the routine that ensures all files are closed is placed in C$END1. This is separate from the atexit system function. The psects C$END0, C$END1, and C$END3 are reserved for use by the PDP11-C RTL. The addresses of routines to be |

called
at task exit can be placed in the psect C$END2. Modules that define this psect may not reside in a resident library.

C$FCSI Instructions for PDP-11 C FCS Extension Library routines. These routines provide support for calling FCS routines.

C$INI0
C$INI1
C$INI2
C$INI3

Similar to the C$ENDx psects, the C$INIx psects are used to provide the addresses of routines to be called at task startup. The psects C$INI0, C$INI1, and C$INI3 are reserved for use by the PDP-11 C RTL. The psect C$INI2 is available to place the addresses of routines to be called at task startup. Modules that define this psect may not reside in a resident library.

C$INIR Code for initialization routines.

C$MFT0
C$MFT2

Monetary formatting table. Used for locale-specific routines to determine the results of monetary formatting functions for each character set.

---

**Name Use**

---

C$NFT0
C$NFT2

Numeric formatting table. Used for locale-specific routines to determine the results of numeric formatting functions for each character set.

C$OTSC Constant data for PDP-11 C Object Time System (OTS) routines.

C$OTSD Read data for the PDP-11 C OTS routines.

C$OTSH RT-11 only. Used to determine size of C$OTSI and C$STDI psects.

C$OTSI Instructions for PDP-11 C OTS routines. These routines handle most of the math and conversion functions.

C$OTSJ RT-11 only. Used to determine size of C$OTSI and C$STDI psects.

C$OTSR Constant data for PDP-11 C OTS routines.

C$OTSW Writeable storage for PDP-11 C OTS routines. Modules that contain this psect may not reside in a resident library.

C$RMSI Instructions for PDP-11 C RMS Extension Library routines. These routines provide

support for calling RMS routines.

C$STDC Constant data for the Standard Library routines.

C$STDD Read data for the Standard Library routines.

C$STDI Instructions for the Standard Library routines.

C$STDR Constant data for the Standard Library routines.

C$TIM0

C$TIM2

Time formatting table. Used for locale-specific routines to determine the results of time formatting functions for each character set.

$PFCXT FCS Transfer Vector. Standard I/O calls to FCS go through this vector. Routines containing this psect may not reside in a resident library because the psect contains references to FCS routines. However, this does allow several other routines to reside in a resident library.

$PIOXT I/O Transfer Vector. This is used to allow PDP-11 C to access several low-level I/O systems easily. $PIOXT contains two addresses for each low-level I/O action used by PDP-11 C. One address is for support for native I/O for that action, the other is for support for either RMS or FCS I/O for that action. Modules that define this psect may not reside in a resident library.

$PRLUN Bit mask used for reserving LUNs. The first word indicates the number of words that follow. These make up the mask. Modules that define this psect may not reside in a resident library.

---

**Name Use**

---

$PRMXT RMS Transfer Vector. All Standard I/O calls to RMS go through this vector. Routines containing this psect may not reside in a resident library, because the psect contains references to several RMS routines. However, this allows a number of other routines to live in resident libraries.

$$C The PDP-11 C OTS work area. This is read/write data space used by the RTL. Modules that define this psect may not reside in a resident library.

$$CAST OTS work area psect containing structure required by asctime ( ) function.

1

$$CCLK OTS work area psect containing storage required for correct use of the clock function.

1

$$CEXI OTS work area psect containing storage required to register the addresses of the functions to be called during the execution of the atexit ( ) routine.

1
$$CGEN OTS work area psect containing storage required to support the getenv ( ) function.

1
$$CLOC OTS work area psect containing storage required to support the locale functions.

1
$$CMLL OTS work area psect containing storage required to support memory allocation functions.

1
$$CSIG OTS work area psect containing storage required to support the signal functions.

1
$$CSIO OTS work area psect containing storage required to support standard I/O operations.

1
$$CTIM OTS work area psect containing storage for the required struct tm.

1

---

1
This read/write psect will only appear in the user task if the related functions are referenced in the user's program.

---

[next] [previous] [contents]

**Table 8-2: Global Symbols**

| Format to Exclude | Symbol[1] |
|---|---|
| d, i, u | $PULON, $PLONG |
| o, p | $POLON |
| x, X | $PHLON |
| f, e, E, g, G | $PFLOA, $PFLOE |

1 Where two symbols are shown, a **globalvalue** statement for both symbols must appear in the program.

[next] [previous] [contents]

**Table B-1: PDP-11 C Standard Library Header Files**

| Module | Description |
|---|---|
| **assert.h** | Definition of the **assert** macro |
| **ctype.h** | Character type and macro definitions for character classification and mapping functions |
| **errno.h** | Error number definitions |
| **float.h** | Macro definitions that provide implementation-specific floating-point limits |
| **limits.h** | Macro definitions that provide implementation-specific constraints |
| **locale.h** | Localization and formatting of dates and times |
| **math.h** | Math functions |
| **setjmp.h** | Mechanism for bypassing normal function call and return protocol |
| **signal.h** | Signal and condition handling value definitions |
| **stdarg.h** | Access to variable length argument lists specified through the ellipsis notation in a function prototype |
| **stddef.h** | Common definitions |
| **stdio.h** | Standard I/O definitions |
| **stdlib.h** | General utility functions |
| **string.h** | String-handling function definitions |
| **time.h** | Time manipulation functions |

[next] [previous] [contents]

### Table B-2: PDP-11 C FCS Extension Library Header Files

| Module | Description |
|---|---|
| **fcs.h** | FCS values, offsets, and data structures |
| **fcsfhb.h** | FCS file header block |
| **fcsiff.h** | FCS index file format |

[next] [previous] [contents]

**Table B-3: PDP-11 C RMS Extension Library Header Files**

| Module | Description |
|---|---|
| **fab.h** | File access block definitions |
| **nam.h** | Name block definitions |
| **rab.h** | Record access block definitions |
| **rms.h** | All RMS structures and return status values |
| **rmsdef.h** | RMS return status values |
| **rmsops.h** | RMS Extension Library operations |
| **rmsorg.h** | Replacement for RMS Extension Library ORG macro |
| **rmspoo.h** | RMS Extension Library pool space |
| **xab.h** | Extended attribute block definitions |

[next] [previous] [contents]

### Table B-4: PDP-11 C System Interface Header Files

| Module | Description |
|---|---|
| **rstsys.h** | Defines an interface to RSTS/E system-provided routines |
| **rsxsys.h** | Defines an interface to RSX system-provided routines |
| **rtsys.h** | Defines an interface to RT-11 system-provided routines |
| **nam.h** | Name block definitions |
| **rab.h** | Record access block definitions |

[next] [previous] [contents]

---

**Table D-1: Data Type Keywords**

---

**Keyword Meaning**

---

**Type specifiers:**

---

**int** Integer
**long** 32-bit integer
**signed** Signed integer
**unsigned** Unsigned integer
**short** 16-bit integer
**char** 8-bit integer
**float** Single-precision, floating-point number
**double** Double-precision, floating-point number
**struct** Structure (aggregate of other types)
**union** Union (aggregate of other types)
**variant_struct**

**1**
Structure (aggregate of other types)

**variant_union**

**1**
Union (aggregate of other types)

**enum** Enumerated scalar type
**void** Function return type
**const** Type qualifier
**volatile** Type qualifier

---

**Storage-class specifiers:**

---

**auto** Allocated at function block activation
**static** Allocated at compile time

**register** Allocated at function block activation
**extern** Allocated at compile time
**globaldef**

**1**
Definition of global variable
**globalref**

**1**
Reference to global variable
**globalvalue**

**1**
Definition or declaration of global value

---

1
Type specifier or storage class specifier provided for compatibility with VAX C. Only available when compiling
/NOSTANDARD.

---

**Keyword Meaning**

---

**Storage-class specifiers:**

---

**typedef** Tagged set of type specifiers

---

**Storage-class qualifier:**

---

**readonly**

**1**
Location may only be read
**noshare**

**1**

| | |
|---|---|
| | Is ignored by PDP-11 C |

---

1
Type specifier or storage class specifier provided for compatibility with VAX C. Only available when compiling
/NOSTANDARD.

---

[next] [previous] [contents]

---

**Table D-2: Precedence of Operators**

---

| Category | Association | Operator |
|---|---|---|
| Primary | Left to right | ( ) [ ] -> . |
| Unary | Right to left | ! ~ ++ - ( *type* ) + - * & **sizeof** |
| Binary | Left to right | * / % + - << >> < <= > >= = = != & ^ \| && \|\| |
| Conditional | Right to left | ?: |
| Assignment | Right to left | = += -= * = /= %= >>= <<= &= ^= \| = |
| Comma | Left to right | , |

---

### 5.4 Floating-Point Numbers (float, double)

When declaring floating-point variables, you determine the amount of precision needed for the stored object. In PDP-11 C, you can have either single-precision or double-precision variables. The representation of the data type **float** is a 32-bit (single precision) floating point object.

The representation of the data type **double** is a 64-bit (double precision) floating point object.

The sizes and supported ranges of PDP-11 C floating-point numbers are as follows:

**float**

Float is a 32-bit keyword with a range of:

FLT_MAX to FLT_MAX

FLT_MAX is approximately equal to:

1 : 7 £ 10

38

The minimum positive floating number is FLT_MIN, which is approximately equal to:

2 : 9 £ 10

39

Float values are precise to 6 decimal digits.

**double**

Double is a 64-bit keyword with a range of:

DBL_MAX to DBL_MAX

DBL_MAX is approximately equal to:

1 : 7 £ 10

38

The minimum positive floating number is DBL_MIN, which is approximately equal to:

2 : 9 £ 10

39

Double values are precise to 16 decimal digits.

The exact values of FLT_MAX, FLT_MIN, DBL_MAX, and DBL_MIN may be found in *float.h* .

A floating-point constant has an integral part, a decimal

point, a fractional part, the letter e or E, and an optionally signed integer exponent. The integral and fractional parts consist of decimal digits; you may omit either the integral or fractional part. You may omit either the decimal point with the following digits or the exponent (e,E), but not both.
By default, floating-point constants are of type **double** . However, using the suffix (F,f) will yield type **float** and the suffix (L,l) will yield **long double** . Note that in PDP-11 C, **long double** is the same as **double** .
The following are examples of floating-point constants:
3.0e10
3.0E-10
3.0e+10
3E10F
3.0L
.120e2
.120

[next] [previous] [contents]

---

1
Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, *Second Edition* (Englewood Cliffs, New Jersey: Prentice-Hall, 1988).

[next] [previous] [contents]

---

1
Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, *Second Edition* (Englewood Cliffs, New Jersey: Prentice-Hall, 1988), p.1.

**Example 2-4: Conditional Execution Using the if Statement**

```
/* This program asks the user to guess a letter. The *
 * program tells whether the answer is correct or *
 * incorrect. The program is hard coded to accept 'a' or *
 * 'A' as the correct letter. */
#include <stdio.h>
int main(void)
{
   int ch; /* Declare a character */
                                /* Ask the user to guess */
   printf("Guess which letter I'm thinking of!\n");
1 ch = getchar(); /* Get the character */
                                /* Correct = "a" or "A" */
2 if (ch == 'a' || ch == 'A')
                                /* If correct guess */
        printf("You're right!");
   else /* If incorrect guess */
        {
           printf("You're wrong.\n");
           printf("You'll have to try again!");
        }
}
```

[next] [previous] [contents]

---

1
Bruce Anderson, ``Type Syntax in the Language C: An Object Lesson in Syntactic Innovation,'' *SIGPLAN Notices* 15, No. 2 (March 1980).

**additive operator**
An operator that performs addition ( + ) or subtraction ( - ). These operators perform arithmetic conversion on each of the operands, if necessary. *See also* arithmetic conversion rules.

**aggregate**
A data structure (array, structure, or union) composed of segments called members. You declare the members to be of either a scalar or aggregate data type. Members of an array are called elements and must be of the same data type. A structure has named members that can be of different data types. A union is a structure that is as long as its longest declared member and that contains the value of only one member at a time.

**ampersand ( & )**
As a unary operator, computes the address of its operand. As a binary operator, performs a bitwise AND on two operands; both must be of integral type. As an assignment operator ( &= ), performs a bitwise AND on two expressions and assigns the result to the left object. The double ampersand ( && ), a binary operator, performs a logical AND on two operands. *See also* binary operator, bitwise operator, logical operator, and unary operator.

**argument**
An expression that appears within the parentheses of a function call. The expression is evaluated and the result is copied into the corresponding parameter of the called function. *See also* argument passing and parameter.

**argument passing**
The mechanism by which the value of the argument in a function call is copied to a parameter in the called function. In PDP-11 C, all arguments are passed by value; that is, the parameter receives a copy of the argument's value. Therefore, a function called in PDP-11 C cannot modify the value of an argument except by using its address. In general, addresses are passed using the ampersand operator

( *see* ampersand ( & )) in the function call or by passing a pointer variable. In addition, using an array or function name (an array with no brackets or function identifier with no parentheses) as an argument results in the passing of the address of the array or function.

**arithmetic conversion rules**
The set of rules that govern the changing of a value of an operand from one data type to another in arithmetic expressions. Conversions take place in assignments by changing the type of the right operand's result to that of the object referred to by the left operand; the resultant type also applies to the assignment expression. Conversions are also performed when arguments are passed to functions.

**arithmetic operator**
A PDP-11 C operator that performs a mathematical operation. In an expression, certain operations take precedence (are performed first) over other operations. The unary minus operator ( - ) is at the highest level of precedence. At the next level are the binary operators for multiplication (
*
), division ( / ), and mod ( % ). At the next level are addition ( + ) and subtraction ( - ). There is no exponentiation operator. If necessary, all the binary operators perform the arithmetic conversions on their operands. *See also* arithmetic conversion rules and binary operator.

**arithmetic type**
One of the integral data types, enumerated types, single- or double-precision floating-point ( **float** or **double** ) types.

**array**
An aggregate data type consisting of subscripted members, called elements, all of the same type. Elements of an array can be one of the fundamental types or can be structures, unions, or other arrays (to form multidimensional arrays).

**assignment expression**
An expression that has the following form:

E1 asgnop E2
Expression E1 must evaluate to an lvalue, the operator asgnop is an assignment operator, and E2 is an expression. The type of an assignment expression is that of its left operand. The value of an assignment expression is that of the left operand after the assignment takes place. If the operator is of the form op=, then the operation E1 op ( E2 ) is performed, and the result is assigned to the object referenced by E1; E1 is evaluated once.

**assignment operator**
The combination of an arithmetic or bitwise operator with the assignment symbol ( = ); also, the assignment symbol by itself. *See also* assignment expression.

**asterisk (**
*****

**)**
As a unary operator, treats its operand as an address and results in the contents of that address. As a binary operator, multiplies two operands, performing the arithmetic conversions, if necessary. As an assignment operator ( * = ), multiplies an expression by the value of the object referenced by the left operand, and assigns the product to that object. *See also* unary operator and binary operator.

**auto storage class**
A storage class that defines a variable whose storage is allocated automatically upon entry into a function or block, and is automatically deallocated upon exit from a function or block. *See also* block.

**binary operator**
An operator that is placed between two operands. The binary operators include arithmetic operators, shift operators, relational operators, equality operators, bitwise operators (AND, OR, and XOR), logical operators (logical AND, logical OR), and the comma operator, in that order of precedence. All binary operators group from left to right. PDP-11 C has no exponentiation operator. The Run-Time Library function **exp** must be used instead.

**bit field**
A structure member that may consist of a specified number of bits, which may be named or unnamed. A colon is used to separate the member's declarator (if any) from a constant-expression that gives the field width in bits. No field may be longer than 16 bits (1 word) in PDP-11 C.

**bitwise operator**
An operator that performs Boolean algebra on the binary values of two operands, which must be integral. If necessary, the operators perform the arithmetic conversions. Both operands are evaluated. All bitwise operators are associative, and expressions using them may be rearranged. The operators include, in order of precedence, the single ampersand ( & ) (bitwise AND), the circumflex ( ^ ) (bitwise exclusive OR), and the single bar ( | ) (bitwise inclusive OR).

**block**
A compound statement when it is not the body of a function. *See also* compound statement.

**block activation**
The run-time activation of a block or function, in which local

**auto** and **register** variables are allocated storage and, if they are declared with initializers, given initial values. Variables of storage class **static** , **extern** , **globaldef** , and **globalvalue** are allocated and initialized at link time. The block activation precedes the execution of any executable statements in the function or block. Functions are activated when they are called. Internal blocks (compound statements) are activated when the program control flows into them. Internal blocks are not activated if they are entered by a **goto** statement, unless the **goto** target is the label of the block rather than the label of some statement within the block. If a block is entered by a **goto** statement, references to **auto** and **register** variables declared in the block are still valid references, but the variables may not be properly initialized. Blocks that make up the body of a **switch** statement are not activated; **auto** or **register** variables declared in the block are not initialized.

**cast**

An expression preceded by a cast operator of the form *type_name* . The cast operator forces the conversion of the evaluated expression to the given type. The expression is assigned to a variable of the specified type, which is then used in place of the whole construction. The cast operator has the same precedence as the other unary operators.

**character**

Character refers to:

.
A member of a supported character set.

.
An object of the PDP-11 C data type **char** , which is stored in a single byte of memory. An object of type **char** always represents a single character, not a string.

.
A constant consisting of up to four ASCII characters for a **long int** , two ASCII characters for a **short int** , and one ASCII character for a **char** size object. The ASCII characters must be enclosed in apostrophes ( ' ' ), not quotation marks ( " " ).

*See also* string.

**comma operator (,)**

A PDP-11 C operator used to separate two expressions as follows:
E1, E2
The expressions E1 and E2 are evaluated left to right, and the value of E1 is discarded. The type and value of the comma expression are those of E2.

**comment**
A sequence of characters introduced by the pair ( / * ) and terminated by ( * / ). Comments are ignored during compilation. They may not be nested.

**compilation unit**
All of the source files compiled to form a single object module. Declarations and definitions within a compilation unit determine the lexical scope of functions and variables.

**compound statement**
Valid PDP-11 C statements enclosed in braces ( { } ). Compound statements can also include declarations. The scope of these variables is local to the compound statement. A compound statement, when it is not the body of a function, is called a block.

**conditional operator (?:)**
The PDP-11 C operator ( ?: ), which is used in conditional expressions of the following form:
E1 ? E2 : E3
E1, E2, and E3 are valid PDP-11 C expressions. E1 is evaluated, and if it is nonzero, the result is the value of E2; otherwise, the result is the value of E3. Either E2 or E3 is evaluated, but not both.

**constant**
A primary expression whose value does not change. A constant may be literal or symbolic.

**constant expression**
An expression involving only constants. Constant expressions are evaluated at compile time so they may be used wherever a constant is valid.

**conversion**
The changing of a value from one data type to another. Conversions take place in assignments by changing the type of the right operand's result to that of the object referred to by the left operand; the resultant type also applies to the assignment expression. Conversions are also performed when arguments are passed to functions **char** and **short** become **int** and **float** becomes **double** . If no function prototype is in scope, **unsigned char** and **unsigned short** become **unsigned int** . Conversions can also be forced by means of a cast. Conversions are performed on operands in arithmetic expressions by the arithmetic conversions. *See also* cast.

**conversion characters**
A character used with the PDP-11 C Standard Library Standard I/O functions that is preceded by a percent sign ( % ) and specifies an input or output format. For example, letter d instructs the function to input/output the value in a decimal format.

**data cache**

The area of the extended buffer pool that stores information relating to RSTS/E read operations. Using the data cache reduces the number of data transfers from the disk.

**data definition**
The syntax that both declares the data type of an object and reserves its storage. For variables that are internal to a function, the data definition is the same as the declaration. For external variables, the data definition is external to any function (an external data definition).

**data type qualifier**
Keywords which affect the allocation or access of data storage. The two data type qualifiers are **const** and **volatile** .

**declaration**
A statement that gives the data type and possibly the storage class of one or more variables.

**declarator**
The part of the declaration that lists the identifiers of the declared objects and may contain operators that declare a pointer, function, or array of objects of the declared type.

**directives**
*See* preprocessor directives.

**elements**
Members of an array, structure, or union. *See also* aggregate.

**enumerated type**
A type defined (with the **enum** keyword) to have an ordered set of integer values. The integer values are associated with constant identifiers named in the declaration. Although **enum** variables are stored internally as integers, use them in programs as if they have a distinct data type named in the **enum** declaration.

**equality operator (= = !=)**
One of the operators, equal to ( = = ), or not equal to ( != ). They are analogous to the relational operators, but at the next lower level of precedence.

**exponentiation operator**
The C language does not have an exponentiation operator. Use the PDP-11 C Run-Time Library function **exp** .

**expression**
A series of tokens that the compiler can use to produce a value. Expressions have one or more operands and, usually, one or more operators. An identifier with no operator is an expression that yields a value directly. Operands are either identifiers (such as variable names) or other expressions, which are sometimes called subexpressions. *See also* operator and tokens.

**extension libraries**
Libraries that contain extensions beyond ANSI standards. PDP-11 C provides these extensions to support file control services (FCS) and record management services (RMS) file operations as well as providing support for RSX, RSTS/E, and RT-11 system directories.

**external storage class**
A storage class that permits identifiers to have a link-time scope that can possibly span object modules. Identifiers of this storage class are defined outside of functions using no storage class specifier, and are declared, optionally, throughout the program using the **extern** specifier. External

variables provide a means other than argument passing for exchanging data between the functions that comprise a PDP-11 C program. *See also* link-time scope.

**file control services (FCS) library**
An extension library containing a set of routines supplied with PDP-11 C that supports the FCS facility.

**file specification**
An identifier that specifies an existing file.

**floating type**
One of the data types **float** or **double** , representing a single- or double-precision floating-point number. The range of values for the **double** variables is the same as for that of **float** variables, but the precision is 16 decimal digits, as opposed to 7.

**function**
The primary unit from which PDP-11 C programs are constructed. A function definition begins with a name and parameter list, followed by the declarations of the parameters (if any) and the body of the function enclosed in braces ( { } ). The function body consists of the declarations of any local variables and the set of statements that perform its action. Functions do not have to return a value to the caller. C functions cannot be nested, that is, a function may not contain another function. *See also* function call.

**function call**
A primary expression, usually a function identifier followed by parentheses, that is used to invoke the function. The parentheses contain a (possibly empty) comma-separated list of expressions that are the arguments to the function. Any previously undeclared identifier followed immediately by parentheses is declared as a function returning **int** . A function may call itself recursively.

**function prototype**
A function prototype is a function declaration that specifies the data types of its arguments in the identifier list. PDP-11 C uses the prototype to ensure that any function definition,

and all declarations and calls within the scope of the prototype, contain the correct number of arguments or parameters, and that each argument or parameter is of the correct data type.