Prime

C User's Guide

Release T3.0–23.0

DOC7534–4LA

C User's Guide

Fourth Edition

Marilyn Hammond

This guide documents the use of the PRIMOS C compiler and libraries as implemented on the PRIMOS operating system at Translator Family Release T3.0-23.0.

Prime Computer, Inc., Prime Park, Natick, MA 01760

Printing History

First Edition (DOC7534-193) June 1985
Second Edition (DOC7534-2LA) January 1986
Third Edition (DOC7534-3LA) January 1988
Fourth Edition (DOC7534-4LA) June 1990

Credits

Editorial: Norma Kellstedt and Judy Goodman
Engineering Support: Wendy Merrill
Illustration: Carol Smith
Production: Judy Gordon

How to Order Technical Documents

To order copies of documents, or to obtain a catalog and price list:

United States Customers	*International*
Call Prime Telemarketing, toll free, at 1-800-343-2533, Monday through Thursday, 8:30 a.m. to 8:00 p.m. and Friday, 8:30 a.m. to 6:00 p.m. (EST).	Contact your local Prime subsidiary or distributor.

PRIME SERVICE℠

Prime provides the following toll-free number for customers in the United States needing service:

1-800-800-PRIME

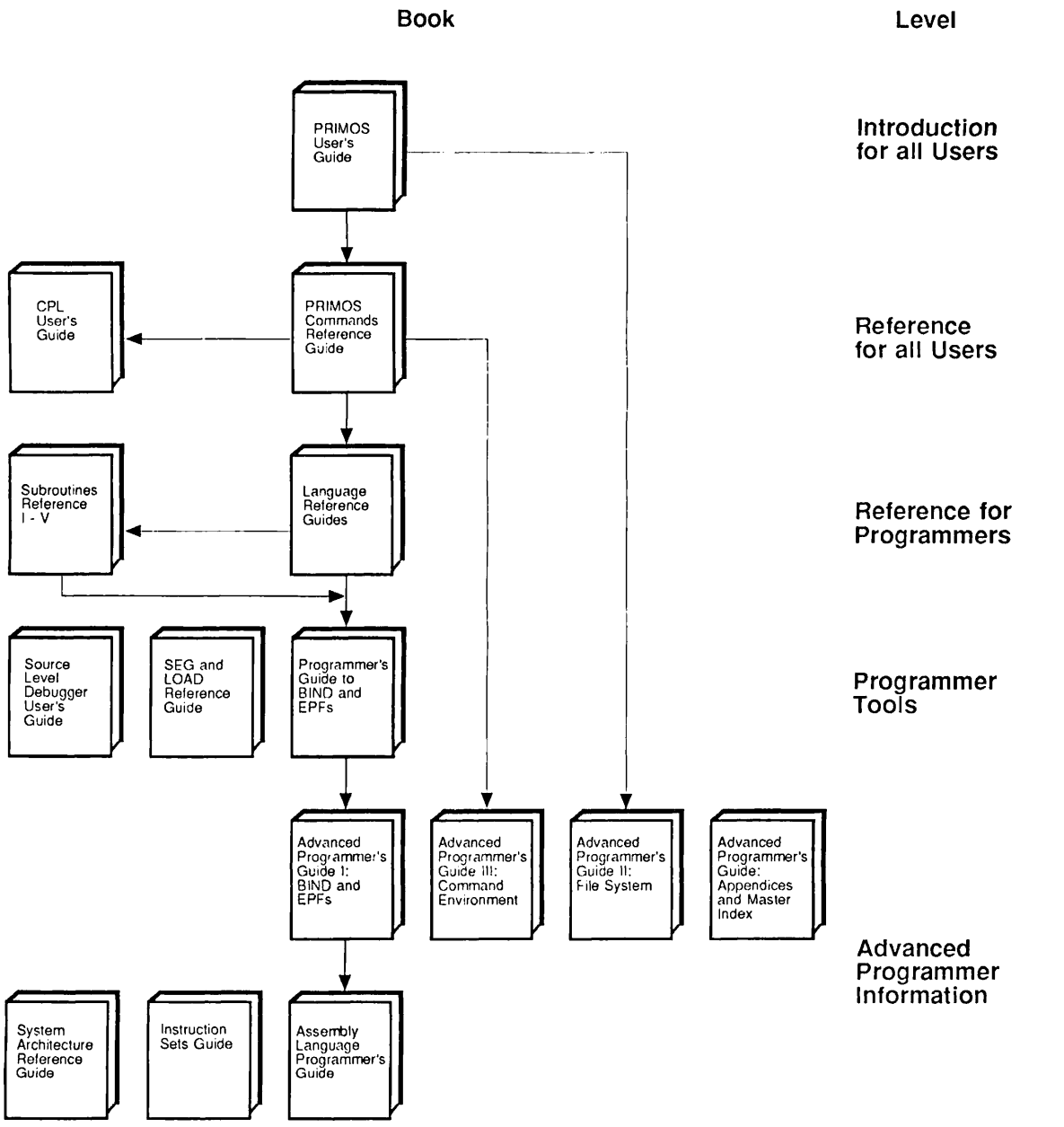
For other locations, contact your Prime representative.

Surveys and Correspondence

Please comment on this manual using the Reader Response Form provided in the back of this book. Address any additional comments on this or other Prime documents to:

Technical Publications Department
Prime Computer, Inc.
500 Old Connecticut Path
Framingham, MA 01701

Reading Path for PRIMOS Documentation



Lpath.D7534.ALA

CONTENTS

APPENDICES

ABOUT THIS BOOK

The *C User's Guide* documents the C compiler of the PRIMOS® operating system and provides all the information necessary to compile, load, execute, and debug C programs under the PRIMOS operating system on 50 Series™ machines.

The PRIMIX™ operating system is a separately priced operating system based on AT&T UNIX® System V and coresident with PRIMOS on the 50 Series. Some of the topics discussed in this book are also relevant to PRIMIX users. These topics include compiler options and interfacing to other Prime® languages. However, the library functions described in this book are different from those supplied with PRIMIX. If you are developing programs under PRIMIX, consult the PRIMIX books listed below under Associated Documents.

Throughout this book, references to PRIMOS C refer to the manner in which the C programming language is implemented under PRIMOS on 50 Series computers.

This guide is not a tutorial on the C programming language. Instead, this book is intended for experienced programmers who have a knowledge of C but who may not be familiar with 50 Series computers. Those users who are unfamiliar with the C programming language should obtain a copy of one of the many commercially available manuals describing the language.

NEW FEATURES OF PRIMOS C

At Release T3.0-23.0, PRIMOS C has added the following new features:

- ANSI standard compliance. This release of the compiler makes the compiler consistent with the ANSI C standard, X3.159-1989, when the -ANSI compiler option is used. New standard-conforming header files and function libraries are also provided. Chapter 8 explains how to compile, link, and run ANSI C programs using PRIMOS C.
- Quadruple precision floating point support. Quadruple precision floating point computations, using the data type **long double**, are now supported in non-ANSI mode with the -QUADFLOATING and -QUADCONSTANTS options. (ANSI mode supports the long double datatype.) See Chapter 2 for information about these options.

- New compiler options. The following new compiler options have been added to PRIMOS C. All are available only in 32IX mode. They are

  -ANSI, -NOANSI
  -CLUSTER, -NO_CLUSTER
  -DISALLOWEXPANSION
  -EXTRACTPROTOTYPES
  -FORCEEXPANSION
  -HARDWAREROUNDING, -NOHARDWAREROUNDING
  -HOLEYSTRUCTURES, -NO_HOLEYSTRUCTURES
  -INTEGEREXCEPTIONS, -NO_INTEGEREXCEPTIONS
  -PACKBYTES, -NO_PACKBYTES
  -PREPROCESSONLY
  -QUADCONSTANTS, -NO_QUADCONSTANTS
  -QUADFLOATING, -NO_QUADFLOATING
  -SEGMENTSPANCHECKING, -NO_SEGMENTSPANCHECKING
  -STRICTCOMPLIANCE, -NOSTRICTCOMPLIANCE

  See Chapter 2 for more information about these options.

- %p format for scanf( ). The %p format specification, previously available only with printf( ), inputs the address of a pointer in the usual Prime format (segment, ring, word number). See the discussion of scanf( ) in Chapter 4.

The following features were first available at Release T2.0-22.1.

- Additional Prime extensions. These are the **#assert, #display,** and **#elif** preprocessor commands, as well as the **defined** unary expression. The handling of **#include** commands now allows the use of preprocessor tokens. Automatic string concatenation is also supported. See Appendix A for information about these commands.

- Two compiler options: **-SPEAK** and **-STANDARDINTRINSICS.** See Chapter 2.

- Two library routines: **assert( )** and **signal( ).** See Chapter 4.

Four additional features were first available at Release T1.3-21.0.

- system( ) library function. Executes its argument as a PRIMOS command line. See Chapter 4.

- Nested #include files. The limit on the levels of nested insert files increased from 9 to 20. See Chapter 7.

- Formal parameters to #define macros. The maximum number of formal parameters to **#define** macros increased from 16 to 128. See Chapter 7.

- STRING.H.INS.CC include file. This file is identical to STRINGS.H.INS.CC. Later revisions of the C compiler will no longer support STRINGS.H.INS.CC. See Chapter 4.

ORGANIZATION OF THIS BOOK

This guide contains eight chapters and seven appendices, as follows:

Chapter 1 — Overview of PRIMOS C. Introduces PRIMOS C, including extensions to the language and 50 Series system resources supporting the C language.

Chapter 2 — Compiling Programs in C. Provides instructions for invoking and using the C compiler. This chapter also contains a description of compiler options.

Chapter 3 — Linking C Programs. Provides information on loading and executing C programs with the BIND and SEG loaders.

Chapter 4 — Using the C Library. Lists and describes the non-ANSI C library functions contained in the CCLIB and C_LIB runtime libraries, and the preprocessor macros defined in the supplied include files.

Chapter 5 — Interfacing to Other Languages. Describes how C can be used to interface to other 50 Series languages.

Chapter 6 — Advanced Topics. Contains information on advanced, system-related topics.

Chapter 7 — Portability Considerations. Describes characteristics of 50 Series machines that you should consider when you port C applications to and from the 50 Series.

Chapter 8 — Using ANSI C. Provides an overview of ANSI C. Explains how to compile, link, and run C programs that conform to the ANSI C standard. Describes the ANSI C library functions contained in C_LIB.

Appendix A — Extensions to the C Language. Describes the extensions to the C language that are available on the 50 Series.

Appendix B — Debugging C Programs. Introduces the Source Level Debugger.

Appendix C — Operator Precedence and Associativity. Lists the C operators and their order of evaluation.

Appendix D — Summary of C Library Functions. Presents a summary of the non-ANSI C library functions by action performed.

Appendix E — C Data Formats. Presents the data formats used by the C language on the 50 Series.

Appendix F — The Prime Extended Character Set. Contains the ASCII reference tables and the mnemonics for character constants and string constants.

Appendix G — Glossary. Explains concepts and conventions basic to 50 Series computers and the PRIMOS operating system.

ASSOCIATED DOCUMENTS

Refer to the guides listed below when using the PRIMOS C compiler. The suggested audience and reading sequence for many of these books are shown in the figure entitled Reading Path for PRIMOS Documentation, opposite the table of contents for this book.

To find out how to order these books, consult the *Guide to Prime User Documents.*

- *Advanced Programmer's Guide: Appendices and Master Index (DOC10066-4LA)*
- *Advanced Programmer's Guide I: BIND and EPFs (DOC10055-2LA)*
- *Advanced Programmer's Guide II: File System (DOC10056-3LA)*
- *Advanced Programmer's Guide III: Command Environment (DOC10057-2LA)*
- *Assembly Language Programmer's Guide (DOC3059-3LA)*
- *CPL User's Guide (DOC4302-3LA)*
- *EMACS Primer (IDR6107)*
- *EMACS Reference Guide (DOC5026-2LA)*
- *Instruction Sets Guide (DOC9474-3LA)*
- *MIDASPLUS User's Guide (DOC9244-2LA)*
- *New User's Guide to EDITOR and RUNOFF (FDR3104-101B)*
- *PRIMOS User's Guide (DOC4130-5LA)*
- *PRIMIX User's Guide (MAN9502-1LA and UPM9502-11A)*
- *PRIMIX Programmer's Guide (MAN9503-2LA)*
- *Programmer's Guide to BIND and EPFs (DOC8691-1LA, UPD8691-11A, and UPD8691-12A)*
- *SEG and LOAD Reference Guide (DOC3524-192L)*
- *Source Level Debugger User's Guide (DOC4033-193L, UPD4033-21A, and UPD4033-22A)*
- *Subroutines Reference I: Using Subroutines (DOC10080-2LA and UPD10080-21A)*
- *Subroutines Reference II: File System (DOC10081-2LA)*
- *Subroutines Reference III: Operating System (DOC10082-2LA)*
- *Subroutines Reference IV: Libraries and I/O (DOC10083-2LA)*
- *Subroutines Reference V: Event Synchronization (DOC10213-1LA and UPD10213-11A)*
- *System Architecture Reference Guide (DOC9473-3LA)*
- *Using PRIMIX on the 50 Series (DOC9709-3LA)*

ACKNOWLEDGEMENTS

The C compiler described in this book was designed and developed by David A. Kosower and Garth Conboy of Pacer Software Inc., a La Jolla, California corporation with technical offices in Westborough, Massachusetts.

PRIME DOCUMENTATION CONVENTIONS

The following conventions may be used throughout this document. The examples in the table illustrate the uses of these conventions.

Convention	*Explanation*	*Example*
UPPERCASE	In command formats, words in uppercase bold indicate the names of commands, options, statements, and keywords. Enter them in either uppercase or lowercase.	**SLIST**
italic	In command formats, words in lowercase bold italic indicate variables for which you must substitute a suitable value. In text and in messages, variables are in non-bold lowercase italic.	**LOGIN *user-id*** Supply a value for *x* between 1 and 10.
Abbreviations in format statements	If a command or option has an abbreviation, the abbreviation is placed immediately below the full form.	**SET_QUOTA** **SQ**
Brackets [ ]	Brackets enclose a list of one or more optional items. Choose none, one, or several of these items.	**LD** [**-BRIEF** / **-SIZE**]
Braces { }	Braces enclose a list of items. Choose one and only one of these items.	**CLOSE** { ***filename*** / **-ALL** }
Braces within brackets [{ }]	Braces within brackets enclose a list of items. Choose either none or only one of these items; do not choose more than one.	**BIND** [{ ***pathname*** / ***options*** }]
Parentheses ( )	In command or statement formats, you must enter parentheses exactly as shown.	**DIM *array* (*row, col*)**
Underscore in examples	In examples, user input is underscored but system prompts and output are not.	OK, RESUME MY_PROG This is the output of MY_PROG.CPL OK,

Convention	*Explanation*	*Example*
Ellipsis ...	An ellipsis indicates that you have the option of entering several items of the same kind on the command line.	**SHUTDN** ***pdev-1*** [***...pdev-n***]
Hyphen -	Wherever a hyphen appears as the first character of an option, it is a required part of that option.	**SPOOL -LIST**
Subscript	A subscript after a number indicates that the number is not in base 10. For example, a subscript 8 is used for octal numbers.	200_8
Key symbol	In examples and text, the name of a key enclosed by a rectangle indicates that you press that key.	Press Return

1

OVERVIEW OF PRIMOS C

This chapter introduces PRIMOS C. The first section describes the implementation of C under PRIMOS and PRIMIX and the characteristics of the C compiler. The second section discusses C language standardization, executable code compatibility within the 50 Series line, and source code compatibility with other C implementations. The last section briefly describes the system resources supporting C language development.

PRIMOS C

The C programming language is a general-purpose language that can be used for a wide variety of applications. Although the C language is widely associated with the UNIX operating systems, it is not dependent on any particular operating system or on any particular hardware architecture.

PRIMOS C is a full implementation of the C programming language. PRIMOS C supports two versions of C:

- The C language as defined in 1978 by Brian W. Kernighan and Dennis M. Ritchie in *The C Programming Language* (Englewood Cliffs, New Jersey: Prentice-Hall, 1978).
- The new ANSI C standard, X3.159-1989.

PRIMOS C provides modern flow control and data structures in addition to a full complement of operators and data types. Other features of PRIMOS C are separate compilation, data sharing, and data initialization.

Prime has also added several extensions to the C language so that it more closely matches the operating environment found in PRIMOS, the proprietary operating system on 50 Series machines. These extensions are listed later in this chapter.

Use of PRIMOS C Under PRIMIX

PRIMIX is a separately-priced operating system based on AT&T UNIX System V and coresident with PRIMOS on the 50 Series. PRIMIX uses the same C compiler as PRIMOS. However, the command syntax and library functions described in this book are different from those available under PRIMIX. PRIMIX programmers may wish to consult this book for information on compiler options, interfacing to other 50 Series languages, and various advanced topics. For information about PRIMIX commands and about the C language libraries supplied with PRIMIX, consult the PRIMIX references listed in the preface to this book.

The C Compiler

The PRIMOS C compiler generates object code in both 32IX and 64V addressing modes, which allows access to 512 megabytes of virtual address space on 50 Series machines. When used in 32IX mode the compiler is fully optimized to take advantage of the new C-oriented architecture changes and enhancements to 32IX mode.

The C compiler is fully compatible with the BIND and SEG loaders, the Symbolic Debuggers (VPSD for 64V mode and IPSD for 32I and 32IX modes), and the Source Level Debugger (DBG).

Application programs written in C can access common data blocks. The data blocks can be defined either by C routines or by routines written in other languages. C can also access data that span segment boundaries, with some restrictions (see Chapter 7). Subroutines written in other languages, as well as PRIMOS system subroutines, can call or be called by C programs and subroutines with full argument transfer where data types permit.

STANDARDIZATION

PRIMOS C provides compile-time and runtime support for the ANSI C standard, X3.159-1989, which is also documented in the second edition of Kernighan and Ritchie's *The C Programming Language*. Chapter 8 provides information about compiling, linking, and running standard-conforming C programs. PRIMOS C continues to support the 1978 version of the C language. Use the first edition of *The C Programming Language* as a reference guide in developing non-ANSI C programs.

At the source level, non-ANSI PRIMOS C is reasonably compatible with the C compilers running under the newest versions of the UNIX operating systems from AT&T and from the University of California at Berkeley. In addition, the command line option -COMPATIBILITY causes the compiler to accept code written for the older AT&T UNIX Version 6. Other portability issues are discussed in Chapter 7.

Runtime Libraries

PRIMOS C provides runtime libraries and header files that fully comply with ANSI C requirements, as well as libraries and header files that support the 1978 C language. The non-ANSI libraries support a subset of the AT&T UNIX System V subroutines. These runtime libraries include

- File I/O functions (for example, open( ), read( ), fopen( ), fprint( ), fscanf( ))
- String and character manipulation functions (for example, isalpha( ), isdigit( ))
- Mathematical functions (for example, abs( ), sqrt( ), tan( ))
- System functions (for example, abort( ), setjmp( ), longjmp( ), sleep( ))

Compatibility

Prime uses a common operating system architecture on all 50 Series machines. Therefore, C application programs compiled in 64V mode on one system can, without any modification, be executed on another 50 Series system. Programs compiled in 32IX mode, however, do not run on older machines.

50 Series Extensions to the C Language

The PRIMOS C programming language contains a number of extensions to the 1978 C language, all of which are part of the ANSI standard unless otherwise stated. These extensions are listed below.

- **enum** data type
- **void** data type
- **long double** data type, supporting quadruple precision floating point numbers
- **fortran** storage class
- Unary plus (+) operator
- Identifier names up to 32 characters in length
- Preprocessor commands **#assert**, **#display**, **#list**, **#nolist**, and **#endincl** (these commands are not in the ANSI standard)
- Preprocessor command **#elif**
- Preprocessor operator **defined**
- Support for preprocessor tokens with the **#include** command
- Automatic string concatenation

A description of each extension is provided in Appendix A, Extensions to the C Language.

SYSTEM RESOURCES SUPPORTING C

Application programs written in PRIMOS C can access a wide range of libraries, system utilities, and file management resources. The sections below describe a few of the major resources accessible to C application programs.

Libraries

Chapter 4, Using the C Library, contains a description of the non-ANSI C library functions and Chapter 8, Using ANSI C, includes a description of the ANSI C library functions.

BIND Linker and SEG Loader

BIND and SEG are the 50 Series linking and loading utilities for both 64V-mode and 32IX-mode programs. BIND and SEG combine separately compiled program modules, subroutines, and libraries into an executable program. A single C source file can be a maximum of 128K bytes. All memory management, linking, and the like are handled by these utilities. Various types of load maps may be obtained. Chapter 3, Linking C Programs, demonstrates the use of BIND and SEG to link C programs.

Editors

The PRIMOS editor (ED) is a line-oriented text editor that enables you to enter and modify source code and text files. A complete description of ED is in the *New User's Guide to EDITOR and RUNOFF.* Prime also offers the screen editor EMACS as a separately priced product. The EMACS screen editor is described in the *EMACS Primer* and the *EMACS Reference Guide.*

Source Level Debugger

The Prime Source Level Debugger (DBG) is an interactive debugger that enables you to debug C code interactively. Appendix B contains some suggestions for using DBG with C programs. The *Source Level Debugger User's Guide* details the operation of DBG.

Multiple Index Data Access System

Prime Multiple Index Data Access System (MIDASPLUS™) is a software subsystem of utilities and subroutines for creating and maintaining keyed-index/direct-access files. MIDASPLUS provides the C programmer with a keyed-index multilevel file structure. All housekeeping functions on the index and data subfiles are performed by MIDASPLUS subroutines called from C programs.

Prime MIDASPLUS files created by programs written in one language may be accessed and manipulated by programs written in other languages, thus ensuring compatibility. *The MIDASPLUS User's Guide* contains a complete description of MIDASPLUS.

Chapter 5, Interfacing to Other Languages, provides information on calling MIDASPLUS from PRIMOS C programs.

Language Interfaces

Object modules generated by the C compiler are capable of calling and being called by object modules generated by the COBOL 74, FORTRAN IV, FORTRAN 77, Pascal, or PL/I compilers. This is possible because all 50 Series high-level languages are similar at the object code level, and all use similar calling conventions. However, certain restrictions must be adhered to.

- Data types must be compatible when variables are passed as parameters.
- All modules must be compiled in 64V, 32IX, or 32I mode.
- The C compiler must be informed that interlanguage calling is taking place by use of the **fortran** keyword or by command line options.

Application programs written in C can also call Prime Macro Assembler (PMA) routines and vice versa. For further information on PMA routines, see the *Assembly Language Programmer's Guide.*

Chapter 5, Interfacing to Other Languages, provides guidelines and examples for interfacing C to other 50 Series languages.

2

COMPILING PROGRAMS IN C

This chapter explains how to compile C programs under PRIMOS on 50 Series systems. The first section of this chapter describes the include files that are provided with the C compiler and libraries. The second section describes the PRIMOS search rules facility and explains how to specify directories to be searched for include files. The last section describes the use of the C compiler, the messages produced during compilation, and the command line options available.

The PRIMOS C compiler generates binary code in 64V and 32IX segmented addressing modes in two passes and three passes, respectively. The C compiler accepts a source program meeting the requirements specified in this guide. The C compiler also accepts both .C and .CC as suffixes for source files.

The C compiler, like the other high-level language compilers on the 50 Series, can generate an object file, a source file listing, error and statistical data, and other helpful information regarding the compilation of C application programs.

STANDARD INCLUDE FILES

The standard C include files are located in the directory SYSCOM. These files have names that end in .INS.CC. The C compiler does not recognize files in SYSCOM that lack the .INS.CC suffix. The .INS.CC suffix is optional in other directories. If you copy a header file from SYSCOM to another directory, you may remove the suffix or not, as you wish.

Table 4-1 in Chapter 4 and Table 8-1 in Chapter 8 list the C include files in SYSCOM that are provided for the non-ANSI and ANSI C libraries, respectively. The files AKEYS.INS.CC, ERRD.INS.CC, and KEYS.INS.CC are installed as part of PRIMOS. They are documented in Volume II of the *Subroutines Reference Guide*. Other products may also provide C include files. For example, if MIDASPLUS is installed on your system, SYSCOM contains the file PARM.K.INS.CC.

INCLUDE FILES AND THE SEARCH RULES FACILITY

The PRIMOS search rules facility enables you to establish an INCLUDE$ search list. An INCLUDE$ search list is a list of directories that are to be searched for an include file whenever a **#include** directive is processed by the compiler. Although there are several kinds of search lists, this section explains only the INCLUDE$ search list. For complete information about the PRIMOS search rules facility, see the *Advanced Programmer's Guide, Volume II.*

In PRIMOS C, you can specify directories to be searched for include files in a number of different ways. When the C compiler encounters a **#include** directive, it searches for the file in the following manner.

1. If the pathname is delimited by angle brackets (<. . .>), the compiler goes to step 2.

   If the pathname is delimited by double quotes (". . ."), the compiler proceeds as follows. If the pathname is a simple filename, the compiler searches the current directory. If the pathname is an absolute pathname (that is, if it begins with a disk partition) the compiler searches that disk partition for the specified path. If the pathname is a full pathname (that is, if it begins with a top-level directory) the compiler searches all the disk partitions for the specified path. If it still cannot find the file, the compiler goes to step 2.

2. The compiler searches the directories specified in command line -INCLUDE options, if any. If it cannot find the file in those directories, the compiler goes to step 3.

3. PRIMOS searches the INCLUDE$ search list and supplies pathnames to the compiler. The compiler then searches these directories. If the compiler cannot find the file in those directories, it goes to step 4.

4. The compiler searches the top-level directory SYSCOM. If it cannot find the file, the compiler reports an error.

Note

The standard C include files in SYSCOM contain the suffix .INS.CC. Files in SYSCOM that lack the .INS.CC suffix are not recognized by the C compiler. The .INS.CC suffix is optional in other directories. If you copy a header file from SYSCOM to another directory, you may remove the suffix or not, as you wish.

In directories other than SYSCOM, the statement

```
#include "name"
```

causes the compiler to search for the following files, in this order:

name.INS.CC
name.INS.C
name

Establishing Search Rules: To establish search rules for include files, perform the following steps:

1. Create a template file called

   [*yourchoice.*]INCLUDE$.SR

   This file should contain a list of the pathnames of the directories that contain your include files. List the directories in the order that you want them searched. For example, you might create a file called MY.INCLUDE$.SR that contains the following directory names:

   ```
   <SYS1>MASTER_DIR>INSERT_FILES
   <SYS2>ME
   ```

2. Activate the template file by using the SET_SEARCH_RULES (SSR) command. For example, if your file is named MY.INCLUDE$.SR, type

   ```
   OK, SSR MY.INCLUDE$
   ```

   This command sets your INCLUDE$ search list. This search list may contain system search rules and administrator search rules in addition to the rules you specified in MY.INCLUDE$.SR.

When you give the SSR command shown in step 2, PRIMOS copies the contents of MY.INCLUDE$.SR into your INCLUDE$ search list. If you have no special system or administrator search rules, your INCLUDE$ search list appears as follows when you type the LIST_SEARCH_RULES (LSR) command:

```
List: INCLUDE$
Pathname of template: <MYSYS>ME>CPROGS>MY.INCLUDE$.SR

    [home_dir]
    <SYS1>MASTER_DIR>INSERT_FILES
    <SYS2>ME
```

[home_dir], your current attach point, is the system default. It is always the first directory searched, unless you remove it from the list or change the order of evaluation by using the -NO_SYSTEM option of the SSR command. Additional search rules, established as systemwide defaults by your System Administrator, may also appear at the beginning of your INCLUDE$ search list. The above search rules initiate the search in [home_dir], then search <SYS1>MASTER_DIR>INSERT_FILES, and finally search <SYS2>ME.

The SET_SEARCH_RULES and LIST_SEARCH_RULES commands are described in the *PRIMOS Commands Reference Guide*. For more information about establishing search rules, see the *Advanced Programmer's Guide, Volume II*.

Using Search Rules: The C compiler searches the contents of the directories according to the pattern described at the beginning of this section.

Using [referencing_dir]: The *Advanced Programmer's Guide, Volume II* describes several expressions that you can use in your list of search rules. One of these, [referencing_dir], has a special meaning for INCLUDE$ search lists. [referencing_dir] is less useful in C than in other Prime languages because include files can be specified in so many ways in C.

Like [home_dir], [referencing_dir] is a variable that PRIMOS replaces with a directory pathname. [referencing_dir] always evaluates to the pathname of the directory from which the request for an include file is made. Thus, if a **#include** directive is located in a source file, [referencing_dir] evaluates to the pathname of the directory that contains the source file.

USING THE C COMPILER

Invoke the C compiler from PRIMOS command level with the command

CC *sourcefile* [*-option 1*] [*-option 2*] . . . [*-option n*]

where CC invokes the C compiler, *sourcefile* denotes the pathname of the C source program to be compiled, and *-option* denotes an option controlling the compiler functions. All compiler options begin with a hyphen.

For example, the command

```
OK, CC TEST_PROGRAM -LISTING -STATISTICS
```

compiles a program named TEST_PROGRAM.CC or TEST_PROGRAM.C with the -LISTING and -STATISTICS options.

The compiler options are listed in Table 2-1, C Command Line Compiler Options, at the end of this section. Each option is described in detail later in this chapter.

Compile-time Error Messages

The C compiler automatically displays an error message at the terminal for each error it encounters during the compilation procedure. The C compiler also records compilation errors in a source listing if one is specified. The format for C compiler error messages is

```
Error# n on source line = y has severity z:
```

descriptive-text

where *n* indicates the cumulative error count, *y* shows the source line on which the error was encountered, and *z* states the severity of the error, as follows:

Verbose Error, warning, or information that is not normally displayed (see description of -VERBOSE option on page 2-35)

Warning Error encountered that may later result in an unsuccessful execution

Fixable Recoverable syntax error that does not prevent code generation

Error Uncorrectable error that prevents code generation

Fatal Error that prevents further compilation

An example of a compile-time error message follows:

```
OK, SLIST BADPROGRAM.C
 main( )
 {
     printf("hello, world\n")
 }

OK, CC BADPROGRAM -32IX
    [CI Rev. T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]

Error# 1 on source line = 4 has severity Error:
     }

   A syntax error was found; a "}" was found where another
   token was expected.  Error recovery was invoked.

 01 Error and 00 Warnings detected in 4 source lines.
```

After any compilation is complete, the C compiler displays an end of compilation message:

nn Error and *xx* Warnings detected in *y* source lines.

where *nn* indicates the total number of compilation errors. (00 indicates an error-free compilation.)

Compiler Options

The C compiler provides a variety of compiler options that enable you to perform many tasks during program compilation. These tasks include

- Generating a binary file
- Defining the properties of generated object code
- Generating a source listing and specifying its contents
- Generating compiler error and statistical information
- Controlling optimization

Many of the compiler options come in pairs. That is, for each option there is an option having the opposite effect. One option of each pair is always the default. Compiler options can be specified in any order.

The PRIMOS command line is not case-sensitive. Commands, compiler options, and arguments can be specified in uppercase or lowercase.

In the summary of compiler options in Table 2-1, the default options are indicated by an asterisk (*). A few of the options require an argument specification in addition to the option specification. The argument specification is not preceded by a hyphen. The short

form of each option is underscored. The second column of the table indicates the valid addressing mode(s) for each option; for example, an option designated 64V can be used only in conjunction with the -64V option. Detailed explanations of the options follow the table.

TABLE 2-1. C Command Line Compiler Options

Option	*Modes*	*Operation Performed*
-32IX	---	Generates 32IX-mode object code.
-64V* **-6**	---	Generates 64V-mode object code.
-ANSI **-AN**	32IX, 64V	Examines a source program for adherence to the ANSI C standard.
-BIG	32IX, 64V	Assumes external arrays and pointed-to objects span segment boundaries.
-BINARY* **-B**	32IX, 64V	Generates binary (object) file. This option may take an argument.
-BIT8* **-BIT**	32IX, 64V	Sets bit 8 in character and string constants.
-CHECKOUT **-CH**	32IX, 64V	Executes only the compiler's first pass.
-CIX **-CI**	64V	Enables 64V to call 32IX code. This option takes an argument.
-CLUSTER **-CLU**	32IX	Causes optimization and code generation for entire source file.
-COMPATIBILITY **-COMPA**	32IX, 64V	Compiles Version 6 source code as well as Version 7, System III and System V source code.
-COPY*	64V	Passes parameters by value.
-DEBUG **-DEB**	32IX, 64V	Generates information for full Source Level Debugger (DBG) support.
-DEFINE **-DEF**	32IX, 64V	Defines a specified name to be a specified value. This option takes two arguments.
-DISALLOWEXPANSION **-DIS**	32IX	Causes named routine not to be expanded inline. This option takes an argument.
-DOUBLEFLOATING* **-DOU**	32IX, 64V	Performs all floating-point math in double precision.
-ERRTTY* **-ERRT**	32IX, 64V	Displays error messages on user's terminal.
-EXPLIST **-EXP**	32IX, 64V	Generates expanded (assembly) listing.

Asterisks (*) indicate defaults.

TABLE 2-1. C Command Line Compiler Options (continued)

Option	*Modes*	*Operation Performed*
-EXTRACTPROTOTYPES **-EXTRAC**	32IX	Creates header file with ANSI-style prototype declarations for all functions in source file. This option may take a pathname argument.
-FORCEEXPANSION **-FORCEE**	32IX	Forces named routine to be expanded inline. This option takes an argument.
-FRN	32IX, 64V	Generates Floating Round Number (FRN) instruction before FST instruction.
-HARDWAREROUNDING **-HARD**	32IX	Turns on hardware rounding.
-HIGHENDPROCESSORS **-HIGH**	64V	Generates code optimized for the 4000, 6000, and 9000 series processors.
-HOLEYSTRUCTURES **-HOLE**	32IX	Causes all non-bit-field structure members 32 bits or larger to be aligned on 32-bit boundaries.
-IGNOREREGISTER **-IG**	32IX	Ignores the **register** keyword.
-INCLUDE **-INC**	32IX, 64V	Specifies include search pathnames. This option takes an argument.
-INPUT **-I**	32IX, 64V	Designates the source file to be compiled. A pathname must be specified. This option is obsolete. Its use is not recommended.
-INTEGEREXCEPTIONS **-INTE**	32IX	Causes runtime errors to be generated for integer overflow, underflow, and divide by zero.
-INTLONG* **-INTL**	32IX, 64V	Generates 4-byte integers.
-INTRINSIC **-INTR**	32IX, 64V	Causes compiler to generate inline code for one of several library functions. This option takes one or two arguments.
-INTSHORT **-INTS**	32IX, 64V	Generates 2-byte integers. Use of this option is not recommended.
-LBSTRING* **-LBS**	32IX, 64V	Places string constants in the linkage area.
-LISTING **-L**	32IX, 64V	Generates listing file. This option may take an argument.

Asterisks (*) indicate defaults.

TABLE 2-1. C Command Line Compiler Options (continued)

Option	*Modes*	*Operation Performed*
-LOWENDPROCESSORS* **-LOW**	64V	Generates code optimized for machines other than the 4000, 6000, and 9000 series processors.
-NEWFORTRAN* **-NEWF**	64V	Uses the new interlanguage interface.
-NOANSI* **-NOAN**	32IX, 64V	Does not examine source program for adherence to the ANSI C standard.
-NOBIG*	32IX, 64V	Assumes external arrays and pointed-to objects do not span segment boundaries. This option may cause use of 16-bit indexing.
-NOBIT8 **-NOBIT**	32IX, 64V	Does not set bit 8 in character and string constants. Use of this option is not recommended.
-NOCHECKOUT **-NOCH**	32IX, 64V	Does not execute only the compiler's first pass.
-NOCLUSTER* **-NOCLU**	32IX	Does not cause optimization and code generation for entire source file.
-NOCOMPATIBILITY* **-NOCOMPA**	32IX, 64V	Does not accept Version 6 source code.
-NOCOPY	64V	Passes parameters by reference.
-NODEBUG* **-NODEB**	32IX, 64V	Does not generate information for full Source Level Debugger (DBG) support.
-NOERRTTY **-NOERRT**	32IX, 64V	Does not display error messages on user's terminal.
-NOEXPLIST* **-NOEXP**	32IX, 64V	Does not generate an expanded listing file.
-NOFRN* **-NOFR**	32IX, 64V	Does not generate FRN instructions.
-NOHARDWAREROUNDING* **-NOHARD**	32IX	Does not turn on hardware rounding.
-NOHOLEYSTRUCTURES* **-NOHOLE**	32IX	Does not cause all non-bit-field structure members 32 bits or larger to be aligned on 32-bit boundaries.
-NOIGNOREREGISTER **-NOIG**	32IX	Respects the **register** keyword.

Asterisks (*) indicate defaults.

TABLE 2-1. C Command Line Compiler Options (continued)

Option	*Modes*	*Operation Performed*
-NOINTEGEREXCEPTIONS* **-NOINTE**	32IX	Does not cause runtime errors to be generated for integer overflow, underflow, and divide by zero.
-NOONUNIT **-NOON**	32IX, 64V	Lets PRIMOS report the occurrence of fatal compiler errors.
-NOOPTIMIZE* **-NOOPT**	32IX, 64V	Performs no object code optimization.
-NOPACKBYTES* **-NOPACK**	32IX	Does not pack adjacent single-byte entities in structures and unions.
-NOOPTSTATISTICS* **-NOOPTS**	32IX	Does not print optimization statistics.
-NOPOP*	32IX, 64V	Removes old constant macro definition.
-NOQUADCONSTANTS* **-NOQUADC**	32IX	Does not support quad-precision constants.
-NOQUADFLOATING* **-NOQUADF**	32IX	Does not support quad-precision variables.
-NOSAFEPOINTERS* **-NOSAFE**	32IX, 64V	Does not always retain byte offset bit.
-NOSEGMENTSPANCHECKING* **-NOSEG**	32IX	Does not cause two block-memory functions to produce correct code for segment-spanning arguments.
-NOSILENT* **-NOSIL**	32IX, 64V	Displays warning messages on the user's terminal.
-NOSTATISTICS* **-NOSTAT**	32IX	Does not generate compiler statistics.
-NO_STORE_OWNER_FIELD* **-NSOF**	32IX, 64V	Suppresses the generation of code that performs a store owner field operation.
-NOSTRICTCOMPLIANCE* **-NOSTRIC**	32IX	Does not detect certain violations of the ANSI standard.
-NOSYSOPTIONS **-NOSYS**	32IX	Does not look for the system options file.
-NOVERBOSE* **-NOVERB**	32IX, 64V	Does not display verbose messages on the user's terminal.
-OLDFORTRAN **-OL**	64V	Uses the old interlanguage interface.

Asterisks (*) indicate defaults.

TABLE 2-1. C Command Line Compiler Options (continued)

Option	*Modes*	*Operation Performed*
-OPTIMIZE **-OPT**	32IX, 64V	Performs object code optimization. This option may take an argument.
-OPTIONSFILE **-OPTIO**	32IX, 64V	Reads command line options from a specified file. This option takes an argument.
-OPTSTATISTICS **-OPTS**	32IX	Prints optimization statistics.
-PACKBYTES **-PACK**	32IX	Packs adjacent single-byte entities in structures and unions.
-PARTIALDEBUG **-PAR**	32IX	Generates debugger symbol information only for variables that are referenced.
-PBSTRING **-PBS**	32IX, 64V	Places string constants in the procedure area.
-POP	32IX, 64V	Pops old macro definitions.
-PREPROCESSONLY **-PRE**	32IX	Generates file that contains source with macros expanded. This option may take a pathname argument.
-PRODUCTION **-PROD**	32IX, 64V	Generates information for partial DBG support.
-QUADCONSTANTS **-QUADC**	32IX	Supports quad-precision constants when the program is compiled without the -ANSI option.
-QUADFLOATING **-QUADF**	32IX	Supports quad-precision variables when the program is compiled without the -ANSI option.
-SAFEPOINTERS **-SAFE**	32IX	Always retains byte offset bit.
-SEGMENTSPANCHECKING **-SEG**	32IX	Causes two block-memory functions to produce correct code for segment-spanning arguments.
-SHORTCALL **-SHORTC**	32IX, 64V	Enables the compiler to generate shortcalls. This option takes an argument.
-SINGLEFLOATING **-SIN**	32IX, 64V	Allows single precision floating-point math.
-SILENT **-SIL**	32IX, 64V	Does not display warning messages on the user's terminal.

Asterisks (*) indicate defaults.

TABLE 2-1. C Command Line Compiler Options (continued)

Option	*Modes*	*Operation Performed*
-SOURCE	32IX, 64V	Identical to -INPUT, this option is obsolete. Its use is not recommended.
-SPEAK	32IX	Enables compile-time progress messages after each stage of the compiler.
-STANDARDINTRINSICS **-STAN**	32IX	Causes compiler to generate inline code for all the options supported by the -INTRINSIC option.
-STATISTICS **-STAT**	32IX, 64V	Generates compiler statistics on the user's terminal.
-STORE_OWNER_FIELD **-SOF**	32IX, 64V	Generates code within a program to store the identity of a called procedure in a known place.
-STRICTCOMPLIANCE **-STRIC**	32IX	Detects certain violations of the ANSI standard, such as PRIMOS-specific extensions, when used in conjunction with -ANSI.
-SYSOPTIONS* **-SYS**	32IX	Looks for the system options file.
-UNDEFINE **-UNDEF**	32IX	Removes an initial definition. This option takes an argument.
-VALUEONLY **-VALUE**	32IX	Identifies a routine that has no side effects. This option takes an argument.
-VERBOSE **-VERB**	32IX, 64V	Displays verbose messages on the user's terminal.
-XREF	32IX, 64V	Generates a full cross-reference listing.
-XREFS	32IX, 64V	Generates a partial cross-reference listing.

Asterisks (*) indicate defaults.

Command Line Compiler Options

This section provides detailed descriptions of C compiler options. Option pairs, that is, options that have opposite effects, are listed together. An asterisk (*) indicates that the option before it is the default.

▶ -32IX

Short form: -32IX

Generates object code in 32IX mode. This option is available on all newer 50 Series processors (that is, all processors with four-digit names except the 2250™ processor). Code produced with the -32IX command line option runs between 1.5 and 4.0 times faster than code produced with the -64V option. When you use the -32IX option, the compiler prints the following banner:

```
[CI Rev. T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]
```

You may not put this option in an options file (see the -OPTIONSFILE option). You must specify it on the command line.

▶ -64V*

Short form: -6

Generates object code in 64V mode. Code produced with this option runs on any 50 Series machine newer than and including the 400™ processor.

Note

Use of the -64V option is not recommended. This option generates code that results in poor performance. If your machine supports 32IX mode, use the -32IX option.

When you compile a C program in 64V mode, the compiler prints the following banner:

```
[CC Rev. T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]
```

You may not put this option in an options file (see the -OPTIONSFILE option). You must specify it on the command line.

▶ -ANSI / -NOANSI*

Short forms: -AN / -NOAN

This option is intended for use with -32IX only. Examines a C source program for adherence to the ANSI C standard. It can be used with -64V, but it only checks for uses of the fortran keyword. For example:

```
main()
{
    fortran mkon$p();
}
OK, CC TEST -NEWFORTRAN -ANSI
[CC Rev. 23.0-T3.0 Copyright (c) 1990, Prime Computer, Inc.]

Error# 1; Error type = 269; Source line = 3;
          Error severity = Warning
    fortran mkon$p();
          ^
    The use of the "fortran" keyword may not be portable construct.

00 Errors and 01 Warning detected in 4 source lines.
```

For information about writing, compiling, and linking ANSI C programs, see Chapter 8.

-ANSI
Instructs the C compiler to perform extra checking for violations of the ANSI C standard.

-NOANSI
Instructs the C compiler to disregard violations of the ANSI C standard. Instead, the compiler checks for violations of the C language as defined in the first (1978) edition of *The C Programming Language*, by Kernighan and Ritchie.

You may not put either of these options in an options file (see the -OPTIONSFILE option). You must specify them on the command line.

▶ -BIG / -NOBIG*

Short forms: -BIG / -NOBIG

Determines the type of code generated for array and pointer references.

-BIG
Assumes that unless they are declared small, all external arrays, structures, and pointed-to objects span segment boundaries and exceed 128K bytes.

-NOBIG
Assumes all external arrays and objects referenced by pointers that are declared with a size smaller than 128K bytes do not span segment boundaries, thus allowing 16-bit array-addressing code to be generated. The object code is faster, but introduces the risk of incorrect array referencing code being generated for arrays that exceed one segment. This option may cause 16-bit pointer increment and decrement code to be generated.

▶ -BINARY [*argument*]

Short form: -B

Generates and names a binary file. This option, however, does not define the properties of the binary file. The argument specification for this option can be one of the following:

Argument	*Meaning*
pathname	Indicates the pathname to which the object code is written.
YES*	Instructs the C compiler to create a binary file named PROGRAM.BIN for all programs compiled in 32IX mode and for 64V-mode programs with a .CC or .C suffix. If you compile in 64V mode and the source filename does not include a .CC or .C suffix, the C compiler generates an object file named B_PROGRAM. This argument is the default.
NO	Does not generate a binary file.

You may not put this option in an options file (see the -OPTIONSFILE option). You must specify it on the command line.

▶ -BIT8* / -NOBIT8

Short forms: -BIT / -NOBIT

Controls the setting of the most significant bit in each byte of string and character constants. This most significant bit is always ON for Prime ASCII, the basic character set in the Prime Extended Character Set (Prime ECS). (For information about Prime ECS, see Appendix F.) The purpose of disabling this bit by using -NOBIT8 is to provide compatibility with algorithms that expect the integer value of character constants to be from 0 through 127 decimal.

Note

Areas of programs that need to use -NOBIT8 must be localized because character constants generated with -NOBIT8 are not supported by any libraries or I/O subsystems.

-BIT8

Produces object code that conforms to Prime ECS. All nonoctal character and string constants have their eighth bit (most significant bit) set.

-NOBIT8

All nonoctal character and string constants have their eighth bit masked off, so that they do not conform to Prime ECS. These constants are not recognized as members of the standard Prime character set. Severe runtime problems occur if you attempt to use these character constants outside of the routine compiled with this option.

▶ -CHECKOUT / -NOCHECKOUT*

Short forms: -CH / -NOCH

Runs only the compiler's first pass to increase compilation speed for detection of initial program errors.

-CHECKOUT
Runs only the first pass of the compiler. No code generation or expanded listings can be produced if this option is specified.

-NOCHECKOUT
Runs all passes of the compiler.

▶ -CIX *routinename*

Short form: -CI

This option is valid in 64V mode only.

Causes the compiler to assume that the external routine *routinename* was compiled in 32IX mode and to generate the correct calling sequence. No Argument Pointers (APs) are used, all pointers are shortened to two 16-bit halfwords, and the address of the argument list is placed in the XB register before the PCL procedure call. See Chapter 5, Interfacing to Other Languages, for more information.

▶ -CLUSTER / -NOCLUSTER*

Short forms: -CLU / -NOCLU

These options are valid in 32IX mode only.

Controls the order of compilation for a source file with multiple routines. -CLUSTER is useful only in conjunction with -OPTIMIZE.

-CLUSTER
Causes the compiler to accumulate intermediate representation for an entire source file before performing optimization and code generation. -CLUSTER behaves differently depending on the optimization level specified:

- With -NOOPTIMIZE (the default), -CLUSTER simply increases the amount of memory required to compile a given source file.
- With -OPTIMIZE 1, -OPTIMIZE 2, or -OPTIMIZE 3, -CLUSTER causes the compiler to build a calling tree (call graph) and to process inline expansion bottom-up. As optimization level increases, the complexity of routines that are considered candidates for inline expansion also increases. If a nonstatic routine is expanded inline, the compiler also generates code for an external version of the routine so that it can be reached from routines in other source files.

-NOCLUSTER
Causes the compiler to accumulate intermediate representation and perform optimization and code generation for each function before moving on to the next function in the source file.

▶ -COMPATIBILITY / -NOCOMPATIBILITY*

Short forms: -COMPA / -NOCOMPA

Controls the handling of AT&T UNIX Version 6 C source incompatibilities.

-COMPATIBILITY
Compiles AT&T UNIX Version 6 C source code as well as the more recent AT&T UNIX Version 7, System III and System V C source code. This option also permits the use of Version 6 syntax (for example, =op and old initialization style).

-NOCOMPATIBILITY
Interprets all occurrences of Version 6 C syntax as errors.

▶ -COPY* / -NOCOPY

Short forms: -COPY / -NOCOPY

These options are valid in 64V mode only.

Controls the passing of arguments from one function to another function either by reference or by value. See Chapter 5, Interfacing to Other Languages, for more information.

-COPY
Passes arguments from one function to another function by value.

-NOCOPY
Passes arguments from one function to another function by reference.

▶ -DEBUG / -NODEBUG*

Short forms: -DEB / -NODEB

Enables full Source Level Debugger (DBG) support.

-DEBUG
Enables DBG support.

-NODEBUG
Does not enable DBG support.

Refer to Appendix B, Debugging C Programs, for more information on DBG.

▶ -DEFINE *name value*

Short form: -DEF

Simulates an initial **#define** C preprocessor command. That is, compiling the source file MYPROGRAM.CC with the command line

```
OK, CC MYPROGRAM -DEFINE MYCONST 20
```

has the same effect as specifying

```
#define MYCONST 20
```

as the first directive in the source file MYPROGRAM.CC. If no *value* is specified following *name*, *value* is assumed to be 1. If *value* contains spaces, enclose it in single quotation marks. For example,

```
OK, CC MYPROGRAM -DEFINE SUM '5 + 2'
```

You may not use the -DEFINE option to define macros with arguments on the command line. You may specify as many -DEFINE options as required on the command line.

▶ -DISALLOWEXPANSION

See -FORCEEXPANSION.

▶ -DOUBLEFLOATING* / -SINGLEFLOATING

Short forms: -DOU / -SIN

Controls the precision of floating-point math operations.

-DOUBLEFLOATING
Performs all floating-point math in double precision.

-SINGLEFLOATING
Allows the calculation to be performed in single precision if both arguments to a floating-point operation are single precision.

▶ -ERRTTY* / -NOERRTTY

Short forms: -ERRT / -NOERRT

Controls the display of compiler error messages on a user's terminal.

-ERRTTY
Displays compiler error messages on the user's terminal.

-NOERRTTY
Does not display compiler error messages on the user's terminal.

See the description of compiler messages earlier in this chapter. The -NOERRTTY option does not affect the contents of a file produced with the -LISTING option. It affects only error messages printed to the terminal.

▶ -EXPLIST / -NOEXPLIST*

Short forms: -EXP / -NOEXP

Controls the insertion of pseudo-assembly code into the source listing file.

-EXPLIST
Causes pseudo-Prime Macro Assembler (PMA) code to be written to the listing file for each C language statement in the source file.

-NOEXPLIST
Does not insert pseudo-PMA statements into the source listing file.

▶ -EXTRACTPROTOTYPES [*pathname*]

Short form: -EXTRAC

This option is valid in 32IX mode only.

Causes the compiler to create a file that contains ANSI-style prototype declarations for all functions defined in the source file.

The *pathname* argument specifies the name of the file to be created by -EXTRACTPROTOTYPES. If the pathname is omitted, the default name of the file is *name*.H, where *name* is the root name of the source file.

For more information about -EXTRACTPROTOTYPES, see Chapter 8.

▶ -FORCEEXPANSION *routine* / -DISALLOWEXPANSION *routine*

Short forms: -FORCEE / -DIS

These options are valid in 32IX mode only.

Overrides the compiler's default algorithm for determining which routines are expanded inline. Both -FORCEEXPANSION and -DISALLOWEXPANSION work only when used in conjunction with -CLUSTER. If you specify either -FORCEEXPANSION or -DISALLOWEXPANSION without specifying -CLUSTER, the compiler ignores it.

-FORCEEXPANSION
Orders the compiler to expand the named routine inline.

-DISALLOWEXPANSION
Forbids the compiler to expand the named routine inline.

▶ -FRN / -NOFRN*

Short forms: -FR / -NOFR

Controls generation of the floating-point round instruction. The -FRN option usually improves the accuracy of calculations involving single precision floating-point numbers. Such numbers are type **float** in C.

For programs compiled in 32IX mode on newer 50 Series systems (that is, all systems with four-digit names except the 2250), -FRN has been superseded by the -HARDWAREROUNDING option (-HARD). If you have one of these systems, use -HARDWAREROUNDING instead of -FRN. (See the discussion of -HARDWAREROUNDING.)

-FRN
Causes all single precision numbers to be rounded each time they are moved from a register to main storage. -FRN adds the Prime Macro Assembler instruction FRN to the generated code at every single precision store. For information about how this instruction works, see the *System Architecture Guide* and the *Instruction Sets Guide.* The rounding method that is used ordinarily reduces loss of accuracy in the low-order bits when many calculations are performed on the same number.

-NOFRN
Does not generate FRN instructions.

Occasionally, a program may give less accurate results with -FRN than without it. Use -FRN only if you are familiar enough with the FRN instruction to know how it will affect the operations in your program.

-FRN does not affect double precision real numbers (**double**) or quadruple precision floating-point numbers (**long double**). Often the best way to gain increased accuracy is to use **double** or **long double** numbers rather than **float.** -FRN causes a slight increase in execution time, and should therefore be used only when maximum accuracy for single precision numbers is a major consideration.

▶ -HARDWAREROUNDING / -NOHARDWAREROUNDING*

Short forms: -HARD / -NOHARD

This option is valid in 32IX mode only.

-HARDWAREROUNDING enables hardware rounding for floating-point operations. This option usually improves the accuracy of calculations involving both single precision and double precision real numbers (float, double). -HARDWAREROUNDING has an effect only with the newer 50 Series systems (that is, all systems with four-digit names except the 2250). On these systems, use -HARDWAREROUNDING instead of -FRN for single precision rounding. Do not use -HARDWAREROUNDING and -FRN together; such use is redundant and degrades runtime performance.

-HARDWAREROUNDING
Enables hardware rounding for the following floating-point operations: add, subtract, multiply, divide, store, and compare. It ordinarily provides greater accuracy than -FRN, which causes rounding to occur only when a number is stored. For information about how hardware rounding is performed, see the *System Architecture Guide* and the *Instruction Sets Guide.* The rounding method that is used ordinarily reduces loss of accuracy in the low-order bits when many calculations are performed on the same number.

-NOHARDWAREROUNDING
Causes no rounding to be performed.

Occasionally, a program may give less accurate results with -HARDWAREROUNDING than without it. Use -HARDWAREROUNDING only if you are familiar enough with hardware rounding to know how it will affect the operations in your program.

-HARDWAREROUNDING does not affect quadruple precision floating-point numbers (**long double**). Often the best way to gain increased accuracy is to use **double** numbers rather than **float**, or **long double** numbers rather than **double**. -HARDWAREROUNDING causes a slight increase in execution time, and should therefore be used only when maximum accuracy is a major consideration, and when it is not possible to convert to the next higher precision.

▶ -HIGHENDPROCESSORS / -LOWENDPROCESSORS*

Short forms: -HIGH / -LOW

These options are valid in 64V mode only.

Controls the target machine optimization for 64V-mode code generation. The distinction between high-end and low-end processors is not valid in 32IX mode, which performs fairly consistently across all machines on which it runs. All newer 50 Series processors -- that is, all processors with four-digit names except the 2250 -- can generate 32IX-mode code.

-HIGHENDPROCESSORS
Generates optimal code for all processors in the 4000, 6000, and 9000 series by avoiding the use of skip instructions. Code generated with this option runs on all 50 Series machines, but runs faster on the high-end processors.

-LOWENDPROCESSORS
Generates optimal code for 50 Series machines other than the 4000, 6000, and 9000 series processors. Code generated under this option runs on all 50 Series machines, but runs faster on the low-end machines.

▶ -HOLEYSTRUCTURES / -NOHOLEYSTRUCTURES*

Short forms: -HOLE / -NOHOLE

These options are valid in 32IX mode only.

Controls the alignment of structure members 32 bits or larger that are not bit fields.

-HOLEYSTRUCTURES
Causes all non-bit-field structure members that are 32 bits or larger to be aligned on even word (32-bit) boundaries.

-NOHOLEYSTRUCTURES
Causes all non-bit-field structure members that are 32 bits or larger to be aligned on halfword (16-bit) boundaries.

▶ -IGNOREREGISTER / -NOIGNOREREGISTER

Short forms: -IG / -NOIG

These options are valid in 32IX mode only.

Controls the meaning of the **register** keyword. In 64V mode, variables declared with the **register** keyword are placed not in registers, but in the stack frame.

-IGNOREREGISTER
Ignores the **register** keyword and allows the compiler's optimizer to control all placement of register variables. This option is the default at optimization levels 1 and higher.

-NOIGNOREREGISTER
Respects the **register** keyword. The option is the default at -NOOPTIMIZE.

▶ -INCLUDE *pathname*

Short form: -INC

Specifies an additional directory to be searched when the compiler is attempting to locate files that were specified with **#include** preprocessor commands. The *pathname* must end in a directory name.

In PRIMOS C, you can specify directories to be searched for **#include** files in a number of different ways. The search algorithm used by the C compiler is described on page 2-2.

▶ -INPUT *pathname*

Short form: -I

Is identical to the -SOURCE option. That is, both options designate the source file pathname to be compiled. For example,

```
CC -INPUT pathname
```

and

```
CC pathname
```

produce the same result. Also, *pathname* must not be designated more than once on the command line. The -INPUT option is obsolete, and its use is not recommended.

You may not put this option in an options file (see the -OPTIONSFILE option). You must specify it on the command line.

▶ -INTEGEREXCEPTIONS / -NOINTEGEREXCEPTIONS*

Short forms: -INTE / -NOINTE

These options are valid in 32IX mode only.

Forces the hardware to take a fault on integer overflow, underflow, and division by zero.

-INTEGEREXCEPTIONS
Enables the integer exception-handling mechanism. When integer arithmetic causes an integer to be larger than the data item to which it is assigned, a FIXEDOVERFLOW runtime error occurs. When a division by zero is encountered, a ZERODIVIDE runtime error occurs.

-NOINTEGEREXCEPTIONS
Does not enable integer exception handling.

▶ -INTLONG* / -INTSHORT

Short forms: -INTL / -INTS

Controls the meaning of the **int** keyword.

-INTLONG
This is the standard operational mode for the PRIMOS C compiler. The **int** keyword means **long int**, or 32-bit integer. All undeclared functions are expected to return a **long int**.

-INTSHORT
This option is useful when debugging code to be ported to a machine where **int** means **short int**, or 16-bit integer. Use of this option is not recommended.

When you call a function from a program compiled with -INTSHORT, parameters of type **short** or **char** are converted to type **long int**, just as they are with the default, -INTLONG. This conversion allows you to use the standard C libraries. You must, however, declare all C library functions that return type **int** as returning type **long**. Otherwise, the C compiler assumes that the functions return type **short**.

▶ -INTRINSIC [*sourcename*] *intrinsicname*

Short form: -INTR

Causes the compiler to generate inline code for, or a shortcall to, any of a limited number of common C library routines. The first argument, *sourcename*, is optional and is the name that is used in the source program to reference the intrinsic function. If this argument is missing, it is assumed to be the same as I{intrinsicname}. The last argument to the option, *intrinsicname*, is the true name of the intrinsic function. The currently supported intrinsic routines are as follows:

- In V mode, strlen( ), strcpy( ), and strncpy( ).
- In IX mode, abs( ), fabs( ), strlen( ), strcmp( ), strcpy( ), strncpy( ), and, if -ANSI is specified, memcpy( ).

If -ANSI is not specified, specifying -INTRINSIC STRNCPY enables strncpy( ) to perform a block move. If -ANSI is specified, specifying -INTRINSIC MEMCPY enables memcpy( ) to perform a block move, and specifying -INTRINSIC STRNCPY enables strncpy( ) to perform according to the standard: that is, it copies the specified number of characters from string 1 to string 2, then pads any remaining spaces in string 2 with '\0's.

For example, suppose you invoke the compiler with the command line

```
OK, CC MYPROGRAM -INTRINSIC MYSTRLEN STRLEN
```

During compilation, each call to the function MYSTRLEN in the program MYPROGRAM causes the compiler to generate inline code for the C library function strlen( ).

The intrinsic versions of strncpy( ) and memcpy( ) do not copy strings that span segment boundaries unless the -SEGMENTSPANCHECKING option is specified.

▶ -LBSTRING* / -PBSTRING

Short forms: -LBS / -PBS

Controls the placement of string constants.

-LBSTRING
: String constants are placed in the linkage area and thus may be modified at runtime. This is the default.

-PBSTRING
String constants are placed in the procedure area and thus may not be modified at runtime. They may, however, be shared between multiple processes. This option improves performance slightly.

▶ -LISTING [*argument*]

Short form: -L

Controls generation of a source listing file. The argument specification can be one of the following:

Argument	*Meaning*
pathname	Source listing is written to the file *pathname.*
YES	Instructs the C compiler to create a listing file named PROGRAM.LIST for all programs compiled in 32IX mode and for 64V-mode programs with a .CC or .C suffix. If you compile in 64V mode and the source filename does not include a .CC or .C suffix, the C compiler generates a listing file named L_PROGRAM. This argument is the default.
NO	Does not generate a source listing file. This is the default when you do not specify the -LISTING option.
TTY	Source listing is displayed on the user's terminal.

You may not put this option in an options file (see the -OPTIONSFILE option). You must specify it on the command line.

▶ -LOWENDPROCESSORS*

See -HIGHENDPROCESSORS.

▶ -NEWFORTRAN* / -OLDFORTRAN

Short forms: -NEWF / -OL

These options are valid in 64V mode only.

Controls the interlanguage interface in 64V mode. In 32IX mode, only the new language interface is available.

-NEWFORTRAN
Uses the new interlanguage interface in 64V mode. This is the default when you compile a 64V-mode program that uses the **fortran** keyword.

-OLDFORTRAN
Uses an old, obsolete interlanguage interface in 64V mode. If you compile a 64V-mode program with the -OLDFORTRAN option, the compiler issues a warning. You are strongly encouraged to use the new interlanguage calling interface instead of the old one.

▶ -NOONUNIT

Short form: -NOON

Lets PRIMOS report the occurrence of fatal compiler errors. Use of this command option is not recommended, as the compiler performs appropriate cleanup procedures and PRIMOS does not.

▶ -OLDFORTRAN

See -NEWFORTRAN.

▶ -OPTIMIZE [*level*] / -NOOPTIMIZE*

Short forms: -OPT / -NOOPT

Controls object code optimization.

-OPTIMIZE
Without an optimization level, this option is valid in 64V mode only. Causes all data to be even halfword-aligned, and causes variables declared **register** to be copied into the local stack frame.

-OPTIMIZE 1
This option is valid in 32IX mode only. Performs first-level optimizations:

- Register allocation
- Complex tree pruning
- Unreachable code elimination
- Peephole optimizations such as branch chaining, instruction changing (complex strength reduction), and instruction elimination

-OPTIMIZE 2
This option is valid in 32IX mode only. Performs second-level optimizations:

- Solving of data flow equations and use of this information
- Post-op improvements
- Loop invariant code removal
- Induction expression identification and elimination

- Copy propagation
- Dead variable elimination
- Useless code removal
- Pointer comparison improvement

-OPTIMIZE 3

This option is valid in 32IX mode only. Performs third-level optimizations:

- Elimination of tail recursion
- Running of copy propagation iteratively until no further changes are made to the intermediate representation
- Running of dead variable elimination iteratively until there are no more dead variables to remove
- Increase of the complexity of routines that are made available for inline expansion
- Provision of more sophisticated register tracking to allow better overlapping of temporary, scratch, and special registers
- More aggressive temporary assignment

Note

When you use second-level or third-level optimizations, compilation time is significantly longer than at lower optimization levels. Debug all code fully before you use the -OPTIMIZE option.

-NOOPTIMIZE

Does not perform any object code optimization. However, constant folding and simple strength reduction are performed in both 64V and 32IX modes. In 32IX mode, simple tree pruning, conversion propagation, and avoidance of code generation for discarded values are also performed.

▶ -OPTIONSFILE *pathname*

Short form: -OPTIO

Specifies that you have placed compiler command line options in a text file called *pathname*. The compiler processes the options file before compiling your program. The optional suffixes for an options file are .OPTIONS.CC and .OPTIONS.C. The compiler automatically appends a suffix if needed. An options file may contain any options except the following: -32IX, -64V, -ANSI, -NOANSI, -BINARY, -INPUT, -LISTING, -SOURCE, -SYSOPTIONS (32IX mode only), -NOSYSOPTIONS (32IX mode only). Options files may be nested for up to nine levels. A single options file may contain no more than 1024 compiler options. You may use CPL style comments in an options file. A /* comments out until the end of line. Blank lines are also allowed.

▶ -OPTSTATISTICS / -NOOPTSTATISTICS*

Short forms: -OPTS / -NOOPTS

These options are valid in 32IX mode only.

Controls the printing of optimization statistics. This option is meaningful only when used in conjunction with an optimization level of 1 or higher. See the discussion of the -OPTIMIZE option.

-OPTSTATISTICS
Prints compiler statistics about optimization.

-NOOPTSTATISTICS
Does not print compiler statistics about optimization.

▶ -PACKBYTES / -NOPACKBYTES

Short forms: -PACK / -NOPACK

These options are valid in 32IX mode only.

Controls the alignment of contiguous single-byte entities in structures and unions. Ordinarily, these entities are structure members of type **char**. Structure or union members that are larger than a single byte (including character arrays) are aligned to start on a halfword (16-bit) boundary.

-PACKBYTES
Packs adjacent single-byte entities in structures or unions, two per halfword.

-NOPACKBYTES
Causes adjacent single-byte entities in structures and unions to be stored in the high byte of a 16-bit halfword, followed by a single-byte hole.

If -PACKBYTES is specified, the structure

```
struct smallpack {
    char a;
    char b;
    char c;
    char arr[3];
};
```

is stored as

```
| 8 bits | 8 bits |
  a        b          (word 1)
  c                   (word 2)
  arr[0]   arr[1]     (word 3)
  arr[2]              (word 4)
```

It is *not* stored as

```
| 8 bits | 8 bits |
  a        b          (word 1)
  c        arr[0]     (word 2)
  arr[1]   arr[2]     (word 3)
```

If -PACKBYTES is not specified, the previous example is stored as

```
| 8 bits | 8 bits |
  a                   (word 1)
  b                   (word 2)
  c                   (word 3)
  arr[0]   arr[1]     (word 4)
  arr[2]              (word 5)
```

▶ -PARTIALDEBUG

Short form: -PAR

This option is valid in 32IX mode only.

Generates symbol information for the Source Level Debugger (DBG) only for variables that are actually referenced in expressions. (Refer to Appendix B, Debugging C Programs, for more information on DBG.) With -PARTIALDEBUG, the debugger does not know about symbols that are declared but never referenced. Use of this option can dramatically decrease the amount of debugger object text emitted for a routine, because most routines include many declarations that are not used.

▶ -PBSTRING

See -LBSTRING.

▶ -POP / -NOPOP*

Short forms: -POP / -NOPOP

Controls pushing and popping of multiple **#define** macro definitions if a macro is defined more than once.

-POP
Pushes previously defined values onto a value stack each time a new macro definition is encountered. A macro definition can be removed with a single **#undef**, which also reinstates the previous value for the macro definition.

-NOPOP
Discards the old value and replaces it with the new value.

▶ -PREPROCESSONLY [*pathname*]

Short form: -PRE

This option is valid in 32IX mode only.

Causes the output of the C preprocessor to be placed in the specified file. If the pathname argument is omitted, the default name of the file is *name*.I, where *name* is the name of the source file.

Subsequent phases of the compilation are not performed. Use this option to make sure that your macros are expanded in the way you intend. If -ANSI is also specified, the resulting file is much more readable than if -ANSI is not specified.

▶ -PRODUCTION

Short form: -PROD

Enables production level DBG support. (Refer to Appendix B, Debugging C Programs, for more information on DBG.) This support includes information about each program block and symbol, but does not include statement information.

▶ -QUADCONSTANTS / -NOQUADCONSTANTS*

Short forms: -QUADC / -NOQUADC

These options are valid in 32IX mode only.

Enables support for quadruple precision floating-point constants when the program is compiled without the -ANSI option.

-QUADCONSTANTS
Allows the use of quad-precision constants (type **long double**). Quad-precision constants are specified in the same way as **double** constants, with an added "L" suffix. ("L" may be in uppercase or lowercase.)

Note

The -ANSI C compiler option automatically includes support for quad-precision constants.

-NOQUADCONSTANTS
Does not allow the use of quad-precision constants.

▶ -QUADFLOATING / -NOQUADFLOATING*

Short forms: -QUADF / -NOQUADF

These options are valid in 32IX mode only.

Enables support for quadruple precision floating-point variables when the program is compiled without the -ANSI option.

-QUADFLOATING
Allows the use of quad-precision variables. Such variables are declared as type **long double**.

Note

The -ANSI C compiler option automatically includes support for quad-precision constants.

-NOQUADFLOATING
Does not allow the use of quad-precision variables.

▶ -SAFEPOINTERS / -NOSAFEPOINTERS*

Short forms: -SAFE / -NOSAFE

These options are valid in 32IX mode only.

Controls the preservation of the byte offset bit when converting pointers to integers.

-SAFEPOINTERS
Always preserves the byte offset bit of pointers when converting them to integers, even if the pointer type is noncharacter and thus the pointer should not have an odd byte offset. Use this option only in tricky C code where noncharacter pointers are used to hold odd integer flags (such as -1). If even integers (such as -2) are used for the flags, use of this option is not necessary. This option is *not* the default, and considerably better code is generated for pointer comparisons if the option is *not* used.

-NOSAFEPOINTERS
When converting pointers to integers (for example, for comparisons), assumes that any noncharacter pointers point to halfword-aligned objects and thus do not have byte offset bits.

▶ -SEGMENTSPANCHECKING / -NOSEGMENTSPANCHECKING*

Short forms: -SEG / -NOSEG

These options are valid in 32IX mode only.

Tells two intrinsic functions to check for arguments that span segment boundaries.

-SEGMENTSPANCHECKING
Causes the two block-memory intrinsic functions strncpy( ) and memcpy( ) to check for segment-spanning arguments before performing their operations. If any of the arguments span segments, the compiler generates slower code that produces the correct results across segment boundaries. This option works only if you specify it in conjunction with one or more of the following: -INTRINSIC STRNCPY, -INTRINSIC MEMCPY, -STANDARDINTRINSICS.

-NOSEGMENTSPANCHECKING
Does not cause the two functions to check for segment-spanning arguments.

▶ -SHORTCALL *routinename*

Short form: -SHORTC

Instructs the compiler to generate a shortcall to the specified *routinename* (either a JSXB or a JMP, depending on addressing mode). The actual behavior of this option varies between 64V and 32IX modes; see Chapter 6, Advanced Topics, for more information.

▶ -SINGLEFLOATING

See -DOUBLEFLOATING.

▶ -SILENT / -NOSILENT*

Short forms: -SIL / -NOSIL

Controls the output of warning and verbose compiler error messages.

-SILENT
Prevents verbose and warning compiler error messages from appearing. These messages are not displayed on the user's terminal or printed in the source listing file.

-NOSILENT
Causes verbose and warning compiler error messages to appear on the user's terminal and in the source listing file.

See the description of compiler error messages on page 2-4.

▶ -SOURCE *pathname*

Short form: -SOURCE

Is identical to the -INPUT option. That is, they both designate a source file pathname to be compiled. For example,

```
CC -SOURCE pathname
```

and

```
CC pathname
```

produce the same result. Also, *pathname* must not be designated more than once on the command line. The -SOURCE option is obsolete, and its use is not recommended.

You may not put this option in an options file (see the -OPTIONSFILE option). You must specify it on the command line.

▶ -SPEAK

Short form: -SPEAK

This option is valid in 32IX mode only.

Enables the display of compile-time progress messages. These messages indicate which include files are being processed and when the compiler is parsing, generating, and emitting code.

▶ -STANDARDINTRINSICS

Short form: -STAN

This option is valid in 32IX mode only.

Causes the compiler to generate inline code for, or a shortcall to, a small group of common C library routines. -STANDARDINTRINSICS generates code for all of the routines that can be enabled separately by the -INTRINSIC option. These routines include abs( ), fabs( ), strcmp( ), strcpy( ), strlen( ), strncpy( ), and, if -ANSI is specified, memcpy( ).

If -ANSI is not specified, strncpy( ) is enabled to perform a block move. If -ANSI is specified, memcpy( ) is enabled to perform a block move, and strncpy( ) is enabled to perform according to the standard: that is, it copies the specified number of characters from string 1 to string 2, then pads any remaining spaces in string 2 with '\0's.

The intrinsic versions of strncpy( ) and memcpy( ) do not copy strings that span segment boundaries unless the -SEGMENTSPANCHECKING option is specified.

▶ -STATISTICS / -NOSTATISTICS*

Short forms: -STAT / -NOSTAT

Controls the display of compiler statistical data on the user's terminal.

-STATISTICS
Displays statistical data on compiler internal table use, number of source lines compiled, and average compilation speed.

-NOSTATISTICS
Does not display compiler statistical data on the user's terminal.

▶ -STORE_OWNER_FIELD / -NO_STORE_OWNER_FIELD*

Short forms: -SOF / -NSOF

Stores the identity of called procedures in a location available to the PRIMOS DUMP_STACK command mechanism. When the PRIMOS DUMP_STACK command is issued, the names of procedures compiled with -STORE_OWNER_FIELD appear within the information displayed for that procedure's stack frame.

-STORE_OWNER_FIELD
Causes the generation of code that stores the name of each procedure directly following its ECB. The procedure name can then be used by the PRIMOS DUMP_STACK mechanism.

-NO_STORE_OWNER_FIELD
Suppresses the generation of code that stores the identity of a called procedure.

▶ -STRICTCOMPLIANCE / -NOSTRICTCOMPLIANCE*

Short forms: -STRIC / -NOSTRIC

These options are valid in 32IX mode only. Used to detect certain violations of the ANSI standard, mainly PRIMOS extensions that are not available in other C compilers. In particular, this option causes uses of the fortran keyword to be flagged as an error. -STRICTCOMPLIANCE should be used in conjunction with the -ANSI option.

-STRICTCOMPLIANCE
Checks for certain ANSI violations and issues warnings/errors if found.

-NOSTRICTCOMPLIANCE
Does not check for certain ANSI violations.

▶ -SYSOPTIONS* / -NOSYSOPTIONS

Short forms: -SYS / -NOSYS

These options are valid in 32IX mode only.

Controls reading of an optional global system options file. This file can be processed in 32IX mode only. If -SYSOPTIONS is active (the default), the system options file is processed at the start of command line processing.

-SYSOPTIONS
Looks for the file SYSOVL>CI.OPTIONS.CC and if the file exists, processes it as if you had typed

```
OK, CC -32IX -OPTIONSFILE SYSOVL>CI.OPTIONS.CC
```

on the command line. The alternate name for this file is SYSOVL>CI.OPTIONS.C.

-NOSYSOPTIONS
Does not look for the file SYSOVL>CI.OPTIONS.CC.

You may not put either of these options in an options file (see the -OPTIONSFILE option). You must specify them on the command line.

▶ -UNDEFINE *macroname*

Short form: -UNDEF

This option is valid in 32IX mode only.

Removes any initial macro definition for *macroname*. The following initial definitions are entered by the compiler in 32IX mode: ___CI as 1, ___50SERIES as 1, ___DEBUG as 1 if the -DEBUG option has been specified, and ___OPTIMIZE as the optimization level specified on the command line, if any.

▶ -VALUEONLY *routinename*

Short form: -VALUE

This option is valid in 32IX mode only.

Identifies the specified *routinename* as having no side effects, thus allowing the optimizer to remove it as loop invariant code.

▶ -VERBOSE / -NOVERBOSE*

Short forms: -VERB / -NOVERB

Controls the display of verbose messages (messages that are not normally displayed).

-VERBOSE
Causes verbose messages to be displayed on the user's terminal and in listing files.

-NOVERBOSE
Prevents verbose messages from appearing. These messages are not displayed on the user's terminal or printed in the source listing file.

See the description of compiler messages earlier in this chapter.

▶ -XREF / -XREFS

Short forms: -XREF / -XREFS

Controls generation of full or partial cross-reference listing.

-XREF
Generates a full symbol cross-reference at the end of the listing file. Information as to the site, type, storage class, line declared, and lines used is printed in the listing file for each declared symbol.

-XREFS
Prints only information about symbols that have been referenced at least once after they have been declared.

3

LINKING C PROGRAMS

This chapter provides you with the basic information you need to link and execute non-ANSI C programs under PRIMOS. (See Chapter 8 for information about linking and running ANSI C programs.) It describes the CCMAIN library, the runtime libraries, and the use of BIND and SEG.

Although PRIMIX uses the same C compiler as PRIMOS, the command syntax and library functions described in this book are different from those available under PRIMIX. For information about linking and executing C programs under PRIMIX and about the C language libraries supplied with PRIMIX, consult the PRIMIX references listed in About This Book.

The CCMAIN library allows you to pass arguments to your program from the command line in the customary C fashion. (If you do not use this library, PRIMOS does not allow you to use command line arguments.) CCMAIN parses the PRIMOS command line used to invoke the program. It then passes the PRIMOS command line to your MAIN program in the argument count/argument value (argc/argv) format. You must link CCMAIN before you link your object files.

Note

There is a new library called ANSI_CCMAIN. It allows you to pass arguments to your program from the command line as CCMAIN does. Additionally, ANSI_CCMAIN enables calls to the ANSI C library functions. (See Chapter 8 for more information about the ANSI C library.)

RUNTIME LIBRARIES

Two runtime libraries come with the PRIMOS C compiler: C_LIB and CCLIB. C_LIB is a binary library containing DYNTs (dynamic links) to two Executable Program Format (EPF) libraries in the directory LIBRARIES*. The EPF libraries contain the runtime library routines described in Chapter 4. Use the C_LIB DYNT library with the BIND utility, not with the SEG utility. For more information about EPFs, see the *Programmer's Guide to BIND and EPFs.* The CCLIB library provides the same functionality as the C_LIB DYNT library, except that CCLIB links much more slowly, results in a larger program, and contains no DYNTs.

Note

If you use the SEG utility to link C programs, you must use the CCLIB library.

Note also that SEG does not provide the improved performance and reduced linking time that BIND provides. The CCLIB library is provided primarily for compatibility with pre-Rev. 19.4 systems. Use of SEG and CCLIB is also required to build shared static-mode programs. See the end of this chapter for more information about shared static-mode programs.

Using the C_LIB Library

By default, each user has search rules in ENTRY$.SR, which is located in the top-level directory SYSTEM. The following entries may appear in the ENTRY$.SR file for a typical system:

```
-PUBLIC SYSTEM_LIB$PRG.RUN
-PUBLIC TRANS_LIB$PRG.RUN
-PUBLIC TRANS_LIB$PRC.RUN
-PUBLIC SYSTEM_LIB$PRC.RUN
-PUBLIC APPLICATION_LIBRARY.RUN
-PUBLIC FTN_LIBRARY.RUN
-STATIC_MODE_LIBRARIES
-PUBLIC CC_LIBRARY.RUN
-PUBLIC ANSI_CC_LIBRARY.RUN
```

Name Conflicts

Two complete versions of the C libraries exist: one for use by 64V-mode code, the other for use by 32IX-mode code. In a dynamic linking environment (programs linked with BIND), library calls made by a 64V-mode routine must link to the 64V mode libraries, and library calls made by a 32IX-mode routine must link to the 32IX library. In addition, calls to a C library routine must link to the C library rather than to a routine with the same name in another language library. The compiler and linkers accomplish this automatically, in the following manner:

- If a 64V-mode routine that calls printf( ) is linked with CCLIB using SEG, the routine printf( ) from CCLIB is statically bound into the user program.

- If a 64V-mode routine that calls printf( ) is linked with C_LIB using BIND, the symbol name printf is changed to CC$printf and a dynamic link is made to CC$printf in the EPF libraries via the DYNTs in C_LIB.
- In 32IX mode, the names of all non-FORTRAN external definitions and references (routines and common blocks) are changed by prepending the prefix G$ to the symbol name in the user code. Thus, a call to printf( ) is treated by the compiler as a call to the routine G$printf. The 32IX library, which was written in C and compiled in 32IX mode, also has the G$ prefix prepended, so either SEG or BIND resolves the correct references from user code to library.

Because of the prefix handling described above, the number of significant characters in external names varies with the addressing mode and linker used. The following table summarizes these differences:

Linker	*Addressing Mode*	*Number of Significant Characters*
SEG	64V	8
BIND	64V	32
SEG	32IX	6
BIND	32IX	30

As a user of the C language under PRIMOS, be aware of potential conflicts between C library subroutine names and subroutine names in the FORTRAN and other libraries. To avoid conflicts, always link C subroutines and libraries before you link subroutines and libraries written in another programming language.

For example,

```
[BIND Rev. T3.0-23.0 Copyright (c) 1990 Prime Computer, Inc.]
: LO C_MODULE_1.BIN
: LO C_MODULE_2.BIN
   .
   .
   .
: LI C_LIB                    /* Resolves calls to C specific
                                 libraries */
: LO FTN_MODULE_1.BIN
: LO FTN_MODULE_2.BIN
   .
   .
   .
: LI                          /* Resolves calls to the system
                                 library PFTN.LIB */
```

GUIDELINES FOR LINKING C PROGRAMS

Before linking C programs, observe the following guidelines:

- If you wish to use the command line argument feature of the C language, link CCMAIN as the first library.
- If you link CCMAIN, BIND expects your main routine to be named main( ); this routine can be located anywhere in your program. If your main routine is not named main( ), use the MAIN subcommand to tell BIND which routine is your main routine. See page 3-6 for information about the MAIN subcommand.
- If you do not link CCMAIN, BIND expects your main routine to be the first routine in the first object file you link. However, this routine does not have to be named main( ). If your main routine is not the first routine in the first file you link, use the MAIN subcommand to tell BIND which routine is your main routine.
- Whether or not you use CCMAIN, you must link the C_LIB DYNT library before you link the PRIMOS system libraries.

Linking Programs

You may use either BIND or SEG to link C programs. Note, however, that use of BIND is recommended on systems at Rev. 19.4 or later. Use of SEG is explained at the end of this chapter.

The BIND utility creates a runfile called an Executable Program Format (EPF). You can link most C programs using BIND by following this procedure:

1. Invoke the BIND utility with the BIND command. The system displays a colon (:) prompt, indicating that you are now interacting with the BIND utility.
2. Issue the LIBRARY (LI) subcommand to link CCMAIN if you want to use the command line argument features of the C language.
3. Issue the LOAD (LO) subcommand to link the main object file and any additional object files from separately compiled subroutines.
4. Issue the LI subcommand to link the C_LIB library. If you get a `BIND COMPLETE` message at this point, you may skip the next step.
5. Issue the LI subcommand to link system subroutines and functions called from libraries. The system responds with a `BIND COMPLETE` message after the subroutines and functions have been linked. If you do not receive the `BIND COMPLETE` message, use the MAP subcommand to identify the modules that were not linked, and link them. The BIND utility automatically appends a .RUN suffix to the end of the executable file.
6. Issue the FILE command to exit from BIND. The FILE command also saves the newly created executable file in your directory and returns you to PRIMOS command level.

BIND Examples

The BIND utility allows you to create an EPF interactively or on a single command line. Three linking sessions are illustrated in this section. Example 1 shows the use of BIND without the CCMAIN runtime library. Example 2 shows the use of BIND with the CCMAIN runtime library. Example 3 shows the use of command line arguments that accomplish the BIND link in a single command without interactive usage of BIND.

Example 1:

```
OK, SLIST EXAMPLE1.CC
main( )
{
   printf("Goodbye, universe\n");
}

OK, CC EXAMPLE1 -32IX          /* Compile EXAMPLE1 */
[CI Rev. T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]
0 Errors and 0 Warnings detected in 4 source lines.
OK, BIND EXAMPLE1              /* Invoke the BIND utility */
[BIND Rev. T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]
: LO EXAMPLE1                  /* Link object file */
: LI C_LIB                     /* Link EPF libraries */
: LI                           /* Link system subroutines called
                                  from program and C library,
                                  if necessary. */

BIND COMPLETE
: FILE                         /* File (save) executable file in
                                  directory and return to PRIMOS */

OK,
```

Example 2:

```
OK, SLIST EXAMPLE2.CC
main (argc, argv)
     int argc;
     char *argv [];
{
     printf("Number of arguments detected = %d.\n", argc);
}


OK, CC EXAMPLE2 -32IX    /* Compile EXAMPLE2 */
[CI Rev. T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]
00 Errors and 00 Warnings detected in 6 source lines.

OK, BIND EXAMPLE2        /* Invoke the BIND utility */
[BIND Rev. T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]
: LI CCMAIN              /* Link CCMAIN runtime library */
: LO EXAMPLE2            /* Link object file */
: LI C_LIB               /* Link EPF libraries */
BIND COMPLETE
: FILE                   /* File (save) executable file in
                            directory and return to PRIMOS*/
OK,
```

Example 3:

```
OK, CC EXAMPLE3 -32IX
[CI Rev. T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]
00 Errors and 00 Warnings detected in 112 source lines.

OK, BIND EXAMPLE3 -LI CCMAIN -LO EXAMPLE3 -LI C_LIB -LI
[BIND Rev. T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]
BIND COMPLETE
```

Unreferenced Routines and Variables

A program may contain declarations for external routines or variables that are not referenced in the program. In such cases, the PRIMOS C compiler behaves differently depending on whether the -DEBUG option was invoked.

- If -DEBUG is invoked, the compiler retains information about all variables and functions that it encounters. If any external variable or function is declared but not referenced in the program, BIND will be unable to resolve the reference and will not issue a BIND COMPLETE message.
- If -DEBUG is not invoked, the compiler ignores references to any variables or functions that are declared but not referenced, and BIND will issue a BIND COMPLETE message.

Using the MAP Subcommand

If BIND does not display the message BIND COMPLETE at the end of the linking procedure, you can issue the MAP subcommand to check for any unresolved subroutine, program, or common block references. The MAP subcommand of BIND has the following format:

MAP [*pathname*] [*options*]

If you specify *pathname*, the MAP subcommand writes the unresolved references to a file instead of displaying them at your terminal. For example,

```
: MAP MYFILE      /* Writes a BIND map of your program to a
                     file called MYFILE */
```

The -UNDEFINED option enables you to display a list of all unresolved references in your program as follows:

```
: MAP -UNDEFINED  /* Displays a list of all unresolved
                     references at your terminal */
```

Using the MAIN Subcommand

The MAIN subcommand tells BIND which routine is the main entrypoint of your program. Use MAIN in either of the following situations:

- You are using CCMAIN, and your main routine is not named main( ).
- You are not using CCMAIN, and your main routine is not the first routine in the first object file you link.

The MAIN subcommand has two formats, one for 64V mode and one for 32IX mode. If you compiled your program in 64V mode, the format is

MAIN ***routine-name***

where *routine-name* is the name of your main routine. If you compiled your program in 32IX mode, the format is

MAIN G$*routine-name*

That is, you must put the G$ prefix before the name of your main routine. (See page 3-3 for more information about the G$ prefix.)

Issue the MAIN subcommand after you load your source file(s), but before you link in the C_LIB library.

For example, suppose that you are compiling the program TEST.C in 64V mode, that you want to use command line arguments, and that your main routine (the only routine in your program) is called test( ). You can compile and link your program as follows:

```
OK, CC TEST
[CC Rev. T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]
00 Errors and 00 Warnings detected in 16 lines and 594 include lines.
OK, BIND
[BIND Rev. T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]
: LI CCMAIN
: LO TEST
: MAIN TEST
Main Program ECB is TEST at -0002/000236
: LI C_LIB
BIND COMPLETE
: FILE
OK,
```

As another example, suppose that you are compiling TEST2.C in 32IX mode, that you are not using command line arguments, and that your main routine, called main( ), is the last routine in your source file. You can compile and link your program as follows:

```
OK, CC TEST2 -32IX
[CI Rev. T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]
00 Errors and 00 Warnings detected in 134 lines and 176 include lines.
OK, BIND
[BIND Rev. T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]
: LO TEST2
: MAIN G$MAIN
Main Program ECB is G$MAIN at -0002/000166
: LI C_LIB
BIND COMPLETE
: FILE
OK,
```

Using the QUIT Command

If for some reason you have to exit prematurely from a BIND session, you can do so by issuing the QUIT command. Simply type the following:

QUIT

The QUIT command aborts a BIND session and does not save the EPF. Before it returns you to PRIMOS command level, BIND prompts you to make sure you really want to abort the session. For example,

```
OK, BIND EXAMPLE
[BIND Rev. T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]
: LI CCMAIN
: LO EXAMPLE.BIN
: LI C_LIB
BIND COMPLETE
: QUIT
EPF not filed, ok to quit? ('Yes', 'Y', 'No', 'N'):Y
OK,
```

Using the HELP Subcommand

The HELP subcommand of BIND is available in case you encounter problems while trying to link a program. The HELP subcommand has the following format:

HELP [*command_name*] [-LIST]

When you issue the HELP subcommand followed by the name of a particular command, BIND replies with concise online information describing the syntax and semantics of the specified command. For example,

```
: HELP MAP
MAp [<map dest>] [<map option>]
     will copy a mapfile to <map dest> with <map option>.
     <map dest> may be a file, -TTY or -SPOOL.
     <map option> may be one of the following:
       -FULL, -RANGES, -BASE, -UNdefined, -FLAGS,
       -Named_SYmbols.
```

If you issue the HELP subcommand followed by the -LIST option, BIND displays a list of its subcommands at your terminal.

Executing an EPF

You can execute an EPF at PRIMOS command level by issuing the RESUME (R) command followed by the program name. For example,

```
OK, RESUME EXAMPLE1.RUN

Goodbye, universe
OK,
```

If you linked your program using the CCMAIN library, you may follow the program name with command line arguments. The RESUME command is not included in the argument count (argc) or the argument vector (argv). For example, if your program, PROG.C, is

```
#include <stdio.h>

main(argc, argv)
int argc;
char *argv[];
{
    int i;

    printf("The arguments are: ");
    for (i = 0; i < argc; i++)
         printf("%s%c", argv[i], (i < argc-1) ? ' ' : '\n');
    printf("%d arguments\n", argc);
}
```

and your command line is

```
OK, RESUME prog how many args is this
```

the program will display

```
The arguments are: prog how many args is this
6 arguments
OK,
```

Loading C Programs With the SEG Linking Loader

You can load most C programs with the SEG loader using the following procedure:

1. Invoke the SEG utility with the SEG -LOAD command. The system displays a

   ```
   $
   ```

   prompt, indicating that you are now interacting with the SEG linker's VLOAD subprocessor.

2. Issue the LIBRARY (LI) command to link CCMAIN if you want to use the UNIX-like command line argument feature.

3. Issue the LOAD (LO) command to load the main object file and any additional object file from separately compiled subroutines.

4. Issue the LI command to link the CCLIB runtine library. If you get a LOAD COMPLETE message at this point, you may skip the next step.

5. Issue the LI command to link the system libraries. SEG then responds with a LOAD COMPLETE message. If this message does not appear, then use the MAP 3 command to identify the modules that were not linked, and link them. (See the *SEG and LOAD Reference Guide* for more information.) The SEG loader automatically appends the .SEG suffix to the runfile.

6. Issue the QUIT command to exit from SEG.

Note that you cannot use numerical command line arguments to a program if you use SEG to link the program. When you execute the program, numerical command line arguments are interpreted as arguments to the SEG command itself.

Example:

```
OK, SLIST EXAMPLE.C
void main(argc, argv)
  int argc;
  char *argv[];
{
  printf("Number of arguments is %d.\n", argc);
}

OK, CC EXAMPLE
[CC Rev. T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]
00 Errors and 00 Warnings detected in 6 source lines.

OK, SEG -LOAD
[SEG Rev. T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]
$ LI CCMAIN
$ LO EXAMPLE
$ LI CCLIB
$ LI
LOAD COMPLETE
$ QU

OK, SEG EXAMPLE every breath you take
Number of arguments is 5.

OK, SEG EXAMPLE money for nothing
Number of arguments is 4.
```

CREATING SHARED C PROGRAMS

Shared programs written in C may be created using the methods outlined in Chapter 4, Advanced SEG Techniques, of the *SEG and LOAD Reference Guide.* However, you must observe some additional restrictions to insure that shared C programs execute successfully.

You must use the nonshared static-mode library, CCLIB. You may not use the DYNT library, C_LIB.

If the C program to be shared does not use CCMAIN, then RUNIT may be used for the shared program. In this case, you must ensure that segment 4000 is used as the data segment. For example, if PROGRAM.BIN is the binary file for a program to be shared into segment 2177, a possible SEG command sequence is as follows:

```
OK, SEG -LOAD
[SEG Rev. T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]
$ COMMON ABS 4000              /* Gets data into segment 4000
$ MIX
$ SPLIT
$ S/LO PROGRAM.BIN 0 2177 4000   /* Shares into segment 2177
                                        with data in segment 4000
$ D/LI CCLIB
$ D/LI
LOAD COMPLETE
```

```
$ RETURN
# SHARE
FILE ID: EX
Creating EX2177
Creating EX4000
# QUIT
```

If the C program to be shared uses CCMAIN, then RUNIT may not be used. In this case, you must ensure that data is placed in some segment above 4000. For example, if PROGRAM.BIN is the binary file for a program to be shared into segment 2177, a possible SEG command sequence is as follows:

```
OK, SEG -LOAD
[SEG Rev. T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]
$ S/LI CCMAIN 0 2177 4001          /* Shares into segment 2177 with
                                          data in segment 4001
$ D/LO PROGRAM.BIN
$ D/LI CCLIB
$ D/LI
LOAD COMPLETE
$ RETURN
# SHARE
FILE ID: EX
Creating EX2177
# QUIT
```

It is strongly recommended that you become completely familiar with Chapter 4, Advanced SEG Techniques, of the *SEG and LOAD Reference Guide*, before attempting to create shared programs. The examples above demonstrate only the use or nonuse of CCMAIN and the placement of data. They are too trivial to demonstrate other features used in creating shared programs.

4

USING THE C LIBRARY

This chapter explains how to use the non-ANSI C library. Following a discussion of the supplied include files is a dictionary of C library functions and macros available under PRIMOS.

The non-ANSI C library is the library you use when you do *not* link your program with the ANSI_CCMAIN library. If you link your program with the ANSI_CCMAIN library, you can call all the functions in the non-ANSI library that are not available in the ANSI library. For a list of these functions, see the section entitled Nonstandard Library Functions in Chapter 8.

Chapter 8 provides an alphabetical list of the ANSI C library functions, along with information on how to write, compile, and link your program in order to call these functions. For complete documentation of the ANSI C library functions, consult the second edition of *The C Programming Language*, by Kernighan and Ritchie (1988).

Note

The library routines described in this chapter are different from those supplied with PRIMIX. If you are developing programs under PRIMIX, consult the PRIMIX books listed under Associated Documents in About This Book.

INCLUDE FILES

Many of the library functions require defined constants and keys in the calling sequence. Other routines are actually implemented as macros, not functions. In addition, some functions must be declared in the calling program because they return values that are not of type **int**. Most of the constants, keys, and macros are defined in a set of **include files** (also called **header files**), which are located in the top-level directory SYSCOM. The return types of most noninteger functions are also declared in the include files. Table 4-1 lists the supplied C include files and the routines that use them.

Using Include Files

To include one of these files in your program, enclose its name, in lowercase and without the .INS.CC suffix, in angle brackets after the **#include** command. For example,

```
#include <math.h>
```

If you copy one of these files to another directory, you may remove the .INS.CC suffix or not, as you wish. For more information about include files, see the description of include search rules on page 2-2 and the description of the command line option -INCLUDE on page 2-21.

TABLE 4-1. C Include Files

Include File	*Routines That Use Include File*
ASSERT.H.INS.CC	assert( )
CTYPE.H.INS.CC	Character classification
MATH.H.INS.CC	Mathematical
PRIME_ECS_CHARS.H.INS.CC	None (see page 4-2)
SETJMP.H.INS.CC	setjmp( ) and longjmp( )
SIGNAL.H.INS.CC	signal( )
STAT.H.INS.CC	stat( ) and fstat( )
STDIO.H.INS.CC	Input/output
STRING.H.INS.CC	String-handling
TERM.H.INS.CC	gterm( ) and sterm( )
TIME.H.INS.CC	ctime( ) and localtime( )
TIMEB.H.INS.CC	ftime( )

Using the Extended Character Set

As of Rev. 21.0, Prime expanded its character set. The Prime Extended Character Set (Prime ECS) includes characters with octal values from 0 through 0377. The include file PRIME_ECS_CHARS.H.INS.CC allows you to use the ECS symbols listed in Appendix F. The basic character set remains the same as it was before Rev. 21.0; it is the ANSI ASCII 7-bit set (called ASCII-7), with the eighth bit turned on.

The C library functions and preprocessor macros have *not* been modified to recognize the new extended character set. In particular, the character evaluation routines isascii( ), ispascii( ), isalpha( ), isdigit( ), isgraph( ), islower( ), isprint( ), ispunct( ), isspace( ), and

isupper( ) behave exactly as they did before Rev. 21.0. With the exception of isascii( ) and ispascii( ), these routines are essentially blind with respect to the 8th bit.

For example, the mnemonic UCUC_CHAR is defined to be octal 0133 in the include file PRIME_ECS_CHARS.H.INS.CC. This character represents an uppercase U circumflex (Û) in the Extended Character Set. When passed UCUC_CHAR, however, isupper( ) returns false, because isupper( ) treats the character octal 0133 the same as the character octal 0333, which is the left bracket character ([).

DICTIONARY OF C LIBRARY FUNCTIONS AND MACROS

The library functions are contained in two EPF libraries, referenced through C_LIB, and in the nonshared static-mode library CCLIB. (Use C_LIB and BIND whenever possible.) The predefined macros are contained in the supplied include files.

Interpreting Definitions of Functions and Macros

Each description in this section contains a format, in boldface, that resembles C code. The format shows the header file required by the function or macro, the parameter list, the data types of the parameters, and the type of value returned. For example, the format used for the ftell( ) function is

```
#include <stdio.h>
int ftell(filePointer)
FILE *filePointer;
```

This format indicates the following:

- You must include the stdio.h header file in your program when you use ftell( ).
- The ftell( ) routine returns an integer value.
- You must pass one parameter to ftell( ).
- The data type of the parameter is FILE *, where FILE is a **typedef** contained in the stdio.h header file.

In the formats, the parameters are given names that are consistent with their use. When several routines use the same kind of parameter, the same name is used in their formats. For example, ftell( ) and fscanf( ) both have parameters called *filePointer*, because both routines use a value returned by fopen( ) or fdopen( ), which is of type FILE *. Similarly, lseek( ) and read( ) both have parameters called *fileID* because both of those routines use a value returned by open( ) or fileno( ), which is of type **int**.

The discussion section that follows each definition contains a fuller explanation of the parameters, return value, and behavior of the routine, including whether it is implemented as a function or as a macro.

Differences Between Functions and Macros

If a routine is implemented as a macro, rather than as a function, it can be undefined with a preprocessor command of the form

#undef ***functionname***

This is useful if you want to substitute a routine with the same name to replace the library routine. In general, substituting a function for a macro reduces the size of the executable program but increases execution time. Macros are expanded inline each time they are encountered, whereas functions are expanded only once. However, a call to a function takes longer than execution of inline code.

C Library Functions in Alphabetical Order

The following section describes the library functions contained in both C_LIB and CCLIB. The library functions and their descriptions are listed alphabetically. Appendix D contains a set of tables listing the library functions by the type of action performed. Chapter 7 contains a summary of differences between these functions and their counterparts in other implementations, such as the UNIX operating systems.

▶ abort( )

Raises the ABORT$ condition, which, under normal circumstances, causes your program to terminate.

abort( )

PRIMOS prints the following message:

CONDITION ABORT$ RAISED AT *segment-number/halfword-number*

You have two ways to regain control after a call to abort( ).

- Use signal( ) in 32IX mode to catch ABORT$ where the signal type is SIGABRT.
- Set up an on-unit for the ABORT$ condition via a call to MKON$P. See page 5-21 in this book, and the *Subroutines Reference III*, for more information about the MKON$P subroutine.

Example: An example of using signal( ) to catch ABORT$ follows.

```
OK, SLIST TMP1.C
#include <signal.h>
main()
{
    void abort_handler();
    signal(SIGABRT, abort_handler);
    abort();
}
```

```
/* function that is called when abort() is invoked */
void
abort_handler(sig)
    int sig;
{
    printf("abort has been called again\n");
    exit(1);
}
OK, CC TMP1 -32IX
[CI Rev. T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]
00 Errors and 00 Warnings detected in 17 lines and 115 include lines.
OK, BIND -LI CCMAIN -LO TMP1 -LI C_LIB
[BIND Rev. T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]
BIND COMPLETE
OK, R TMP1
abort has been called again
OK,
```

▶ abs( ), fabs( )

Returns the absolute value of an integer and the absolute value of a floating-point number, respectively.

```
int abs(integer)
int integer;

#include <math.h>
double fabs(floating)
double floating;
```

▶ access( )

Checks a specified PRIMOS pathname to determine whether the specified access rights are permitted.

```
#include <stdio.h>
int access(pathname, mode)
char *pathname;
int mode;
```

Zero is returned if the rights are allowed. This function returns -1 either on any error or if the access rights are not allowed. It sets *errno* (defined in stdio.h) to the file system error code. The following modes are allowed:

0 The file exists.

2 The file exists and the process has write access.

4 The file exists and the process has read access.

Combinations of access modes can be specified by summing the above values. For example, 6 indicates ERW (exist/read/write).

▶ acos( )

Returns the arc cosine of its argument. The range of the returned value is from 0 through π radians.

```
#include <math.h>
double acos(x)
double x;
```

▶ asin( )

Returns the arc sine of its argument. The range of the returned value is from - $\pi/2$ through $\pi/2$ radians.

```
#include <math.h>
double asin(x)
double x;
```

▶ assert( )

Adds runtime diagnostics to programs. Available in 32IX mode only.

```
#include <assert.h>
void assert(expression)
int expression;
```

The assert( ) function is implemented as a macro. If the expression argument is false or equal to zero when it is executed, then the information about the failure is written to the standard error file named **stderr**. This information includes the text of the argument, the name of the source file, and the source line number. The format of the message is:

`Assertion failed: "`*expression*`", file "`*pathname*`", line` *line-number*.

The abort( ) function is then called to terminate execution. If, however, the expression is true, then assert( ) has no effect.

Another macro, NDEBUG, is referenced, but not defined, in the ASSERT.H.INS.CC file. If, however, NDEBUG is defined in a user's program at the point in the source file where <assert.h> is included, the assert macro will always be ignored, regardless of its evaluated value.

▶ atan( )

Returns the arc tangent of its argument. The range of the returned value is from $-\pi/2$ through $\pi/2$ radians.

```
#include <math.h>
double atan(x)
double x;
```

▶ atan2( )

Returns a value in the range $-\pi$ through π. The returned value is the arc tangent of x/y, where x and y are the two arguments.

```
#include <math.h>
double atan2(x, y)
double x, y;
```

▶ atof( ), atoi( ), atol( )

Converts strings of ASCII characters to the appropriate numeric values.

```
#include <math.h>
double atof(inputPointer)
char *inputPointer;

int atoi(inputPointer)
char *inputPointer;

long atol(inputPointer)
char *inputPointer;
```

These functions recognize strings in various formats, depending on the returned data type. The string for atof( ) may contain leading white space (space or tab). This is followed by an optional sign, then a string of digits (optionally containing a decimal point), then an optional exponent composed of an e or E, and then an (optionally signed) integer. The first unrecognized character ends the string.

The string for atoi( ) and atol( ) may contain a series of leading tabs and spaces, then an optional sign, and finally a series of digits (with no decimal point). The first unrecognized character ends the string.

These functions do not account for overflows resulting from the conversion. In the 50 Series implementation, **long** is synonymous with **int**; thus atoi( ) and atol( ) are equivalent.

▶ atoi( )

```
int atoi(inputPointer)
char *inputPointer;
```

For more information, see the atof( ) function.

▶ atol( )

```
long atol(inputPointer)
char *inputPointer;
```

For more information, see the atof( ) function.

▶ bio$primosfileunit( )

```
#include <stdio.h>
int bio$primosfileunit(fileID)
int fileID;
```

The function bio$primosfileunit( ) allows you to determine PRIMOS file unit that is being used to access a disk file. Given a *fileID* returned from open( ) or fileno( ), bio$primosfileunit( ) returns the corresponding PRIMOS file unit. For example,

```
#include <stdio.h>
int fileID;
FILE *filePointer;
int primosUnit1, primosUnit2;

fileID = open("aFileName", 2);
filePointer = fopen("anotherFileName", "w");
primosUnit1 = bio$primosfileunit(fileID);
primosUnit2 = bio$primosfileunit(fileno(filePointer));
```

▶ cabs( )

```
#include <math.h>
double cabs(z)
struct {double x, y;} z;
```

For more information, see the hypot( ) function.

▶ calloc( )

This function allocates an area of memory.

```
char *calloc(numberOfElements, elementSize)
unsigned numberOfElements, elementSize;
```

The calloc( ) function allocates space for an array of *numberOfElements* elements of size *elementSize*. If calloc( ) is unable to allocate the space, it returns 0.

For more information, see the malloc( ) function on page 4-34.

▶ ceil( )

Returns as a **double** the smallest integer that is equal to or greater than its argument.

```
#include <math.h>
double ceil(x)
double x;
```

▶ cfree( )

```
int cfree(pointer)
char *pointer;
```

For more information, see the free( ) function.

▶ chdir( )

Changes the current home directory (that is, attaches) to the specified PRIMOS pathname.

```
#include <stdio.h>
int chdir(pathname)
char *pathname;
```

The target pathname is set as the new working directory. This function calls the PRIMOS subroutine AT$ to do the attach. Zero is returned if the change of directory is executed correctly. The routine returns -1 on any error and sets errno (defined in stdio.h) to the file system error code.

▶ chrcheck( )

Returns 1 if a character has been typed but not read. Returns 0 if no character has been typed but not read. If you use chrcheck( ) instead of getc( )/fgetc( ), the program can continue with other processing when there has been no terminal input. (See gterm( ) and sterm( ).)

```
int chrcheck( )
```

▶ clearerr( )

Resets the error and end-of-file indications for a file, so that ferror( ) and feof( ) no longer return a nonzero value. The clearerr( ) function is implemented as a macro.

```
#include <stdio.h>
clearerr(filePointer)
FILE *filePointer;
```

▶ close( )

Closes a file specified by a *fileID*. The *fileID* is the return value from the open( ) function.

```
#include <stdio.h>
int close(fileID)
int fileID;
```

If the file was opened for write or update, any buffered data is written to the file. The function returns 0 if the file is successfully closed. On any error, the function returns a -1 and the file system error code is set in the external variable errno (defined in stdio.h). Use C library routines to close all files opened by C library routines.

▶ copy( )

Copies a file to a new location.

```
#include <stdio.h>
int copy(oldPathname, newPathname)
char *oldPathname;
char *newPathname;
```

Zero is returned if the copy is executed correctly. The function returns -1 on any error and sets errno (defined in stdio.h) to the file system error code. Both specified pathnames may be PRIMOS pathnames.

▶ cos( )

Returns the cosine of the argument expressed in radians.

```
#include <math.h>
double cos(x)
double x;
```

▶ cosh( )

Returns the hyperbolic cosine of the argument.

```
#include <math.h>
double cosh(x)
double x;
```

▶ creat( )

Opens a specified file and assigns specified access rights to the file. See open( ).

Note

You are strongly advised to use open( ) instead of creat( ) in the PRIMOS environment. The open( ) function creates files opened for write or read/write if they do not previously exist.

```
#include <stdio.h>
int creat(pathname, createMode)
char *pathname;
int createMode;
```

If the specified file does not exist, it is created. If the file already exists, its length is truncated to 0. The file is opened for binary read/write. If the call executes correctly, the function returns an integer *fileID*. This *fileID* may then be supplied as an argument to routines such as read( ), write( ), lseek( ) and close( ). On any error, the function returns -1 and sets errno (defined in stdio.h) to the file system error code. The values for *createMode* are discussed with the setmod( ) and getmod( ) functions.

▶ ctime( )

Converts a time in seconds since 00:00:00 Jan. 1, 1970 to a 30-byte ASCII string of the form ***DD MMM YY HH:MM:SS <day-of-week>\n\0***.

```
char *ctime(seconds)
int *seconds;
```

The argument to ctime( ) is a pointer to the time to be converted. (This can be obtained from the time( ) routine.) ctime( ) returns a pointer to a 30-byte ASCII string, which contains the result.

▶ cuserid( )

Returns a pointer to a character string containing the user ID of the current process.

```
#include <stdio.h>
char *cuserid(string)
char *string;
```

If the argument is null, the user name is stored internally. If not null, the argument must point to a storage area of length L_cuserid (defined in stdio.h), and the name is written into that storage area.

▶ delete( )

Deletes a specified file.

```
#include <stdio.h>
int delete(pathname)
char *pathname;
```

The file to be deleted may be specified by a fully qualified PRIMOS pathname. Zero is returned if the file is deleted correctly. The function returns -1 on any error and sets errno (defined in stdio.h) to the file system error code.

▶ ecvt( ), fcvt( )

Converts a **double** value to a NULL-terminated string of ASCII digits and returns the address of the string.

```
char *ecvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *fcvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;
```

In both functions, *value* is the double precision value to be converted, and *ndigit* is the number of ASCII digits (not including the terminating NULL) to be used in the converted string. Calls to these functions overwrite any existing string. The integer pointed to by **decpt* returns the position of the decimal point relative to the first character in the returned string. A negative **decpt* value means that the decimal point is to the left of the returned digits, and a 0 means that the decimal point is immediately to the left of the first digit. The integer pointed to by **sign* is set to nonzero if *value* was negative; otherwise, **sign* is set to zero.

The following example uses the ecvt( ) function to convert a **double** value called `dblval` and prints the information returned.

```
/* ECVT EXAMPLE */
#include <stdio.h>
main( )

{

char *ecvt( );

double dblval;          /* Value to be converted */

int sign, point;        /* Output for sign, decimal point */

static char string[20]; /* Array for converted string */

dblval = -4.6389240e-4;

printf("input value: %e\n", dblval);
strcpy(string, ecvt(dblval, 6, &point, &sign));
printf("converted string: %s\n", string);
printf("value is %s\n", (sign) ? "negative" : "positive");
printf("decimal point is at position %d\n", point);

}
```

The output of the program is

```
input value : -4.6389240e-4
converted string: 463892
value is negative
decimal point is at position -3
```

▶ exit( )

Terminates a user process and returns to PRIMOS.

```
exit(status)
int status;
```

The exit( ) function returns the specified status to PRIMOS. This function also flushes all buffers and closes all open files before performing the exit.

▶ exp( )

Returns base e raised to the power given by the argument.

```
#include <math.h>
double exp(x)
double x;
```

▶ fabs( )

```
#include <math.h>
double fabs(floating)
double floating;
```

For more information, see the abs( ) function.

▶ fclose( )

Closes a file, flushing any buffers associated with the file pointer.

```
#include <stdio.h>
int fclose(filePointer)
FILE *filePointer;
```

This function returns 0 on success. If the buffered data cannot be written to the file, or if the file control block is not associated with an open file, fclose( ) returns EOF (a preprocessor constant defined in stdio.h).

▶ fcvt( )

```
char *fcvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;
```

For more information, see the ecvt( ) function.

▶ fdopen( )

Associates a file pointer with a *fileID* returned by the open( ) or creat( ) functions.

```
#include <stdio.h>
FILE *fdopen(fileID, accessMode)
int fileID;
char *accessMode;
```

The fdopen( ) function allows you to access a file originally opened by a call to open( ) or creat( ) as if it had been opened by a call to fopen( ). Generally, a file can be accessed either by *fileID* if opened by open( ) or creat( ), or by *filePointer* if opened by fopen( ). A file cannot be accessed by both *fileID* and *filePointer*.

The first argument to fdopen( ) is the *fileID* returned by open( ) or creat( ). The second argument is the same as the second argument to fopen( ). This access mode must agree with the original mode with which the file was opened. If the operation is completed

successfully, a nonzero file descriptor is returned. The fdopen( ) function returns NULL (defined in stdio.h) on any error and sets errno (defined in stdio.h) to the file system error code. The values of *accessMode* are discussed with fopen( ).

▶ fdtm( )

Returns the modification time for the specified file. The time is as it would be returned by the time( ) function.

```
#include <stdio.h>
int fdtm(pathname)
char *pathname;
```

The fdtm( ) function returns -1 on any error and sets errno (defined in stdio.h) to the file system error code.

▶ feof( )

Tests a file to see if the end of file has been reached. If so, feof( ) returns a nonzero integer; if not, it returns 0. The feof( ) function is implemented as a macro.

```
#include <stdio.h>
int feof(filePointer)
FILE *filePointer;
```

A call to this function continues to return a nonzero integer until the file is closed or until clearerr( ) is called.

▶ ferror( )

Returns a nonzero integer if an error occurred while the file was being written or read.

```
#include <stdio.h>
int ferror(filePointer)
FILE *filePointer;
```

A call to this function continues to return a nonzero integer until the file is closed or until clearerr( ) is called. The ferror( ) function is implemented as a macro.

▶ fexists( )

Returns 1 if the specified PRIMOS pathname exists and 0 if it does not.

```
#include <stdio.h>
int fexists(pathname)
char *pathname;
```

The fexists( ) function returns -1 on any error and sets errno (defined in stdio.h) to the file system error code. The pathname may terminate with either a filename or a directory name.

▶ fflush( )

Writes out any buffered information for the specified file. Output files are normally buffered only if they are not directed to a terminal.

```
#include <stdio.h>
int fflush(filePointer)
FILE *filePointer;
```

The smallest data item that can be written to a PRIMOS disk file is a 16-bit item; thus, if there is an odd number of bytes in the buffer when fflush( ) is called, the last byte is not dumped to disk. When fflush( ) returns, the buffer still contains the leftover byte. In order to force this byte out to disk with a zero-padding byte, the file may be closed. The fflush( ) function returns 0 when it is successful. If the buffered data cannot be written to the file, or if the file control block is not associated with an output file, fflush( ) returns EOF (a preprocessor constant defined in stdio.h).

▶ fgetc( )

```
#include <stdio.h>
int fgetc(filePointer)
FILE *filePointer;
```

For more information, see the getc( ) function.

▶ fgetname( )

```
#include <stdio.h>
char *fgetname(filePointer, buffer)
FILE *filePointer;
char *buffer;
```

For more information, see the getname( ) function.

▶ fgets( )

```
#include <stdio.h>
char *fgets(string, maxline, filePointer)
char *string;
int maxline;
FILE *filePointer;
```

For more information, see the gets( ) function.

▶ fileno( )

Returns an integer *fileID*. The fileno( ) function is implemented as a macro.

```
#include <stdio.h>
int fileno(filePointer)
FILE *filePointer;
```

The argument *filePointer* is a file pointer returned by fopen( ).

▶ floor( )

Returns (as a **double**) the largest integer that is less than or equal to its argument.

```
#include <math.h>
double floor(x)
double x;
```

▶ fopen( )

Opens a file, returning the address of a FILE structure, denoting a file control block.

```
#include <stdio.h>
FILE *fopen(pathname, accessMode)
char *pathname, *accessMode;
```

The file pointer (type FILE *) returned by fopen( ) may be used as an argument to the following functions: clearerr( ), fclose( ), feof( ), ferror( ), fflush( ), fgetc( ), fgetname( ), fgets( ), fileno( ), fprintf( ), fputc( ), fputs( ), fread( ), freopen( ), fscanf( ), fseek( ), fstat( ), ftell( ), fwrite( ), getc( ), geth( ), getname( ), getw( ), putc( ), puth( ), putw( ), rewind( ), setbuf( ), and ungetc( ).

The file control block may be freed with the fclose( ) function or by default on normal program termination (a call to exit( )).

The first argument to fopen( ) is a character string containing a valid PRIMOS pathname. The second argument, *accessMode*, is one of the character strings listed in Table 4-2.

Output may not be directly followed by input without an intervening call to fflush( ) or to one of the file positioning functions fseek( ) and rewind( ). Similarly, input may not be directly followed by output without an intervening call to the fflush( ) function or to a file positioning function, unless the input operation encounters the end of file.

An ASCII file is a file in PRIMOS standard text format, that is, an editable file. Space compression is used in an ASCII file, and the new line (\n) at the end of a line may be padded with a NULL (\0) byte to make each record contain an even number of bytes. A binary file can contain arbitrary data with no translation done for either input or output. Data written to ASCII files may be changed/compressed as it is actually written to disk; however, as the file is read back in, these changes are undone and the data appears exactly as it was written. The disk format of a binary file is exactly what was written.

The **r** or **i** character strings open an existing file for input. Conversely, the **w** or **o** character strings open a file for output. If the file does not exist, fopen( ) creates a new file. If the file exists, fopen( ) truncates the file. The **wa** and **oa** character strings are virtually the same as **w** and **o**, except that the initial file position is set to the end of file (no truncation occurs).

TABLE 4-2. Character Strings for fopen

Character String	*Action Performed*
r	Reads ASCII
w	Writes ASCII
i	Reads binary
o	Writes binary
wa	Writes ASCII append
oa	Writes binary append
i+	Updates binary
o+	Updates binary (truncate when opening)
oa+	Updates binary append

The "i+" character string opens a file for read/write with the initial position at the beginning of the file. The "o+" character string opens a file for read and write and initially truncates the file. The "oa+" character string opens a file for read and write with the initial position at the end of file.

Note

The smallest unit of data that can be written to a PRIMOS disk file is a 16-bit entity. The f routines (for example, fread( ) and fopen( )) attempt to hide this fact from the user. (See the comments in the fflush( ) documentation.)

If fopen( ) is forced to create a file (opening a write file that does not previously exist), it creates a DAM file.

The function returns a null pointer (defined in stdio.h) to signal errors. Use C library routines to close all files opened with C library routines.

▶ fprintf( )

```
#include <stdio.h>
int fprintf(filePointer,
            formatSpecification [, outputSource,. . . ])
FILE *filePointer;
char *formatSpecification;
```

For more information, see the printf( ) function.

▶ fputc( )

```
#include <stdio.h>
int fputc(character, filePointer)
char character;
FILE *filePointer;
```

For more information, see the putc( ) function.

▶ fputs( )

```
int fputs(string, filePointer)
char *string;
FILE *filePointer;
```

For more information, see the puts( ) function.

▶ fread( )

Reads a specified number of items from the file.

```
#include <stdio.h>
int fread(pointer, sizeOfItem, numberOfItems, filePointer)
char *pointer;
int numberOfItems, sizeOfItem;
FILE *filePointer;
```

The first argument, *pointer*, points to a buffer into which data is read from the file pointed to by the fourth argument, *filePointer*. The reading of the specified items begins at the current location in the file. The items read are placed in storage beginning at the location given by the first argument. The second argument, *sizeOfItem*, specifies the size of an item in bytes. The function returns the number of items actually read. If fread( ) encounters the end of file or an error, it returns 0 (not EOF).

▶ free( ), cfree( )

Makes available for reallocation the area allocated by a previous calloc( ), malloc( ), or realloc( ) call.

```
int free(pointer)
char *pointer;

int cfree(pointer)
char *pointer;
```

The argument is the address returned by a previous call to malloc( ), calloc( ), or realloc( ). The functions return 0 if the area is successfully freed, -1 if an error occurs.

Note

The C library's routines for dynamic memory management (malloc( ), calloc( ), realloc( ), free( ), and cfree( )) are designed for use only with each other. If you allocate memory with code written in another language, do not deallocate it with a C routine. Similarly, if you allocate memory with a C routine, do not deallocate it with code written in another language.

▶ freopen( )

Substitutes the file specified by a pathname for the open file addressed by a file pointer. The latter file is closed.

```
#include <stdio.h>
FILE *freopen(pathname, accessMode, filePointer)
char *pathname, *accessMode;
FILE *filePointer;
```

The freopen( ) function is typically used to associate one of the predefined names stdin, stdout, or stderr with a file.

The first two arguments to freopen( ) have the same meaning as the arguments to fopen( ). The third argument is a pointer to a FILE structure, denoting a currently open file. After the function call, the open file is closed.

If the attempt to reopen fails, the function returns a null pointer (defined in stdio.h); otherwise, the function returns the address of the reopened file control block, which is the third argument.

▶ frexp( )

Returns the mantissa and exponent of a **double** argument.

```
#include <math.h>
double frexp(value, eptr)
double value;
int *eptr;
```

The mantissa is a **double**, and its magnitude is less than one. The second argument is a pointer to an **int**, to which frexp( ) returns an integer *n* such that *value* = mantissa * 2^n.

▶ frwlock( )

Returns the current read/write lock for the specified file. The read/write lock is discussed in the *PRIMOS Commands Reference Guide* under the commands COPY and RWLOCK.

```
#include <stdio.h>
int frwlock(pathname)
char *pathname;
```

The frwlock( ) function returns -1 on any error and sets errno (defined in stdio.h) to the file system error code.

The valid return values are

0	System default
1	N readers or one writer
2	N readers and one writer
3	N readers and N writers

▶ fscanf( )

```
#include <stdio>
int fscanf(filePointer, formatSpecification [, inputPointer,. . . ])
FILE *filePointer;
char *formatSpecification;
```

For more information, see the scanf( ) function.

▶ fseek( )

Positions the file to the specified byte offset in the file.

```
#include <stdio.h>
int fseek(filePointer, offset, direction)
FILE *filePointer;
int offset, direction;
```

The argument *direction* is an integer indicating whether the offset is measured from the current read or write address (1), from the beginning of the file (0), or from the end of the file (2). The fseek( ) function returns EOF (a preprocessor constant defined in stdio.h) for improper seeks, 0 for successful seeks. To position into ASCII files that have been opened for writing or updating, a previous call to ftell( ) must have been made to obtain a valid byte position in the disk file. The only operations that can be performed successfully on an ASCII file opened for reading are seeks to the beginning or end of the file where the offset is zero. Any other operation causes fseek( ) to return EOF. Arbitrary seeks on binary files are permitted. See the fopen( ) description.

Note

Under PRIMOS, ASCII text files are stored on disk with as many as 128 spaces stored in only two bytes. The C library routines generally hide this fact by compressing the data on the way out to disk and expanding it on the way in from disk. This compression can cause problems if you update a file after it has been written. For example, you cannot write the string `"abcedf"` on top of six spaces in an ASCII disk file without overwriting data following the spaces, because only two bytes on the disk file are used to store six spaces. Additionally, space compression causes problems when reading ASCII files because the 32-bit offset is not large enough to always store a unique file position. These problems are, of course, solved by using binary files rather than ASCII files.

▶ fsize( )

Returns the size of a specified file in bytes.

```
int fsize(pathname)
char *pathname;
```

The fsize( ) function opens the file and positions to the end of file to perform this calculation. (This can be a time-consuming task for large SAM files.) The function returns -1 on any error and sets errno (defined in stdio.h) to the file system error code.

▶ fstat( )

This function is equivalent to the following stat( ) call:

```
stat(fgetname(filePointer, charBuf), buffer)
```

```
#include <stdio.h>
#include <stat.h>
int fstat(filePointer, buffer)
FILE *filePointer;
struct stat *buffer;
```

The information returned by fstat( ) is virtually identical to the information returned by stat( ); however, fstat( ) returns the status information for an already open file, while stat( ) returns the same information for a specified pathname. The first argument to fstat( ) is a file pointer as returned by the fopen( ) routine. See the stat( ) function for more information.

▶ ftell( )

Returns the current byte offset to the specified stream file.

```
#include <stdio.h>
int ftell(filePointer)
FILE *filePointer;
```

The offset is measured from the beginning of the file. This function is useful only for obtaining an offset that is later passed to fseek( ). Any error causes ftell() to return EOF (a preprocessor constant defined in stdio.h). Note that ftell( ) cannot reliably measure offsets into ASCII files opened for reading, and will return EOF. For more information, see the fseek( ) description.

▶ ftime( )

Returns the elapsed time since 00:00:00, Jan. 1, 1970, in a timeb structure. The structure layout is as follows (structure defined in timeb.h):

```
struct timeb { int time;        /* Time in seconds */
               short millitm;   /* Fractional milliseconds */
               short timezone;  /* Always zero */
               short dstflag;   /* Always zero */
             } ;
```

```
#include <timeb.h>
ftime(timePointer)
struct timeb *timePointer;
```

▶ ftype( )

Returns the type of a specified file.

```
#include <stdio.h>
int ftype(pathname)
char *pathname;
```

The ftype( ) function returns -1 on any error and sets errno (defined in stdio.h) to the file system error code. The valid return values are

Value	*Meaning*
0	SAM file
1	DAM file
2	SAM segment directory
3	DAM segment directory
4	Directory
6	Access Category (ACAT)
7	CAM file

▶ fwrite( )

Writes a specified number of items to the file.

```
#include <stdio.h>
int fwrite(pointer, sizeOfItem, numberOfItems, filePointer)
char *pointer;
int numberOfItems, sizeOfItem;
FILE *filePointer;
```

The first argument, *pointer*, points to a buffer from which data items are written. *sizeOfItem* is the size in bytes. The fourth argument, *filePointer*, is the pointer that was returned by the function fopen( ) or fdopen( ). The writing begins at the current location in the file. The function returns the number of items actually written. It returns 0 if there is an error.

▶ g$amiix( )

Determines if the current machine is capable of executing C 32IX-mode code.

int g$amiix( )

This routine is an integer function that returns true (1) if the current machine is capable of executing C 32IX-mode code; it returns false (0) otherwise. Call g$amiix( ) from 64V-mode C code only.

▶ getc( ), fgetc( )

Returns the next character as an **int** from a specified file.

```
#include <stdio.h>
int getc(filePointer)
FILE *filePointer;

#include <stdio.h>
int fgetc(filePointer)
FILE *filePointer;
```

The getc( ) function positions the file after the character is returned, and the next getc( ) call takes the character from that position. The getc( ) function is implemented as a macro. The argument, *filePointer*, is the pointer that was returned by the function fopen( ) or fdopen( ).

The fgetc( ) function is almost identical to the getc( ) function; however, the fgetc( ) function generates an actual function call and not a macro substitution.

Normally, when a program is reading from a terminal, input is not available until the user types a newline. If the terminal is in raw mode, however, the program can read each character as it is typed. See sterm( ), gterm( ), and chrcheck( ).

▶ getchar( )

Returns the next character from the standard input and is identical to using getc(stdin). This function is implemented as a macro.

```
#include <stdio.h>
int getchar( )
```

▶ geth( )

Similar to getw( ), except that the next two characters are read from the file and returned as an **int**. (Sign extension occurs.)

```
#include <stdio.h>
int geth(filePointer)
FILE *filePointer;
```

The argument, *filePointer*, is the pointer that was returned by the function fopen( ) or fdopen( ). The getc( ), fgetc( ), getchar( ), getw( ), and geth( ) functions all return EOF on end of file or error, but because EOF is an integer, you must use feof( ) and ferror( ) to check the success of getw( ) or geth( ).

▶ getmod( )

Returns the access available to the current user to a specified file or directory.

```
#include <stdio.h>
int getmod(pathname, user)
char *pathname, *user;
```

If the specified file is ACL protected, all return bits are valid. If the file is password protected, then only the read, write, and delete bits are valid. The second argument is usually ignored in the current implementation. The only exception is that for a file in a password protected directory, the user name can be specified as _non-owner_; then getmod( ) returns the access held by a nonowner to the file. The function returns -1 on any error and sets errno (defined in stdio.h) to the file system error code.

getmod( ) returns the following bit settings:

Bit Setting	*Meaning*
01	Read
02	Write
04	Use
010	List
020	Add
040	Delete
0100	Protect

▶ getname( ), fgetname( )

Returns the PRIMOS pathname associated with an integer *fileID* (getname( )) or a file pointer (fgetname( )).

```
#include <stdio.h>
char *getname(fileID, buffer)
int fileID;
char *buffer;
#include <stdio.h>
char *fgetname(filePointer, buffer)
FILE *filePointer;
char *buffer;
```

fileID is the integer returned by open( ), creat( ), or fileno( ). *filePointer* is the pointer returned by fopen( ) or fdopen( ). Both functions place the filename in *buffer* and return the address of *buffer*. The filename is padded with one NULL to form a correct C string. The buffer must be at least 129 bytes in length. If an error occurs, both functions return NULL and set errno (defined in stdio.h) to the file system error code.

▶ gets( ), fgets( )

Reads a line from a specified file.

```
#include <stdio.h>
char *gets(string)
char *string;

#include <stdio.h>
char *fgets(string, maxline, filePointer)
char *string;
int maxline;
FILE *filePointer;
```

The gets( ) function reads a line from standard input into the buffer specified by *string*. This function replaces the newline character (\n) with a NULL (\0). The gets( ) function returns its argument, which is a pointer to a character string containing the acquired line. If an error occurs or if an EOF is encountered before the newline character is encountered, the function returns NULL (defined in stdio.h). The fgets( ) function reads from the file until it has read a newline character (\n) or until it has read *maxline* - 1 characters, whichever comes first. The function puts the characters into the buffer *string*. *filePointer* is a value of type FILE * that was returned by fopen( ) or fdopen( ).

The fgets( ) function terminates the line with NULL (\0). Unlike gets( ), fgets( ) places the newline that terminates the input record into the user buffer as it fits. On end of file or error, the functions return NULL (defined in stdio.h); otherwise, they return the address of the first character in the line.

▶ getw( )

Returns the next four characters from a specified input file as an **int** value. No type conversion is performed.

```
#include <stdio.h>
int getw(filePointer)
FILE *filePointer;
```

If the getw( ) function encounters an end of file (EOF) during the retrieval of any of the four characters, the EOF (a preprocessor constant defined in stdio.h) is returned, and the four characters are lost.

▶ gterm( )

Obtains the current terminal characteristics and puts them in the specified structure.

```
#include <term.h>
void gterm(buffer)
struct term *buffer;
```

All terminal attributes can be set with the sterm( ) function. Read and Write mode (RAW) indicates that each character can be read (with fgetc( )) as it is typed without waiting for a terminating line feed. The structure and flag bits (defined in term.h) are as follows:

```
                              /* Flag bits              */

#define BREAK  01             /* Break enabled?         */
#define CRLF   02             /* Echo LF after CR?      */
#define ECHO   04             /* Echo characters?       */
#define RAW    010            /* Single character reads? */
#define XON    020            /* Flow control enabled   */

struct term { short tt_flags; /* Flag bits              */
              char  tt_erase; /* Erase character        */
              char  tt_kill;  /* Kill character         */
            };
```

▶ gvget( )

Returns a pointer to a static character array that contains the value of the named PRIMOS global variable set by the gvset( ) routine or the PRIMOS command DEFINE_GVAR.

```
#include <stdio.h>
char *gvget(name)
char *name;
```

Returns 0 on any error or if the variable is undefined, and it sets errno (defined in stdio.h) to the file system error code. gvget( ) is an interlude to the PRIMOS subroutine GV$GET.

▶ gvset( )

The **gvset( )** function changes the value of a PRIMOS global variable.

```
#include <stdio.h>
int gvset(name, value)
char *name, *value;
```

If the specified name does not exist, the name is created as a global variable. Zero is returned if the function is executed successfully. gvset( ) returns -1 on any error and sets

errno (defined in stdio.h) to the file system error code. This function is an interlude to the PRIMOS subroutine GV$SET.

▶ hypot( ), cabs( )

hypot( ) returns sqrt($x^2 + y^2$). cabs( ) returns the complex absolute value sqrt($z.x^2 + z.y^2$).

```
#include <math.h>
double hypot(x, y)
double x, y;

double cabs(z)
struct {double x, y;} z;
```

▶ index( )

```
#include <string.h>
char *index(string, character)
char *string, character;
```

For more information, see the strchr( ) function.

▶ isalnum( )

Returns a nonzero integer if its argument is one of the alphanumeric ASCII characters; otherwise, it returns zero. This function is implemented as a macro.

```
#include <ctype.h>
int isalnum(character)
char character;
```

▶ isalpha( )

Returns a nonzero integer if its argument is an alphabetic ASCII character; otherwise, it returns zero. This function is implemented as a macro.

```
#include <ctype.h>
int isalpha(character);
char character;
```

▶ isascii( )

Returns a nonzero integer if its argument is any ASCII character (value less than 0400 octal); otherwise, it returns zero. This function is implemented as a macro.

```
#include <ctype.h>
int isascii(character)
char character;
```

▶ isatty( )

Returns 1 if the current process is running from a terminal; zero if not.

```
int isatty(fileID)
int fileID;
```

The required argument is a dummy argument; it need not be the actual *fileID* of the process.

▶ iscntrl( )

Returns a nonzero integer if its argument is an ASCII DEL character (0177 or 0377 octal) or any nonprinting ASCII character (code between 00 and 040 octal or between 0200 and 0240 octal). Zero is returned otherwise. This function is implemented as a macro.

```
#include <ctype.h>
int iscntrl(character)
char character;
```

▶ isdigit( )

Returns a nonzero integer if its argument is a decimal digit character in the range 0 through 9. Returns zero if not. This function is implemented as a macro.

```
#include <ctype.h>
int isdigit(character)
char character;
```

▶ isgraph( )

Returns a nonzero integer if its argument is a graphic ASCII character; otherwise, it returns zero. Graphic characters are not control characters and are not the space characters (040 and 0240). This function is implemented as a macro.

```
#include <ctype.h>
int isgraph(character)
char character;
```

▶ islower( )

Returns a nonzero integer if its argument is a lowercase alphabetic ASCII character; otherwise it returns zero. This function is implemented as a macro.

```
#include <ctype.h>
int islower(character)
char character;
```

▶ ispascii( )

Returns a nonzero integer if its argument is any valid Prime ASCII character in the range 0200 through 0377 octal; otherwise, it returns zero. This function is implemented as a macro.

```
#include <ctype.h>
int ispascii(character)
char character;
```

▶ isprint( )

Returns a nonzero integer if its argument is any ASCII printing character. ASCII printing characters have values from 040 through 0176 octal and from 0240 through 0376 octal. The routine returns zero otherwise. This routine is implemented as a macro.

```
#include <ctype.h>
int isprint(character)
char character;
```

▶ ispunct( )

Returns a nonzero integer if its argument is an ASCII punctuation character; that is, if it is nonalphanumeric and greater than 040 and less than 0177 octal or greater than 0240 and less than 0377 octal. It returns zero otherwise. This function is implemented as a macro.

```
#include <ctype.h>
int ispunct(character)
char character;
```

▶ isspace( )

Returns a nonzero integer if its argument is white space, that is, if it is an ASCII space, tab, RETURN, form feed, or newline character. It returns zero otherwise. This function is implemented as a macro.

```
#include <ctype.h>
int isspace(character)
char character;
```

▶ isupper( )

Returns a nonzero integer if its argument is an uppercase alphabetic ASCII character; otherwise, it returns zero. This function is implemented as a macro.

```
#include <ctype.h>
int isupper(character)
char character;
```

▶ isxdigit( )

Returns a nonzero integer if its argument is a hexadecimal digit (0-9, A-F, a-f); otherwise, it returns zero. This function is implemented as a macro.

```
#include <ctype.h>
int isxdigit(character)
char character;
```

▶ ldexp( )

Returns the following quantity: *value* times 2 to the power of *exp*.

```
#include <math.h>
double ldexp(value, exp)
double value;
int exp;
```

▶ localtime( )

Converts a time (as returned from the time( ) function) to a time structure.

```
#include <time.h>
struct tm *localtime(seconds)
int *seconds;
```

The localtime( ) function returns a pointer to the time structure; successive calls overwrite the structure. The following is the structure layout (structure defined in time.h):

```
struct tm { int tm_sec,    /* seconds */
                tm_min,    /* minutes */
                tm_hour,   /* hours (24) */
                tm_mday,   /* day in month (1-31) */
                tm_mon,    /* month (0-11) */
                tm_year,   /* year (00-99) */
                tm_wday,   /* day in week (0-6) */
                tm_yday,   /* day in year (0-365) */
                tm_isdst;  /* 0 */
          };
```

▶ log( )

Returns the natural (base e) logarithm of the argument, which must be of type **double.** (The returned value is also **double.**)

```
#include <math.h>
double log(x)
double x;
```

▶ log10( )

Returns the base 10 logarithm of the argument, which must be of type **double.** (The returned value is also **double.**)

```
#include <math.h>
double log10(x)
double x;
```

▶ longjmp( )

```
#include <setjmp.h>
longjmp(env, val)
jmp_buf env;
```

For more information, see the setjmp( ) function.

▶ lsdir( )

Returns a pointer to a static character array containing the next filename in an open directory.

```
#include <stdio.h>
char *lsdir(fileID)
int fileID;
```

The directory is specified by the integer value *fileID* returned from open( ), creat( ), or fileno( ). If you pass the negative integer *-fileID*, lsdir( ) positions the directory to entry 0 before the next name is read. The function returns 0 on any error and sets errno (defined in stdio.h) to the file system error code.

▶ lseek( ), seek( )

Positions a file to an arbitrary byte position and returns the new position as an **int**.

```
#include <stdio.h>
int lseek(fileID, offset, direction)
int fileID, offset, direction;

#include <stdio.h>
int seek(fileID, offset, direction)
int fileID, offset, direction;
```

These functions set the position relative to the beginning of the file (*direction* = 0 or 3), the current position (*direction* = 1 or 4), or the end of file (*direction* = 2 or 5). The target byte position is specified by the *offset* argument. *direction*s of 3, 4, and 5 cause the *offset* to be multiplied by 2048 before the positioning is performed. The size of a physical disk record is 2048 bytes on 50 Series systems, and is, thus, system dependent. The target file is specified by a *fileID* returned from open( ).

For disk devices, lseek( ) returns the new byte offset within the file, or -1 on any error. For magnetic tape devices, lseek( ) returns 0 if the operation was successful and -1 on error. In either error case, the external variable errno (defined in stdio.h) is set to the PRIMOS error code.

seek( ) and lseek( ) are valid operations for disk files and magnetic tape devices, but not for TTY devices or asynchronous lines. lseek( ) requests for tape devices are specified in records instead of bytes. 0 and 1 are the only *direction* keys allowed for lseek( ) on magnetic tape devices. Their meanings are as follows:

0	Record from beginning of tape
1	Record from current position

For more information, see the description of the open( ) function on page 4-36.

▶ malloc( )

Allocates a contiguous area of memory whose size in bytes is supplied as an argument.

```
char *malloc(size)
unsigned size;
```

The malloc( ) function returns the address of the first byte, which is aligned on a 16-bit boundary. A value of 0 is returned if malloc( ) is unable to allocate enough memory.

When you call the library routines malloc( ) and calloc( ) in 32IX mode, you *must* declare them as returning pointer types.

Note

The C library's routines for dynamic memory management (malloc( ), calloc( ), realloc( ), free( ), and cfree( )) are designed for use only with each other. If you allocate memory with code written in another language, do not deallocate it with a C routine. Similarly, if you allocate memory with a C routine, do not deallocate it with code written in another language.

▶ mkdir( )

Creates a specified directory (this may be a PRIMOS pathname).

```
#include <stdio.h>
int mkdir(pathname)
char *pathname;
```

The new directory has default protections, which can be altered with setmod( ). mkdir( ) returns 0 if the directory is created successfully. The function returns -1 on any error and sets *errno* (defined in stdio.h) to the file system error code.

▶ modf( )

Returns the positive fractional part of a specified **double** and stores the integer part in the **double** pointed to by *integerPart*.

```
#include <math.h>
double modf(value, integerPart)
double value, *integerPart;
```

▶ move( )

Moves a specified file to a specified new location.

```
#include <stdio.h>
int move(oldPathname, newPathname)
char *oldPathname, *newPathname;
```

This function performs a change of name if the old and new pathnames refer to the same directory; otherwise, it performs a copy and delete operation. The function returns -1 on all errors and sets errno (defined in stdio.h) to the file system error code.

▶ open( )

Opens a specified file.

```
#include <stdio.h>
int open(pathname, openMode,
         [fileUnit])

int open(pathname, openMode,
         protocol, config, lword)

int open(pathname, openMode,
         magTapeOptions)

char *pathname;
int  openMode;
int  fileUnit;
char *protocol;
int  config, lword;
char *magTapeOptions;
```

If successful, open( ) returns an integer *fileID*. You use *fileID* as an argument to the following functions: bio$primosfileunit( ), close( ), fdopen( ), fgetname( ), getname( ), lsdir( ), lseek( ), read( ), seek( ), tell( ) and write( ).

If unsuccessful, open( ) returns -1 and the global variable errno (defined in stdio.h) is set to the PRIMOS error code.

pathname is of type **char** *. It may be specified by any of the following character strings:

pathname
A normal PRIMOS style pathname specifying a disk file. The special name "_current-dir_" may be used to open the current directory for reading. All keys are ignored in this case. For compatibility with previous releases, you may use "_current-ufd_" as a synonym for "_current-dir_".

"Device=TTY"
Device type TTY, specifying the current user's terminal.

"Device=ASYNCxxx"
Device type ASYNC, specifying an assignable asynchronous line, where *xxx* is the decimal line number. The asynchronous line is assigned by the open( ) call and unassigned by its corresponding close( ) call. You can disable automatic assigning and unassigning by using the 04000 additive key. Use the 01000000 additive key to disable unassigning only.

"Device=MTx"

Device type tape drive, specifying an available magnetic tape device, where *x* is the tape unit number. The tape drive is assigned by the open( ) call and unassigned by its corresponding close( ) call. You can disable automatic assigning and unassigning by using the 04000 additive key. Use the 01000000 additive key to disable unassigning only. PRIMOS prints a message when a tape device is assigned or unassigned by the C libraries. For example, opening and closing MT0 produces the messages `Device MT0 assigned` and `Device MT0 released`.

The values for the *openMode* argument include several additive keys. These keys are octal numbers representing bit patterns that can be ORed together with other additive keys. You must retain the initial zero so that the C compiler interprets them as octal numbers. The values for *openMode* appear in Table 4-3.

TABLE 4-3. Values for the openMode Argument of open

Value	*Meaning*
0	Open for reading.
1	Open for writing. A binary file opened for write only is actually opened for read/write because for certain I/O operations, PRIMOS must read in a halfword (16-bit) quantity to write out a byte quantity. To write one byte to an existing file, one halfword must be read in so that the halfword containing the new character can be written.
2	Open for reading and writing. This key is not valid for magnetic tape devices.
-1	Open for reading. The compiler assumes that the disk file is already open on PRIMOS file unit *fileUnit*. No additive keys are allowed when this *openMode* is used. Whenever possible, use additive key 02000 rather than *openMode* -1.
-2	Open for writing. The compiler assumes that the disk file is already open on PRIMOS file unit *fileUnit*. No additive keys are allowed when this *openMode* is used. Whenever possible, use additive key 02000 rather than *openMode* -2.
-3	Open for reading and writing. The compiler assumes that the disk file is already open on PRIMOS file unit *fileUnit*. No additive keys are allowed when this *openMode* is used. Whenever possible, use additive key 02000 rather than *openMode* -3.
0100	Additive key to enable no-wait mode I/O. This key is valid only for TTY, asynchronous, and magnetic tape devices.
0200	Additive key to cause mapping of \n to \n\r on output. This key is valid only for TTY and asynchronous devices.
0400	Additive key to cause truncation of an already existing disk file when it is opened for writing, that is, with *openMode* 1, 2, -2, or -3. This key is valid only for disk devices. If this key is used with additive key 02000 (or *openMode*s -2 or -3), the file is truncated at its current position, rather than at the beginning of file.

TABLE 4-3. Values for the openMode Argument of open (continued)

Value	*Meaning*
01000	Additive key to disable disk write buffering. This causes all write( ) and seek( ) requests to be flushed immediately to PRIMOS, with the possible exception of a single odd byte at the end of file. This key is valid only for disk devices.
02000	Additive key to signal that a disk file is already open on PRIMOS file unit *fileUnit* for the specified *openMode* (0, 1, or 2). The current file position is not altered. This key is valid only for disk devices. Use this key rather than -1, -2, or -3 whenever possible. Other additive keys may be used with this key.
04000	Additive key to disable automatic assigning (on open( )) and unassigning (on close( )) of asynchronous and magnetic tape devices. This key is valid only for asynchronous and magnetic tape devices.
010000	Additive key to cause a SAM file to be created if key 1 or 2 are used and the specified file does not already exist. Creation of a new DAM file is the default. This key is valid only for write or read/write open( ) requests for disk devices.
020000	Additive key to cause a CAM file to be created if key 1 or 2 is used and the specified file does not already exist. Creation of a new DAM file is the default. This key is valid only for write or read/write open( ) requests for disk devices.
040000	Additive key to disable disk read buffering. This causes all read( ) requests to come directly from PRIMOS rather than through a local buffer. Similarly, all seek( ) requests are executed immediately instead of being deferred until the next physical disk I/O operation. This key is useful if one process is reading a file that is being concurrently written by another process and the most up-to-date data must be available for reading at all times. This key is valid only for disk devices.
0200000	Additive key to cause a magnetic tape device to be rewound when closed. This key is valid only for magnetic tape devices.
0400000	Additive key to cause a magnetic tape device to be unloaded when closed. This key is valid only for magnetic tape devices.
01000000	Additive key to disable unassigning of asynchronous or magnetic tape devices when they are closed. This key causes the device to be assigned when it is opened, but leaves it assigned to the user process after it is closed. This is useful for writing multiple tape marks to magnetic tape devices. This key is valid only for magnetic tape devices.
02000000	Additive key to signal that the optional arguments *protocol*, *config* and *lword* are present. This key causes open( ) to use these arguments for asynchronous device assignment. These arguments are passed through to the PRIMOS routine ASNLN$. If this key is not specified then the following values are used. This key is valid only for asynchonous devices.

TABLE 4-3. Values for the openMode Argument of open (continued)

Value	*Meaning*
	Value — *Meaning* *protocol* — null string *config* — 0 *lword* — 0
04000000	Additive key to signal that the optional argument *magTapeOptions* is present. This key causes open( ) to use this argument for magnetic tape device assignment. The argument *magTapeOptions* must be a NULL-terminated string containing command line options acceptable to the PRIMOS ASSIGN command, for example, "-density 6250 -speed 100". See the *PRIMOS Commands Reference Guide* for other options to the ASSIGN command. If this key is not specified, no additional options are used. The MTX syntax supported by the PRIMOS ASSIGN command is not supported by the C libraries, which support only 9-track tape I/O. This key is valid only with magnetic tape devices.
010000000	Additive key that causes all PRWF$$ writes to be done in force write mode. This means that PRWF$$ does not return until the disk records involved are written to disk. This key should be combined with the unbuffered write key (01000) to achieve the desired results.

The opened file is a binary file. No translation takes place between the user program and the disk. Any type of data can be written and read back correctly. However, ASCII data written to these types of files is not translated into PRIMOS standard text format and thus is not valid data for other PRIMOS commands such as ED and SPOOL. Reading and writing these types of files with read( ) and write( ) is much faster than using fopen( ) with fread( ) and fwrite( ).

All I/O to TTY and asynchronous devices is in raw mode; that is, your kill and erase characters are not interpreted. Normally, the output is not filtered in any way, but if you use the 0200 key, the \n (newline character) is translated to \n\r (newline carriage-return) on output.

Because of disk buffering, a maximum of one page (2048 bytes) may be buffered by the low-level I/O routines before it is actually written to disk. You can use the function fflush( ) to write the contents of the buffer to disk. You must first use fdopen( ) to obtain a *filePointer*, which you then pass to fflush( ).

While a file is being accessed by C's I/O libraries it may appear to have an odd size in bytes. The libraries maintain this illusion. However, PRIMOS does not support odd length files, so when a file with an odd length is closed, it may be padded with one null byte to bring it up to an even length.

Three predefined *fileIDs* (values 0 through 2) do not have to be opened before I/O may be performed using them.

Value	*Meaning*
0	TTY input (stdin)
1	TTY output (stdout)
2	TTY output (stderr)

These *fileIDs* always refer to the user's terminal and may not be redirected. The two output *fileIDs* have the 0200 additive key set.

The *fileIDs* returned for nondisk devices, including the three predefined *fileIDs*, are for use only with the I/O routines read( ), write( ), lseek( ) and tell( ). You cannot pass any of these *fileIDs* to fdopen( ) to obtain a *filePointer*.

You can perform four low-level I/O operations on an open device: read( ), write( ), lseek( )/seek( ), and tell( ). All four operations are permitted on disk devices. Only read( ), write( ) and tell( ) operations are valid for TTY and asynchronous devices. The operations read( ), write( ) and lseek( ) are valid for magnetic tape devices. close( ) is valid for all devices. Any files opened by C library routines must be closed by C library routines.

For TTY devices, tell( ) returns 0 if no characters are available to read, and returns 1 if characters are available to read.

For asynchronous devices, tell( ) returns two pieces of information packed into the returned 32-bit integer. The most significant 16 bits contain the amount of free space in the output buffer, in bytes. This corresponds to the value returned by the PRIMOS subroutine T$AMLC called with a value of 7 for the key argument. (See Volume IV of the *Subroutines Reference Guide* for more information about T$AMLC.) The least significant 16 bits contain the number of bytes waiting to be read in the input buffer. This corresponds to the value returned by T$AMLC with a key value of 4.

For example,

```
unsigned int Asyncstatus;
short freeOutputBytes, waitingInputBytes;

Asyncstatus = tell(fileID);
freeOutputBytes = Asyncstatus >> 16;
waitingInputBytes = Asyncstatus & 0xFFFF;
```

Magnetic tape devices may be opened for read or write, but not read/write. All magnetic tape operations are in raw mode and affect an entire tape record. The operations read( ), write( ) and lseek( ) are valid for magnetic tape devices. However, each read( ) or write( ) causes an entire tape record to be read or written, and all position (lseek( )) requests are specified in records rather than bytes. 0 and 1 are the only direction keys allowed for lseek( ) on magnetic tape devices. 0 means record from beginning of tape, 1 means record from current postion.

When a magnetic tape device that has been opened for write is closed, an end-of-file tape mark is written. You can write a double tape mark by using the following procedure.

1. Open the device using the 01000000 additive key. When you use this key, the device is not unassigned when closed.
2. Close the device. This writes one tape mark.
3. Open the device again using no additive keys. The device is reassigned even though it is already assigned, but that does no harm.
4. Close the device. This writes the second tape mark and unassigns the device.

No-wait mode I/O, enabled by the 0100 *openMode*, has different effects depending on the device type. For TTY and asynchronous devices, no-wait mode causes read( ) requests not to wait if fewer than the specified number of characters are actually available to read. For example, in no-wait mode, if a request is made to read 10 bytes and the user has typed only two characters, then read( ) returns only two bytes. In wait mode the read( ) blocks until 10 bytes are actually available, and all 10 are returned.

For tape devices, no-wait mode affects read( ), write( ) and lseek( ) operations. Control returns to the caller as soon as the requested operation is started. Thus, for a read( ) request, control returns to the caller before the user's buffer is filled with the requested data. Similarly, for write( ) requests, control returns to the caller before all of the data is actually written to tape. This enables you to implement double buffering tape I/O mechanisms. If your program makes a second tape request before the previous one has completed, then control does not return to the caller until the first operation is complete and the second one has started. In order to wait for the completion of an operation in no-wait mode, make the following lseek( ) call for the tape device:

```
lseek(fileID, 0, 1);
```

In wait mode, for tape devices, all operations are completed before control is returned to the caller.

For disk devices, lseek( ) returns the new byte offset within the file, or -1 on any error. For magnetic tape devices, lseek( ) returns 0 if the operation was successful and -1 on error. In either error case, the external variable errno (defined in stdio.h) is set to the PRIMOS error code.

The C library I/O routines use the PRIMOS subroutine T$MT to move data to and from magnetic tape. (See Volume IV of the *Subroutines Reference Guide.*) This places the following constraints upon magnetic tape I/O:

- You cannot write records that contain an odd number of bytes. If you request an odd number of bytes, T$MT rounds your request up to a even number.
- The largest tape record that you can read or write is, at most, 12K bytes. It may be as small as 10K bytes, depending on the page alignment of the buffer. You are not allowed to use larger records.
- The buffer address used for tape I/O (read( ) or write( )) must be aligned on a 16-bit boundary. If you try to pass an odd byte aligned buffer, these routines return -1 and set errno to E$BPAR.

The external variable errno (defined in stdio.h) is used for two distinct purposes in conjunction with magnetic tape I/O. If the function read( ), write( ) or lseek( ) fails because of a PRIMOS error, such as `Device not assigned`, the function returns -1 and errno is set to the PRIMOS error code. If the function succeeds, however, and an operation is successfully started, read( ) or write( ) returns the size of the request in bytes, and lseek( ) returns 0. In this case, errno is set to the current hardware status, which is the second element of the *statv* argument to T$MT. It is your responsibility to check errno for any errors that occur during tape operations, such as `End of tape detected` or `Parity error`.

The meaning of the current hardware status depends upon whether you are performing wait mode or no-wait mode I/O. If you have enabled no-wait mode by using the 0100 key, when a tape operation is successfully started, errno is set to the current magnetic tape hardware status before the read( ), write( ) or lseek( ) operation is started. If you are using wait mode I/O, however, errno reflects the hardware status after the operation is complete. In either mode, you can use the special call

```
lseek(fileID, 0, 1)
```

to cause errno to be set to the current hardware status, after completion of any pending operation. The function bio$primosfileunit( ) allows you to determine the PRIMOS file unit that is being used to access a disk file. Given a *fileID* returned from open( ) or fileno( ), bio$primosfileunit( ) returns the corresponding PRIMOS file unit.

▶ perror( )

Writes a short error message to the user's terminal describing the last error encountered during a call to the C runtime library from a C program.

```
#include <stdio.h>
extern int errno;
perror(string)
char *string;
```

The perror( ) function writes out its argument (a user-supplied prefix to the error message), followed by a colon, followed by the message itself, followed by a newline. The argument is typically the name of the program that incurred the error.

The external symbol errno (defined in stdio.h) contains the number of the most recent error. This is a standard system error code as defined in SYSCOM>ERRD.INS.CC; a value of 0 indicates no previous error.

▶ pow( )

Returns the first argument raised to the power of the second argument.

```
#include <math.h>
double pow(x, y)
double x, y;
```

The first argument cannot be negative. Both arguments must be **double**, and the returned value is **double**.

▶ primospath( )

Takes a pathname such as those used by the UNIX operating systems and converts it to a PRIMOS pathname.

```
char *primospath(unixPathname)
char *unixPathname;
```

The UNIX operating systems use the slash (/) character as a separator in pathnames instead of the PRIMOS greater-than (>) symbol. The symbol . specifies the current directory, and .. specifies the parent directory. By definition, .. of / is /. (The parent of the root is the root.) Pathnames that do not start with a slash are considered to be relative to the current attach point. The characters @, +, and = are passed through unchanged.

The function primospath( ) returns a pointer to a static character array containing the PRIMOS pathname.

▶ printf( ), fprintf( ), sprintf( )

Perform formatted output to the standard output (printf( )), to a specified file (fprintf( )), or to a character string in memory (sprintf( )).

```
#include <stdio.h>
int printf(formatSpecification [, outputSource,. . . ])
char *formatSpecification;

#include <stdio.h>
int fprintf(filePointer,
            formatSpecification [, outputSource,. . . ])
FILE *filePointer;
char *formatSpecification;

#include <stdio.h>
int sprintf(string, formatSpecification [, outputSource,. . . ])
char *string;
char *formatSpecification;
```

All three functions take a format-specification argument containing characters to be written literally to the output and/or conversion specifications corresponding to the list of optional output sources.

All three functions return the number of characters actually written out. The printf( ) and fprintf( ) functions return -1 if an I/O error occurs.

The output sources are expressions whose types correspond to conversion specifications given in the format specification. If no conversion specifications are given, the output sources may be omitted; otherwise, the function call must have exactly as many output sources as there are conversion specifications, and the conversion specifications must match the types of the output sources. Conversion specifications are matched to output sources in simple left-to-right order.

formatSpecification is a character string that specifies the output format. *formatSpecification* may contain ordinary characters, conversion specifications, or both. Ordinary characters are printed to the output literally.

Conversion Specifications

A conversion specification causes the conversion of a corresponding *outputSource* to a formatted character string. Each conversion specification begins with a percent sign (%) and ends with a conversion character that specifies an output format. Conversion characters and their corresponding output formats are listed in Table 4-4.

Optionally, conversion specifiers may be inserted between the percent sign and the conversion character. Table 4-5 lists the conversion specifiers and their effect upon the output format. If you use conversion specifiers, you must place them in the order shown in that table.

TABLE 4-4. Conversion Characters for Formatting Output

Character	*Meaning*
%d	Converts to decimal format.
%o	Converts to unsigned octal format without leading 0.
%x	Converts to unsigned hexadecimal format without leading 0x.
%u	Converts to unsigned decimal format, returning a number in the range 0 through 4,294,967,295.
%c	Outputs a single character. (A NULL is ignored.)
%s	Outputs a character string. (Characters are written until NULL is encountered or until the number of characters indicated by the precision specification is exhausted. If the precision specification is 0 or omitted, all characters up to a NULL are output.)

TABLE 4-4. Conversion Characters for Formatting Output (continued)

Character	*Meaning*
%e, %E	Converts **float** or **double** to the format **[-]m.nnnnnnE[+|-]xx** where the number of *n*s specifies the precision (default = 6). If the precision is explicitly 0, the decimal point is displayed but no *n*s are displayed. An E is printed if the conversion character is an uppercase E. An e is printed if the conversion character is a lowercase e.
%f	Converts **float** or **double** to the format **[-]m. . . m.nnnnnn** where the number of *n*s specifies the precision (default = 6). Note that the precision does not determine the number of significant digits printed. If the precision is explicitly 0, no decimal point and no *n*s appear.
%g	Converts **float** or **double** to %d, %e, or %f format, whichever is shorter. (Suppresses insignificant zeros.)
%Lf, %Le, %Lg	Same as %e, %f, %g, except that they convert a **long double** number to the corresponding format. Use this specification in conjunction with either the -ANSI option, or with the -QUADCONSTANTS and -QUADFLOATING options. (The -ANSI option includes support for the **long double** data type.)
%p	Converts the address of a pointer to the format **[*]ssss[(r)]/wwwwww[+8b]** The asterisk (*) indicates that the fault bit is set. *ssss* is the segment number, in octal. *r* is the ring number (0, 1, 2, or 3). *wwwwww* is the word number, in octal. +8b indicates that the byte bit (also called the extension bit) is set, indicating that the address is 48 bits in length. (If +8b does not appear, the address is 32 bits in length.)
%%	Writes out the percent (%) symbol. No conversion is performed.

TABLE 4-5. Field Specification for Output Formats

Character	*Meaning*
-	Left-justifies the converted output source in its field.
width	Designates the minimum field width. The value is an integer constant. If the converted output source is wider than this minimum, write it out anyway. If the converted output source is smaller than the minimum width, pad it to make up the field width. Padding is normally done with spaces, and with 0 if the field width is specified with a leading 0. (This does not mean that the width is an octal number.) Padding normally occurs on the left. If a minus sign is used, however, padding occurs on the right.
.	Separates field width from precision.
precision	Designates the maximum number of characters to print with the %s format, or the number of fractional digits with the %e, %f or %g format. The value is an integer constant.
l	Indicates that a following %d, %o, %x, or %u specification corresponds to a **long** output source. (Note that in PRIMOS C, all **int**s are **long** by default.) Indicates that a following %e, %f or %g specification corresponds to a **double** output source.
*	Can be used to replace the field-width specification and the precision specification. The corresponding width or precision is given in the output source.

▶ putc( ), fputc( ), putchar( ), putw( ), puth( )

Write characters to a specified file.

```
#include <stdio.h>
int putc(character, filePointer)
char character;
FILE *filePointer;

#include <stdio.h>
int fputc(character, filePointer)
char character;
FILE *filePointer;

#include <stdio.h>
int putchar(character)
char character;
```

```
#include <stdio.h>
int putw(integer, filePointer)
int integer;
FILE *filePointer;

#include <stdio.h>
int puth(integer, filePointer)
int integer;
FILE *filePointer;
```

The putc( ) function writes a single character to a specified file and returns the character. The file pointer is left positioned after the character. This function is implemented as a macro.

The fputc( ) function generates the same results as the putc( ) function, but it is not implemented as a macro.

The putchar( ) function writes a single character to the standard output and returns the character. This function is identical to putc(stdout). The putchar( ) function is implemented as a macro.

The putw( ) function writes an **int** to an output file as four characters. No type conversion is performed.

The puth( ) function writes the low-order two bytes of an **int** to a specified output file as two characters. No type conversion is performed.

All of these functions return EOF (defined in stdio.h) to designate output errors. Because EOF is itself an integer, use ferror( ) to detect errors encountered by the putw( ) function.

▶ puts( ), fputs( )

Write a character string.

```
#include <stdio.h>
int puts(string)
char *string;

int fputs(string, filePointer)
char *string;
FILE *filePointer;
```

The puts( ) function writes a string to the standard output, followed by a newline. The fputs( ) function writes a character string to a specified file, but it does not append a newline to the string. Neither function copies the terminating NULL to the output stream.

▶ putw( )

```
#include <stdio.h>
int putw(integer, filePointer)
int integer;
FILE *filePointer;
```

For more information, see the putc( ) function.

▶ rand( ), srand( )

rand( ) returns pseudo-random numbers that range from 0 through $2^{31} - 1$. srand( ) can be called at any time to reset the random number generator to a random starting point.

```
int rand( )
int srand(seed)
int seed;
```

The rand( ) function uses a multiplicative congruential random number generator with a repeat factor (period) of 2^{32}. The random number generator is reinitialized by calling srand( ) with the argument 1, or it can be set to a specific point by calling srand( ) with any other number. If rand( ) is called before a call to srand( ), an initial seed of 1 is used.

▶ read( )

Reads bytes from a file specified by a *fileID* returned from the open( ) or creat( ) function, and places them in a buffer.

```
#include <stdio.h>
int read(fileID, pointer, nbytes)
int fileID, nbytes;
char *pointer;
```

pointer points to a buffer into which data is read from the file specified by *fileID.* The buffer must be large enough to hold at least *nbytes* of contiguous storage. The function returns the number of bytes actually read. The return value does not equal *nbytes* if an end of file is encountered before the read( ) can be completed.

Use of open( ) and read( ) is more efficient than use of fopen( ) and fread( ). To get maximum speed from this routine, *pointer* and the file pointer should have the same alignment. Both should be at an odd byte or an even byte.

A return value of 0 means that an end of file was encountered before any bytes of data could be read. The function returns -1 on any error and sets errno (defined in stdio.h) to the file system error code.

The read( ) function is valid for disk devices, TTY and asynchronous devices, and magnetic tape devices.

For more information, see the description of the open( ) function on page 4-36.

▶ realloc( )

Changes the size of the area pointed to by the first argument to the number of bytes given by the second argument.

```
char *realloc(pointer, size)
char *pointer;
unsigned size;
```

The realloc( ) function returns the address of the area, because the area may have moved to a new address. If the area was moved, the space previously occupied is freed. If realloc( ) is unable to reallocate the space (for example, if there is not enough room) it returns 0.

The realloc( ) function must be used on currently allocated space and cannot be called with an area that was previously freed. The contents of the area are unchanged up to the lesser of the old and new sizes. If the old size is less than the new size and is not an even multiple of 8, then any bytes between the old size and the next highest multiple of 8 will contain garbage. After that, new space in the reallocated area is initialized with 0. For example, if the old size is 6 and realloc( ) is called with a *size* of 50, the contents of the first 6 bytes remain the same; the next two bytes contain garbage; and the last 42 bytes are initialized with 0.

Note

The C library's routines for dynamic memory management (malloc( ), calloc( ), realloc( ), free( ), and cfree( )) are designed for use only with each other. If you allocate memory with code written in another language, do not deallocate it with a C routine. Similarly, if you allocate memory with a C routine, do not deallocate it with code written in another language.

▶ rewind( )

Positions the file to the beginning.

```
#include <stdio.h>
int rewind(filePointer)
FILE *filePointer;
```

The rewind( ) function is equivalent to `fseek(filePointer, 0, 0)`. It returns -1 to indicate failure and 0 to indicate success.

▶ rindex( )

```
#include <string.h>
char *rindex(string, character)
char *string, character;
```

For more information, see the strrchr( ) function.

▶ scanf( ), fscanf( ), sscanf( )

Perform formatted input from the standard input (scanf( )), from a specified file (fscanf( )), or from a character string in memory (sscanf( )).

```
#include <stdio.h>
int scanf(formatSpecification [, inputPointer,. . . ])
char *formatSpecification;

#include <stdio.h>
int fscanf(filePointer, formatSpecification [, inputPointer,. . . ])
FILE *filePointer;
char *formatSpecification;

#include <stdio.h>
int sscanf(string, formatSpecification [, inputPointer,. . . ])
char *string, *formatSpecification;
```

In each function, *formatSpecification* may contain ordinary characters, conversion specifications, or both. Ordinary characters are matched literally. Each conversion specification converts a portion of the input to a target designated by an *inputPointer*.

Each function returns the number of successfully matched and assigned input items. If end of file (or string) is encountered, the functions return EOF (a preprocessor constant defined in stdio.h).

The *formatSpecification* is a character string that can include three kinds of items:

- White-space characters (spaces, tabs, and newlines), which cause input to be read up to the next non-white-space character.
- Ordinary characters (except %), which must match the next non-white-space character in the input.
- Conversion specifications (beginning with %), which govern the conversion of the characters in an input field and their assignment to an object indicated by a corresponding *inputPointer*. (See Table 4-6, at the end of this section.)

Each *inputPointer* is an address expression indicating an object whose type must match that of the corresponding conversion specification. The indicated object is the target that receives the input value. The number of *inputPointer*s must match the number of conversion

specifications, and the addressed objects must match the types of the conversion specifications.

On successive calls, scanf( ) and fscanf( ) resume searching at a point immediately after the last character processed by the previous call. The sscanf( ) function lacks this functionality. The string is searched from the beginning on each call to sscanf( ).

Conversion Specifications

Each conversion specification begins with a percent sign (%). This sign is followed by an optional assignment-suppression character (*), an optional number giving the maximum field width, and a character indicating the type of conversion. The conversion characters are described in Table 4-6.

An input field is defined as a string of non-white-space characters. It extends either to the next white-space character or until the optionally specified field width is exhausted. (Note that because the newline character belongs to the set of white-space characters, the function reads across line/record boundaries.) The delimiters of the input field can be changed with the bracket conversion specification, %[. . .], described in Table 4-6.

If the assignment-suppression character (*) appears in the format specification, the corresponding input field is interpreted and then skipped, without making any assignment. For example, you can use assignment suppression to read a character followed by a newline:

```
scanf("%c%*c", &c);
```

This call to scanf( ) reads one character, then drops the next character it sees. If the assignment suppression character is not used here, the next call to scanf( ) will pick up the next character, with possibly unexpected results.

The *inputPointer* arguments must be pointers or other address-valued expressions. To read a number in decimal format and assign its value to *n*, you must use the code

```
scanf("%d", &n)
```

not

```
scanf("%d", n)
```

White space in a format specification matches optional white space in the input field. That is, the format specification `string = %d` matches

```
string = 1234
string=1234
string= 1234
string =1234
```

but not

```
strin g=1234
```

TABLE 4-6. Conversion Specifications for Formatting Input

Character	*Meaning*
%d	A decimal integer is input. The corresponding argument must be an integer pointer.
%o	An octal integer is input with or without a leading sign and 0. The corresponding argument must be an integer pointer.
%x	A hexadecimal integer is input with or without the leading 0x. The corresponding argument must be an integer pointer.
%c	A single character is input. The corresponding argument must point to a character. White space is not skipped in this case, so that *n* white-space characters can be read with *%nc*. If a field width is given with %c, the given number of characters is read, and the corresponding argument must be a character array pointer.
%s	A character string is input. The corresponding argument must point to an array of characters that is large enough to contain the string plus the terminating NULL (\0). The input field is terminated by a white-space character (space, tab, or newline).
%f, %he, %hf	A floating-point number is input. The corresponding argument must be a pointer to a floating-point number. The input format for a floating-point number is **[+|-]nnn[.[ddd]][{E|e}[+|-]nn]** where *d* and *n* are decimal digits.
%e	Same as %f format.
%ld, %lo, %lx	Same as %d, %o, %x.
%lf, %le	Same as %e, %f, except that the corresponding argument is a pointer to a **double** rather than a floating-point number. The same effect can be obtained by using an uppercase E or F.
%Lf, %Le, %Lg	Same as %e, %f, %g, except that the corresponding argument is a pointer to a **long double** rather than a floating-point number. Use this specification in conjunction with either the -ANSI option, or with the -QUADCONSTANTS and -QUADFLOATING options. (The -ANSI option includes support for the **long double** data type.)

TABLE 4-6. Conversion Specifications for Formatting Input (continued)

Character	*Meaning*

%p	The address of a pointer is input, in the format
	[*]ssss[(r)]/wwwwww[+8b | +0b]
	The asterisk (*) causes the fault bit to be set. *ssss* is the segment number, in octal. *r* is the ring number (0, 1, 2, or 3). *wwwwww* is the word number, in octal. +8b sets the byte bit (also called the extension bit), indicating that the address is 48 bits in length. +0b leaves the byte bit reset, indicating a 32-bit address.
	Note
	For scanf( ) to read the %p specification correctly, the argument must be a pointer to a pointer to **void**. For example, the declaration
	`void *ptr;`
	and the statement
	`scanf("%p", &ptr);`
	will read in a pointer value correctly.

%hd, %ho, %hx	Same as %d, %o, %x, except that the corresponding argument is a pointer to a **short** rather than an **int**.
%[. . .]	A string that is not delimited by white-space characters is input. The brackets enclose a set of characters. Ordinarily, this set is made up of the characters that comprise the string field. Any character not in the set terminates the field; however, if the first character in this set is a caret (^), the set specifies the characters that terminate the field. The corresponding argument must be a character array pointer.

▶ seek( )

```
#include <stdio.h>
int seek(fileID, offset, direction)
int fileID, offset, direction;
```

For more information, see the lseek( ) function.

▶ setbuf( )

Associates a buffer with an input or output file.

```
#include<stdio.h>
setbuf(filePointer, buffer)
FILE *filePointer;
char *buffer;
```

The setbuf( ) function may be called only after the file has been opened and before any I/O is done with respect to the file. It causes file operations to use the specified buffer for all subsequent I/O operations on the file instead of using an automatically allocated buffer. The buffer must be large enough to hold an entire input record. The BUFSIZ constant defined in the stdio.h module is available for your use in defining the size of the buffer. If the buffer is NULL (defined in stdio.h), the file is unbuffered. The buffer *must* be obtained by calling malloc( ) because it is freed when a call to fclose( ) is made with *filePointer*.

Note that unbuffered I/O is permitted on binary files only. That is, the file must have been opened by a call to fopen( ) with an open mode of "i" or "o", not "r" or "w". The setbuf( ) routine can also be used with the defined files stdout and stderr to cause terminal output to be buffered. Buffered terminal input is not allowed.

setbuf( ) returns the defined constant EOF if the requested operation cannot be performed. See the discussion of input and output buffering in Chapter 7.

▶ setjmp( ), longjmp( )

Provide a way to transfer control from a nested series of functions back to a predefined point without returning normally (that is, not by a series of return statements).

```
#include <setjmp.h>
setjmp(env)
jmp_buf env;

#include <setjmp.h>
longjmp(env, val)
jmp_buf env;
```

The setjmp( ) function saves the context of the calling function in an environment buffer. The longjmp( ) function restores the context of the environment buffer.

The environment buffer is declared as an array of integers long enough to hold the context of the calling function.

When setjmp( ) is first called, it returns the value 1. If longjmp( ) is then called and the same environment is named as in the call to setjmp( ), control is returned to the setjmp( ) call as if it had returned normally a second time. The value then returned by the setjmp( ) routine is second argument to longjmp( ) in 32IX mode.

In 64V mode the value returned by setjmp( ) is undefined when setjmp( ) returns by virtue of a call to longjmp( ). (The second argument to longjmp( ) is retained for compatibility only.) Since this value could coincidentally be 1 (which also indicates that setjmp( ) has been called for the first time), it is best to use a static external data object to communicate between the code that calls longjmp( ) and the code that calls setjmp( ). Never rely on the value returned by setjmp( ) in V-mode code.

Routines that call setjmp( ) should be compiled with low levels of optimization (that is, -NOOPT or -OPT 1).

For an example of the use of setjmp( ) and longjmp( ), see page 5-21.

▶ setmod( )

Sets access rights for a specified file. This function is analogous to the getmod( ) function, which reads access rights for a specified file.

```
#include <stdio.h>
int setmod(pathname, user, mode)
char *pathname, *user;
int mode;
```

Specify the *user* argument to set access rights for a particular named user or group. If the object is protected by a default ACL or by an access category, then a copy of that ACL is made as a specific ACL for the object. This routine modifies an existing ACL rather than replacing it. For objects in password-protected directories, only the read, write, delete, and default bits are used. Specify a *user* name of "_nonowner_" to set nonowner rights for an object in a password-protected directory.

setmod( ) uses three additional bit settings that getmod( ) does not use: 00 for no access, 0200 to remove an ACL, and 0400 to return the object to default protection. The bit settings for access rights are listed with getmod( ) above.

The setmod( ) function returns -1 on all errors and sets errno (defined in stdio.h) to the file system error code.

▶ signal( )

Provides a mechanism for handling conditions that arise during execution. In addition, the CCMAIN.BIN C library must be linked in for signal( ) to work properly. This function is available in 32IX mode only.

```
#include <signal.h>
void (*signal(sig, func))( )
int sig;
void (*func)( );
```

Note

See the descriptions of the abort( ) and timer( ) functions in this chapter for examples of programs that use signal( ).

The signal( ) function determines how conditions that occur during execution will be handled. The first argument, *sig*, is a constant that specifies the type of signal to be

handled. These constants, or macros, are defined in the SIGNAL.H.INS.CC file and are listed in Table 4-7 as follows.

TABLE 4-7. Conditions Raised by sig Argument of signal Function

sig Arg	*Cause*	*Condition Raised*
SIGHUP	Hangup	LOGOUT$
SIGINT	Interrupt	QUIT$ first choice
SIGQUIT	Quits	QUIT$ second choice
SIGILL	Illegal instruction	RESTRICTED_INST$ UII$
SIGABRT	Abnormal termination	ABORT$
SIGFPE	Arithmetic error	ARITH$ ERROR
SIGBUS	Bus error	ACCESS_VIOLATION$
SIGSEGV	Segmentation violation	ILLEGAL_SEGNO$ OUT_OF_BOUNDS$ NO_AVAIL_SEGS$ NULL_POINTER$ POINTER_FAULT$
SIGSYS	Bad arg to system call	LINKAGE_FAULT$ LINKAGE_ERROR$ SVC_INST$
SIGALRM	Alarm clock	ALARM$
SIGTERM	Software termination	CLEANUP$ (STOP$)

The second argument, *func*, specifies the function to be called if the condition named by *sig* is raised. This can be either a user-defined function or one of the following macros defined in SIGNAL.H.INS.CC.

Func Arg	*Action to be Taken*
SIG_DFL	Implementation-defined default behavior
SIG_IGN	Ignore the condition

The user-supplied handler, *func*, for any signal is invoked as:

```
func(sig)
```

where the argument, *sig*, specifies the condition that was raised.

▶ sin( )

Returns the sine of its radian argument. Both the argument and the sine value must be **double**.

```
#include <math.h>
double sin(x)
double x;
```

▶ sinh( )

Returns the hyperbolic sine of the argument. Both the argument and the hyperbolic sine value must be **double**.

```
#include <math.h>
double sinh(x)
double x;
```

▶ sleep( )

Suspends the execution of the current process for at least the number of seconds specified by its argument. When successful, sleep( ) returns the number of seconds that the process slept.

```
int sleep(seconds)
unsigned seconds;
```

▶ sprintf( )

```
#include <stdio.h>
int sprintf(string, formatSpecification [, outputSource,. . . ])
char *string;
char *formatSpecification;
```

For more information, see the printf( ) function.

▶ sqrt( )

Returns the square root of the argument. The argument and the returned value are both **double**. The argument must not be negative.

```
#include <math.h>
double sqrt(x)
double x;
```

▶ srand( )

```
int srand(seed)
int seed;
```

For more information, see the rand( ) function.

▶ sscanf( )

```
#include <stdio.h>
int sscanf(string, formatSpecification [, inputPointer,. . . ])
char *string,*formatSpecification;
```

For more information, see the scanf( ) function.

▶ stat( )

Fills a stat structure with information about a specified file.

```
#include <stdio.h>
int stat(pathname, buffer)
char *pathname;
struct stat *buffer;
```

The stat structure contains all information returned from the fsize( ), fdtm( ), ftype( ), and frwlock( ) routines. The stat( ) function returns -1 on all errors and sets errno (defined in stdio.h) to the file system error code. The structure layout is as follows (structure defined in stat.h):

```
struct stat { long st_size;     /* File size
                                   in bytes */
              long st_mtime;    /* DTM of file */
              short st_type;    /* Type of file */
              short st_rwlock; /* Read/write
                                   lock of the
                                   file */
            };
```

▶ sterm( )

Sets terminal characteristics.

```
#include <term.h>
void sterm(buffer)
struct sterm *buffer;
```

The BREAK bit is ignored in the passed-flags structure. The two additional bit settings of INHIBIT_BREAK 040 and ENABLE_BREAK 0100 are used in its place. See also the description of gterm( ) on page 4-28.

▶ strcat( ), strncat( )

Concatenate character strings.

```
#include <string.h>
char *strcat(string1, string2)
char *string1, *string2;

#include <string.h>
char *strncat(string1, string2, max)
char *string1, *string2;
int max;
```

The strcat( ) function concatenates its second argument to the end of its first argument. Both arguments must be character strings, and, in the case of strcat( ), NULL-terminated.

The strncat( ) function performs the same operation as strcat( ), but it uses characters from the second argument up through the specified maximum unless a NULL terminator is encountered first. The argument *max* is an integer giving the maximum number of characters to use from *string2*. If a strncat( ) call reaches the specified maximum, strncat( ) sets the next byte in *string1* to NULL.

Both functions return the address of the first argument, *string1*. The argument is assumed to be large enough to hold the concatenated result.

▶ strchr( ), index( ), strrchr( ), rindex( )

Find the first (or last) occurrence of a character in a string.

```
#include <string.h>
char *strchr(string, character)
char *string, character;

#include <string.h>
char *strrchr(string, character)
char *string, character;
```

These functions perform similar tasks. The strchr( ) function returns the address of the first occurrence of a given *character* in a NULL-terminated *string*. It returns 0 if the *character* does not occur in the *string*. The strrchr( ) function is similar to strchr( ), but it returns the address of the last (rightmost) occurrence of the *character*.

The index( ) function is a synonym for strchr( ), and the rindex( ) function is a synonym for strrchr( ).

▶ strcmp( ), strncmp( )

Compare two ASCII character strings.

```
#include <string.h>
int strcmp(string1, string2)
char *string1, *string2;

#include <string.h>
int strncmp(string1, string2, max)
char *string1, *string2;
int max;
```

The strcmp( ) function compares two ASCII character strings and returns a negative, 0, or positive integer, indicating that *string1* is lexicographically less than, equal to, or greater than *string2.* The returned value is obtained by subtracting the ASCII values of the characters at the first position where the two strings disagree.

The strncmp( ) function performs the same operation as strcmp( ), but it compares a specific maximum number of characters in the two strings. The argument *max* gives the maximum number of characters, beginning with the first to be compared.

With either function, the comparison is terminated when a NULL is encountered.

▶ strcpy( ), strncpy( )

Copy argument strings.

```
#include <string.h>
char *strcpy(string1, string2)
char *string1, *string2;

#include <string.h>
char *strncpy(string1, string2, max)
char *string1, *string2;
int max;
```

The strcpy( ) function copies *string2* into *string1.* This function terminates when a NULL is encountered in *string2.*

The strncpy( ) function copies a specified number of characters from *string2* to *string1.* Exactly *max* characters are copied, including NULLs. This is a block memory move. If *string2* contains more than *max* characters, the copy in *string1* is not necessarily terminated by a NULL. Both functions return the address of *string1.*

▶ strcspn( )

Searches a string for a character in a specified set of characters.

```
#include <string.h>
int strcspn(string, charset)
char *string, *charset;
```

The strcspn( ) function returns the number of characters that precede the matched one. That is, the function spans the characters not in *charset* and returns the number of such leading characters.

If the argument *string* is a null string, 0 is returned. If no characters in *string* are in *charset*, the length of *string* is returned.

▶ strlen( )

Returns the length of a string of ASCII characters. The returned length does not include the terminating NULL (\0).

```
#include <string.h>
int strlen(string)
char *string;
```

▶ strncat( )

```
#include <string.h>
char *strncat(string1, string2)
char *string1, *string2;
int max;
```

For more information, see the strcat( ) function.

▶ strncmp( )

```
#include <string.h>
int strncmp(string1, string2, max)
char *string1, *string2;
int max;
```

For more information, see the strcmp( ) function.

▶ strncpy( )

```
#include <string.h>
char *strncpy(string1, string2, max)
char *string1, *string2;
int max;
```

For more information, see the strcpy( ) function.

▶ strpbrk( )

Searches a string for an occurrence of one of a specified set of characters.

```
#include <string.h>
char *strpbrk(string, charset)
char *string, *charset;
```

The strpbrk( ) function returns the address of the first character in *string* that is in *charset*, or NULL if no character is in the set.

▶ strrchr( )

```
#include <string.h>
char *strrchr(string, character)
char *string, character;
```

For more information, see the strchr( ) function.

▶ strspn( )

Searches a string for the occurrence of a character not in a specified set of characters.

```
#include <string.h>
int strspn(string, charset)
char *string, *charset;
```

The strspn( ) function returns the number of characters that precede the mismatched character. That is, the function spans the characters in *charset* and returns the number of such leading characters.

If *charset* is a null string, a value of 0 is returned. If all the characters in *string* are also in *charset*, the length of *string* is returned.

▶ system( )

Executes the command contained in its argument as a PRIMOS command, and then resumes execution of the current program.

```
int system(command)
char *command;
```

Abbreviations are expanded. If the call succeeds, system( ) returns 0. If the call fails, system( ) returns a positive integer.

▶ tan( )

Returns a **double** value that is the tangent of the argument expressed in radians, which must also be **double**.

```
#include <math.h>
double tan(x)
double x;
```

▶ tanh( )

Returns a **double** value that is the hyperbolic tangent of the **double** argument.

```
#include <math.h>
double tanh(x)
double x;
```

▶ tell( )

Returns the current byte position in a file specified by a *fileID* returned from open( ).

```
#include <stdio.h>
int tell(fileID)
int fileID;
```

The byte position returned by the tell( ) function can be used in future calls to seek( ) or lseek( ). The function returns -1 on any error and sets errno (defined in stdio.h) to the file system error code.

The use of tell( ) is valid on disk, TTY, and asynchronous devices, but not on magnetic tape devices. For TTY devices, tell( ) returns 0 if no characters are available to read and 1 if characters are available to read. See open( ) for more information.

▶ time( )

Returns the time, in seconds, elapsed since 00:00:00, Jan. 1, 1970.

```
int time(seconds)
int *seconds;
```

If the pointer *seconds* is not NULL (0), the returned value is also stored in the location to which *seconds* points.

▶ timer( )

Causes the PRIMOS ALARM$ condition to be raised after a specified number of elapsed minutes.

```
void timer(mins)
int mins;
```

The timer( ) function is analogous to the alarm routine found on UNIX operating systems, except that a separate call to either signal( ) or MKON$P must be made to set up the handler for the condition.

The following examples demonstrate how to catch the ALARM$ condition using signal( ) and MKON$P. In the examples, note that you can turn off the timer by calling the function with a 0 argument.

Example 1:

```
OK, SLIST TMP1.C
#include <signal.h>
static short minutes = 0;

main()
{
    void alarm_handler();

    signal(SIGALRM, alarm_handler);

    timer(1);      /* raise ALARM$ in one minute */

    for (;;) {
        if (minutes == 2) {
            timer(0);      /* turn timer off */
            return;
        }
        printf("looping...\n");
        sleep(10);
    }
}
/* function that is called when timer is up */
void
alarm_handler(sig)
    int sig;
```

```
{
    printf("A minute has passed.\n");
    ++minutes;
    timer(1);      /* reset timer */
    return;
}
OK, CC TMP1 -32IX
[CI Rev. T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]
00 Errors and 00 Warnings detected in 31 lines and 115 include lines.
OK, BIND -LI CCMAIN -LO TMP1 -LI C_LIB
[BIND Rev. T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]
BIND COMPLETE
OK, R TMP1
looping...
looping...
looping...
looping...
looping...
looping...
A minute has passed.
looping...
looping...
looping...
looping...
looping...
looping...
A minute has passed.
OK,
```

Note the use of the 32IX-mode predefined symbol __CI in the next example. The __CI symbol enables the compilation unit to be correct for both 64V and 32IX mode. The __CI symbol is discussed in more detail on page 5-20. See the description of the -UNDEFINE option on page 2-34 for information about other predefined symbols.

Example 2:

```
static short minutes = 0;  /* Minute count;
                              incremented by
                              handler */

main( )
{
  #ifdef __CI
     fortran void handler( ); /* Condition handler */
  #else
     extern void handler( );  /* Condition handler */
  #endif
  fortran void mkon$p( );     /* Make an onunit */
  fortran void sleep$( );     /* PRIMOS sleep routine */

  /* *** Start of code *** */

  mkon$p("ALARM$", 6, handler);
                             /* Set up ALARM$ handler */

  timer(1);
                             /* Raise ALARM$ in one minute */
```

```
    for(;;)
    {
       if (minutes == 2)
       {
          timer(0);              /* Turn timer off */
          return;
       }
       printf("Looping. . . \n");
       sleep$((long)10000);      /* Sleep for 10 seconds */
    }
  }  /* main */

  #ifdef __CI
    fortran
  #endif
  void handler(cfh)
    int *cfh;                   /* PRIMOS passed condition frame
                                   header pointer;
                                   we'll ignore it. */
  {
    printf("A minute has passed.\n");
    ++minutes;                  /* Bump count */
    timer(1);                   /* Reset timer */
    return;                     /* Back to where we were */
  }  /* handler */
```

▶ tmpnam( )

Creates a character string that can be used in place of the *pathname* argument in other function calls such as open( ) and fopen( ).

```
#include <stdio.h>
char *tmpnam(name)
char *name;
```

If the *name* argument is null, tmpnam( ) places the string in an internal storage area and returns a pointer to it. If it is not null, it is taken to be the address of an area of length L_tmpnam (defined in stdio.h). In this case, the string is written into this location, and *name* is returned. Multiple calls to tmpnam( ) with a null argument cause the current name to be overwritten.

▶ toascii( )

Converts a character or integer to an ASCII character by ANDing the value with 0377. Note that this is a mathematical operation only; it is not intended to convert noncharacter data to printable characters. This function is implemented as a macro.

```
#include <ctype.h>
int toascii(character)
char character;
```

▶ tolower( ), _tolower( )

Converts its argument, an uppercase alphabetic ASCII character, to lowercase.

```
#include <ctype.h>
int tolower(character)
char character;
```

The algorithm used is

character - 'A' + 'a'

The function tolower( ) first checks the range of the argument to make sure that it is an uppercase character. If so, it returns the lowercase form of argument; otherwise, it returns the argument unmodified. However, _tolower( ) is implemented as a macro and operates on any passed argument. It does not check the range of the argument.

▶ topascii( )

Converts a character or integer to a Prime ASCII character by ANDing the value with 0377 and then ORing it with 0200. Note that this is a mathematical operation only, and is not intended to convert noncharacter data to printable characters. This function is implemented as a macro.

```
#include <ctype.h>
int topascii(character)
char character;
```

▶ toupper( ), _toupper( )

Returns its argument, an ASCII lowercase alphabetic character, converted to uppercase.

```
#include <ctype.h>
int toupper(character)
char character;
```

The algorithm used for the conversion is

character - 'a' + 'A'

The function toupper( ) first checks the argument to make sure it is a lowercase character. If so it returns the uppercase form of the character. Otherwise, it returns the argument unmodified. However, _toupper( ) is implemented as a macro and operates on any passed argument. It does not check the range of its argument.

▶ ungetc( )

Writes a character to the buffer of a file and leaves the file positioned before the character.

```
#include <stdio.h>
int ungetc(character, filePointer)
char character;
FILE *filePointer;
```

The written *character* is said to be pushed back onto the file, because it is returned by the next getc( ) call. The function returns the pushed-back *character* or EOF if it cannot push the *character* back.

One character is guaranteed to be pushed back, provided something has previously been read from the file. The fseek( ) function erases all memory of pushed-back characters.

▶ write( )

Writes a specified number of bytes from a buffer to a file specified by a *fileID* returned from the open( ) or creat( ) functions.

```
#include <stdio.h>
int write(fileID, pointer, nbytes)
int fileID, nbytes;
char *poin;
```

pointer is the address of *nbytes* of contiguous storage. The write( ) function returns the number of bytes actually written. write( ) returns -1 on all errors and sets errno (defined in stdio.h) to the file system error code.

The write( ) function is valid for disk devices, TTY and asynchronous devices, and magnetic tape devices.

For more information, see the description of the open( ) function on page 4-36.

5

INTERFACING TO OTHER LANGUAGES

On 50 Series systems, you can write programs in C that call subroutines written in other 50 Series languages such as PL/I, Pascal, and F77. Similarly, you can write programs in other 50 Series languages that call C subroutines. Interlanguage calling, however, is somewhat complicated. Each of the high-level languages has its own conventions for declaring and accessing different data types and for passing data to a called function or procedure. In addition, differences exist in the way individual compilers are implemented on the 50 Series. Consequently, you must use special programming techniques to bridge this gap between different languages.

On 50 Series systems, you can compile C programs in either 64V or 32IX mode. Both modes are standard C, so they share the same language-specific conventions for declaring and accessing data types and for passing data to functions. Some differences exist, however, in the ways the two modes are implemented on the 50 Series. As a result, the techniques for interlanguage calling are somewhat different for the two modes.

Organization of This Chapter

This chapter contains information about the following topics:

- The differences between C and other languages that are important to interlanguage calling
- How to call other languages from either 64V-mode or 32IX-mode C
- How to call 64V-mode C from other languages
- How to call 32IX-mode C from other languages
- How to call 64V-mode C from 32IX-mode C
- How to call 32IX-mode C from 64V-mode C
- Function return types from C and other language routines
- How to use conditional compilation to make your interlanguage C code correct for both modes

- How to use the PRIMOS condition mechanism from C
- How to create and access common blocks from C
- How to access MIDASPLUS files

DIFFERENCES BETWEEN C AND OTHER LANGUAGES

This section begins with a list that summarizes the major differences between C and other high-level languages. The list is followed by more detailed descriptions of each of the differences. These descriptions contain references to examples that occur in later sections of the chapter.

Differences common to both 64V-mode and 32IX-mode C are the following:

- C passes parameters to a function by value. Other languages pass parameters to a function or procedure by reference.
- Certain data types are promoted when they are passed as parameters to a function in C. This does not occur in other languages.
- Arrays begin with element zero in C. In many other languages, arrays begin with element one.
- Strings in C are NULL-terminated arrays of characters. Some other languages use a different representation for strings.
- When C passes an array as a parameter to a function, it actually passes a pointer to the first element of the array. Most other high-level languages actually pass the first element itself.

Differences specific to 64V-mode C or 32IX-mode C are the following:

- In 64V-mode C, pointers are 48 bits long. In 32IX-mode C, pointers are 32 bits long. Some of the other 50 Series languages use 48-bit pointers, some use 32-bit pointers, and some use both.
- In 32IX mode, the compiler changes the names of external identifiers. The prefix G$ is prepended to the names of external variables, routines, and common blocks. This does not occur in 64V mode.
- 32IX-mode C programs cannot easily access pointers returned by routines written in other languages. This problem does not exist in 64V mode.

Pass by Value Versus Pass by Reference

In a C program, if you pass a scalar variable to a function, the called function receives a copy of the variable. Therefore, the called function cannot change the value of the original variable. This is called passing by value. In C, if you want a function to change the value of a variable, you must expressly pass a pointer to the variable. The called function then receives a copy of the pointer, but both the original pointer and the copy point to the

same place. Therefore, if the called function changes the variable pointed to, the original variable is changed. The following program example shows both ways of passing a variable to a function in C.

```
#include <stdio.h>
main( )
{
    int var;
    int *ptr;
    var = 5;
    func1(var);
    printf("After call to func1, var = %d. \n",var);
    ptr = &var;
    func2(ptr);
    printf("After call to func2, var = %d. \n",var);
}

func1(myvar)
int myvar;
{
    myvar += 2;  /* This doesn't change the value of var. */
}

func2(myptr)
int *myptr;
{
    *myptr += 2;  /* This changes the value of var. */
}
```

This program prints

```
After call to func1, var = 5.
After call to func2, var = 7.
```

Note that `func1` does not change the value of `var`, but `func2` does. That is because `var` is passed to `func1` by value, so `func1` gets only a copy of `var`. The function `func2` gets a copy of the pointer `myptr`. The copy of `myptr`, however, points to the same place as `myptr`, so `func2` can change the value of `var`. The only way you can pass a scalar variable by reference in C is to pass a pointer.

Other high-level languages are different from C. Most high-level languages pass parameters to a function by reference. That is, you do not have to pass a pointer in order to change the value of the original variable. The called function actually gets the original variable, not a copy of the variable.

On 50 Series systems, special interlanguage calling conventions allow you to pass most types of arguments by reference from C language programs to non-C functions without expressly passing a pointer. See Examples 1 and 2 on pages 5-8 and 5-9.

To pass a pointer by reference, you must use an integer as a dummy variable and cast it to a pointer. See Example 7 on page 5-12.

Promotion of Argument Types

In a non-ANSI C program, when you pass a **char**, **short int**, or **short unsigned int** as a parameter to a function, it is promoted to type **int**. Similarly, if you pass a **float**, it is converted to type **double**. The following example shows the proper way to code a function that receives these data types.

```
main( )
{
   char c;
   short int s;
   int l;
   float f;
   double d;
   foo(c,s,l,f,d);
}

foo(c,s,l,f,d)
int c,s,l;
double f,d;
{
   /* Any code here */
}
```

Notice that the passed variables of type **char** and **short int** are received as type **int**, and the variable of type **float** is received as type **double**. This method of passing and picking up arguments conforms to the Kernighan and Ritchie standard. The important point here is that other high-level languages do not promote data types; the called function or procedure receives the same data type that was originally passed.

For the reasons just described, the only data types that can be passed from a non-C program to a C function on 50 Series systems are types **long int**, **double**, and pointers to all types. When you call a non-C routine from a C program, the interlanguage calling conventions available with PRIMOS C automatically suppress promotion. See Examples 1 and 2 on pages 5-8 and 5-9.

Be particularly cautious when you write code in which a C function receives a parameter, then passes it to a non-C routine. The C compiler lets you write code that hides the fact that data type promotion has occurred, as shown in the following example:

```
foo(c,s,l,f,d)
char c;
short s;
int l;
float f;
double d;
{
   /* Any code here */
}
```

With the PRIMOS C compiler, this method works correctly, even though it is not standard C. The compiler makes the assumption that `c` and `s` were really passed as type **int** and that `f` was really passed as type **double**. The compiler warns you about the type change only when you use the -VERBOSE option. If you pass one of these parameters to a non-C

routine, and you do not use -VERBOSE on the command line, you can easily forget what data type you are really passing. See Example 6 on page 5-11.

In an ANSI C program using function prototypes, variables passed to a function are converted to the parameter types of the function's prototype. If the parameter list ends with (, ...), however, the default argument promotion is done in the same way as for non-ANSI C.

First Element of an Array

In a C program, arrays are indexed starting with item zero, as shown in the following example:

```
static int array[3] = {1, 2, 3};
main( )
{
   printf("array[2] = %d. \n",array[2]);
}
```

This program prints

```
array[2] = 3.
```

because `array[2]` is the third element of the array, which is 3. In FORTRAN and many other languages, the first element of an array is item one. This simple difference needs no elaboration, but remember to watch for it in interlanguage programs. See Example 9 on page 5-13.

Representation of Strings

In C, a string is simply a NULL-terminated array of type **char**. Most C library functions that operate on strings use the NULL byte to determine when they have reached the end of the string. For example, if you use the conversion specification %s in a format specification for the printf( ) function, printf( ) expects the corresponding variable to be a NULL-terminated array of characters. If the NULL byte is missing, printf( ) cannot format the data properly.

Other high-level languages do not terminate their strings with a NULL byte. Instead, they store the length of the string along with its contents. If your C program uses strings received from non-C routines, you may wish to add a NULL byte so that you can treat them like ordinary C strings. Alternatively, if you pass such a string to a formatting function such as printf( ) or sprintf( ), you can use a precision specification so that the function accesses only a specific number of characters. When you pass a string from a C program to certain non-C routines, you may be required to pass its size, also. See Examples 1, 4, and 6 on pages 5-8, 5-10, and 5-11.

Passing Arrays as Parameters

In C, the name of an array is actually a pointer to the first element of the array. You pass this pointer when you pass an array to a function. Other 50 Series languages, such as F77 and PL/I, pass the first element of the array. Thus, array handling in C has one more level of indirection than in other languages.

When you pass an array to a C function from another language, you may use either of two methods to deal with the extra level of indirection. One method is to force the other language to pass C the address of the array. You can do this by using the LOC function in F77 or the ADDR function in PL/I. See Examples 9 and 12 on pages 5-13 and 5-16. Another method is to use a dummy integer variable to receive the array, then cast its address to a pointer type. See Examples 10 and 13 on pages 5-14 and 5-16. When you pass an array from a C program to a routine written in another language using the special interlanguage calling conventions, the array is passed in a manner that is compatible with the other language. See Examples 1, 2, and 4 on pages 5-8, 5-9, and 5-10.

Be cautious when you write code in which a C function receives an array as a parameter, then passes the same array to a non-C routine. The C function actually receives a pointer, and you must cast the pointer to an array type before you can pass it to a non-C routine. See Example 4 on page 5-10.

Pointer Size in 64V-Mode and 32IX-Mode C

The pointer formats for 64V-mode and 32IX-mode C are described in Appendix E. 64V-mode C uses 48-bit pointers; 32IX-mode C uses 32-bit pointers. Other 50 Series languages vary with respect to pointer size. For example, Pascal uses 48-bit pointers. PL/I uses 48-bit pointers by default, but you can pass a 32-bit pointer from PL/I by using the SHORT option. F77 does not use pointers, but you can get the address of an F77 variable with the LOC function, which generates a 32-bit pointer.

When you pass a pointer to a C function from a PL/I program, pass a full 48-bit pointer if the function is compiled in 64V mode. You may pass either a 32-bit or a 48-bit pointer if the function is compiled in 32IX mode. When you call a C function from Pascal, the pointer is the correct size for 64V-mode C, and 32IX-mode C simply ignores the extra information. When you use LOC to pass an address from F77 to a C program, the pointer is the correct size for 32IX-mode C, but is missing some information expected by 64V-mode C. This is not a problem, however, because all data types in F77 are aligned on an even byte, so the byte offset bit is never set. As a result, the short pointer is interpreted correctly by 64V-mode C. See Examples 8, 9, 11, and 12 on pages 5-12, 5-13, 5-15, and 5-16.

The special interlanguage calling conventions, described later in this chapter, allow you to pass 48-bit pointers from both 64V-mode C and 32IX-mode C when you call other language routines. See Examples 1 and 2 on pages 5-8 and 5-9.

External Identifier Names in 32IX Mode

Two versions of the C libraries exist on your system: one for use by 64V-mode programs and the other for use by 32IX-mode programs. A special naming convention helps to insure that the correct routine is linked. In 32IX mode, the prefix G$ is prepended to each external symbol name in the user code. The 32IX mode library, which was also compiled in 32IX mode, also has G$ prepended to its routines, so that the correct link is made.

This naming mechanism becomes visible to you, as a programmer, only when you share a common block between 32IX mode-C and another language. The non-C code must explicitly specify the G$ prefix. See Examples 19 through 22 on pages 5-23 through 5-25.

Functions Returning Pointers

Most non-C routines return pointers in a register that is the wrong one for 32IX-mode C programs. The interlanguage calling conventions on 50 Series systems do not resolve this incompatibility. Non-C functions that return pointers must be declared as type **int** in a C program. The returned value must then be cast to a pointer type. See Example 16 on page 5-20.

Note

The C library's routines for dynamic memory management (malloc( ), calloc( ), realloc( ), free( ), and cfree( )) are designed for use only with each other. If you allocate memory with code written in another language, do not deallocate it with a C routine. Similarly, if you allocate memory with a C routine, do not deallocate it with code written in another language.

CALLING OTHER LANGUAGE ROUTINES FROM C PROGRAMS

Prior to Rev. 19.4, the PRIMOS C compiler supported code generation in 64V mode only. These early versions supported an awkward, antiquated interlanguage-calling convention now known as -oldFORTRAN. With -oldFORTRAN, the **fortran** keyword was used to declare any non-C routines. For example, the PRIMOS subroutine SRCH$$ was declared as follows:

```
fortran void srch$$( );
```

In the argument lists to these routines, the & character was given special meaning. When placed in front of simple variables, the & character caused them to be passed by reference. For compatibility reasons, this convention, now known as the -OLDFORTRAN option, is still available in 64V mode, although -NEWFORTRAN is the default. We strongly suggest that you make whatever source changes are required and use the -NEWFORTRAN compile line option in 64V mode.

In 32IX mode, only -NEWFORTRAN is available. All the explanations and examples in this chapter assume that you are using the -NEWFORTRAN option when you compile your program.

The fortran Storage Class

When your C program calls a subroutine written in another language, you must suppress the default action of converting **char** and **short int** arguments to **int** and **float** arguments to **double**. You must also enable the passing of arguments by reference. Further, if your C program is compiled in 32IX mode, you must force the compiler to use a compatible stack frame format and to pass 48-bit pointers. You do all of these things by declaring the non-C routine with a storage class of **fortran**. Note that you use the **fortran** storage class for all non-C routines, not just FORTRAN language routines.

In general, PRIMOS C passes arguments to routines declared with the **fortran** storage class by reference, if possible. However, certain data types, parenthesized lvalues, and non-lvalues are passed by value. (An lvalue is a data item that may appear on the left side of an assignment statement.) The specific rules are as follows.

Arguments of the following types are passed by value, as described in the following table:

Type	*How Passed*
Characters	As short integers (that is, as the low order byte of a 16-bit halfword)
Pointers	As 48-bit pointers
Bit fields	As long integers
Constants	As whatever type they are
Non-lvalues	As whatever type they are
Parenthesized lvalues	As whatever type they are

Lvalues of the following types are passed by reference: short integer, long integer, float, double, structure, union, and array (including string constants).

Notes

You can cause any lvalue to be passed by value by putting it in parentheses.

You cannot pass a pointer by reference.

Example 1

The PRIMOS routine TNOU expects an array of characters, followed by a 16-bit integer containing a count of characters in the array. Valid calls to TNOU from C are shown in the following program.

```
main( )
{
   fortran tnou( );
   static char buffer[] = "Hi there";
   char *p = "Another test";
   tnou("This is a test", 14);
   tnou(buffer, (short)strlen(buffer));
   tnou((char [])p, (short)strlen(p));
}
```

Note the following points about this example:

- TNOU is declared with the storage class **fortran.**
- The constant number 14 is passed correctly.
- The constant string ''This is a test" is passed correctly.
- The string called buffer is passed correctly because it is declared as an array.
- The string called p is declared as a pointer, so it must be cast to an array before it is passed.

Note that the following call is incorrect.

```
main( )       /* THIS EXAMPLE IS WRONG!! */
{
   fortran tnou( );
   char *p = "Another test";
   tnou(*p, (short)strlen(p));  /* This won't work!! */
}
```

This is wrong because *p has type **char.** If you passed *p, the first character of the string would be converted to a short integer and passed by value.

Example 2

This example shows a C program that passes various data types to non-C routines called FUN and MOREFUN.

```
main( )
{
   short s, *p;
   struct {short a; char buf[10], int f1:5,f2:4;} str;
   long a[10];
   fortran fun( );
   fortran morefun( );
   fun(s, (s), (float)s, a[s], (a[s]), *p, p);
   morefun(str, str.buf, str.buf[str.a], str.f1);
}
```

The parameters are passed to fun as follows:

- s is passed by reference as short.
- (s) is passed by value as short.
- (float)s is passed by value as float.
- a[s] is passed by reference as long.
- (a[s]) is passed by value as long.
- *p is passed by reference as short.
- p is passed by value as a pointer.
- str is passed by reference as a structure.

- `str.buf` is passed by reference as an interlanguage compatible array.
- `str.buf[str.a]` is passed by value as a short integer.
- `str.fl` is passed by value as a long integer.

Example 3

Some PRIMOS subroutines expect to receive a character as a 16-bit integer. In the following example, the character pointed to by `cptr` is passed correctly to the subroutine T1OU.

```
stillAnotherMain( )
{
   char *cptr;
   fortran t1ou( );
   t1ou(*cptr);
}
```

Example 4

When a C routine is passed an array as a parameter, it actually gets a pointer. If you want a C routine that receives an array to pass it to a non-C routine, you must cast it first, as shown in the following example:

```
foo(string)
char string[];   /* The compiler changes this to char *string */
{
   fortran tnou( );
   tnou((char [])string, (short)strlen(string));  /* OK */
}
```

The following call does not work correctly, because `string` is a pointer, not an array, when it is received as a parameter by the routine `foo`.

```
foo(string)      /* THIS EXAMPLE IS WRONG!! */
char string[];   /* The compiler changes this to char *string */
{
   fortran tnou( );
   tnou(string, (short)strlen(string));  /* THIS DOESN'T WORK!! */
}
```

Example 5

As noted previously, certain data types are promoted when they are passed as parameters to a function, but you can write C code that hides the fact that promotion has occurred. The following program contains a C routine that receives three parameters, then passes the same parameters to a non-C routine. Although the C routine's parameters are declared as **char**, **short**, and **float**, they are actually received as **long int**, **long int**, and **double**, respectively. As a result, NON_C_ROUTINE receives two **long int**s and a **double**.

```
Subroutine (c,s,f)
     char c;          /* Changed to long int */
     short s;         /* Changed to long int */
     float f;         /* Changed to double */
main( )
{
     fortran non_C_routine( );
     non_C_routine(c,s,f);
}
```

Example 6

Some F77 and PL/I routines are coded to accept a string of variable size as a parameter. When you pass a string argument from a C program to such a routine, you must pass an additional parameter in order to describe the true length of the string argument. Furthermore, if nonstring parameters are passed to such a routine along with the string, you must pass length arguments for all the parameters, even the nonstring parameters. The argument list must contain the actual arguments followed by the length arguments, in the same order. In the case of a nonstring argument, the value zero is used as a length argument.

In the following example, a string variable, a **short**, and a string constant are passed to the F77 or PL/I routine DEMO. The F77 and PL/I code are also shown.

C program:

```
main( )
{
   static char string[] = "hi there";
   short idummy;
   fortran demo( );
   demo(string, idummy, "other string",strlen(string), 0, 12);
}
```

F77 routine:

```
SUBROUTINE DEMO(S1, SHORT, S2)
CHAR*(*) S1, S2
INTEGER*2 SHORT
RETURN
END
```

PL/I routine:

```
demo: proc(s1, short, s2);
      dcl (s1,s2) char(*);
      dcl short fixed bin(15);
      return;
end;
```

Note that in the call to the subroutine DEMO, a value of 0 was passed in the fifth position, corresponding to the argument whose type was **short**.

Example 7

Some non-C routines change the value of pointers that they receive as parameters. Such routines require you to pass pointers by reference. Unfortunately, C programs pass pointers by value, even to routines declared with the **fortran** storage class. To pass a variable by reference, you must declare it to be an integer type. After the call to the routine, the integer contains the desired value. If you want to use this value as a pointer in your program, you must then cast the integer to a pointer type. On the 50 Series, a direct cast from an integer type to a pointer type, or vice versa, alters the bit pattern. To perform such a cast without altering the bit pattern you must add a level of indirection.

The following program uses a dummy integer variable to pass a pointer to a non-C routine by reference. (See also the discussion of casting between pointer and integer types in Chapter 7.)

```
main( )
{
   long *ptr;
   int dummy;
   fortran void non_c_routine( );

   non_c_routine(dummy);   /* Dummy is passed by reference as an int */
   ptr = *(char **)&dummy;  /* This doesn't alter the bit pattern */
               /* Any code using ptr */

}
```

Note the following points about this example:

- The dummy argument is declared as type **int**.
- Because NON_C_ROUTINE is declared **fortran**, dummy is passed by reference.
- You must cast the value of dummy to a pointer type before you can use it as a pointer.
- You must use a complex cast to avoid altering the bit pattern of the value.

CALLING 64V-MODE ROUTINES FROM OTHER LANGUAGES

Because of the argument type conversion expected by C functions, only integral types (**long int**), double precision real (**double**), and pointers (to any type) may be passed to C routines.

Example 8

The following example shows a PL/I main routine that calls a 64V-mode C subroutine.

PL/I program:

```
main: proc;
    dcl C64Vroutine entry(fixed bin(31), float bin(47), pointer);
    dcl charArray char(100) static
                 init('This is a test, only a test');

    call C64Vroutine(123, 3.14159, addr(charArray));
end;
```

C subroutine:

```
void C64Vroutine(longInt, doubleReal, charPointer)
int longInt;
double doubleReal;
char *charPointer;
{
   printf("Arguments are: %d, %f, %.20s\n",
                          longInt, doubleReal, charPointer);
}
```

Note the following points about this example:

- The character array passed from PL/I is not NULL terminated. Therefore, the call to printf( ) contains the precision specification .20 so that printf( ) accesses only a specified number of characters.
- The declaration of the C subroutine in the PL/I program specifies a full 48-bit pointer.
- The PL/I program specifically passes the address of the string.

Example 9

The following C routine expects to be called with an array of short integers passed in from another language:

```
void JustForFun(array)
short array[];              <-- Two equivalent */
short *array;               <-- declaration options */
{
   /* Use of the array */
}
```

This C routine can be called correctly by the two following programs:

F77 program:

```
INTEGER*2 ARRAY(10)
CALL JUSTFORFUN(LOC(ARRAY))
STOP
END
```

PL/I program:

```
main: proc;
      dcl JustForFun entry(pointer);
      dcl array(10) fixed bin(15);
      call JustForFun(addr(array));
      return;
end;
```

Note the following points about this example:

- Whether the incoming array is declared as `short array[]` or as `short *array`, the C routine always expects a pointer to the first element of the array.
- F77 and PL/I normally pass an array by passing the first member of the array, so you need to add a level of indirection to the array before you pass it from F77 or PL/I to C.
- You add a level of indirection by passing the address of the array returned by LOC in F77 or ADDR in PL/I.
- Because this C routine is compiled in 64V mode, it expects a full 48-bit pointer from PL/I, but it can correctly interpret the 32-bit pointer from F77.

Example 10

The C routine and F77 program below accomplish the same task as Example 9 above.

C routine:

```
void test(dummy)
int dummy;
{
   short *array = (short *)&dummy;
   printf("First two elements = %d %d\n",array[0], array[1]);
}
```

F77 program:

```
INTEGER*2 ARRAY(10)
ARRAY(1) = 3
ARRAY(2) = 10
CALL TEST(ARRAY)
STOP
END
```

Note the following points about this method:

- The F77 program passes the array in normal F77 fashion, so the C routine actually receives the first member of the array by reference.
- The incoming argument to the C routine is declared to be of type **int**.
- The address of the dummy integer is cast to type **short** * within the C routine.
- `ARRAY(n)` in the F77 program becomes `array[n-1]` in the C routine, where *n* is an index value.

CALLING 32IX-MODE C FROM OTHER LANGUAGES

Because of the argument type conversion expected by C functions, only integral types (**long int**), double precision real (**double**), and pointers (to any type) may be passed to C routines. As mentioned previously, and as defined fully in Chapter 6, 32IX mode uses a unique stack frame format. A 32IX-mode C routine that is called from another language must be defined as having the **fortran** storage class in its definition line. This syntax is legal in 32IX mode only.

Example 11

The following example shows the same program that was used in Example 7, but with the C routine compiled in 32IX mode. The two differences between this example and Example 7 are the use of the **fortran** storage class in the C subroutine's definition line and the size of the pointer passed from PL/I. Once again, a PL/I main routine calls a 32IX-mode C subroutine.

PL/I program:

```
main: proc;
    dcl C32IXroutine entry(fixed bin(31),float bin(47),
                           pointer options(short));
    dcl charArray char(100) static
                  init('This is a test, only a test');

    call C32IXroutine(123, 3.14159, addr(charArray));
end;
```

C subroutine:

```
fortran void C32IXroutine(longInt, doubleReal, charPointer)
int longInt;
double doubleReal;
char *charPointer;
{
   printf("Arguments are: %d, %f, %.20s\n",
                          longInt,doubleReal, charPointer);
}
```

Note the following points about this example:

- The definition of the C function begins with the word **fortran.**
- Again, the character array passed from PL/I is not NULL terminated. Therefore, the call to printf( ) contains the precision specification .20 so that printf( ) accesses only a specified number of characters.
- The declaration of the C subroutine in the PL/I program specifies a short pointer. (This is optional.)
- The PL/I program specifically passes the address of the string.

Example 12

This example is the same as Example 9, but the C routine is compiled in 32IX mode. The C routine expects to be called with an array of short integers passed in from another language:

```
fortran void JustForFun(array)
short *array;
{
   /* Use of the array */
}
```

The following two programs can correctly call this C routine:

F77 program:

```
INTEGER*2 ARRAY(10)
CALL JUSTFORFUN(LOC(ARRAY))
STOP
END
```

PL/I program:

```
main: proc;
      dcl JustForFun entry(pointer options(short));
      dcl array(10) fixed bin(15);
      call JustForFun(addr(array));
      return;
end;
```

Note the following points about this example:

- The 32IX-mode C routine has the word **fortran** in its definition line.
- The incoming array is declared as `short *array`, and the C routine always expects a pointer to the first element of the array.
- F77 and PL/I normally pass an array by passing the first member of the array by address, so you need to add a level of indirection to the array before you pass it from F77 or PL/I to C. You do this by passing the array to LOC in F77 or ADDR in PL/I.
- Because this C routine is compiled in 32IX mode, it can handle either a 48-bit pointer or a 32-bit pointer.

Example 13

The C routine and F77 program below are the same as Example 10, but the C routine is compiled in 32IX mode.

C routine:

```
fortran void test(dummy)
int dummy;
{
   short *array = (short *)&dummy;
   printf("First two elements = %d %d\n",array[0], array[1]);
}
```

F77 program:

```
INTEGER*2 ARRAY(10)
ARRAY(1) = 3
ARRAY(2) = 10
CALL TEST(ARRAY)
STOP
END
```

Note the following points about this method:

- The 32IX-mode C routine has the word **fortran** in its definition line.
- The F77 program passes the array in normal F77 fashion, so the C routine actually receives the first member of the array by reference.
- The incoming argument to the C routine is declared to be of type **int**.
- The address of the dummy integer is cast to type **short** * within the C routine.
- Array members `ARRAY(1)` and `ARRAY(2)` in the F77 program become `array[0]` and `array[1]` in the C routine.

CALLING 64V-MODE C FROM 32IX-MODE C

Calling a C routine compiled in 64V mode from a C routine compiled in 32IX mode is similar to calling another language from 32IX-mode C, as described earlier in this chapter. You must declare the 64V-mode routine with storage class **fortran** so that the compiler in 32IX mode uses argument-passing conventions compatible with 64V mode. The 64V-mode routine, however, differs from a foreign language routine in two important respects:

- It expects its parameters to be passed by value.
- It assumes that standard C argument conversions have been performed by the caller.

Because you are using the **fortran** storage class, you must use parentheses if you want arguments to be passed to the 64V-mode routine by value. The use of the **fortran** storage class also disables standard C argument promotion. However, you can expressly cast each **char**, **short int**, or **short unsigned int** to type **int** and each **float** to **double** before passing it. This insures that the 64V-mode routine gets the data types it expects.

CALLING 32IX-MODE C FROM 64V-MODE C

There are two ways to call 32IX-mode routines from 64V-mode C. One method is similar the one discussed above in the previous section, Calling 32IX-mode C From Other Languages. That is, you include the word **fortran** in the definition line for the 32IX-mode routine. For example,

```
fortran void ixroutine(arg1)
int arg1;
{
   /* Code for the 32IX-mode routine */
}
```

This causes the 32IX-mode routine to use a stack frame format compatible with 64V mode. In all other respects, you code as you normally would.

An alternate method is to use the -CIX command line option when you compile the 64V-mode program. In this case, you do not use the **fortran** storage class. The format of the compile line option is

CC *program* -CIX *routine*

where *program* is the name of the 64V-mode source program and *routine* is the name of the routine that was separately compiled in 32IX mode. This causes the compiler to assume that the external routine name was compiled by the C compiler in 32IX mode, and causes the correct 32IX-style calling sequence to be generated. Argument type conversions are compatible between 64V and 32IX C code, so these issues do not cause any problems here. This method is not recommended, because your code is wrong if you forget to use the -CIX option.

Example 14

This example shows the 64V assembler code generated when a 64V-mode C program calls a 32IX-mode routine.

C code to be compiled in 64V mode:

```
main( )
{
   int i;
   double d;
   char *p;
   ci(i,p,d);      */ Assume "-CIX ci" on the command line */
}
```

Generated 64V-mode assembler code:

```
LDL       I               <-- Copy the arguments for pass
STL       SB%+OFFSET+0        by value
LDL       P
STL       SB%+OFFSET+2
DFLD      D
DFST      SB%+OFFSET+4
EAXB      SB%+OFFSET+0    <-- Set up 32IX "argument pointer"
PCL       CI,*            <-- Call 32IX routine
```

Note that no APs are used, all pointers are shortened to two halfwords, and the address of the argument list is placed in the XB register before the PCL. See Chapter 6 for more information.

FUNCTION RETURN TYPES FROM C AND OTHER LANGUAGE ROUTINES

Functions written in C may return values to routines written in other languages, and vice versa. Type compatibilities are shown in Table 5-1.

TABLE 5-1. Language Type Compatibilities

C	*F77*	*PL/I*	*Pascal*
short	INTEGER*2	fixed bin(15)	integer
long	INTEGER*4	fixed bin(31)	longinteger
float	REAL*4	float bin(23)	real
double	REAL*8	float bin(47)	longreal

See Volume I of the *Subroutines Reference* for other language type compatibilities.

Avoid writing interlanguage functions that return characters, structures, or pointers, if possible.

Example 15

Normally, avoid writing interlanguage functions that return structures. However, the following instance is a valid use of a C structure to receive a string returned by a PL/I routine. Assume that TEST is written in PL/I and returns a CHAR(100) VAR.

```
main( )
{
   typedef struct {short length; char data[100];} Cvar;
   Cvar string;
   fortran Cvar test( );
   string = test( );
   printf("String = %.*s\n", string.length, string.data);
}
```

Note the following points about this example:

- The **struct** is declared to contain a **short** followed by an array of 100 characters.
- The **short** variable is used as an argument to printf( ). It corresponds to the * character, which replaces the precision specification, so printf( ) accesses the correct number of characters.

Example 16

Normally, avoid calling routines in other languages that return pointers. However, you may wish to use certain PRIMOS subroutines, such as memory allocation routines, that return pointers. If you are using 32IX-mode C, this presents a problem because certain routines return pointers in a register that is the wrong one for 32IX-mode C programs. To pick up the returned pointer in the correct register, you must declare such a routine type **int** in your 32IX-mode C program. In order to use the returned value as a pointer, you must cast it to a pointer type. However, casting between pointer and integer types on the 50 Series alters the bit pattern, unless you add a level of indirection. (See the discussion of casting in Chapter 7.)

The following example shows a 32IX-mode C function that uses a pointer returned by the PRIMOS routine STR$AL.

```
char *foo(bytes)
unsigned int bytes;
{
   fortran int str$al( );    /* Returns a pointer value as an integer */
   short ecode;
   int temp;                /* Intermediate storage for pointer value */
   char *pointer;

   temp = str$al(0, bytes >> 1, 0, ecode);
   pointer = *(char **)&temp;   /* Convert integer to pointer without
                                   changing the bit pattern */
}
```

Note the following points about this example:

- STR$AL is declared as type **int**, with storage class **fortran**.
- A temporary variable of type **int** holds the returned value.
- The returned value is cast to a pointer type so that it can be used as a pointer.
- A complex cast is used so that the bit pattern is not altered.

MAKING YOUR CODE CORRECT FOR BOTH MODES

Sometimes you must write C code that can be safely compiled in either 64V mode or 32IX mode. For your convenience, PRIMOS C provides the predefined symbol __CI. The symbol __CI is always defined when you compile in 32IX mode, but not when you compile in 64V mode. You can use this symbol with the preprocessor commands **#ifdef**, **#else**, and **#endif** to make your code correct for both 64V and 32IX mode. (See the description of the -UNDEFINE compile line option on page 2-34 for more information about predefined symbols.)

Example 17

This example shows a C function that is called from another language. If you compile this in 32IX mode, you must declare the storage class of this funcion as **fortran** in the definition line. However, this syntax is illegal if the function is compiled in 64V mode. Note the use of the symbol __CI.

```
#ifdef __CI
fortran short foo( )
#else
short foo( )
#endif
{
   /* Code */
}
```

USING THE PRIMOS CONDITION MECHANISM FROM C

You can access the PRIMOS condition mechanism from C. Use the PRIMOS routine MKON$P to set up on-units, and the C library functions setjmp( ) and longjmp( ) to perform non-local gotos. When a condition is signaled, PRIMOS makes a call on your behalf to the routine specified as the handler for the condition in a previous call to MKON$P. PRIMOS passes one argument to the handler, a pointer to the condition stack frame. PRIMOS expects to be able to transfer this argument in the standard fashion. Therefore, in 32IX mode, the handler must be declared with storage class **fortran**.

Example 18

This example shows correct use of the MKON$P routine and of the setjmp( ) and longjmp( ) library functions. Note the use of the predefined symbol __CI to make the C code correct for both 64V mode and 32IX mode.

Main C program:

```
#include <stdio.h>
#include <setjmp.h>

static int s, val;
static jmp_buf env;

main( )
{
#ifdef __CI
   fortran void handler( );
#else
   void handler( );
#endif
   fortran mkon$p( );
   fortran sleep$( );

/* Set up on-unit for "QUIT$" */
   mkon$p("QUIT$", 5, handler);
   val = setjmp(env);
   printf("Starting right here!\n");
   while(1)
   {
       sleep$((long)400);
       printf("Looping. . . \n");
   }
}
```

Handler routine:

```
#ifdef __CI
fortran void handler(cfh)
#else
void handler(cfh)
#endif
int *cfh;
{
   char c;

   printf("\nCaught QUIT$ condition.\n");
   printf("\nContinue? y or n: ");
   scanf("%c%*c", &c);
   if (c != 'n')
      longjmp(env, val);
   exit(0);
}
```

COMMON BLOCKS

C may access common blocks defined in other languages, and other languages may access common blocks defined in C. To access a common block created in another language from a C subroutine, you must declare the common block name with a storage class of **extern** in the C routine. To declare a common block in C, declare any variable at level 0 (outside of any procedure) without any storage class.

Data types must be compatible between the C common block descriptions and the common block descriptions in the other languages. Therefore, avoid putting either single characters or pointer types in common blocks shared between C and other languages. You may use character arrays.

As mentioned previously, all external symbols referenced or defined in 32IX mode have the G$ prefix prepended to their names. This poses a potential problem. If you define a common block in 32IX-mode C and reference it from another language, you must explicitly specify the G$ prefix in your reference. Similarly, if you define a common block in another language and reference it from 32IX-mode C, you must define the common block with a G$ prefix in the other language's code. Also, if you use the advanced symbol placement commands of the SEG loader to move 32IX-mode C common blocks, you must specify the G$ prefix (for example, A/SYM G$EXAMPLE 100).

Example 19

The following example shows one common block created in F77, two common blocks created in PL/I, and a 64V-mode C function that imports all of them.

F77 declaration for 64V-mode C:

```
INTEGER*2  S
REAL*4 F
COMPLEX*16 C
COMMON /F77COM/ S,F,C
```

PL/I declaration for 64V-mode C:

```
dcl aLongInteger fixed bin(31) external;
dcl 1 complex external,
    2 real_part float bin(23),
    2 cplx_part float bin(23);
```

64V-mode C function:

```
LetsGetSomeExternalData( )
{
   extern struct { short s;
                   float f;
                   struct { double r_part;
                            double c_part; } c; } F77com;
   extern int aLongInteger;
   extern struct { float r_part;
                   float c_part; } complex;
}
```

Example 20

This example is the same as Example 19, except that the C routine is compiled in 32IX mode. One common block is created in F77, two common blocks are created in PL/I, and a 32IX-mode C function imports all of them. Note the G$ characters in the non-C declarations, but not in the 32IX-mode C function.

F77 declaration for 32IX-mode C:

```
INTEGER*2  S
REAL*4 F
COMPLEX*16 C
COMMON /G$F77COM/ S,F,C
```

PL/I declaration for 32IX-mode C:

```
dcl G$aLongInteger fixed bin(31) external;
dcl 1 G$complex external,
    2 real_part float bin(23),
    2 cplx_part float bin(23);
```

32IX-mode C function:

```
LetsGetSomeExternalData( )
{
   extern struct { short s;
                   float f;
                   struct { double r_part;
                            double c_part; } c; } F77com;
   extern int aLongInteger;
   extern struct { float r_part;
                   float c_part; } complex;
}
```

Example 21

If you must declare a common block that is acceptable to both 64V-mode C and 32IX-mode C, you must code the block with the G$ characters prepended, then use the ___CI symbol in the C function to add the G$ characters in 64V mode but not in 32IX mode. This example is the same as Examples 17 and 18. It is correct for both modes of C. One common block is created in F77, two common blocks are created in PL/I, and a C function imports all of them.

F77 declaration:

```
INTEGER*2  S
REAL*4 F
COMPLEX*16 C
COMMON /G$F77COM/ S,F,C
```

PL/I declaration:

```
dcl G$aLongInteger fixed bin(31) external;
dcl 1 G$complex external,
    2 real_part float bin(23),
    2 cplx_part float bin(23);
```

C function, correct for both modes:

```
LetsGetSomeExternalData( )
{
   extern struct { short s;
                   float f;
                   struct { double r_part;
                            double c_part; } c;
#ifdef __CI
                                                  } F77com;
#else
                                                  } G$F77com;
#endif
#ifdef __CI
   extern int aLongInteger;
#else
   extern int G$aLongInteger;
#endif
   extern struct { float r_part;
                   float c_part;

#ifdef __CI
                                  } complex;
#else
                                  } G$complex;
#endif
}
```

Example 22

The following example shows a common block created in C and accessed by F77 and PL/I routines. The predefined symbol __CI is used to make the C code correct for both 64V mode and 32IX mode. (Compare Examples 19 and 20 with Example 21, to determine how to code this for 64V alone or for 32IX alone.)

C declaration of a common block:

```
#ifdef __CI
short s;
#else
short G$s;
#endif
struct { float f;
         long anotherLongInteger;
#ifdef __CI
                                  } ss;
#else
                                  } G$ss;
#endif
main( )
{
   /* Code for main program */
}
```

F77 routine:

```
INTEGER*2 S_VALUE
COMMON /G$S/ S_VALUE
REAL*4 F
INTEGER*2 L
COMMON /G$SS/ F,L
```

PL/I routine:

```
dcl G$s fixed bin(15) extern;
dcl 1 G$ss extern,
    2  f float bin(23);
    2  l fixed bin(31);
```

CALLING MIDASPLUS FROM C

You can call the MIDASPLUS data management system from C routines. Be sure that you understand MIDASPLUS thoroughly before you attempt to access it from C.

You must do the following things when you use the callable interface to MIDASPLUS:

- Use the (default) -NEWFORTRAN option on your CC command line.
- Use the **fortran** storage class to define all MIDASPLUS routines.
- Use OPENM$ and CLOSM$ to open and close MIDASPLUS files. Do not use PRIMOS file system routines, such as SRCH$$ and TSRC$$.
- Do not use hard-coded file units when you open MIDASPLUS files. Use the K$GETU key to allow PRIMOS to select an available file unit.
- Include the files PARM.K and KEYS. They contain flags and keys needed by MIDASPLUS.
- When you handle MIDASPLUS errors in C programs, use a zero for the alternate return argument, and check the communications array after the call for any error conditions. C does not support label variables, so it does not support alternate return arguments.

Under some circumstances, using zero for the alternate return argument is not satisfactory. For example, some MIDASPLUS routines, such as PRIBLD, do not support a communications array argument that is cast to a long integer (that is, `(long)0`). These routines terminate the calling program with an error message if you use `(long)0` as the alternate return argument. Also, some MIDASPLUS routines, such as ADD1$, support the communications array and classify errors as fatal or nonfatal. These terminate if a fatal error occurs and 0 was used for the alternate return argument.

Example 23

To handle the error conditions described above, you can write a routine in a language that supports alternate returns, such as FTN, F77, or PL/I. The following is an example of an interlude written in FORTRAN for ADD1$. It returns 0 if no fatal error occurs, and 1 if a fatal error occurs.

FORTRAN interlude to MIDASPLUS:

```
      INTEGER*4 FUNCTION ADD1$C(FUNIT, BUFFER, KEY, ARRAY, FLAGS,
     &                          INDEX, FILENO, PLENTH, KEYLNT)
      CALL ADD1$(FUNIT, BUFFER, KEY, ARRAY, FLAGS, $10, INDEX,
     &           FILENO, PLENTH, KEYLNT)
      ADD1$C = 0
      RETURN
10    ADD1$C = 1
      RETURN
      END
```

Example 24

The following C program accesses a MIDASPLUS file.

```
/* This is a C program that opens an indexed file,          */
/* reads a record, and displays it.                         */

#include <keys>                    /* Primos I/O keys */
#include <parm.k>                  /* Flags used by MIDASPLUS */
#include <stdio.h>                 /* Needed for getchar( ) */

main( )
{
/* Data structures:  */
   fortran closm$( ), find$( ), openm$( );
   short int funit, i, status, routine, buffer[43];
   short int array[14];
   char choice;
   static struct thekey {char one[9]; };
   static struct thekey findkey;

/* START EXECUTION:  */

/* Open file: */
   openm$((short)(k$rdwr+k$getu), "bank", 4, funit, status);
   if (status != 0)
      abort( );

/* Ask for key to be entered from terminal:  */
   choice = 'Y';   /*  Next while is repeated as long as choice is yes */
   while ((choice == 'Y') | (choice == 'y'))
   {
      printf("ENTER KEY VALUE (9 NUMBERS): \n");
      i = 0;
```

```
      while (i <=8)
      {
         findkey.one[i] = getchar( );
         i++;
      }  /* end while */

/* Read and display sequential record: */
      find$(funit,
            buffer,
            findkey,
            array,
            (short)(FL$RET + FL$KEY),
            (long) 0,                        /* ALTRTN -- no use in C but
                                                must be long */
            0,                               /* Search on primary key */
            0,                               /* Obsolete for MIDASPLUS */
            0,                               /* Return all data */
            0);                              /* Full key */

/* Check error code in array:  */

      if (array[0] == 0);                    /* Do nothing, 0 is normal */
      else

         if (array[0] == 7)                  /* Key not found   */
            printf("THERE IS NO RECORD WITH THIS KEY\n");
         else
         {
            printf("ERROR -- ASK FOR HELP\n");
               abort( );
         } /* end else */

/* Display what is returned in buffer:  */
      printf("%s\n", buffer);
      printf("\n");
      printf("DO YOU WANT TO CONTINUE? Y or N:\n");
      i= 0;
      getchar( );  /* Throw away last CR */
      choice = getchar( );
      getchar( );  /* Throw away last CR */
   }  /* end while for choice*/

/* Close file:  */
   closm$(funit, status);
   if (status == 0)
      printf ("NORMAL END OF RUN");
   else
      printf ("STATUS IS", "%d\n", status);

}  /* end program */
```

Note the following points about this program:

- The program must be compiled with -NEWFORTRAN.
- The MIDASPLUS routines are declared with storage class **fortran.**
- OPENM$ and CLOSM$ are used to open and close the file.
- The K$GETU key is used in the call to OPENM$.
- The header files PARM.K and KEYS are included.
- The argument `(long)0` signifies to FIND$ that no alternate return point has been specified.

6

ADVANCED TOPICS

This chapter contains information about stack frame formats and shortcalls in 32IX-mode and 64V-mode C. These topics are provided purely for your interest. You do not need to understand this information to program in C.

C STACK FRAME FORMATS

32IX Mode

C in 32IX mode uses a nonstandard stack frame format. One bit of the flags halfword of the standard stack frame header (SFH.FLAGS.MBZ) is used to tag 32IX stack frames. This bit is always 0 for standard stack frames. Setting this bit allows proper handling of the C 32IX stack frame format, register tracking across procedure calls, and shortcall capability. This use is similar to the use of the USER PROC bit, which is set by the -STORE_OWNER_FIELD option of many 50 Series compilers, including C in 32IX mode.

When this extended frame bit is set, it signifies that extension flags exist at SB%+042 and SB%+043. The three Most Significant Bits (MSBs) of SB%+042 tag the type of information present in the following halfwords. Currently, only type 0 (all three bits 0) is defined. Type 0 indicates a C 32IX extended stack frame.

Figure 6-1 shows the stack frame header format for 32IX routines. All numbers are in octal.

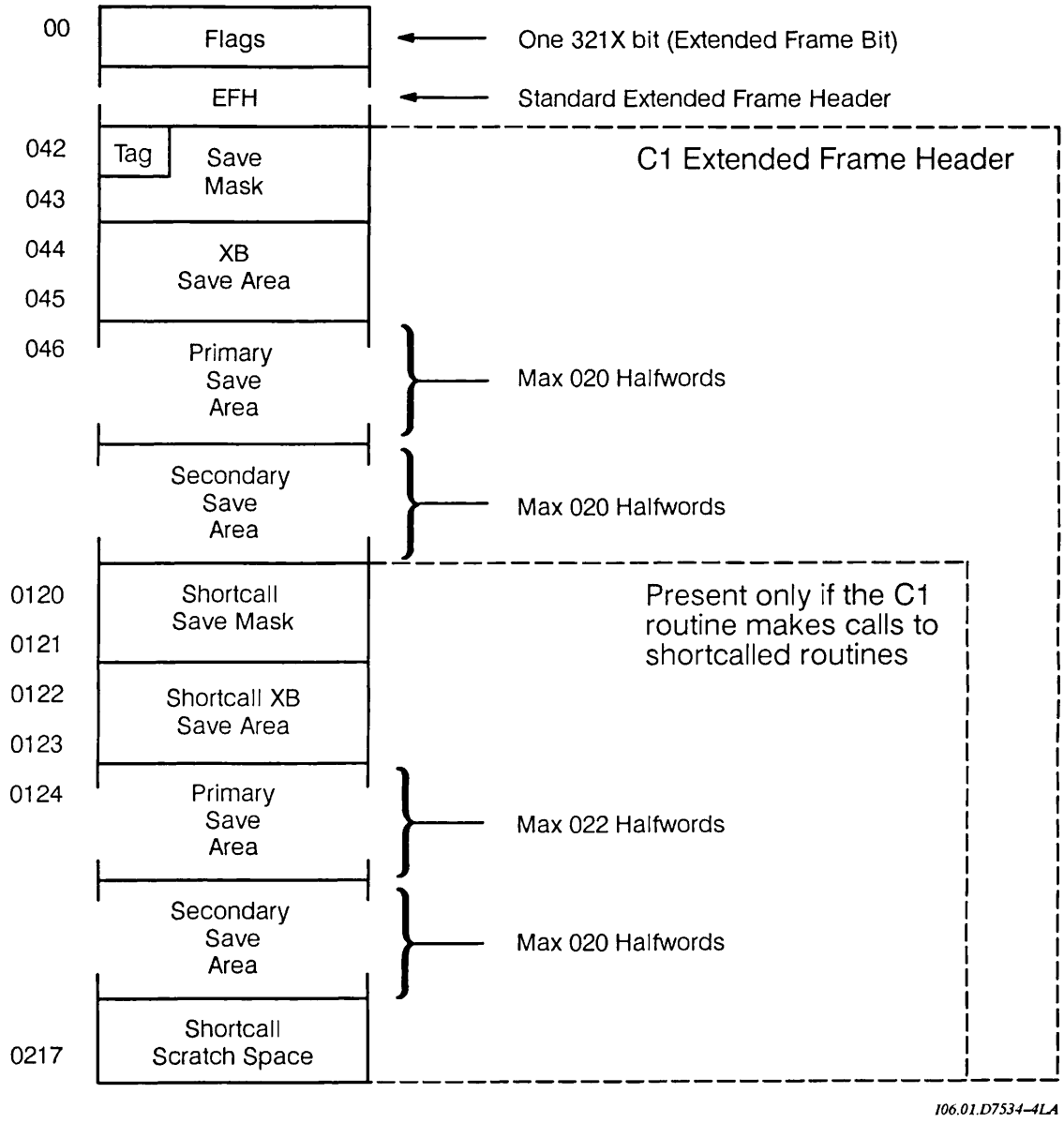


FIGURE 6-1. Stack Frame Header Format for 321X Routines

The format of the first save mask is shown in Figure 6-2.

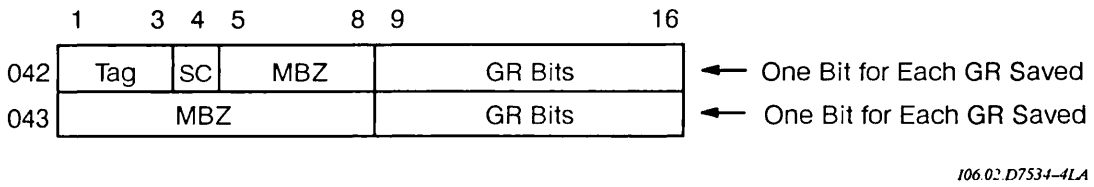


FIGURE 6-2. Format of the First Save Mask

The three tag bits are 0, which signifies that this is a C 32IX extended stack frame. The SC bit is used to tag shortcall frames. This bit is set on entry to all 32IX-generated shortcall routines and reset on exit. Setting the SC bit signifies that a shortcall routine is currently executing and that the shortcall frame header (starting at SB%+0120) is active. The GR bits at halfword SB%+042 comprise the save mask for the secondary save area. The GR bits at halfword SB%+043 comprise the save mask for the primary save area. All MBZ bits are reserved and may be used for future expansion of the 32IX frame header.

The format of the shortcall save mask is shown in Figure 6-3.

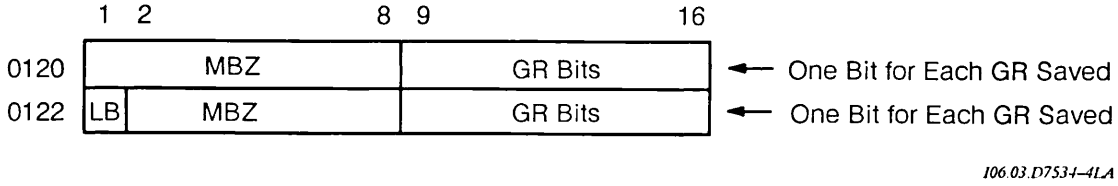


FIGURE 6-3. Format of the Shortcall Save Mask

The GR bits at halfword SB%+0120 represent the save mask for the shortcall secondary save area. The GR bits at halfword SB%+0121 represent the save mask for the shortcall primary save area. The LB bit is used to denote the saving of the link base register (LB%). This is why the shortcall primary save area can be as many as 022 halfwords long, rather than 020. All MBZ bits are reserved and may be used for future expansion of the 32IX frame header.

When a C 32IX routine is entered, all nontemporary registers to be used by the routine are saved in the primary save area. The extended frame bit and the primary save mask are set. This is done as a long store, so the secondary save mask is set to 0 also. Currently, registers R3 to R7 are considered nontemporary and are tracked across procedure calls. Each

C 32IX routine has a single return point where any saved registers are restored before the PRTN.

With 32IX C, APs are never used to pass arguments. Rather, arguments are placed on the caller's stack in contiguous memory. Before the PCL or shortcall, the XB% is set to point to the start of this argument template. Thus, the current XB% must usually be saved on the caller's side before a procedure call and restored after the return. The 32IX stack frame header reserves a long (32-bit) word for this purpose. Because this is a consistent operation, and because the save location is constant, there is no need to update save masks here.

Before doing a ZMVD or calling a **fortran** storage class routine, a C 32IX routine must save registers that are currently in use and that may be corrupted by the operation. These saves are done into the secondary save area, and the secondary save mask is set. When the operation is completed, the secondary save mask is reset to 0.

The primary save area contains registers saved by the callee. The secondary save area contains registers saved by the caller. Except for unusual conditions (ZMVD, **fortran** routines, and intrinsics) all registers are saved by the callee.

The primary save area of a stack frame must be restored when the stack is unwound, via a longjmp( ), past the frame. The restoration is necessary because the primary save area contains the registers that must be active for the previous frame. The secondary save area of a frame must be restored only when that frame is the target frame of a stack unwind.

A 32IX C routine that calls a shortcall routine reserves a 0100 halfword block of stack space at SB%+0120. The first halfwords of this area are used as the stack frame header for the shortcalled routine, because it does not have its own stack frame. The rest of the space is used for automatic variables.

The shortcalled routine's stack frame header has the same format and meaning as a normal 32IX C stack frame header, but the former starts at SB%+0120 rather than SB%+042. When any C-32IX-generated shortcalled routine is entered, the shortcalled bit (SC bit of SB%+042 of the extended stack frame) is set. When the shortcalled routine returns, the bit is reset.

During stack unwind, if the shortcalled bit of a frame is set, the stack frame header in the shortcall area is processed before the normal stack frame header. This handles the case of a shortcall routine longcalling another routine that then calls longjmp( ) and causes a stack unwind.

64V Mode

The following example shows a brief but complete C program.

```
main( )
{
     int i;
     char c;
     float f;
     double d;

     foo(i,c,f,d);
}

foo (i,c,f,d)
int i;
int c;
double f;
double d;
{
     /* Any code here */
}
```

This program can be compiled and loaded with either BIND or SEG. Figure 6-4 represents the runtime environment during execution of the program.

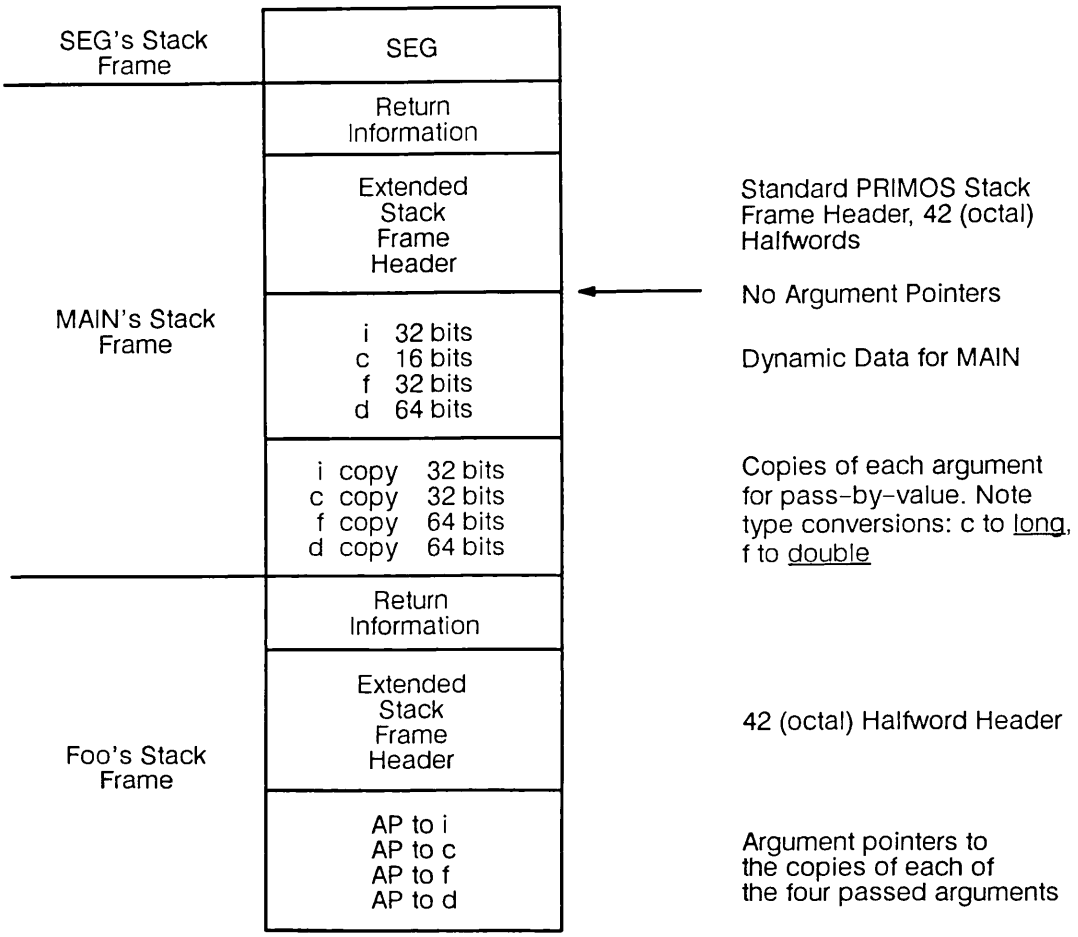


FIGURE 6-4. 64V-mode Runtime Environment

The subroutine foo could perform another procedure call and pass any of its parameters on to the called procedure as arguments. In that case, a new copy of each parameter would be made and the passed argument pointers would point to these copies. This standard, pass-by-value method works correctly in all normal cases.

Problems arise, however, if a routine that accepts many arguments is called with too few arguments. The called routine attempts to pass all of its arguments on to yet another routine. When the middle routine attempts to copy its arguments to pass them by value, an argument is missing, so a pointer fault occurs.

The following example shows a program with such a bug.

```
buggy( )
{
     error(format,a1,a2);
}

error(format,a1,a2,a3,a4,a5)
char *format;
{
     fprintf(stderr, format, a1, a2, a3, a4, a5);
}
```

When the function called error attempts to copy a3 for the call to fprintf( ), a pointer fault is raised. This problem exists in 64V mode only, since 32IX-mode C does not use argument pointers.

Avoid calling a function with more or fewer parameters than the function expects. (See Chapter 7.) If you must code in this manner, however, you can avoid problems in 64V mode by using the -NOCOPY command line option. (-COPY is the default.) This causes function parameters passed on to other functions to be passed by reference rather than by value. The two compile line options -NOCOPY and -COPY exist in 64V mode only.

SHORTCALLS

For general information about the shortcall mechanism, see the *Assembly Language Programmer's Guide* and the *Instruction Sets Guide.*

Shortcalls From 32IX Mode

When you use the command line option -SHORTCALL *name* in 32IX mode, the compiler generates a JMP rather than a PCL to the external routine name. The normal C-style (pass-by-reference) argument template is built up before the JMP is generated. The address of the start of the argument list is placed in the XB% register (via an EAXB). The return address for these routines is placed in R0 before the JMP. These shortcalled routines may be written in PMA, or they may be written in C and compiled in 32IX mode with a -SHORTCALL command line option specifying the name of the routine in the source file that is to be generated as shortcallable.

Unlike other 50 Series language implementations of shortcall, 32IX C shortcalled routines have an ECB. Calls to the routines are made indirectly through the first halfword of the ECB using a JMP rather than a PCL. All calls to shortcalled routines pass the address of the shortcalled routine's ECB in a register (R1). The shortcalled routine can thus find the value for its link base (by looking in its ECB) and create one for itself. This involves saving the current link base on entry and restoring it on exit from the shortcalled routine. A bit in the shortcall primary save mask indicates that the link base has been saved. If the shortcall routine does not use any static or external data, the generated code does not save and restore the previous link base value.

The ECBs of shortcalled routines must be tagged so that the determination of the type of call to make (short or long) can be made at runtime. At runtime, the address of shortcalled routines is taken, and calls are made through pointers to functions that may point to either shortcalled or longcalled routines. The tag for ECBs belonging to shortcalled routines is the value -1 in the number of arguments field. The use of an invalid number in this field does not cause any problems because the compiler never generates a PCL through a shortcalled routine's ECB. This mechanism offers a great deal more flexibility than other shortcall implementations.

Taking the address of static shortcalled routines is permitted. The address of a static shortcalled routine is the address of the routine's ECB, as is the address of other routines. However, static shortcalled routines assume that they can share the link base of their caller. Unlike other shortcalled routines, static shortcalled routines do not expect their ECB address to be passed to them. Thus, the address of a shortcalled routine cannot be passed to and called from another routine because the wrong link base would be referenced. This is consistent with C's concept of static.

The following code sequence is used by 32IX C to perform shortcalls. Note that expanded listings produced by 32IX C use Rx+<offset>,* to denote GRR addressing, that is, register indirect through Rx. However, this format is not accepted by Prime Macro Assembler (PMA).

```
<Create the argument template by copying
any arguments into contiguous memory>

EAXB    <first argument>        <-- Set "argument pointer"
LIP     R1,<IP to <name>s ECB>
L       Rx,R1+00,*              (GRR)
EAR     R0,<return address>
JMP     Rx+00,*                 (GRR)
```

If the routine to be called is in the same source file as the caller, the last two lines are

```
EAR     R0,<return address>
JMP     <first instruction>
```

The following code sequence is used by 32IX C to return from shortcalled routines.

```
JMP     R0+00,*       (GRR)
                          or, if coding in PMA (with no GRR):
ST      R0,<temp>
JMP     <temp>,*
```

The following restrictions apply to all shortcalled routines:

- They may directly call other shortcalled routines. This is implemented by longcalling a dummy routine that then shortcalls the target routine.
- They have limited automatic data space.

Shortcalled routines may call setjmp( ), the C library equivalent of MKLB$F. However, a bit in the label, the fault bit of the target PB%, must be set. Setting this bit tells longjmp( ), the C library equivalent of PL1$NL, that it is resuming execution in shortcalled code. When longjmp( ) resumes execution, it restores registers from the secondary save area in the shortcall stack frame header (starting at halfword 0120). (For more information about setjmp( ) and longjmp( ), see page 4-54).

The format of a C 32IX label variable created by setjmp( ) is incompatible with the format created by other languages, using MKLB$F, and by 64V-mode C. This incompatibility prevents a label from being created in a C 32IX routine and then passed to another language that could try to do a PL1$NL through the passed label. All nonlocal gotos to C 32IX stack frames go through longjmp( ) so that registers are restored correctly.

The format of the C 32IX label variable is shown in Figure 6-5. Note that the last two halfwords of the label variable are the reverse of the standard label variable.

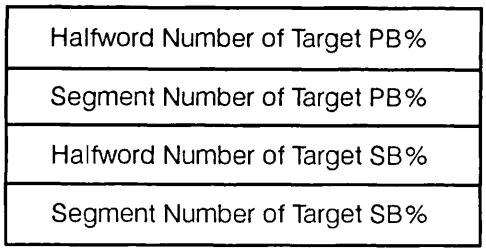


FIGURE 6-5. Format of the C 32IX Label Variable

The implementation of longjmp( ) for 32IX C is more complicated than simply calling PL1$NL. On a nonlocal goto, registers are restored as the stack is being unwound. The sequence of steps is as follows:

1. longjmp( ) walks back the stack to the target frame specified by the SB% entry in the target label. As it does so, longjmp( ) examines each stack frame passed.
2. longjmp( ) then builds a structure containing the state of the register file that should be reinstated before execution is continued in the target frame.

3. The PB% entry in the label variable is modified to point back into longjmp( ).
4. The stack is unwound by a call to PL1$NL.
5. The register file is restored.
6. longjmp( ) does a JMP to the original target PB% location, and execution continues in the target frame.

Shortcalls From 64V Mode

If the compile line option -SHORTCALL name is used in 64V mode, the compiler generates a JSXB rather than a PCL to the external routine name. The normal C-style (pass-by-reference) argument template is built up, and the address of the start of the argument list is placed in the L register (via an EAL) before the JSXB is generated. The shortcalled routine must be written in PMA. The following example shows a program that calls the routine SC.

```
main( )
{
     int i;
     double d;
     char *p;
     sc(i,p,d);
}
```

If this program is compiled in 64V mode with the command line option -SHORTCALL SC, the following code is generated.

```
LDL       I              <-- Create argument template.
STL       SB%+OFFSET+0
LDL       P
STL       SB%+OFFSET+2
LDA       P+2
STA       SB%+OFFSET+4
DFLD      D
DFST      SB%+OFFSET+5
EAL       SB%+OFFSET+0   <-- Point L to first argument.
JSXB      SC,*           <-- Perform the shortcall.
```

7

PORTABILITY CONSIDERATIONS

The first section of this chapter describes features of PRIMOS C that may differ from those of other C implementations.

The second section of this chapter, PRIMOS C Library Functions, contains two lists. The first list compares the functions in the PRIMOS C library with like-named functions in other C implementations. The second list compares functions not provided in the PRIMOS C library with suggested alternative functions that are available in the PRIMOS C library.

FEATURES OF PRIMOS C

This section describes features of 50 Series machines and of PRIMOS C that may differ from those of other implementations. You should take these features into consideration whenever you port C applications to and from PRIMOS C.

Character Set

The basic character set used internally under PRIMOS is the ANSI, ASCII 7-bit set with the 8 parity bit always on. This character set, known as Prime ASCII, is a proper subset of the Prime Extended Character Set (Prime ECS). If your terminal or printer supports Prime ECS, the 8th bit is significant. For terminals and printers that do not support Prime ECS, symbolic characters or Prime ASCII values (decimal 128-255) must be used within programs for character comparisons, and characters may not be used as array indices 0-127. Note that, on a 50 Series machine, a NULL character pointer does not point to a zero. Some code written for other machines uses the 8th character bit as a flag. Such code must be modified for terminals that do not support Prime ECS. (For information about Prime ECS, see Appendix F.)

Blank Compression and Null Padding in ASCII Text Files

On 50 Series machines, ASCII text files are stored on disk with multiple blanks compressed and lines padded to an even number of bytes with the NULL character. All utilities that manipulate files as standard Prime ASCII text manage this blank compression in a manner that is transparent to the user. These utilities include

- C library functions that explicitly manipulate ASCII files and data
- System subroutines that explicitly manipulate ASCII files and data
- PRIMOS text editors

Problems can arise when programs that manipulate ASCII text files using direct access or binary file I/O methods are ported from other machines to a 50 Series machine. Neither direct access nor binary file I/O methods manage the blank compression for the user. See the discussion of fopen( ), fseek( ), and ftell( ) in Chapter 4, Using the C Library.

Text Files Generated by Programs

Some PRIMOS utilities require their input files to have a specific format. Prime EMACS, for example, expects text files to consist of lines terminated by newline characters (0212). If your program generates a text file that lacks newline characters, you cannot use EMACS to view that file.

Parameters Passed to a Function

The number of parameters passed to a function must be equal to the number of parameters expected by that function. On some other machines, you may write code in which a function is called with more or fewer parameters than the function actually expects. Such code may work correctly on the 50 Series, but only if the missing or extra parameter is never referenced. A program fails when it tries to reference a parameter that was not supplied. A function that is expecting an integer parameter does not assume 0 as a default.

Function Return Values

On some other machines, programs run correctly if function return value data types are left undeclared. For example, a program may contain a function that returns a pointer. If this function is not explicitly defined as returning a pointer, the default return value is type **int**. Such a program may run correctly on some machines, but not on a 50 Series machine. All functions must be declared with the proper return value data type to insure proper operation.

Size of Pointers

When a program is compiled in 64V mode, its pointers are 48 bits long. An **int** is 32 bits long. In 32IX mode, pointers and **int**s are the same size, 32 bits. The pointer formats are shown in Appendix E.

Casting Between Pointer and Integer Types

Under some unusual circumstances, you may have to cast a pointer to an integer type, or an integer type to a pointer. On 50 Series systems, pointers are complex data types. If you perform ordinary casts, such as

```
ptr = (char *) num;      /* Changes the bit pattern */
num = (int)ptr;          /* Changes the bit pattern */
```

where `num` is an integer type and `ptr` is a pointer, the C compiler changes the bit pattern of the value. If you add a level of indirection, as shown below, the compiler does not alter the bit pattern.

```
ptr = *(char **)&num;
num = *(int *)&ptr;
```

High Bit of a Pointer or Character

Some code written for other machines uses the most significant bit of a pointer or character as a flag. You cannot use the high bit of a pointer or character as a flag on the 50 Series.

Null Pointers

On some other machines, a NULL pointer points to address zero or to a memory location guaranteed to contain zero. On 50 Series machines, no user has access to word zero of segment zero. In the following example, a pointer is set to zero, then dereferenced. Such code fails and raises the condition ACCESS_VIOLATION$ on a 50 Series machine.

```
main( )  /* This function won't work! */
{
int *p;
p = (int *)0;   /* p is now a NULL pointer */
if (*p = 0) printf("Hello world.\n");
   else printf("Goodbye world.\n");
}
```

Segment-spanning Data Objects

On 50 Series machines, you cannot reference an atomic data object that is split across a segment boundary. Atomic data objects include types **char**, **short**, **int**, **long**, **float**, **double**, and their **unsigned** counterparts.

Under certain circumstances, however, you can reference a non-atomic data object -- a string, array, or structure -- that spans a segment boundary. If your program contains an array of structures that exceeds 128K bytes, a structure member may be split across the boundary if it is not aligned according to its size.

In 64V mode, if you plan to use arrays of a **struct**, align the data objects by padding the **struct** with extra members. Align the types **int**, **long**, and **float**, for example, on addresses that have offsets that are multiples of 4. See Appendix E for details about data formats. In 32IX mode, you can use the compiler option -HOLEYSTRUCTURES to align these data types for you.

Arrays of type **char** may not span a segment in 64V mode, although they may do so in 32IX mode.

If your program contains arrays that span a segment, use the -BIG compiler option. In 32IX mode, if a string argument to strncpy( ), memcpy( ), or strfil( ) is likely to span a segment, use the -SEGMENTSPANCHECKING compiler option in conjunction with either -INTRINSIC or -STANDARDINTRINSICS. See Chapter 2 for more information.

Command Line Arguments

PRIMOS C allows you to pass arguments to a program from the command line. The argument handling is functionally the same as in the UNIX operating system. However, you must link the library CCMAIN or ANSI_CCMAIN before your main program when you use SEG or BIND. See Chapter 3 for more information. You can use numerical command line arguments to a program, provided you use BIND, not SEG, to link the program. When you execute a SEG-loaded program, numerical arguments on the command line are interpreted as options to the SEG command itself.

Input and Output Buffering

Under the UNIX operating systems, the high-level I/O routines fread, fwrite, fscanf, and so forth, are buffered, but the low-level I/O routines read and write are unbuffered. The situation is more complex in the case of the PRIMOS C library functions because more than one level of buffering exists.

Like their UNIX counterparts, fread( ), fwrite( ), and the other high-level C I/O functions use a high-level buffer that is automatically allocated when you call fopen( ). You can eliminate this level of buffering from the high-level I/O functions by calling setbuf( ) with a buffer value of NULL after you call fopen( ). The low-level functions read( ) and write( ), like their UNIX counterparts, do not use this level of buffering.

By default, all C library I/O functions employ very low-level disk read and write buffering as a performance enhancement. Ordinarily, this level of buffering is transparent. You can see it, however, if you quit out of an executing program and type STAT UNIT. For example, the following program reads 80 characters from a file, then prompts you to quit.

```
#include <stdio.h>
main( )
{
  char buf[150], temp[80];
  int i, fileID, open( ), fread( );
  if ((fileID = open("testfile", 0)) == -1)
     {  printf("file system error %d\n", errno);
        exit(errno);
     }
  i = read(fileID, buf, 80);
  printf("%d bytes read from testfile.\n", i);
  puts("Hit CONTROL-P, then type STAT UNIT");
  gets(temp);
}
```

When you quit out of the program and type STAT UNIT, you see the following display:

```
User YOU                                SYSTEM

File    File       Open    File
Unit    Position   Mode    Type  RWlock  Treename
   31  000001024     R      SAM   NR-1W  <DIR>TESTFILE
   32  000000000   VMr      DAM   NR-1W  <DIR>TESTBUF.RUN
```

Note that you are positioned more than 80 bytes into the file TESTFILE.
You can disable this level of read and write buffering by using the additive keys 040000 and 01000, respectively, as *openMode* values when you call open( ). If you need to use high-level I/O functions, such as fread( ) and fwrite( ), with no buffering of either kind, you must use a series of function calls, such as the following:

```
fileID = open("mydir>myfile", 040000 | 01000);
filePointer = fdopen(fileID, "r");
setbuf(filePointer,NULL);
```

Interlanguage Calling

In general, the mechanisms used on other machines are not the same as those needed for the 50 Series. See Chapter 5 for a full discussion of these mechanisms.

Macro Preprocessor

Nested Include Files: In PRIMOS C, include files may be nested up to 20 levels deep.

#define Commands: Unlike some C preprocessors, the macro processor in non-ANSI PRIMOS C performs syntax checking on the arguments to **#define** commands, instead of waiting until the macro is expanded.

For example, non-ANSI C accepts only single character arguments in single quotation marks in **#define** commands. You cannot use a multiple character argument in single quotation marks within a macro definition. For example, the definition

```
#define ctrl(letter) ('letter' & 077)
```

is incorrect and results in an error message. Instead, use the definition

```
#define ctrl(L) ('L' & 077)
```

which expands, for example, ctrl(G) to ('G' & 077).

Similarly, commands like

```
#define HUGE 12345678901234567890
#define HEX  0x
#define OCT  0778
#define CH   't
```

will draw compiler error messages if -ANSI is not specified. If -ANSI is specified, these macros will draw errors only when their expansion results in a syntactically invalid program.

A **#define** macro may have up to 128 formal parameters.

Character Boundary

The boundary for a character on the 50 Series is a 16-bit halfword, not a byte. Character arrays, however, are packed two characters per halfword. Therefore, adjacent character variables do not ordinarily reside on adjacent bytes unless they are members of an array.

Some code written for other machines assumes that adjacent characters always reside on adjacent bytes. If you wish to port such code to a Prime machine, use the -PACKBYTES compiler option to pack adjacent characters within structures or unions. For information about -PACKBYTES, see Chapter 2.

Note that character arrays still start on an even 16-bit halfword boundary, whether or not the code is compiled with the -PACKBYTES option. Moreover, since the C language treats a multi-dimensional character array as an array of arrays, each array starts on an even boundary. This results in "holes" between dimensions if the *n*th dimension contains an odd number of character elements.

Promotion of Character Arguments

If the declaration of a function is old-style (as defined by the ANSI standard), all character arguments are promoted to **int** when they are received as a parameter to a function. However, taking the address of a **char** argument will yield the address of the **int** in which it is stored. If the user desires to use the address of the actual character, then the **char** parameter should be assigned to a locally declared **char** variable. Then the address of the locally declared **char** variable can be used for whatever the user desires.

Identifier Names

In PRIMOS C, identifier names are significant for a maximum of 32 characters. This may cause a problem if a program written on a 50 Series machine is ported to a system on which only eight characters are significant.

Vertical Tab Character

The vertical tab character \v is not recognized by the PRIMOS C compiler. If used, this construct yields a lowercase v.

Case Sensitivity

The PRIMOS C compiler is case sensitive, but neither PRIMOS itself nor the BIND, SEG, and DBG utilities are case sensitive. Therefore, the PRIMOS implementation of C is not case sensitive with respect to external (common) identifier names. In the following program, for example, `var1` and `VAR1` are interpreted as the same variable:

```
int var1;
int VAR1;
main( )
    {
    var1=10;
    VAR1=20;
    printf("var1=%d VAR1=%d",var1,VAR1);
    }
```

On a 50 Series machine, the output of this program is

```
var1=20 VAR1=20
```

Quadruple Precision Floating Point Support

PRIMOS C supports quadruple precision floating point numbers, which are declared **long double**. The -ANSI compiler option includes support for the **long double** data type. If the -ANSI option is not used, then include the -QUADCONSTANTS compiler option to enable support for quad-precision constants; use the -QUADFLOATING option to enable support for quad-precision variables.

PRIMOS C LIBRARY FUNCTIONS

This section contains two lists of library functions in alphabetical order. The first list compares the functions in the PRIMOS C library with like-named functions in other C implementations. The second list pairs functions not provided in the PRIMOS C library with suggested alternative functions that are available in the PRIMOS C library.

PRIMOS C Library Functions Compared With Other C Implementations

abort( )
Does not generate a core dump.

abs( )
Equivalent.

access( )
Modes may differ slightly.

acos( )
Equivalent.

asin( )
Equivalent.

assert( )
Equivalent.

atan( )
Equivalent.

atan2( )
Equivalent.

atof( )
Equivalent.

atoi( )
Equivalent.

atol( )
Equivalent.

cabs( )
Equivalent.

calloc( )
Equivalent.

ceil( )
Equivalent.

cfree( )
Equivalent.

chdir( )
Equivalent.

chrcheck( )
Specific to the PRIMOS C library.

clearerr( )
Equivalent.

close( )
Equivalent.

copy( )
Specific to PRIMOS C library. Similar functionality is provided by link( ) in the UNIX operating systems.

cos( )
Equivalent.

cosh( )
Equivalent.

creat( )
Modes are different. Use open( ), not creat( ), whenever possible.

ctime( )
Format and length of string may differ. Some installations may not support Daylight Saving Time. Consult your System Administrator.

cuserid( )
Equivalent.

delete( )
Specific to the PRIMOS C library. Similar functionality is provided by unlink( ) in the UNIX operating systems.

ecvt( )
Equivalent.

exit( )
The parameter *status* must be passed.

exp( )
Equivalent.

fabs( )
Equivalent.

fclose( )
Equivalent.

fcvt( )
Equivalent.

fdopen( )
Equivalent.

fdtm( )
Specific to the PRIMOS C library. The information provided by this function is a subset of that provided by stat( ) in the UNIX operating systems.

feof( )
Equivalent.

ferror( )
Equivalent.

fexists( )
Specific to the PRIMOS C library.

fflush( )
Equivalent.

fgetc( )
Equivalent.

fgetname( )
Returns a PRIMOS pathname.

fgets( )
Equivalent.

fileno( )
Equivalent.

floor( )
Equivalent.

fopen( )
Access modes differ.

fprintf( )
Equivalent.

fputc( )
Equivalent.

fputs( )
Equivalent.

fread( )
Equivalent.

free( )
Equivalent.

freopen( )
Access modes differ.

frexp( )
Equivalent.

frwlock( )
Specific to the PRIMOS C library.

fscanf( )
Extended. Conversion specification characters differ.

fseek( )
A valid byte position must be obtained with ftell( ) when fseek( ) is used with ASCII files.

fsize( )
Specific to the PRIMOS C library. The information provided by this function is a subset of that provided by stat( ) in the UNIX operating systems.

fstat( )
Equivalent.

ftell( )
Return value differs when used with ASCII files because of disk file space compression.

ftime( )
Equivalent. Some installations may not support Daylight Saving Time. Consult your System Administrator.

ftype( )
Specific to the PRIMOS C library. Similar information is provided by stat( ) in the UNIX operating systems.

fwrite( )
Equivalent.

g$amiix( )
Specific to the PRIMOS C library.

getc( )
Equivalent.

getchar( )
Equivalent.

geth( )
Specific to the PRIMOS C library.

getmod( )
Specific to the PRIMOS C library.

getname( )
Returns a PRIMOS pathname.

gets( )
Equivalent.

getw( )
Equivalent.

gterm( )
Specific to the PRIMOS C library. Similar information is provided by ioctl( ) in the UNIX operating systems.

gvget( )
Specific to the PRIMOS C library.

gvset( )
Specific to the PRIMOS C library.

hypot( )
Equivalent.

index( )
Equivalent. (Synonym for strchr( ).)

isalnum( )
Equivalent.

isalpha( )
Equivalent.

isascii( )
Equivalent.

isatty( )
Specific to the PRIMOS C library.

iscntrl( )
Equivalent.

isdigit( )
Equivalent.

isgraph( )
Equivalent.

islower( )
Equivalent.

ispascii( )
Specific to the PRIMOS C library.

isprint( )
Equivalent.

ispunct( )
Equivalent.

isspace( )
Equivalent.

isupper( )
Equivalent.

isxdigit( )
Equivalent.

ldexp( )
Specific to the PRIMOS C library.

localtime( )
Equivalent, but some installations may not support Daylight Saving Time. Consult your System Administrator.

log( )
Equivalent.

log10( )
Equivalent.

longjmp( )
Equivalent.

lsdir( )
Specific to the PRIMOS C library.

lseek( )
Extended. New values for the direction argument allow positioning by physical disk record.

malloc( )
The first byte of the allocated area is always aligned on a 16-bit halfword boundary.

mkdir( )
Takes one argument, a PRIMOS pathname.

modf( )
Equivalent.

move( )
Specific to the PRIMOS C library. Similar functionality is provided by calling link( ) followed by unlink( ) in the UNIX operating systems.

open( )
Values for *openMode* differ. Additive keys provide extended functionality.

perror( )
Equivalent.

pow( )
Equivalent.

primospath( )
Specific to the PRIMOS C library.

printf( )
Equivalent.

putc( )
Equivalent.

putchar( )
Equivalent.

puts( )
Equivalent.

putw( )
Equivalent.

rand( )
Equivalent.

read( )
Equivalent.

realloc( )
May be used only to change the size of currently allocated space.

rewind( )
Equivalent.

rindex( )
Equivalent. (Synonym for strrchr( ).)

scanf( )
Extended. Conversion specification characters differ.

seek( )
Equivalent to lseek( ).

setbuf( )
Equivalent.

setjmp( )
Does not return a value in 64V mode. Equivalent in 32IX mode.

setmod( )
Specific to the PRIMOS C library. Similar functionality is provided by chmod( ) in the UNIX operating systems.

signal( )
Equivalent.

sin( )
Equivalent.

sinh( )
Equivalent.

sleep( )
Equivalent.

sprintf( )
Equivalent.

sqrt( )
Equivalent.

srand( )
Equivalent.

sscanf( )
Extended. Conversion specification characters differ.

stat( )
Equivalent.

sterm( )
Specific to the PRIMOS C library. Similar functionality is provided by ioctl( ) in the UNIX operating systems.

strcat( )
Equivalent.

strchr( )
Equivalent.

strcmp( )
Equivalent.

strcpy( )
Equivalent.

strcspn( )
Equivalent.

strlen( )
Equivalent.

strncat( )
Equivalent.

strncmp( )
Equivalent.

strncpy( )
The specified number of bytes is copied, regardless of whether a NULL byte is encountered.

strpbrk( )
Equivalent.

strrchr( )
Equivalent.

strspn( )
Equivalent.

system( )
The argument, command, will differ because PRIMOS and other operating systems (such as UNIX) use different commands for the same operation.

tan( )
Equivalent.

tanh( )
Equivalent.

tell( )
Specific to the PRIMOS C library. Similar to ftell( ), but used with files opened with open( ).

time( )
Returns only the time, in seconds. Some installations may not support Daylight Saving Time. Consult your System Administrator.

timer( )
Specific to the PRIMOS C library. Similar functionality is provided by alarm( ) in the UNIX operating systems.

tmpnam( )
Equivalent.

toascii( )
Equivalent.

tolower( )
Equivalent.

_tolower( )
Equivalent.

topascii( )
Specific to the PRIMOS C library.

toupper( )
Equivalent.

_toupper( )
Equivalent.

ungetc( )
Equivalent.

write( )
Equivalent.

Library Functions Not Supported in PRIMOS Compared With Suggested Alternatives

Non-supported Function	*Similar Function Available in the PRIMOS C Library*
alarm( )	**timer( )**
chmod( )	**setmod( )**
ioctl( )	**gterm( ), sterm( )**
link( )	**copy( ), move( )**
unlink( )	**delete( ), move( )**

8

USING ANSI C

This chapter provides the information you need in order to compile, link, and run PRIMOS C programs that conform to the ANSI C standard. It does not provide a reference guide to the ANSI C language. To write standard-conforming programs, consult an appropriate reference work. The two most authoritative references to ANSI C are described below.

The Standard: The definitive reference work for the C language is the ANSI standard, *American National Standard for Information Systems -- Programming Language C, X3.159-1989.* To obtain a copy of this document, write to the American National Standards Institute, 1430 Broadway, New York, New York 10018.

K&R 2: Almost equally definitive is the second edition of *The C Programming Language*, by Brian W. Kernighan and Dennis M. Ritchie (Englewood Cliffs, N.J.: Prentice-Hall, 1988), informally known as K&R 2. This book appeared before the ANSI standard was approved, but the information in it is consistent with the standard.

The changes that were made to the C language for ANSI are briefly described below.

- New header files are available containing function prototypes for all of the ANSI library functions, new constants, and new structures.
- The behavior of several existing library functions has been modified, and many new functions are now available.
- A new processor command **#error** has been introduced which produces a diagnostic message while preprocessing is performed.
- Two processor operators (# and ##) have been added which surround a parameter with quotes and concatenate adjacent tokens, respectively.
- Parameters inside macro strings are no longer replaced.
- The new keywords **const**, **volatile**, and **signed** have been added.
    - The type qualifier **const** specifies that the value of an object cannot be changed.
    - The type qualifier **volatile** announces that optimization should be suppressed for an object.

  - The type-specifier **signed** can be applied to **int** or **char**, but is mainly used to force **char** objects to carry a sign.

- New escape sequences are available for character constants, such as the hexadecimal character representation **\xhh**.
- A quad precision data type, **long double**, is available.
- Rules have been established about mixing pointers of different types without the use of casts.
- Ranges of arithmetic types for a particular implementation are given in the headers, <limits.h> and <float.h>.
- Function prototyping has been added, providing error detection of arguments across function calls. (The mixture of function prototyping and the old style of function declaration should be avoided.)
- Variable-length argument lists using the ellipsis notation ", ..." and the macros in the header <stdarg.h> are also new.
- Name spaces of identifiers have been changed such that labels are placed in a separate name space.
- Unions may be initialized. The initializer refers to the first member.
- Automatic structures, unions, and arrays may be initialized.

The first two sections of this chapter describe how to write, compile, link, and run C programs that conform to the ANSI standard. The third section describes how to avoid some potential problems in converting older PRIMOS C programs into standard-conforming programs. The fourth section provides a quick alphabetical reference to ANSI C library functions.

WRITING AND COMPILING STANDARD-CONFORMING C PROGRAMS

This section explains

- How to use the standard ANSI C header files
- How to use the -ANSI option to check the syntax of your program
- How to use the -EXTRACTPROTOTYPES option to obtain a header file of new-style function declarations

Using the Standard Header Files

The C library functions and the header files that define them are specified by the ANSI C standard. The header files are located in the directory SYSCOM. Table 8-1 lists these header files and the routines that use them. For information on how PRIMOS C handles header files, see Chapter 2.

Note

Some ANSI C library function declarations are not in the same header file as their non-ANSI C equivalents. Some functions did not have a function declaration in a header file in non-ANSI C, but do in ANSI C. To find out whether you need to modify any **#include** commands in your program, compare the description of the function in Chapter 4 with the description in the last section of this chapter.

TABLE 8-1. ANSI C Header Files

Header File	*Contents of Header File*
ASSERT.H.INS.CC	Diagnostic macro definition
CTYPE.H.INS.CC	Character classification functions
ERRNO.H.INS.CC	Error condition macros
FLOAT.H.INS.CC	Constant definitions for floating-point type sizes
LIMITS.H.INS.CC	Constant definitions for integral type sizes
LOCALE.H.INS.CC	Numeric value formats and macros
MATH.H.INS.CC	Mathematical functions
SETJMP.H.INS.CC	Non-local jump functions
SIGNAL.H.INS.CC	Signal handling functions
STDARG.H.INS.CC	Variable argument macros
STDDEF.H.INS.CC	Miscellaneous types and macros
STDIO.H.INS.CC	Input and output functions
STDLIB.H.INS.CC	Utility functions
STRING.H.INS.CC	String handling functions
TIME.H.INS.CC	Date and time functions

Many of these files (STDIO.H.INS.CC, for instance) are the same for both standard-conforming and non-standard-conforming programs. The files use the preprocessor macro **__STDC__** (STanDard-Conforming) to separate standard-conforming and non-standard-

conforming header information. If a program is compiled with the -ANSI option, **__STDC__** is defined as 1, and the appropriate parts of the header file are used. Therefore, the preprocessor command

```
#include <stdio.h>
```

pulls in different parts of the file STDIO.H.INS.CC depending on whether or not the -ANSI option was specified.

SYSCOM also contains the following nonstandard header files:

PRIME_ECS_CHARS.H.INS.CC
STAT.H.INS.CC
STRINGS.H.INS.CC
TERM.H.INS.CC
TIMEB.H.INS.CC

For information about using Prime ECS, see page 4-2 and Appendix F.

Syntax Checking: The -ANSI Option

Use the -ANSI compiler option to check your program's syntax for violations of the ANSI standard. You must use -ANSI in conjunction with the -32IX option.

```
OK, CC ANSIPROG -32IX -ANSI
```

The -ANSI option is described in Chapter 2. Information about the -ANSI option can also be found in the discussions of the following options in Chapter 2: -INTRINSIC, -PREPROCESSONLY, -STANDARDINTRINSICS.

Function Declarations: The -EXTRACTPROTOTYPES Option

The greatest change that the ANSI standard has made to the C language is the addition of a new syntax for function declarations and definitions. Although the standard allows programs to use the old style of function declaration and definition, users are encouraged to use the new style.

The -EXTRACTPROTOTYPES option makes it easier for you to convert your programs to the new style. If you compile an old-style C source file with this option, the compiler creates a header file with new-style declarations for all the functions in your source file.

For example, suppose your program contains three function definitions:

```
main( )

func1(myvar)
int myvar;

func2(myptr)
int *myptr;
```

If you name the source file EX.C, then the command

```
OK, CC EX -32IX -EXTRACTPROTOTYPES
```

creates a header file, EX.H, that contains the following declarations:

```
long int main(void);
long int func1(long int);
long int func2(long int *);
```

To add the new-style function declarations to your program, put the preprocessor command

```
#include "ex.h"
```

before any of the function definitions in your source file.

Caution

If you use this option, do not name one of your include files *program-name*.H; if you do, it will be overwritten.

Function Definitions: If you want to make your function definitions conform to the standard, you must change them by hand. New-style definitions for main( ), func1( ), and func2( ) look like this:

```
int main(void)

int func1(int myvar)

int func2(int *myvar)
```

LINKING STANDARD-CONFORMING C PROGRAMS

Use BIND, not SEG, to link a program that has been compiled with the -ANSI option. If you use SEG, you cannot link in the ANSI C runtime libraries.

Use the following steps to link your program with BIND:

1. After you invoke BIND, give the subcommand

   ```
   : LI ANSI_CCMAIN
   ```

   to link in the ANSI libraries. If you give the subcommand **LI CCMAIN**, you must use the -AnsiLibs command line option (discussed on page 8-7) in order to access the ANSI libraries when you execute your program.

2. To load each compiled program unit, give the subcommand

   ```
   : LO sourcename
   ```

   where *sourcename* is the name of the program unit.

3. BIND expects your main routine to be named main( ). If your main routine is not named main( ), use the MAIN subcommand to tell BIND which routine is your main routine.

   ```
   : MAIN G$routine-name
   ```

4. To link in the C DYNT library, give the subcommand

   ```
   : LI C_LIB
   ```

5. If you do not receive a BIND COMPLETE message, give the subcommand

   ```
   : LI
   ```

   to load the system libraries.

6. If you still do not receive a BIND COMPLETE message, give the subcommand

   ```
   : MAP -UN
   ```

   to obtain a list of unresolved references. You can then exit BIND by giving the QUIT subcommand.

7. When you receive a BIND COMPLETE message, give the subcommand

   ```
   : FILE
   ```

   to save your runfile and exit from BIND.

Below is an example of a BIND command line for ANSI C programs.

```
OK, BIND EXAMPLE -LI ANSI_CCMAIN -LO EXAMPLE -LI C_LIB -LI
```

See Chapter 3 for more information about linking with BIND.

RUNNING STANDARD-CONFORMING C PROGRAMS

ANSI C programs are executed in the same manner as non-ANSI C programs with the RESUME command:

```
OK, RESUME progname
```

or

```
OK, RESUME progname [args]
```

Two command line options are available to switch between the ANSI and non-ANSI libraries, if your program has linked in either CCMAIN or ANSI_CCMAIN. The command line options -AnsiLibs and -NoAnsiLibs cause RESUME to invoke the ANSI runtime library and the non-ANSI runtime library, respectively.

Note

The command line options -AnsiLibs and -NoAnsiLibs must be entered in full and capitalized exactly as shown.

If you linked your program with the ANSI_CCMAIN library and decide to access the non-ANSI C runtime library instead, issue the following command

```
OK, RESUME progname -NoAnsiLibs
```

where *progname* is the name of your program. Enter the **-NoAnsiLibs** option exactly as shown; case is significant, and there is no short form.

If you linked your program with CCMAIN, you can access the ANSI C runtime libraries by giving the following command

```
OK, RESUME progname -AnsiLibs
```

where *progname* is the name of your program. Enter **-AnsiLibs** exactly as shown.

To use the -AnsiLibs and -NoAnsiLibs options,

- You must have compiled your program in -32IX mode.
- You must have linked your program with the CCMAIN or ANSI_CCMAIN library.
- You must have linked your program with C_LIB, not CCLIB.

If you use command line arguments, you can place the -AnsiLibs or -NoAnsiLibs option in any position on the command line after the program name. For example, if your program is

```
#include <stdio.h>

main(argc, argv)
int argc;
char *argv[];
{
    int i;

    printf("The arguments are: ");
    for (i = 0; i < argc; i++)
        printf("%s%c", argv[i], (i < argc-1) ? ' ' : '\n');
    printf("%d arguments\n", argc);
}
```

and your command line is

```
OK, RESUME prog how many args -AnsiLibs is this
```

the program will display

```
The arguments are: prog how many args is this
6 arguments
OK,
```

PRIMOS C has a preprocessor macro, ___ANSILIBRARIES, which you can put in your program as a flag to indicate whether you are using the ANSI libraries or not. The following example shows how to use the ___ANSILIBRARIES macro:

```
#include <stdio.h>

main( )
{
    extern short __ANSILIBRARIES;

    if (__ANSILIBRARIES)
        printf("I'm in ANSI mode.\n");
    else
        printf("I'm not in ANSI mode.\n");
}
```

You can declare the ___ANSILIBRARIES macro in uppercase, lowercase, or both, as long as you refer to it consistently throughout your program.

CONVERTING OLDER PRIMOS C PROGRAMS TO ANSI C

This section describes a few potential problems that await users who want to make older PRIMOS C programs conform to the ANSI C standard. Most of these problems have to do with the C library, the macro preprocessor, and mixing old-style and new-style function declarations, the areas of greatest difference between PRIMOS C and the ANSI standard.

Examine carefully your program's use of library functions. Make sure

- That you include the correct header file
- That you declare correctly the variables that hold function arguments and returned values

The list of ANSI C library functions in the next section provides a quick reference both to the required header files and to the required types for function arguments and returned values.

The following list of specific differences between ANSI PRIMOS C and non-ANSI PRIMOS C is not exhaustive.

Using the Extended Character Set With ANSI

The ANSI C library recognizes the extended character set (Prime ECS). The library functions that perform character evaluation recognize the setting of the 8th bit when the code that invokes these functions is compiled with the -ANSI option in 32IX mode. These library functions are isalnum( ), isalpha( ), iscntrl( ), isdigit( ), isgraph( ), islower( ), isprint( ), ispunct( ), isspace( ), isupper( ), and isxdigit( ).

The following example illustrates the difference between compiling these functions with the -ANSI option and without the -ANSI option.

```
OK, SLIST TEST.C
#include <ctype.h>

#define UPPERCASE_A              0301
#define UPPERCASE_A_WITH_ACCENT 0101

main()
{
    if (isupper(UPPERCASE_A))
         printf("%o is uppercase\n", UPPERCASE_A);
    if (isupper(UPPERCASE_A_WITH_ACCENT))
         printf("%o is uppercase\n", UPPERCASE_A_WITH_ACCENT);
}

OK, CC TEST -32IX
[CI Rev. T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]
00 Errors and 00 Warnings detected in 12 lines and 107 include lines.
OK, BIND -LI CCMAIN -LO TEST -LI C_LIB
[BIND Rev. T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]
BIND COMPLETE
OK, R TEST
301 is uppercase
101 is uppercase

OK, CC TEST -32IX -ANSI
[CI Rev. T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]
00 Errors and 00 Warnings detected in 12 lines and 107 include lines.
OK, BIND -LI ANSI_CCMAIN -LO TEST -LI C_LIB
[BIND Rev. T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]
BIND COMPLETE
OK, R TEST
301 is uppercase
```

The atof( ) Library Function

If you compile your program without -ANSI, the atof( ) function expects the math.h header file.

If you compile your program with -ANSI, the function expects the stdlib.h header file.

Caution

If you include neither the math.h header file nor the stdlib.h header file, or if you include the wrong header file, your program will compile and link, but it will produce erroneous output.

The strncpy( ) Library Function

The non-ANSI version of strncpy( ) performs a block move from *string-2* to *string-1* of a specified number of characters, including null characters.

The ANSI version of strncpy( ) also copies chararacters from *string-2* to *string-1* but does not copy anything from *string-2* that follows a null character. Therefore, if strncpy( ) encounters a null character in *string-2* before *n* characters have been copied, it appends null characters to *string-1* until *n* characters have been written.

The following example illustrates the difference between the non-ANSI and ANSI version of the strncpy( ) function.

```
OK, SLIST TEST.C
#include <stdio.h>
#include <string.h>

main()
{
    int i;
    char str[10];

    strncpy(str, "12345\0abc", 9);
    for (i = 0; i < 9; i++)
         putchar(str[i]);
    putchar('\n');
}

OK, CC TEST.C -32IX
[CI Rev. T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]
00 Errors and 00 Warnings detected in 14 lines and 248 include lines.
OK, BIND -LI ANSI_CCMAIN -LO TEST -LI C_LIB
[BIND Rev. T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]
BIND COMPLETE

OK, R TEST -NoAnsiLibs
12345abc

OK, R TEST -AnsiLibs
12345
```

The ctime( ), localtime( ), and time( ) Library Functions

The non-ANSI versions of ctime( ), localtime( ), and time( ) each expect an argument that is a pointer to **int**. The time( ) function returns an **int** value.

The ANSI version of these functions each expect one argument that is a pointer to **time_t**, which is defined in the time.h header file. The ANSI version of time( ) returns a **time_t** value.

The ANSI and non-ANSI versions of the ctime( ) function return the date and time in different formats. The following program prints out the value returned from ctime( ):

```
#include <time.h>

main( )
{
  time_t sec;
  time(&sec);
  printf("%s\n", ctime(&sec));
}
```

If you use the ANSI library, the string looks like this:

```
OK, R EXAMPLE -AnsiLibs
Mon Mar  3 13:06:07 1990
```

If you use the non-ANSI library, it looks like this:

```
OK, R EXAMPLE -NoAnsiLibs
03 Mar 90 13:06:07 Monday
```

The #endif Preprocessor Command

Non-ANSI PRIMOS C allows you to follow an **#endif** command with an identifier name, as in the following example:

```
#ifndef PRIME
  #define PRIME 1
#endif PRIME

main( )
{
}
```

The 1978 C language does not sanction this practice, but it does not forbid it. If you compile such a program with -ANSI, you will receive the following error message:

```
OK, CC PROG -32IX -ANSI
[CI Rev. T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]
Error #1 on line 4 in file <mysys>myid>prog.c

  Extraneous input seen after #endif directive.


01 Error and 00 Warnings detected in 7 source lines.
```

Redefining Macros

The non-ANSI C preprocessor allows you to redefine a macro in a program without first using **#undef** to disable the previous macro definition. For example, the non-ANSI C compiler accepts the following program:

```
#define color "green"
#define color "blue"

main()
{
    printf("color is %s\n", color);
}
```

The ANSI C preprocessor requires that you undefine a macro with the **#undef** command before you redefine it with a different value. If you compile the program above with -ANSI, you receive the following error message:

```
OK, CC PROG -32IX -ANSI
[CI Rev. T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]
Error #1 on line 2 in file <mysys>myid>prog.c

  Redefinition of 'color' must match (ignored).


01 Error and 00 Warnings detected in 7 source lines.
```

Macro Expansion

Formal Parameters in Quoted Strings: The non-ANSI C macro preprocessor replaces formal parameters that occur within quoted strings. For example, in the following program, the parameters `value` and `format` in the macro definition are replaced within the string argument to printf( ):

```
#include <stdio.h>
#define PR(format,value) printf("value = %format\t", value)
main( )
{
   int x;

   x = 35;
   PR(d,x);
}
```

The program displays the following:

```
x = 35
```

The ANSI C macro preprocessor does not automatically replace formal parameters within quoted strings. To obtain the same result when you compile with -ANSI, precede the parameter name with a #. The preprocessor expands the parameter as a quoted string, which is then concatenated with any other strings in the macro definition. The following command performs the same function as the one in the program above:

```
#define PR(format,value) printf(#value " = %"#format"\t", value)
```

If you use the old-style macro definition (without the #) and compile with -ANSI, you obtain the unintended output

```
value = 0.000000ormat
```

Syntax Checking in #define Commands: In non-ANSI PRIMOS C, the macro processor performs syntax checking on the arguments to **#define** commands.

In ANSI PRIMOS C, the macro preprocessor checks the syntax only when the macro is expanded in the program.

For example, non-ANSI C accepts only single character arguments in single quotation marks in **#define** commands. For example, the definition

```
#define ctrl(letter) ('letter' & 077)
```

is incorrect and results in an error message. Similarly, commands like

```
#define HUGE 12345678901234567890
#define HEX  0x
#define OCT  0778
#define CH   't
```

will draw compiler error messages if -ANSI is not specified.

If -ANSI is specified, these macros will draw errors only when their expansion results in a syntactically invalid program.

The fopen( ), fdopen( ), and freopen( ) Library Functions

The non-ANSI and ANSI versions of fopen( ), fdopen( ), and freopen( ) have different expectations for the contents of the second argument, *accessMode* (*mode* in the ANSI function descriptions).

If you link your program with the CCMAIN library, you must use the non-ANSI argument contents. The non-ANSI argument contents are described in the discussion of fopen( ) in Chapter 4.

If you link your program with the ANSI_CCMAIN library, you must use the ANSI argument contents that are described in the discussion of fopen( ) in the ANSI C Library Functions section of this chapter.

Table 8-2 shows how the non-ANSI arguments correspond to the ANSI arguments.

TABLE 8-2. Non-ANSI and ANSI fopen Argument Strings

Non-ANSI String	*ANSI String*
"r"	"r"
"w"	"w"
"wa"	"a"
"i"	"rb"
"o"	"wb"
"oa"	"ab"
"i+"	"r+b"
"o+"	"w+b"
"oa+"	"a+b"
"r+"	
"w+"	
"a+"	

If you want a program to be able to use both the ANSI and the non-ANSI libraries, you can use the __ANSILIBRARIES macro to switch from one argument string to another, as in the following example:

```
#include <stdio.h>

main( )
{
    extern short __ANSILIBRARIES;
    char writeBinary[3];
    FILE *fp, *fopen( );

    if (__ANSILIBRARIES)
        strcpy(writeBinary, "wb");
    else
        strcpy(writeBinary, "o");

    if ((fp = fopen("tmp.file", writeBinary)) == NULL)
        printf("can't open tmp.file\n");
    else {
        fprintf(fp, "put stuff in tmp.file");
        fclose(fp);
    }
}
```

Mixing Old and New Style Function Definitions and Declarations

A program should not mix function prototypes and old-style function definitions and declarations because default argument promotion is performed on arguments if the old style is used. This means that **char** and **short int** arguments are converted to **int**, and **float** arguments are converted to **double.**

In the following example, the character argument passed to print_char is promoted to an **int** due to the old-style function declaration. Since the function definition is in the new style, however, no type promotion is expected and the parameter is treated as a **char.** This leads to incorrect results.

```
OK, SLIST MAIN.C
extern void print_char();
main()
{
    print_char('a');
}
OK, SLIST PRINT.C
void print_char(char ch)
{
    printf("character is %c\n", ch);
}
OK, CC (MAIN PRINT) -32IX -ANSI
[CI Rev. T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]
00 Errors and 00 Warnings detected in 6 source lines.
[CI Rev. T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]
00 Errors and 00 Warnings detected in 4 source lines.
OK, BIND -LI ANSI_CCMAIN -LO MAIN PRINT -LI C_LIB
[BIND Rev. T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]
BIND COMPLETE
OK, R MAIN
character is
```

Nonstandard Library Functions

The following non-ANSI C library functions are implementation-dependent system calls; they are not part of the ANSI C standard library. However, you can reference them whether you compile with -ANSI or without it, and whether you link the ANSI_CCMAIN or CCMAIN library:

access( )	bio$primosfileunit( )	cabs( )	cfree( )
chdir( )	chrcheck( )	close( )	copy( )
creat( )	cuserid( )	delete( )	ecvt( )
fcvt( )	fdopen( )	fdtm( )	fexists( )
fgetname( )	fileno( )	frwlock( )	fsize( )
fstat( )	ftime( )	ftype( )	g$amiix( )
geth( )	getmod( )	getname( )	getw( )
gterm( )	gvget( )	gvset( )	hypot( )
index( )	isascii( )	isatty( )	ispascii( )
lsdir( )	lseek( )	mkdir( )	move( )
open( )	primospath( )	puth( )	putw( )
read( )	rindex( )	seek( )	setmod( )
sleep( )	stat( )	sterm( )	tell( )
timer( )	toascii( )	topascii( )	write( )

See Chapter 4 for information about these functions.

ANSI C LIBRARY FUNCTIONS

This section lists all the ANSI C library function and macro names in alphabetical order. It shows the returned value and argument types for each function as well as the header file in which its declaration resides. For more information, see Chapter 4 if the function existed prior to ANSI. If it did not, consult the ANSI standard or the second edition of *The C Programming Language* by Kernighan and Ritchie.

abort( )	abs( )	acos( )	asctime( ) *
asin( )	assert( )	atan( )	atan2( )
atexit( ) *	atof( )	atoi( )	atol( )
bsearch( ) *	calloc( )	ceil( )	clearerr( )
clock( ) *	cos( )	cosh( )	ctime( )
difftime( ) *	div( ) *	exit( )	exp( )
fabs( )	fclose( )	feof( )	ferror( )
fflush( )	fgetc( )	fgetpos( ) *	fgets( )
floor( )	fmod( ) *	fopen( )	fprintf( )
fputc( )	fputs( )	fread( )	free( )
freopen( )	frexp( )	fscanf( )	fseek( )
fsetpos( ) *	ftell( )	fwrite( )	getc( )
getchar( )	getenv( ) *	gets( )	gmtime( ) *
isalnum( )	isalpha( )	iscntrl( )	isdigit( )
isgraph( )	islower( )	isprint( )	ispunct( )
isspace( )	isupper( )	isxdigit( )	labs( ) *

ldexp( )	ldiv( ) *	localeconv( ) *	localtime
log( )	log10( )	longjmp( )	malloc( )
mblen( ) *	mbstowcs( ) *	mbtowc( ) *	memchr( ) *
memcmp( ) *	memcpy( ) *	memmove( ) *	memset( ) *
mktime( ) *	modf( )	perror( )	pow( )
printf( )	putc( )	putchar( )	puts( )
qsort( ) *	raise( ) *	rand( )	realloc( )
rewind( )	remove( ) *	rename( ) *	scanf( )
setbuf( )	setjmp( )	setlocale( ) *	setvbuf( ) *
signal( )	sin( )	sinh( )	sprintf( )
sqrt( )	srand( )	sscanf( )	strcat( )
strchr( )	strcmp( )	strcoll( ) *	strcpy( )
strcspn( )	strerror( ) *	strftime( ) *	strlen( )
strncat( )	strncmp( )	strncpy( )	strpbrk( )
strrchr( )	strspn( )	strstr( ) *	strtod( ) *
strtok( ) *	strtol( ) *	strtoul( ) *	strxfrm( ) *
system( )	tan( )	tanh( )	time( )
tmpfile( ) *	tmpnam( )	tolower( )	toupper( )
ungetc( )	va_arg( ) *	va_end( ) *	va_start( ) *
vfprintf( ) *	vprintf( ) *	vsprintf( ) *	wcstombs( ) *
wctomb( ) *			

Asterisks (*) denote functions that are new for ANSI.

These functions are described in the following pages.

▶ abort( )

Causes abnormal program termination to occur.

```
#include <stdlib.h>
void abort(void);
```

This function cannot return to its caller.

▶ abs( )

Computes and returns the absolute value of an integer.

```
#include <stdlib.h>
int abs(int j);
```

▶ acos( )

Computes and returns the arc cosine of *x*.

```
#include <math.h>
double acos(double x);
```

The returned value is in the range [0, π] radians. If an argument is not in the range [-1, +1], a domain error occurs.

▶ asctime( )

Converts the time in the structure *timeptr* into a string that has the following form.

```
Fri Feb 09 13:15:59 1990\n\0
```

```
#include <time.h>
char *asctime(const struct tm *timeptr);
```

This function returns a pointer to the string.

▶ asin( )

Computes and returns the arc sine of *x*.

```
#include <math.h>
double asin(double x);
```

The returned value is in the range $[-\pi/2, +\pi/2]$ radians. If an argument is not in the range [-1, +1], a domain error occurs.

▶ assert( )

Places diagnostics into programs.

```
#include <assert.h>
void assert(int expression);
```

If the argument is false when it is executed, the macro writes information about the failure to *stderr*.

▶ atan( )

Computes and returns the arc tangent of x.

```
#include <math.h>
double atan(double x);
```

The returned value is in the range $[-\pi/2, +\pi/2]$ radians.

▶ atan2( )

Computes and returns the arc tangent of y/x.

```
#include <math.h>
double atan2(double y, double x);
```

The returned value is in the range $[-\pi, +\pi]$ radians. If both arguments are zero, a domain error occurs.

▶ atexit( )

Registers the function *func*, to be called with no arguments at the normal termination of the program. A maximum of 32 functions can be registered.

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

If the registration succeeds, this function returns zero. If it fails, atexit( ) returns nonzero.

▶ atof( )

Converts the string *nptr* to **double** representation and returns the converted value.

```
#include <stdlib.h>
double atof(const char *nptr);
```

▶ atoi( )

Converts the string *nptr* to **int** representation and returns the converted value.

```
#include <stdlib.h>
int atoi(const char *nptr);
```

▶ atol( )

Converts the string *nptr* to **long int** representation and returns the converted value.

```
#include <stdlib.h>
long int atol(const char *nptr);
```

▶ bsearch( )

Searches an array *base* of *nmemb* objects for an element that matches the object pointed to by *key*.

```
#include <stdlib.h>
void *bsearch(const void *key, const void *base,
              size_t nmemb, size_t size,
              int (*compar)(const void *, const void *));
```

size specifies the size of each array element. *compar* points to a comparison function that is called with two arguments that point to the *key* object and an array element.

If the *key* object is less than, matches, or is greater than the array element, bsearch( ) returns an integer less than, equal to, or greater than zero, respectively. The array is sorted according to the comparison function.

This function returns a pointer to a matching element of the array. If no match is found, however, it returns a null pointer.

▶ calloc( )

Allocates space for an array of *nmemb* objects of size *size*. This function initializes all bits in the space to zero.

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
```

calloc( ) returns a pointer to the allocated space or a null pointer.

▶ ceil( )

Computes and returns the smallest integer that is equal to or greater than *x*.

```
#include <math.h>
double ceil(double x);
```

▶ clearerr( )

Clears the error and end-of-file indicators for the stream *stream.*

```
#include <stdio.h>
void clearerr(FILE *stream);
```

▶ clock( )

Calculates and returns the processor time used.

```
#include <time.h>
clock_t clock(void);
```

The macro CLOCKS_PER_SEC converts the estimated processor time into time in seconds.

▶ cos( )

Computes and returns the cosine of *x* expressed in radians.

```
#include <math.h>
double cos(double x);
```

▶ cosh( )

Computes and returns the hyperbolic cosine of *x*. If the magnitude of *x* is too large, a range error occurs.

```
#include <math.h>
double cosh(double x);
```

▶ ctime( )

Converts the calendar time pointed to by *timer* to the local time with the following form

```
<day-of-week> MMM DD HH:MM:SS YYYY\n\0.
```

```
#include <time.h>
char *ctime(const time_t *timer);
```

This function returns a pointer to the local time string.

▶ difftime( )

Computes and returns the difference in seconds between two calendar times: *time1* - *time0*.

```
#include <time.h>
double difftime(time_t time1, time_t time0);
```

▶ div( )

Computes the quotient and remainder of the division of *numer* by *denom* such that quotient * *denom* + remainder = *numer*.

```
#include <stdlib.h>
div_t div(int numer, int denom);
```

This function operates on **int** types and returns a structure of type **div_t** that consists of the quotient and the remainder.

▶ exit( )

Causes normal program termination to occur.

```
#include <stdlib.h>
void exit(int status);
```

All functions registered by atexit( ) are called in the reverse order of their registration. This function flushes all open output streams, closes all open streams, and removes all files created by tmpfile( ). Control returns to PRIMOS.

A nonzero *status*, however, is used as a severity code that is returned to the invoker of the program.

▶ exp( )

Computes and returns base e raised to the *x* power.

```
#include <math.h>
double exp(double x);
```

If the magnitude of *x* is too large, a range error occurs.

▶ fabs( )

Computes and returns the absolute value of a floating-point *x*.

```
#include <math.h>
double fabs(double x);
```

▶ fclose( )

Flushes the stream *stream* and closes the associated file.

```
#include <stdio.h>
int fclose(FILE *stream);
```

If the stream was successfully closed, fclose( ) returns zero. If any errors were detected, the function returns EOF.

▶ feof( )

Tests the end-of-file indicator for the stream *stream.*

```
#include <stdio.h>
int feof(FILE *stream);
```

If the end-of-file indicator is set for *stream,* this function returns nonzero.

▶ ferror( )

Tests the error indicator for the stream *stream.*

```
#include <stdio.h>
int ferror(FILE *stream)
```

If the *stream's* error indicator is set, this function returns nonzero.

▶ fflush( )

Writes any unwritten data to the file if *stream* points to an output stream or to an update stream in which the most recent operation was not input.

```
#include <stdio.h>
int fflush(FILE *stream);
```

If *stream* is a null pointer, fflush( ) performs the flushing action on all appropriate streams. If a write error occurs, this function returns EOF; otherwise it returns zero. Refer to the fflush( ) description in Chapter 4 for PRIMOS limitations.

▶ fgetc( )

From the input stream *stream*, this function gets the next character as an **unsigned char** converted to an **int**, and advances the file position indicator.

```
#include <stdio.h>
int fgetc(FILE *stream);
```

fgetc( ) returns the character. If *stream* is at the end-of-file, its end-of-file indicator is set and EOF is returned. If a read error occurs, *stream*'s error indicator is set and EOF is returned.

▶ fgetpos( )

Saves the current value of the file position indicator for the stream *stream* in the object *pos*.

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
```

The saved value contains information that fsetpos( ) can use for repositioning *stream* to its position at the time of the call to fgetpos( ). If successful, fgetpos( ) returns zero. If it fails, the function returns nonzero.

▶ fgets( )

Reads at most one less than *n* characters from the stream *stream* into the array *s*.

```
#include <stdio.h>
char *fgets(char *s, int n, FILE *stream);
```

When fgets( ) encounters a new-line character or end-of-file, it reads no additional characters. When the last character is read into the array, the function immediately writes a null character.

If successful, fgets( ) returns *s*. If it encounters an end-of-file, the contents of the array remain unchanged and a null pointer is returned. If a read error occurs, the array contents are indeterminate and a null pointer is returned.

▶ floor( )

Computes and returns the largest integer that is less than or equal to *x*.

```
#include <math.h>
double floor(double x);
```

▶ fmod( )

Computes and returns the remainder of *x/y*. If *y* is zero, this function returns a zero.

```
#include <math.h>
double fmod(double x, double y);
```

▶ fopen( )

Opens a file whose name is the string *filename*, and associates a stream with it. The argument *mode* points to a string that begins with one of the sequences listed in Table 8-3.

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
```

Opening a file with read mode fails for a nonexistent or unreadable file. Opening a file with append mode forces subsequent writes to the file's current end-of-file, regardless of intervening calls to fseek( ).

Opening a file with update mode, allows input and output to be performed on the associated stream. Output followed directly by input requires an intervening call to fflush( ) or a file positioning function (fseek( ), fsetpos( ), or rewind( )). Input followed directly by output requires an intervening call to a file positioning function, unless the input operation encounters end-of-file.

fopen( ) returns a pointer to the stream. If the open operation failed, the function returns a null pointer.

▶ fprintf( )

Writes output to the stream *stream* under control of the string *format*, which specifies how the subsequent arguments are to be converted for output.

TABLE 8-3. Values for mode Argument of fopen Function

Argument	*Action Performed*
"r"	Open text file for reading.
"w"	Truncate to zero length or create text file for writing.
"a"	Append; open or create text file for writing at end-of-file.
"rb"	Open binary file for reading.
"wb"	Truncate to zero length or create binary file for writing.
"ab"	Append; open or create binary file for writing at end-of-file.
"r+"	Open text file for update (reading and writing).
"w+"	Truncate to zero length or create text file for update.
"a+"	Append; open or create text file for update, writing at end-of-file.
"r+b" or "rb+"	Open binary file for update (reading and writing).
"w+b" or "wb+"	Truncate to zero length or create binary file for update.
"a+b" or "ab+"	Append; open or create binary file for update, writing at end-of-file.

```
#include <stdio.h>
int fprintf(FILE *stream, const char *format, ...);
```

This function returns the number of characters written. If an output error occurred, it returns a negative value.

In addition to the conversion characters listed in Table 4-4 on page 4-44, the following also exist:

TABLE 8-4. Additional Conversion Characters for Formatting Output

Character	*Meaning*
%i	Converts to decimal format.
%x	Converts to hexadecimal format using letters abcdef.
%X	Converts to hexadecimal format using letters ABCDEF.
%G	Converts **double** to %f or %E.
%n	The argument should be a pointer to an integer that represents the number of characters written to the output stream so far by this call.

In addition to the field specifiers in Table 4-5 on page 4-46, the following also exist:

TABLE 8-5. Additional Field Specification for Output Formats

Character	*Meaning*
+	The result of a signed conversion always begins with a plus or minus sign.
h	Indicates that a following %d, %i, %o, %u, %x, or %X specification corresponds to a short output source.
0	Uses leading zeros to pad to the field width for d, i, o, u, x, X, e, E, f, g, and G conversions,
#	Converts the result to an alternate form. For o conversion, it forces the first digit to zero. For x and X conversions, it prefixes 0x or 0X to a nonzero result. For e, E, f, g, and G conversion, the result always contains a decimal-point, even with no following digits. For g and G conversions, it does not remove trailing zeros from the result.

▶ fputc( )

Writes *c*, converted to an **unsigned char**, to the stream *stream* at the place specified by the associated file position indicator, and advances the indicator.

```
#include <stdio.h>
int fputc(int c, FILE *stream);
```

fputc( ) returns the character written. If a write error occurs, the function sets the stream's error indicator and returns EOF.

▶ fputs( )

Writes the string *s* to the stream *stream*, excluding the terminating null character.

```
#include <stdio.h>
int fputs(const char *s, FILE *stream);
```

If successful, fputs( ) returns a nonnegative value. If a write error occurs, it returns EOF.

▶ fread( )

Reads up to *nmemb* elements of size *size* from the stream *stream* into the array *ptr*. The function advances the stream's file position indicator by the number of successfully read characters.

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

fread( ) returns the number of successfully read elements. This number may be less than *nmemb* if a read error or end-of-file is encountered. If *size* or *nmemb* is zero, the function returns zero.

▶ free( )

Deallocates the space pointed to by *ptr*.

```
#include <stdlib.h>
void free(void *ptr);
```

If *ptr* is a null pointer, no action occurs.

▶ freopen( )

Opens *filename* and associates the stream *stream* with it.

```
#include <stdio.h>
FILE *freopen(const char *filename, const char *mode, FILE *stream);
```

freopen( ) uses the *mode* argument just as in fopen( ). It attempts to close any file associated with the specified *stream*, and clears the stream's error and end-of-file indicators.

If successful, freopen( ) returns the value of *stream*. If the open operation fails, the function returns a null pointer.

▶ frexp( )

Splits a floating-point number into a normalized fraction and an integral power of 2. The function stores the integer in *exp*.

```
#include <math.h>
double frexp(double value, int *exp);
```

frexp( ) returns a **double** x such that *value* = x * 2 raised to the power **exp*.

▶ fscanf( )

Reads input from the stream *stream* under the control of *format*, which specifies allowable input sequences and their conversion for assignment. This function uses the following arguments as pointers to the objects that are to receive the converted input.

```
#include <stdio.h>
int fscanf(FILE *stream, const char *format, ...);
```

In addition to the conversion characters listed in Table 4-6 on page 4-52, the following also exist.

TABLE 8-6. Additional Conversion Specifications for Formatting Input

Character	*Meaning*
%i	A decimal integer is input. The corresponding argument must be an integer pointer.
%x	A hexadecimal integer is input using letters abcdef. The corresponding argument must be an integer pointer.
%X	A hexadecimal integer is input using letters ABCDEF. The corresponding argument must be an integer pointer.
%G	A floating-point number is input.
%n	No input is consumed. The argument is a pointer to an integer that represents the number of characters read from the input stream so far by this call. This directive does not increment the assignment count returned at the completion of fscanf( ).

fscanf( ) returns the number of input items assigned, which can be fewer than provided for, or which can be zero for an early matching failure. If an input failure occurs before any conversion, this function returns EOF.

▶ fseek( )

Sets the file position indicator for the stream *stream.*

```
#include <stdio.h>
int fseek(FILE *stream, long int offset, int whence);
```

The new position for a binary stream is measured in characters from the beginning of the file. It is obtained by adding *offset* to the *whence* position. If *whence* is SEEK_SET, the position is the beginning of the file; for SEEK_CUR, it is the file position indicator's current value; for SEEK_END, it is end-of-file.

For a text stream, *offset* is either zero or a value returned by an earlier call to ftell( ) on the same stream; *whence* is SEEK_SET.

If successful, fseek( ) clears the end-of-file indicator for the stream and undoes any effects of ungetc( ) on it. For a request that cannot be satisfied, this function returns nonzero. Refer to fseek( ) in Chapter 4 for PRIMOS limitations.

▶ fsetpos( )

Sets *stream*'s file position indicator according to *pos*, which comes from an earlier call to fgetpos( ) on the same stream.

```
#include <stdio.h>
int fsetpos(FILE *stream, const fpos_t *pos);
```

If successful, fsetpos( ) clears the stream's end-of-file indicator, undoes any effects of ungetc( ) on it, and returns zero. On failure, the function returns nonzero.

▶ ftell( )

Gets the file position indicator's current value for the stream *stream*.

```
#include <stdio.h>
long int ftell(FILE *stream);
```

The value for a binary stream is the number of characters from the beginning of the file. The value for a text stream contains information that fseek( ) can use.

ftel( ) returns the current value of the file position indicator for the stream. On failure, it returns -1L.

▶ fwrite( )

Writes up to *nmemb* elements of size *size*, from the array *ptr* to the stream *stream*. The function advances the stream's file position indicator by the number of successfully written characters.

```
#include <stdio.h>
size_t fwrite(const void *ptr, size_t size, size_t nmemb,
              FILE *stream);
```

fwrite( ) returns the number of elements successfully written. If the function encounters a write error, this number is less than *nmemb*.

▶ getc( )

Equivalent to fgetc( ), except that getc( ) is implemented as a macro.

```
#include <stdio.h>
int getc(FILE *stream);
```

getc( ) returns the next character from *stream.* If the stream is at end-of-file, the function sets the stream's end-of-file indicator and returns EOF. If a read error occurs, getc( ) sets the stream's error indicator and returns EOF.

▶ getchar( )

Equivalent to getc( ) with the argument *stdin.*

```
#include <stdio.h>
int getchar(void);
```

getchar( ) returns the next character from *stdin.* If the stream is at end-of-file, the function sets the stream's end-of-file indicator and returns EOF. If a read error occurs, getchar( ) sets the stream's error indicator and returns EOF.

▶ getenv( )

Searches an environment list (for PRIMOS, a global variable list) for a string that matches *name.*

```
#include <stdlib.h>
char *getenv(const char *name);
```

getenv( ) returns a pointer to the string that is associated with the matching list member. If *name* cannot be found, the function returns a null pointer.

▶ gets( )

Reads characters from *stdin* into the array *s*, until it encounters a newline character or end-of-file. This function discards any newline character. It immediately writes a null character after reading the last character into the array.

```
#include <stdio.h>
char *gets(char *s);
```

If successful, gets( ) returns *s.* If it encounters end-of-file, the contents of the array remain unchanged and a null pointer is returned. If a read error occurs, the array contents are indeterminate and a null pointer is returned.

▶ gmtime( )

Converts the calendar time pointed to by *timer* into a time that is expressed as Coordinated Universal Time (UTC).

```
#include <time.h>
struct tm *gmtime(const time_t *timer);
```

This function returns a null pointer since UTC is not available under PRIMOS.

▶ isalnum( )

Tests for any character for which isalpha( ) or isdigit( ) is true.

```
#include <ctype.h>
int isalnum(int c);
```

This function is implemented as a macro.

▶ isalpha( )

Tests for any character for which isupper( ) or islower( ) is true.

```
#include <ctype.h>
int isalpha(int c);
```

This function is implemented as a macro.

▶ iscntrl( )

Tests for any control character.

```
#include <ctype.h>
int iscntrl(int c);
```

This function is implemented as a macro.

▶ isdigit( )

Tests for any decimal-digit character.

```
#include <ctype.h>
int isdigit(int c);
```

This function is implemented as a macro.

▶ isgraph( )

Tests for any printing character except space.

```
#include <ctype.h>
int isgraph(int c);
```

This function is implemented as a macro.

▶ islower( )

Tests for any lowercase alphabetic ASCII character.

```
#include <ctype.h>
int islower(int c);
```

This function is implemented as a macro.

▶ isprint( )

Tests for any printing character including a space.

```
#include <ctype.h>
int isprint(int c);
```

This function is implemented as a macro.

▶ ispunct( )

Tests for any printing character that is neither a space nor a character for which isalnum( ) is true.

```
#include <ctype.h>
int ispunct(int c);
```

This function is implemented as a macro.

▶ isspace( )

Tests for any of the following white-space characters: space (' '), form feed ('\f'), newline ('\n'), carriage return ('\r'), horizontal tab ('\t'), and vertical tab ('\v'). This function is implemented as a macro.

```
#include <ctype.h>
int isspace(int c);
```

▶ isupper( )

Tests for any uppercase alphabetic ASCII character.

```
#include <ctype.h>
int isupper(int c);
```

This function is implemented as a macro.

▶ isxdigit( )

Tests for any hexadecimal digit character.

```
#include <ctype.h>
int isxdigit(int c);
```

This function is implemented as a macro.

▶ labs( )

Computes and returns the absolute value of a long integer.

```
#include <stdlib.h>
long int labs(long int j);
```

▶ lconv( )

Sets the components of an object of type struct *lconv* with values that are appropriate for formatting numeric quantities according to the current locale's rules.

```
#include <locale.h>
struct lconv *localeconv(void);
```

This function returns a pointer to the object. At this release, only the "C" locale is supported.

▶ ldexp( )

Multiplies a floating-point number by an integral power of 2.

```
#include <math.h>
double ldexp(double x, int exp);
```

This function returns $x * 2$ raised to the power *exp*.

▶ ldiv( )

Computes the quotient and remainder of the division of *numer* by *denom* such that quotient * *denom* + remainder = *numer*.

```
#include <stdlib.h>
ldiv_t ldiv(long int numer, long int denom);
```

This function operates on **long int** types and returns a structure of type **ldiv_t** that consists of the quotient and the remainder.

▶ localtime( )

Converts the calendar time in *timer* into a broken-down time.

```
#include <time.h>
struct tm *localtime(const time_t *timer);
```

This function returns a pointer to struct *tm*.

▶ log( )

Computes and returns the natural logarithm of *x*.

```
#include <math.h>
double log(double x);
```

If *x* is negative, a domain error occurs. If *x* is zero, a range error occurs.

▶ log10( )

Computes and returns the base-ten logarithm of *x*.

```
#include <math.h>
double log10(double x);
```

If *x* is negative, a domain error occurs. If *x* is zero, a range error occurs.

▶ longjmp( )

Restores the environment that the most recent setjmp( ) saved in the program invocation that has the corresponding *jmp_buf* argument.

```
#include <setjmp.h>
void longjmp(jmp_buf env, int val);
```

After longjmp( ) completes, the program continues to execute as if setjmp( ) had just returned *val.*

▶ malloc( )

Allocates space for an object of size *size.*

```
#include <stdlib.h>
void *malloc(size_t size);
```

This function returns a pointer to the allocated space or a null pointer.

▶ mblen( )

Determines the number of bytes in the multibyte character pointed to by *s*, when *s* is not a null pointer.

```
#include <stdlib.h>
int mblen(const char *s, size_t n);
```

If *s* is not a null pointer, this function returns the following: the number of bytes if the next *n* or fewer bytes form a valid multibyte character; 0 if *s* points to the null character; or -1.

If *s* is a null pointer, this function returns a nonzero value if the multibyte character encodings are state-dependent. It returns zero if the encodings are not state-dependent.

Note

At this release, mblen( ) returns 0 if *s* is a null pointer, if *s* points to the null character, or if *n* is zero. Otherwise, this function returns 1.

▶ mbstowcs( )

Converts a sequence of multibyte characters beginning in the initial shift state from the array *s* to a sequence of corresponding codes. A maximum of *n* codes are stored into the array *pwcs*. mbstowcs( ) does not examine or convert multibyte characters that follow a null character.

```
#include <stdlib.h>
size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);
```

This function returns the number of modified array elements, which does not include any terminating zero code.

▶ mbtowc( )

Determines the number of bytes in the multibyte character pointed to by *s*, when *s* is not a null pointer. mbtowc( ) next determines the code for the type *wchar_t* value corresponding to the multibyte character. If the multibyte character is valid and *pwc* is not a null pointer, the code is stored in *pwc*. A maximum of *n* bytes of the array *s* are examined.

```
#include <stdlib.h>
int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

If *s* is not a null pointer, this function returns the following: the number of bytes in the converted multibyte character if the next *n* or fewer bytes form a valid multibyte character; 0 if *s* points to the null character; or -1.

If *s* is a null pointer, this function returns a nonzero value if multibyte character encodings are state-dependent. It returns zero if the encodings are not state-dependent.

Note

At this release, mbtowc( ) returns 0 if *s* is a null pointer, if *s* points to the null character, or if *n* is zero. Otherwise, this function returns 1.

▶ memchr( )

Finds the first occurrence of *c*, converted to an unsigned **char**, in the first *n* characters of the object *s*.

```
#include <string.h>
void *memchr(const void *s, int c, size_t n);
```

This function returns a pointer to the character found. If it does not find the character in the object, it returns a null pointer.

▶ memcmp( )

Compares the first *n* characters of the object *s1* to the first *n* characters of the object *s2*.

```
#include <string.h>
int memcmp(const void *s1, const void *s2, size_t n);
```

Depending on whether *s1* is greater than, equal to, or less than *s2*, this function returns an integer that is greater than, equal to, or less than zero, respectively.

▶ memcpy( )

Copies *n* characters from the object *s2* into the object *s1* and returns a pointer to *s1*.

```
#include <string.h>
void *memcpy(void *s1, const void *s2, size_t n);
```

▶ memmove( )

Copies *n* characters from the object *s2* into the object *s1*.

```
#include <string.h>
void *memmove(void *s1, const void *s2, size_t n);
```

The *n* characters from *s2* are first copied into a temporary array of *n* characters that does not overlap the objects pointed to by *s1* and *s2*. The *n* characters from the temporary array are then copied into *s1*. This function returns the value of *s1*.

▶ memset( )

Copies the value of *c*, converted to an unsigned **char**, into each of the first *n* characters of the object *s*, and returns the value of *s*.

```
#include <string.h>
void *memset(void *s, int c, size_t n);
```

▶ mktime( )

Converts the broken-down time, expressed in local time, in *timeptr* to a calendar time value that has the same encoding as the values returned by time( ), and returns that value.

```
#include <time.h>
time_t mktime(struct tm *timeptr);
```

▶ modf( )

Splits *value* into integral and fractional parts whose signs are the same as *value*'s sign.

```
#include <math.h>
double modf(double value, double *iptr);
```

This function stores the integral part as a **double** in *iptr*, and returns the fractional part.

▶ perror( )

Maps errno to an error message and writes a sequence of characters to stderr.

```
#include <stdio.h>
void perror(const char *s);
```

▶ pow( )

Computes and returns *x* raised to the power *y*.

```
#include <math.h>
double pow(double x, double y);
```

If *x* is negative and *y* is not an integral value, a domain error occurs.

▶ printf( )

Equivalent to fprintf( ) with *stdout* for the stream.

```
#include <stdio.h>
int printf(const char *format, ...);
```

This function returns the number of characters written. If an output error occurred, it returns a negative value.

▶ putc( )

Equivalent to fputc( ), except that putc( ) is implemented as a macro.

```
#include <stdio.h>
int putc(int c, FILE *stream);
```

This function returns the character written. If a write error occurs, it sets the error indicator for the stream and returns EOF.

▶ putchar( )

Equivalent to putc( ) with *stdout* for the stream.

```
#include <stdio.h>
int putchar(int c);
```

This function returns the character written. If a write error occurs, it sets the error indicator for the stream and returns EOF.

▶ puts( )

Writes the string *s* to *stdout* and appends a newline character to the output.

```
#include <stdio.h>
int puts(const char *s);
```

If successful, this function returns a nonnegative value. If a write error occurs, it returns EOF.

▶ qsort( )

Sorts an array *base* of *nmemb* objects. *size* specifies the size of each object. qsort( ) sorts the array contents into ascending order according to a comparison function *compar* that is called with two arguments pointing to the objects being compared.

```
#include <stdlib.h>
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

If the first argument is less than, equal to, or greater than the second, this function returns an integer that is less than, equal to, or greater than zero, respectively.

▶ raise( )

Sends the signal *sig* to the executing program.

```
#include <signal.h>
int raise(int sig);
```

If successful, this function returns zero. If it is unsuccessful, it returns nonzero.

▶ rand( )

Computes and returns a pseudo-random integer in the range 0 to RAND_MAX.

```
#include <stdlib.h>
int rand(void);
```

▶ realloc( )

Changes the size of the object pointed to by *ptr* to the size specified by *size*.

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

The contents of the object are unchanged up to the lesser of the new and old sizes. If the new size is larger, the value of the newly allocated portion of the object is indeterminate.

▶ remove( )

Causes the file whose name is the string *filename* to be no longer accessible.

```
#include <stdio.h>
int remove(const char *filename);
```

The next attempt to open that file using the name in *filename* fails, unless the file is to be created. This function returns zero if it is successful, and nonzero if it fails.

▶ rename( )

Causes the file whose name is the string *old* to be known by the name given by the string *new*.

```
#include <stdio.h>
int rename(const char *old, const char *new);
```

The file named *old* is no longer accessible. If *new* exists prior to the call, it will be overwritten. If successful, this function returns zero. If it fails, it returns nonzero; in this case if the file existed previously, it is still known by its original name.

▶ rewind( )

Sets the file position indicator for the stream *stream* to the beginning of the file.

```
#include <stdio.h>
void rewind(FILE *stream);
```

▶ scanf( )

Equivalent to fscanf( ) with *stdin* for the stream.

```
#include <stdio.h>
int scanf(const char *format, ...);
```

This function returns the number of assigned input items. If there is an early matching failure, this number can be fewer than provided for, or even zero. If an input failure occurs before any conversion, scanf( ) returns EOF.

▶ setbuf( )

This function may be used only after *stream* is associated with an open file, and before any operation is performed on the stream.

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
```

If *buf* is not a null pointer, this function may use the array it points to in place of an automatically allocated buffer. Input and output are fully buffered, and the array has a size of BUFSIZE.

If *buf* is a null pointer, input and output are unbuffered, which is permitted on binary files only.

▶ setjmp( )

Retains the calling environment in the *jmp_buf* argument for later use by longjmp( ).

```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

If the return is from a direct invocation, this function returns zero. If the return is from a call to longjmp( ), it returns nonzero.

▶ setlocale( )

Selects the portion of the program's locale that is specified by *category* and *locale*.

```
#include <locale.h>
char *setlocale(int category, const char *locale);
```

This function queries or changes all or portions of the program's current locale. The LC_ALL category names the program's entire locale. The "C" *locale* specifies the minimal environment for C translation. The " " *locale* specifies the implementation-defined native environment.

If *locale* is not a null pointer and the selection can be honored, this function returns a pointer to the string associated with *category* for the new locale. If the selection cannot be honored, a null pointer is returned and the locale is not changed.

If *locale* is a null pointer, this function returns a pointer to the string associated with *category* for the current locale. The locale is not changed. At this release, only the "C" locale is supported.

▶ setvbuf( )

This function may be used only after the stream *stream* is associated with an open file, and before any operation is performed on the stream.

```
#include <stdio.h>
int setvbuf(FILE *stream, char *buf, int mode, size_t size);
```

The argument *mode* determines how *stream* will be buffered. _IOFBF causes fully buffered input and output. _IOLBF causes line buffered input and output. _IONBF causes unbuffered input and output.

If *buf* is not a null pointer, setvbuf( ) may use the array it points to in place of an automatically allocated buffer. *size* specifies the array size.

If successful, this function returns zero. If an invalid value is given for *mode* or if the request cannot be honored, it returns nonzero.

▶ signal( )

Specifies how the receipt of *sig* is to be handled.

```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
```

If *func* is SIG_DFL, default handling occurs for the signal. If *func* is SIG_IGN, the signal is ignored. Otherwise, *func* points to a function to be called when the signal occurs.

If the request can be honored, this function returns the value of *func* for the most recent call to signal( ) for the specified *sig*. Otherwise, it returns SIG_ERR.

▶ sin( )

Computes and returns the sine of x expressed in radians.

```
#include <math.h>
double sin(double x);
```

▶ sinh( )

Computes and returns the hyperbolic sine of x.

```
#include <math.h>
double sinh(double double x);
```

If the magnitude of x is too large, a range error occurs.

▶ sprintf( )

Equivalent to fprintf( ), except that *s* specifies an array into which the generated output is to be written, rather than a stream.

```
#include <stdio.h>
int sprintf(char *s, const char *format, ...);
```

This function returns the number of characters written in *s*, excluding the terminating null character.

▶ sqrt( )

Computes and returns the square root of *x*.

```
#include <math.h>
double sqrt(double x);
```

If *x* is negative, a domain error occurs.

▶ srand( )

Uses *seed* as a seed for a new sequence of pseudo-random numbers that are to be returned by later calls to rand( ).

```
#include <stdlib.h>
void srand(unsigned int seed);
```

If srand( ) is then called with the same *seed* value, the sequence of pseudo-random numbers is repeated. If rand( ) is called before any calls to srand( ), the same sequence is generated as when srand( ) is first called with a *seed* value of 1.

▶ sscanf( )

Equivalent to fscanf( ), except that *s* specifies a string, rather than a stream, from which the input is to be obtained.

```
#include <stdio.h>
int sscanf(const char *s, const char *format, ...);
```

This function returns the number of assigned input items. If there is an early matching failure, this number can be fewer than provided for, or even zero. If an input failure occurs before any conversion, sscanf( ) returns EOF.

▶ strcat( )

Appends a copy of the string *s2*, including the null character, to the end of the string *s1*.

```
#include <string.h>
char *strcat(char *s1, const char *s2);
```

The first character of *s2* overwrites the null character at the end of *s1*. This function returns the value of *s1*.

▶ strchr( )

Finds the first occurrence of *c*, converted to **char**, in the string *s*.

```
#include <string.h>
char *strchr(const char *s, int c);
```

This function returns a pointer to the character found. If the character is not in *s*, it returns a null pointer.

▶ strcmp( )

Compares the string *s1* to the string *s2*.

```
#include <string.h>
int strcmp(cont char *s1, const char *s2);
```

If *s1* is greater than, equal to, or less than *s2*, this function returns an integer that is greater than, equal to, or less than zero, respectively.

▶ strcoll( )

Compares the string *s1* to the string *s2*, both of which are interpreted as appropriate to the LC_COLLATE category of the current locale.

```
#include <string.h>
int strcoll(const char *s1, const char *s2);
```

If *s1* is greater than, equal to, or less than *s2*, this function returns an integer that is greater than, equal to, or less than zero, respectively.

▶ strcpy( )

Copies the string *s2*, including the null character, into the array *s1*.

```
#include <string.h>
char *strcpy(char *s1, const char *s2);
```

This function returns the value of *s1*.

▶ strcspn( )

Computes the length of the first segment of string *s1* that has no characters that are also in string *s2*.

```
#include <string.h>
size_t strcspn(const char *s1, const char *s2);
```

This function returns the length of the segment.

▶ strerror( )

Maps *errnum* to an error message string and returns a pointer to the string.

```
#include <string.h>
char *strerror(int errnum);
```

This function maps a PRIMOS error code or a code from <errno.h> to the appropriate error message.

▶ strftime( )

Puts characters into the array *s* as controlled by the string *format*.

```
#include <time.h>
size_t strftime(char *s, size_t maxsize, const char *format,
                const struct tm *timeptr);
```

The string *format* has zero or more conversion specifiers and ordinary multibyte characters. A maximum of *maxsize* characters are placed into the array. The appropriate characters for replacement of each conversion specifier are determined by the LC_TIME category of the current locale and by the values contained in the structure pointed to by *timeptr*. Table 8-7 lists the conversion specifiers for this function.

If the total number of characters, including the null character, is not more than *maxsize*, this function returns the number of characters placed in *s*, not including the null character. Otherwise, it returns zero.

TABLE 8-7. Conversion Specifiers for strftime Function

Specifier	*Action Performed*
%a	Replaced by locale's abbreviated weekday name.
%A	Replaced by locale's full weekday name.
%b	Replaced by locale's abbreviated month name.
%B	Replaced by locale's full month name.
%c	Replaced by locale's date and time representation.
%d	Replaced by the day of the month as a decimal number (01 to 31).
%H	Replaced by the hour (24-hour clock) as a decimal number (00 to 23).
%I	Replaced by the hour (12-hour clock) as a decimal number (01 to 12).
%j	Replaced by the day of the year as a decimal number (001 to 366).
%m	Replaced by the month as a decimal number (01 to 12).
%M	Replaced by the minute as decimal number (00 to 59).
%p	Replaced by locale's equivalent of the AM/PM designations associated with a 12-hour clock.
%S	Replaced by the second as a decimal number (00 to 61).
%U	Replaced by the week number of the year (Sunday as the first day of week 1) as a decimal number (00 to 53).
%w	Replaced by the weekday as a decimal number (0 to 6 where Sunday is 0).
%W	Replaced by the week number of the year (Monday as the first day of week 1) as a decimal number (00 to 53).
%x	Replaced by locale's date representation.
%X	Replaced by locale's time representation.
%y	Replaced by the year without century as a decimal number (00 to 99).
%Y	Replaced by the year with century as a decimal number.
%Z	Replaced by the time zone name or abbreviation.
%%	Replaced by %.

▶ strlen( )

Computes the length of the string *s* and returns the number of characters that precede the null character.

```
#include <string.h>
size_t strlen(const char *s);
```

▶ strncat( )

Appends *n* characters from the array *s2* to the end of the string *s1*. This function does not append a null character from *s2* and the characters that follow it.

```
#include <string.h>
char *strncat(char *s1, const char *s2, size_t n);
```

The first character of *s2* overwrites the null character at the end of *s1*. A null character is appended to the result. This function returns the value of *s1*.

▶ strncmp( )

Compares *n* characters from the array *s1* to the array *s2*. This function does not compare the characters from *s2* that follow a null character.

```
#include <string.h>
int strncmp(const char *s1, const char *s2, size_t n);
```

If *s1* is greater than, equal to, or less than *s2*, this function returns an integer that is greater than, equal to, or less than zero, respectively.

▶ strncpy( )

Copies *n* characters from the array *s2* to the array *s1*. This function does not copy characters from *s2* that follow a null character.

```
#include <string.h>
char *strncpy(char *s1, const char *s2, size_t n);
```

If *s2* is shorter than *n* characters, null characters are appended to the copy in *s1* until *n* characters have been written. This function returns the value of *s1*.

▶ strpbrk( )

Finds the first occurrence in the string *s1* of any character from the string *s2*.

```
#include <string.h>
char *strpbrk(cont char *s1, const char *s2);
```

This function returns a pointer to the character found. If no character from *s2* is in *s1*, a null pointer is returned.

▶ strrchr( )

Finds the last occurrence of *c*, converted to **char**, in the string *s*.

```
#include <string.h>
char *strrchr(const char *s, int c);
```

This function returns a pointer to the character found. If *c* is not in *s*, a null pointer is returned.

▶ strspn( )

Computes the length of the first segment of string *s1* that consists solely of characters from string *s2*.

```
#include <string.h>
size_t strspn(const char *s1, const char *s2);
```

This function returns the length of the segment.

▶ strstr( )

Finds the first occurrence in string *s1* of the sequence of characters in string *s2*, excluding the null character.

```
#include <string.h>
char *strstr(const char *s1, const char *s2);
```

This function returns a pointer to the located string. If the string is not found, a null pointer is returned. If *s2* has a length of zero, *s1* is returned.

▶ strtod( )

Converts the first part of *nptr* to **double** representation.

```
#include <stdlib.h>
double strtod(const char *nptr, char **endptr);
```

The expected form of the sequence has the following order: an optional plus or minus sign; a nonempty sequence of digits with an optional decimal point character; an optional exponent. Any unconverted part of *nptr* is pointed to by *endptr*.

This function returns the converted value. If it cannot perform a conversion, zero is returned. If the correct value is outside the range of representable values, strtod( ) returns plus or minus HUGE_VAL, according to the sign of the value. If the correct value would cause underflow, zero is returned.

▶ strtok( )

A sequence of calls to this function breaks string *s1* into a sequence of tokens. A character from string *s2* delimits each of these tokens.

```
#include <string.h>
char *strtok(char *s1, const char *s2);
```

s1 is the first argument of the first call in the sequence. Following calls have a null pointer as their first argument. *s2* is a separator string, and it may be different from call to call.

The first call searches *s1* for the first character not in the current *s2* separator string. If the search is successful, the character found is the start of the first token. If no such character is found, the function returns a null pointer.

strtok( ) searches from there for a character that is in the current separator string. If the search is successful, the character found is overwritten by a null character that terminates the current token; the function saves a pointer to the following character, from which the next search for a token will start. If no such character is found, the current token extends to the end of *s1*; subsequent searches for a token return a null pointer. Each subsequent call has a null pointer for the first argument and starts searching from the saved pointer.

▶ strtol( )

Converts the first part of *nptr* to **long int** representation.

```
#include <stdlib.h>
long int strtol(const char *nptr, char **endptr, int base);
```

If *base* is zero, the expected order of the sequence is as follows: an optional plus or minus sign; an integer constant.

If *base* is between 2 and 36, the expected order of the sequence is as follows: an optional plus or minus sign; a series of letters and digits that represent an integer whose radix is

specified by *base*. The letters from a (or A) through z (or Z) are used for the values 10 to 35. Only letters whose values are less than that of *base* are permitted.

If *base* is 16, the expected order of the sequence is as follows: an optional plus or minus sign; an optional 0x or 0X; the sequence of letters and digits.

endptr points to any unconverted part of *nptr*. This function returns the converted value. If the correct value is outside the range of representable values, LONG_MAX or LONG_MIN is returned, according to the sign of the value.

▶ stroul( )

Converts the first part of *nptr* to **unsigned long int** representation.

```
#include <stdlib.h>
unsigned long stroul(const char *nptr, char **endptr, int base);
```

If *base* is zero, the expected order is as follows: an optional plus or minus sign; and an integer constant.

If *base* is between 2 and 36, the expected order is as follows: an optional plus or minus sign; and a series of letters and digits that represent an integer with a *base*-specified radix. The letters from a (or A) through z (or Z) are used for values 10 to 35. Only the letters whose values are less than *base* are permitted.

If *base* is 16, the expected order is as follows: an optional plus or minus sign; an optional 0x or 0X; and a series of letters and digits that represent an integer.

endptr points to any unconverted portion of *nptr*. This function returns the converted value. If the correct value is outside the range of values that can be represented, ULONG_MAX is returned.

▶ strxfrm( )

Transforms the string *s2* and puts the string result into the array *s1*. A maximum of *n* characters are put into *s1*, including the null character. If *n* is zero, *s1* is a null pointer.

```
#include <string.h>
size_t strxfrm(char *s1, const char *s2, size_t n);
```

The transformation is such that applying strcmp( ) to two transformed strings returns a value greater than, equal to, or less than zero. This corresponds to the result of applying strcoll( ) to the same two original strings.

This function returns the length of the transformed string, not including the null character.

▶ system( )

Passes *string* to the host environment, where a command processor executes it.

```
#include <stdlib.h>
int system(const char *string);
```

string may be a null pointer to ask if a command processor exists: in this case if it does exist, nonzero is returned.

If *string* is not a null pointer and the command executed successfully, this function returns zero. If it failed, nonzero is returned.

▶ tan( )

Computes and returns the tangent of x expressed in radians.

```
#include <math.h>
double tan(double x);
```

▶ tanh( )

Computes and returns the hyperbolic tangent of x.

```
#include <math.h>
double tanh(double x);
```

▶ time( )

Returns the present calendar time in an encoded form.

```
#include <time.h>
time_t time(time_t *timer);
```

▶ tmpfile( )

Creates a temporary binary file that is opened for update with "wb+" mode. This temporary file is automatically removed either when it is closed or when the program terminates normally.

```
#include <stdio.h>
FILE *tmpfile(void);
```

If the program terminates abnormally, the temporary file is not removed. If the file is created, this function returns a pointer to the stream of the file. If the file could not be created, a null pointer is returned.

▶ tmpnam( )

Creates a string that is a valid filename and not the same as an existing filename. This function creates a new string each time it is called for a maximum of TMP_MAX times.

```
#include <stdio.h>
char *tmpnam(char *s);
```

If *s* is a null pointer, tmpnam( ) leaves the result in an internal static object and returns a pointer to that object.

If *s* is not a null pointer, it points to an array of at least L_tmpnam characters where tmpnam( ) writes the result; *s* is returned.

▶ tolower( )

Converts an uppercase letter to its corresponding lowercase form.

```
#include <ctype.h>
int tolower(int c);
```

This function returns the corresponding character.

▶ toupper( )

Converts a lowercase letter to its corresponding uppercase form.

```
#include <ctype.h>
int toupper(int c);
```

This function returns the corresponding character.

▶ ungetc( )

Pushes *c* back onto *stream*. If successful, this function returns the character pushed back. If it fails, it returns EOF.

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

▶ va_arg( )

This is a macro for functions that may be called with a variable number of arguments of varying types.

```
#include <stdarg.h>
type va_arg(va_list ap, type);
```

The macro expands to an expression whose value and type are the same as the next argument in the call. The parameter *ap* is the same as the *va_list ap* that va_start( ) initializes. Each va_arg( ) invocation modifies *ap* to return the values of successive arguments.

The parameter *type* is a type name such that the type of a pointer to the specified-type object can be gotten by appending an asterisk (*) to *type*. va_arg( )'s first invocation returns the value of the argument after that specified by *parmN*. Successive invocations return in succession the remaining argument values.

▶ va_end( )

This macro facilitates a normal return from the function with a variable argument list referred to by va_start( ).

```
#include <stdarg.h>
void va_end(va_list ap);
```

va_end( ) modifies *ap*, which makes it unusable without invoking va_start( ) again.

▶ va_start( )

This is a macro for functions that may be called with a variable number of arguments of varying types.

```
#include <stdarg.h>
void va_start(va_list ap, parmN);
```

va_start( ) initializes *ap* for later use by va_arg( ) and va_end( ). This macro must be invoked before any access to the unnamed arguments. *parmN* is the rightmost parameter in the variable parameter list of the function definition.

▶ vfprintf( )

Equivalent to fprintf( ) except that the variable argument list is replaced by *arg*, which va_start( ) must initialize.

```
#include <stdarg.h>
#include <stdio.h>
int vfprintf(FILE *stream, const char *format, va_list arg);
```

This function returns the number of written characters. If an output error occurred, a negative value is returned.

▶ vprintf( )

Equivalent to printf( ), except that the variable argument is replaced by *arg*, which va_start( ) must initialize.

```
#include <stdarg.h>
#include <stdio.h>
int vprintf(const char *format, va_list arg);
```

This function returns the number of written characters. If an output error occurred, a negative value is returned.

▶ vsprintf( )

Equivalent to sprintf( ), except that the variable argument list is replaced by *arg*, which va_start( ) must initialize.

```
#include <stdarg.h>
#include <stdio.h>
int vsprintf(char *s, const char *format, va_list arg);
```

This function returns the number of characters written in the array, excluding the terminating null character.

▶ wcstombs( )

Converts a series of codes that correspond to multibyte characters from the array *pwcs*.

```
#include <stdlib.h>
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

The conversion is into a sequence of multibyte characters that begins in the initial shift state. These multibyte characters are put into the array *s*. The conversion stops if a null character is stored or if a multibyte character would exceed the limit of *n* total bytes. This function returns the number of modified bytes, excluding the null character.

▶ wctomb( )

Finds out how many bytes are needed to represent the multibyte character that corresponds to the code *wchar*.

```
#include <stdlib.h>
int wctomb(char *s, wchar_t wchar);
```

If *s* is not a null pointer, the multibyte character representation is stored in the array *s*. A maximum of MB_CUR_MAX characters are stored. This function returns the number of bytes in the multibyte character that corresponds to *wchar*. If *wchar*'s value does not correspond to a valid multibyte character, -1 is returned.

If *s* is a null pointer, this function returns the following: a nonzero value if the multibyte character encodings are not state-dependent; zero if they are state-dependent.

Note

At this release if *s* is a null pointer, this function returns zero; otherwise it returns 1.

APPENDICES

A

EXTENSIONS TO THE C LANGUAGE

This appendix describes the extensions Prime has made to the C language as described by Kernighan and Ritchie in the first (1978) edition of *The C Programming Language.*

Most of these extensions are similar to those of other vendors, and most are now part of the ANSI X3J11 C standard.

The Prime extensions to the 1978 C language are

- The enumeration data type (**enum**)
- The **void** data type
- The **long double** data type and quadruple precision floating point numbers (32IX mode only)
- The **fortran** storage class
- The unary plus (+) operator (32IX mode only)
- Identifier names up to 32 characters in length
- New and enhanced preprocessor commands
- Automatic string concatenation

ENUMERATION DATA TYPE

The enumeration data type is an extension to the 1978 C language, but is included in the ANSI C standard.

The enumeration data type is analogous to the scalar data type found in Pascal. New enumeration data types in C can be defined by writing a type specifier followed by an ordered list of identifiers. These identifiers are declared as constants. Enumeration constants must all be unique. The declaration format for type **enum** is as follows:

enum [tagName] {enumConst1[=value], enumConst2[=value], ...} [iden1, iden2, ...]:

The values of enumeration may be explicitly set by using the *=value* clause. If this clause is not specified, the compiler chooses a value that is one higher than the previous enumeration constant. The value for the first enumeration constant in each enumeration list is 1 unless otherwise specified.

Also, objects of a given enumeration data type are regarded as having a type distinct from objects of all other data types. All enumeration variables are treated as if they were of type **int**.

The following examples show how to use the **enum** data type:

```
enum color {red, green, blue, yellow};
main( )
{
     enum color a,b,c;
     a = red;
     b = yellow;
     c = red;
     if (a == c && b == yellow)
          printf("Pass.\n");
     else
          printf("Fail.\n");
}
```

```
typedef enum {Read = 01, Write = 02, Update = 04,
              Scribble = 010,
              PrettyMuchAnythingAtAll = 020,
              TotalPower = 040} Rights;
Rights myAccessRights, yourAccessRights;
```

VOID DATA TYPE

The **void** data type is an extension to the 1978 C language, but is included in the ANSI C standard.

Use the **void** data type to ensure that a value is never used. You cannot use a **void** value in any way, nor can you convert it to another data type. A **void** expression may be used only as an expression statement or as the left operand of a comma expression.

You can use the **void** keyword in a declaration statement, a function definition, or a cast statement.

The following example shows the use of the **void** data type in a declaration statement.

```
main( )
{
     fortran void sleep$( );

     sleep$((long)1000);
}
```

If you attempt to use a **void** value in any way, the C compiler displays the following error message:

```
This expression attempted to use the value of a sub-expression that
had the data type of "void"; this is illegal.
```

THE LONG DOUBLE DATA TYPE

The **long double** data type is an extension to the 1978 C language, but is included in the ANSI C standard.

In 32IX mode, PRIMOS C supports quadruple precision floating point constants and variables, which programs can declare as type **long double**. The -ANSI compiler option supports the long double data type. Without the -ANSI option, to enable support for quadruple precision constants, use the -QUADCONSTANTS compiler option. Similarly, to enable support for quadruple precision variables without using -ANSI, use the -QUADFLOATING compiler option. For more information, see Chapter 2.

FORTRAN STORAGE CLASS

The **fortran** keyword is an extension to the 1978 C language, but is permitted by the ANSI C standard unless you compile your program with the -ANSI and -STRICTCOMPLIANCE compiler options.

When you call procedures, the **fortran** storage class forces certain arguments to be passed by reference. See Chapter 5, Interfacing to Other Languages, for more information.

UNARY PLUS OPERATOR

The unary plus operator is an extension to the 1978 C language, but is part of the ANSI C standard.

In 32IX mode, programs can use the unary plus (+) operator in front of a numeric expression to indicate explicitly that the value of the expression is positive.

IDENTIFIER NAMES

All internal identifier names may contain a maximum of 32 characters. If the -NOCOMPATIBILITY option is used, however, the C compiler truncates identifier names to eight characters.

The dollar sign ($) can be used as a character in an identifier name, thus allowing compatibility with PRIMOS routine names and arguments.

Allowing identifier names up to 32 characters in length is an extension to the 1978 C language, but is part of the ANSI C standard. Use of the dollar sign in identifiers is an extension to both the 1978 C language and the ANSI C standard.

PREPROCESSOR COMMANDS

Extensions to the 1978 C Language

The following are extensions to the 1978 C language but are part of the ANSI C standard. They are available only in 32IX mode.

#elif command
defined operator
Use of preprocessor tokens in **#include** commands

#elif Command: Any number of **#elif** (else if) commands can occur after an **#if**, **#ifdef**, or **#ifndef** command and before an **#else** command. The syntax for this command is

#elif <constant expression>

For example:

```
#define Version 2
#if Version == 0
    #define Rev "1.0"
#elif Version == 1
    #define Rev "1.1"
#elif Version == 2
```

```
    #define Rev "1.2"
#else
    #define Rev "1.unknown"
#endif
```

defined Operator: The defined operator can take either of two forms:

defined identifier

defined (identifier)

The expression

#if defined (ident)

has the same meaning as

#ifdef ident

For example:

```
#define Fun(a) (a = a)
#if defined(Fun) && !defined(fun)
    int thisWillBeIncluded;
#endif
#if defined JustForFun
    int thisWillNotBeIncluded;
#elif defined Fun
    int butThisWillBe;
#endif
```

Use of Tokens With #include Commands: The #include preprocessor command in 32IX mode allows the use of preprocessor tokens. For example, the following are legal:

```
#define PathName "foo>bar>silly"
#include PathName

#define MasterTree "SourceMaster>ins"
#define AnIncludeFile "file1"
#include MasterTree ">" AnIncludeFile
#include MasterTree ">file2"
```

Extensions to the 1978 C Language and the ANSI C Standard

Two PRIMOS C preprocessor commands are extensions to both the 1978 C language and the ANSI C standard:

#list
#nolist

The **#list** and **#nolist** commands start and stop output to a specified listing file. These commands are available in both 64V and 32IX modes.

Extensions to the 1978 Language That Are Not Available in ANSI C

The following PRIMOS C preprocessor commands are available only if you compile without the -ANSI option:

#assert (available only in 32IX mode)
#display (available only in 32IX mode)
#endincl (available in 64V and 32IX modes)

#assert Command: This preprocessor command takes a constant expression argument and is used for compile-time diagnostics. If the constant expression evaluates to true, no action is taken. If the constant expression evaluates to false, compilation is aborted, and an `Assertion failed` message appears indicating the source file name and the line number. The **#assert** command has the following format:

#assert constExpr

For example, if `array` is a previously declared variable, the statements

```
#define SEGMENTSIZE (2048 * 64)
#assert sizeof(array) <= SEGMENTSIZE
```

abort compilation and print a message if the array is larger than a segment.

```
OK, cc prog -32ix
[CI Rev. T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]
Assertion failed: file "<mysys>me>prog.c", line 4.

01 Error and 00 Warnings detected in 4 source lines.

#assert failed! (CC)
ER!
```

#display Command: This preprocessor command takes a string argument, which it displays on the user's terminal during compilation. In this way, the user can observe compile-time progress through the program. The **#display** command has this format:

#display stringConst

For example:

```
#display "I have reached this point"
```

#endincl Command: This preprocessor command can be placed in an **#include** file. It logically terminates the **#include** file prior to its physical end.

AUTOMATIC STRING CONCATENATION

Automatic string concatenation is an extension to the 1978 C language, but is included in the ANSI C standard. PRIMOS C automatically concatenates two string literals. That is, the string

```
"Hello ""there.\n"
```

is functionally equivalent to the string

```
"Hello there.\n"
```

B

DEBUGGING C PROGRAMS

This appendix contains information about the Prime Source Level Debugger (DBG). The first section provides a general description of DBG. The next section describes how to use DBG. The last section explains how DBG interacts with certain constructs that are specific to the C language.

DBG enables you to debug C programs using nearly all the same high-level language constructs as are available in the C programming language. To use DBG, you need not be an experienced programmer or know anything about assembly language or machine architecture. All you need is the ability to write programs.

DBG contains powerful features designed to make the debugging task as efficient as possible. Program execution can be controlled at all levels by either single-statement stepping, procedure-by-procedure stepping, or setting breakpoints at statements, procedure entries, and procedure exits. Data can be manipulated at any time using all legal expressions and operations available in the C programming language. DBG streamlines the effort by providing command editing and resubmission, conditional breakpointing, and a macro facility that closely resembles the PRIMOS abbreviation facility. Many more features, especially for the most sophisticated programmers, exist; for information, see the *Source Level Debugger User's Guide.*

DBG is multilingual. In addition to C, it supports all of the 50 Series high-level programming languages (FTN, F77, Pascal, PL/I, CBL, and VRPG). Therefore, when C programs are mixed with any other language, all debugging can be done under DBG using all of the same features. DBG automatically switches contexts from one programming language to the other, supporting the syntax and constructs belonging to the language in which a given procedure is written.

USING DBG

If you are unfamiliar with DBG operation, see the *Source Level Debugger User's Guide.*

The -DEBUG Option

Before using the debugger, you must compile your program using the -DEBUG option to the CC command. (See Chapter 2 for more information about compiling programs.) The -DEBUG option instructs the C compiler to generate specially embedded information required by DBG in order to debug that program.

Link your program as usual. (See Chapter 3 for information about linking C programs.)

Running Under the Debugger

DBG is used interactively. It is invoked from PRIMOS command level with the DBG command. The format of the DBG command is

DBG program-name [option-1 option-2 . . .]

where *program-name* indicates the name of the executable file to debug, and *option-1 option-2* . . . indicates DBG command line options. (You do not need to know these options if you are a novice.)

After you enter the command line, the DBG identification banner is displayed and you are prompted for DBG commands with the DBG prompt, >. For example,

```
OK, DBG MYPROGRAM.RUN

[DBG Rev. T3.0-23.0 20-May Copyright (c) Prime Computer, Inc. 1990]

>
```

You are now ready to begin debugging. See the *Source Level Debugger User's Guide* for complete information on using DBG.

Debugging a C Program That Requires Command Line Arguments

The method used to provide a C program with its required command line arguments from DBG differs slightly from that used with other 50 Series languages. This section applies *only* to programs that take command line arguments. If you write such programs, read this section carefully.

When you debug a C program that uses the CCMAIN or ANSI_CCMAIN library's command line argument feature, you must use DBG's CMDLINE command to inform the debugger about command line arguments before it begins execution of a program. Because arguments to a program cannot be entered on DBG's command line, the CMDLINE command must provide the necessary arguments before DBG executes the program.

The format of the CMDLINE command is

```
CMDLINE
```

For example, if you execute the C program MYPROGRAM at PRIMOS command level by entering

```
OK, RESUME MYPROGRAM.RUN -OPTION_1 -OPTION_2 ARG
```

you execute it under DBG by the following series of commands:

```
OK, DBG MYPROGRAM.RUN

[DBG Rev. T3.0-23.0 20-May Copyright (c) Prime Computer, Inc. 1990]

> CMDLINE
Enter command line:
MYPROGRAM -OPTION_1 -OPTION_2 ARG
>
```

You are now ready to begin debugging. The program is given the proper arguments when execution begins.

Debugging Programs That Use the CCMAIN or ANSI_CCMAIN Library

If you link your program with one of the CCMAIN libraries (either CCMAIN or ANSI_CCMAIN) in order to emulate the command line argument features of the UNIX operating system, your program will start not with your own main() routine but with a routine called cc$main(). If you wish to debug such a program, you should be aware of two limitations:

- In a program linked with a CCMAIN library, the CCMAIN library's routines are the first routines to execute. Because the CCMAIN library's routines are not compiled in debug mode, you cannot single-step through them.
- To set a breakpoint at the entry to your main routine, do not give the DBG command

  ```
  BRK
  ```

  with no arguments. This command will result in an error message. Instead, give the command

  ```
  BRK main\\ENTRY
  ```

  where main is the name of your main() routine.
- Another way of getting to your main() routine is by giving the command

  ```
  ENV main
  ```

  DBG will respond "New language is C."

For information about setting breakpoints and single-stepping, see the *Source Level Debugger Guide.*

DBG AND C LANGUAGE CONSTRUCTS

Assignment

When you use DBG's evaluation command (:) along with any of the special C assignment operators = += -= *= /= %= >>= <<= &= ^= |=, assignment is implicitly performed. This means that expressions are evaluated in DBG in exactly the same way they are evaluated in C programs. (Ordinarily, you must use DBG's LET command to assign values.) For example, if x equals 1, you can use the evaluation command with the += operator in the following way:

```
> : x += 2
X = 3
```

In the example above, the value of 3 is assigned to x.

An attempt to assign a value to an **rvalue** (for example, an expression enclosed within parentheses) does *not* cause an error and appears to be successful. However, this is an illegal operation. DBG does not report the error. (An **rvalue** is a value that may appear on the right side of an assignment statement.)

C Operators

The only C operator not supported by DBG is the cast operator. DBG supports all other PRIMOS C operators. Furthermore, these supported operators used to evaluate expressions in DBG are functionally identical to the corresponding operators in the PRIMOS C compiler. All expected side effects that occur in C programs also occur when the operators are used from DBG. For example, an increment operator (++) that precedes an integer variable returns the value of the variable plus 1 *and* has the side effect of incrementing that variable by 1. For example,

```
> : i = 1
I = 1
> : ++i
2
> : i
I = 2
```

Special Characters

DBG does not support the C escape character (\). Instead, use DBG's escape character (^). You can generate a null character ('\0') by evaluating a null string (""). To generate a literal newline ('\n'), enter ^ followed by RETURN; to generate a literal single quote, enter ^'.

Defaults for Constants

The default for a floating-point constant is **double.** The default for an integer constant is **long.**

The ?: Construct

DBG does not support the ?: construct. Use the **if. . .else** construct instead.

Character Strings

In C, character strings are stored as arrays of character, delimited by a byte that is set to 0. Use the Dollar Extent character ($) to display such an array of characters as a string instead of as an array.

Use the Dollar Extent character in the same manner as the Star Extent character. (See the *Source Level Debugger User's Guide* for a full explanation of the Star Extent character.) However, you can use the Dollar Extent only with arrays of character or pointers to character.

The following example illustrates the difference between the Star Extent character and the Dollar Extent character. The program consists of the following code:

```
main( )

{   char array_of_char[10], *ptr_to_string;
    int array_of_int[10], *ptr_to_int;

    strcpy (array_of_char, "testline");
    ptr_to_string = array_of_char;
    ptr_to_int = array_of_int;
    array_of_int[1] = 5;
}
```

To inspect the entire array value of ARRAY_OF_CHAR, use the Star Extent character and enter

```
> : ARRAY_OF_CHAR[*]
```

DBG displays

```
ARRAY_OF_CHAR(0) = 't'
ARRAY_OF_CHAR(1) = 'e'
ARRAY_OF_CHAR(2) = 's'
ARRAY_OF_CHAR(3) = 't'
ARRAY_OF_CHAR(4) = 'l'
ARRAY_OF_CHAR(5) = 'i'
ARRAY_OF_CHAR(6) = 'n'
ARRAY_OF_CHAR(7) = 'e'
ARRAY_OF_CHAR(8) = ''
ARRAY_OF_CHAR(9) = ''
```

To inspect the value of ARRAY_OF_CHAR as a character string, use the Dollar Extent character instead and enter

```
> : ARRAY_OF_CHAR[$]
```

DBG displays

```
ARRAY_OF_CHAR = 'testline'
```

You can also use the Dollar Extent character with a pointer to character:

```
> : PTR_TO_STRING[$]
```

DBG displays

```
PTR_TO_STRING = 'testline'
```

Using the Dollar Extent character on something other than an array of characters is illegal. For example,

```
> : ARRAY_OF_INT[$]
Illegal operation with dollar extent.  Only arrays of character are
permitted.
ARRAY_OF_INT[$]
             ^
```

DBG displays the promoted type and value of an argument within a function. For example, a **char** with value 'c' shows as 195, and its type is **int** within the subroutine. This is also true when you use the DBG command ARGS to display the values of the arguments of a function. You may evaluate a character variable within a function as a **char** by designating a print mode of ASCII in the evaluation statement.

SAMPLE DBG SESSION

Suppose you have written a C program that takes a positive number and returns the next power of 2 that is greater than this number. The program compiles and links successfully, but watch what happens when you execute it:

```
OK, CC TEST -DEBUG
[CC Rev. T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]
00 Errors and 00 Warnings detected in 23 lines and 90 include lines.
OK, BIND -LO TEST -LI CCLIB -LI
[BIND T3.0-23.0 Copyright (c) 1990, Prime Computer, Inc.]
BIND COMPLETE
OK, RESUME TEST
Get power of 2 greater than any positive number
To exit the program, enter 0
Enter a positive number
13
The power of 2 greater than this number is. . .
16
Enter a positive number
4
The power of 2 greater than this number is. . .
 4
Enter a positive number
0
OK,
```

Notice that the program correctly gives the next power of 2 greater than 13, which is 16.

But it also returns, incorrectly, 4 as the next power of 2 greater than 4. You enter DBG and look at the source program:

```
OK, DBG TEST

[DBG Rev. T3.0-23.0 20-May Copyright (c) Prime Computer, Inc. 1990]

> SOURCE TOP
> SOURCE PRINT 24
.NULL.
    1: #include <stdio.h>
    2: main( )
    3: {
    4:   int x;
    5:   printf("Get power of 2 greater than any positive number\n");
    6:   printf("To exit the program, enter 0\n");
    7:   printf("Enter a positive number\n");
    8:   scanf("%d", &x);
    9:   while (x != 0)
   10:   {
   11:     fun(x);
   12:     printf("Enter a positive number\n");
   13:     scanf("%d", &x);
   14:   }
   15: }
   16: fun(x)
   17: int x;
   18: {
   19:   int i;
   20:     for (i=1; i < x; i <<= 1);
   21:   printf("The power of 2 greater than this number is. . .\n");
   22:   printf("%2d\n", i);
   23: }
>
```

To make sure your input is read correctly, place a breakpoint on source line 9, restart the program, and check the value of variable `x`:

```
> BREAKPOINT 9
> RESTART
Get power of 2 greater than any positive number
To exit the program, enter 0
Enter a positive number
4

**** breakpointed at MAIN\9
> : X
X = 4
>
```

Now that you know that the value of `x` has been assigned correctly, you suspect that tne problem lies in the function `fun`. You step into this function at source line 11, then make sure the argument for the function has been successfully passed:

```
> BREAKPOINT 11
> CONTINUE

**** breakpointed at MAIN\11
> STEPIN
```

```
**** "in" completion at FUN\20
> ARGUMENTS
X = 4
>
```

The argument x for function fun has been passed correctly. Therefore, you decide to trace the value of i, because it is the only variable whose value changes in this function:

```
> WATCH I
> CONTINUE
The value of FUN\I has been changed at FUN\20
  from 1
  to   2
The value of FUN\I has been changed at FUN\20
  from 2
  to   4
The power of 2 greater than this number is. . .
 4
Enter a positive number
3

**** breakpointed at MAIN\9
> QUIT
OK,
```

You now notice that variable i failed to loop the appropriate number of times. Your loop should stop as soon as the value of i is greater than or equal to the value of x; and, in this case, i's value is equal to x's value. Instead of

```
i < x
```

your final loop value for i should read

```
i <= x
```

After you correct the mistake, you run the program again, and it executes successfully:

```
OK, RESUME TEST
Get power of 2 greater than any positive number
To exit the program, enter 0
Enter a positive number
33
The power of 2 greater than this number is. . .
64
Enter a positive number
12
The power of 2 greater than this number is. . .
16
Enter a positive number
4
The power of 2 greater than this number is. . .
 8
Enter a positive number
0
OK,
```

C

OPERATOR PRECEDENCE AND ASSOCIATIVITY

Table C-1 lists the C operators and their order of evaluation.

TABLE C-1. Operator Precedence Table

Operator		*Associativity*
Primary:	( ) [] -> .	Left to right
Unary:	! ~ ++ -- - (type) * & sizeof	Right to left
Multiplicative:	* / %	Left to right
Additive:	+ -	Left to right
Shift:	<< >>	Left to right
Relational:	< <= > >=	Left to right
Equality:	== !=	Left to right
Bitwise:	&	Left to right
Bitwise:	|	Left to right
Bitwise:	^	Left to right
Logical:	&&	Left to right
Logical:	||	Left to right
Conditional:	? :	Right to left
Assignment:	= += -= etc.	Right to left
comma:	,	Left to right

Asterisks (*) indicate defaults.

D

SUMMARY OF C LIBRARY FUNCTIONS

This appendix contains a set of tables listing C functions by the type of action performed. A list of these tables appears below.

In these tables, an asterisk (*) after a function name indicates that the function is part of the ANSI C library.

TABLE D-1. Diagnostics Library Functions

Function	*Description*
assert( ) *	Adds runtime diagnostics to programs. Available in 32IX mode only.

TABLE D-2. Character Handling Library Functions

Function	*Argument that Results in Nonzero Return Value*
isalnum( ) *	ASCII alphanumeric character
isalpha( ) *	ASCII alphabetic character
isascii( )	Valid ASCII character
iscntrl( ) *	Nonprinting ASCII character
isdigit( ) *	Decimal digit character
isgraph( ) *	Graphic ASCII character
islower( ) *	Lowercase alphabetic ASCII character
ispascii( )	Valid Prime ASCII character
isprint( ) *	ASCII printing character
ispunct( ) *	ASCII punctuation character
isspace( ) *	White space character (tab, return, form feed, newline)
isupper( ) *	Uppercase ASCII character
isxdigit( ) *	Hexadecimal digit

TABLE D-3. Localization Library Functions

Function	*Description*
localeconv( ) *	Sets components of the current locale.
setlocale( ) *	Changes or queries a program's entire current locale or portions of it.

TABLE D-4. Mathematics Library Functions

Function	*Returned Value*
acos( ) *	A value in the range 0 to π, which is the arc cosine of the argument expressed in radians.
asin( ) *	A value in the range $-\pi/2$ to $\pi/2$, which is the arc sine of the argument expressed in radians.
atan( ) *	A value in the range $-\pi/2$ to $\pi/2$, which is the arc tangent of the argument expressed in radians.
atan2( ) *	A value in the range $-\pi$ to π, which is the arc tangent of the two values.
cabs( )	sqrt($x^2 + y^2$).
ceil( ) *	(As a **double**) the smallest integer that equals or is greater than the argument.
cos( ) *	The cosine of the argument expressed in radians.
cosh( ) *	The hyperbolic cosine of the argument.
exp( ) *	The base e raised to the power of the argument.
fabs( ) *	The absolute value of a floating-point number.
floor( ) *	(As a **double**) the largest integer that is less than or equal to the argument.
fmod( ) *	The floating-point remainder of the first argument divided by the second argument.
frexp( ) *	The mantissa of a **double** value.
hypot( )	sqrt($x^2 + y^2$).
ldexp( ) *	The quantity x * 2^{exp}.
log( ) *	The natural (base e) logarithm of the argument, which must be **double**.
log10( ) *	The natural (base 10) logarithm of the argument, which must be **double**.
modf( ) *	The positive fraction part of a specified **double**.
pow( ) *	The first argument raised to the power of the second argument.
sin( ) *	A **double** value that represents the sine of the argument expressed in radians.
sinh( ) *	A **double** value that represents the hyperbolic sine of the argument.

TABLE D-4. Mathematics Library Functions (continued)

Function	*Returned Value*
sqrt( ) *	The square root of the argument.
tan( ) *	A **double** value, which is the tangent of the argument expressed in radians.
tanh( ) *	A **double** value, which is the hyperbolic tangent of the **double** argument.

TABLE D-5. Non-local Jumps Library Functions

Function	*Description*
longjmp( ) *	Restores the context of a calling function that was stored in an environment buffer with the setjmp( ) function.
setjmp( ) *	Saves the context of the calling function in an environment buffer for a subsequent longjmp( ) call.

TABLE D-6. Signal Handling Library Functions

Function	*Description*
raise( ) *	Sends the signal specified by the argument to the program
signal( ) *	Determines how conditions that occur during execution will be handled. Available in 32IX mode only.

TABLE D-7. Variable Arguments Library Functions

Function	*Description*
va_start( ) *	Initializes a variable argument list.
va_arg( ) *	Retrieves the next argument in a variable argument list.
va_end( ) *	Enables a normal return from a function with a variable argument list.

TABLE D-8. Input/Output Library Functions

Function	*Description*
bio$primosfileunit( )	Returns the PRIMOS file unit corresponding to a *fileID.*
chrcheck( )	Performs character checking.
close( )	Closes a specified file.
copy( )	Copies a file to a new location.
creat( )	Opens a specified file and assigns access rights to the file.
delete( )	Deletes a specified file.
fclose( ) *	Closes a specified file and flushes any associated buffer.
fdopen( )	Associates a file pointer with a specified integer *fileID.*
fdtm( )	Returns the modification time for a specified file.
fflush( ) *	Writes out buffered information to a specified file.
fgetc( ) *	Returns the next character from a specified file. This function generates an actual function call.
fgetname( )	Returns the PRIMOS pathname associated with an integer *fileID.*
fgetpos( ) *	Stores the current value of the specified stream's file position indicator.
fgets( ) *	Reads a line from a specified file. The read line is terminated by an ASCII NULL character (\0).
fileno( )	Returns an integer *fileID.*
fopen( ) *	Opens a specified file.
fprintf( ) *	Performs formatted output to a specified file.
fputc( ) *	Writes a single character to specified file.
fputs( ) *	Writes a character string to a specified file.
fread( ) *	Reads a specified number of items from a file.
freopen( ) *	Reassigns the address of a specified file and opens the file.
frwlock( )	Returns the current read/write lock for a specified file.
fscanf( ) *	Performs formatted output from a specified file.
fseek( ) *	Positions the file to a specified byte offset in the file.
fsetpos( ) *	Sets the file position indicator for the specified stream.
fsize( )	Returns the size in bytes of a specified file.

TABLE D-8. Input/Output Library Functions (continued)

Function	*Returned Value*
fstat( )	Returns the status of a previously open file.
ftell( ) *	Returns the current byte offset to the specified stream file.
ftype( )	Returns the type of a specified file.
fwrite( ) *	Writes a specified number of items to a file.
getc( ) *	Returns the next character from a specified file (implemented as a macro).
getchar( ) *	Returns the next character from the standard input device (implemented as a macro).
geth( )	Reads two characters from a specified file.
getname( )	Returns the PRIMOS pathname associated with an integer *fileID*.
gets( ) *	Reads a line from the standard input device. The newline character is replaced by the ASCII NULL character (\0).
getw( )	Reads four characters from a specified input file.
lseek( )	Positions a file to an arbitrary byte position and returns the new position as a long integer value.
move( )	Moves a specified file to a specified new location.
open( )	Opens a specified file.
printf( ) *	Performs formatted output to the standard output device.
putc( ) *	Writes a single character to a specified file (implemented as a macro).
putchar( ) *	Writes a single character to the standard output device (implemented as a macro).
puth( )	Writes two characters to an output file as a **short int**. (Type conversion does occur.)
puts( ) *	Writes a character string to the standard output device. A newline character is appended to the output.
putw( )	Writes four characters to a specified output file.
read( )	Reads bytes from a specified file and places them in a buffer.
remove( ) *	Causes a file to become inaccessible.
rename( ) *	Causes a file to be known by a new name.
rewind( ) *	Rewinds to the beginning of the file.

TABLE D-8. Input/Output Library Functions (continued)

Function	*Returned Value*
scanf( ) *	Performs formatted input from the standard input device.
seek( )	Positions a file to an arbitrary byte position.
setbuf( ) *	Associates a buffer with an input or output file.
setmod( )	Sets access rights on a specified file.
setvbuf( ) *	Determines how a stream will be buffered.
sprintf( ) *	Performs formatted output to a character string in memory.
sscanf( ) *	Performs formatted input from a specified character string in memory.
stat( )	Fills a stat structure with information about a specified file.
tell( )	Returns the current byte position in a file specified by a *fileID*.
tmpfile( ) *	Creates a temporary binary file that will be removed when closed or upon program termination.
tmpnam( ) *	Creates a character string that can be used in place of the *s* argument in function calls.
ungetc( ) *	Writes a character to a file buffer and positions the file before the character.
vfprintf( ) *	Performs formatted output of a variable argument list to a specified file.
vprintf( ) *	Performs formatted output of a variable argument list to the standard output device.
vsprintf( ) *	Performs formatted output of a variable argument list to a character string in memory.
write( )	Writes a specified number of bytes from file buffer to a file.

TABLE D-9. Random Sequence Generation Library Functions

Function	*Description*
rand( ) *	Pseudo-random numbers in the range 0 through $2^{31} - 1$.
srand( ) *	Reinitializes the random number generator.

TABLE D-10. Error Handling Library Functions

Function	*Description*
clearerr( ) *	Resets the error and end-of-file indicators for a file.
feof( ) *	Tests a specified file to determine if the end-of-file has been reached.
ferror( ) *	Returns a nonzero integer if an error condition is encountered during a read or write operation.
perror( ) *	Displays a brief message at your terminal describing the last error encountered.

TABLE D-11. String Conversion Library Functions

Function	*Conversion Performed*
atof( ) *	ASCII string to floating-point numeric value.
atoi( ) *	ASCII string to integer numeric value.
atol( ) *	ASCII string to long integer numeric value.
ecvt( )	**double** value to NULL-terminated ASCII string.
fcvt( )	**double** value to NULL-terminated ASCII string.
strtod( ) *	ASCII string to double representation.
strtol( ) *	ASCII string to long integer representation.
strtoul( ) *	ASCII string to unsigned long integer representation.
toascii( )	Character or integer to ASCII character (by ANDing the value with 0377).
tolower( ) *	Uppercase ASCII character to lowercase ASCII character.
topascii( )	Character or integer to Prime ASCII character (by ANDing the value with 0377 and then ORing the value with 0200).
toupper( ) *	Lowercase ASCII character to uppercase ASCII character.

TABLE D-12. Memory Management Library Functions

Function	*Description*
calloc( ) *	Allocates an area of memory.
cfree( )	Frees a previously allocated area of memory.
free( ) *	Frees a previously allocated area of memory.
malloc( ) *	Allocates a contiguous area of memory whose size in bytes is supplied as an argument.
realloc( ) *	Changes the size of an area of memory that was previously allocated.

TABLE D-13. Communication With Environment Library Functions

Function	*Description*
abort( ) *	Aborts program execution.
atexit( ) *	Registers a function to be called at normal program termination.
cuserid( )	Returns a pointer to a character string containing the user ID of the current process.
exit( ) *	Terminates an executing process.
g$amiix( )	Determines if the current machine is capable of executing C 32IX-mode code.
getenv( ) *	Searches the environment list for the specified global variable and returns its value.
gterm( )	Obtains current terminal characteristics.
gvget( )	Returns a pointer to a static character array that contains the value of the named PRIMOS global variable.
gvset( )	Changes the value of a PRIMOS global variable.
isatty( )	Determines if the current process is running from a terminal.
sterm( )	Sets terminal characteristics.
system( ) *	Executes its argument as a PRIMOS command line.

TABLE D-14. Searching and Sorting Utilities Library Functions

Function	*Returned Value*
bsearch( ) *	Searches an array for the given object using the supplied comparison function.
qsort( ) *	Sorts an array using the supplied comparison function.

TABLE D-15. Integer Arithmetic Library Functions

Function	*Returned Value*
abs( ) *	The absolute value of an integer.
div( ) *	The quotient and remainder of the division on integer types.
labs( ) *	The absolute value of a long integer.
ldiv( ) *	The quotient and remainder of the division on long integer types.

TABLE D-16. Multibyte Character and String Handling Library Functions

Function	*Description*
mblen( ) *	Determines the number of bytes comprising a multibyte character.
mbstowcs( ) *	Converts a sequence of multibyte characters into a sequence of corresponding codes.
mbtowc( ) *	Determines the number of bytes comprising a multibyte character and its corresponding code.
wcstombs( ) *	Converts a sequence of codes into a sequence of multibyte characters.
wctomb( ) *	Determines the number of bytes needed to represent a multibyte character and stores it.

TABLE D-17. String Handling Library Functions

Function	*Description*
index( )	Returns the address of the first occurrence of a specified character in a NULL-terminated string.
memchr( ) *	Locates the first occurrence of a character in an object.
memcmp( ) *	Compares a given number of characters in one object to another.
memcpy( ) *	Copies a specified number of characters from one object to another.
memmove( ) *	Copies a specified number of characters from one object to another.
memset( ) *	Copies a character into each element of an object.
rindex( )	Returns the address of the last occurrence (rightmost) of a specified character.
strcat( ) *	Concatenates two strings.
strchr( ) *	Returns the address of the first occurrence of a given character in a NULL-terminated string (synonymous with the index( ) library function).
strcmp( ) *	Compares two ASCII character strings.
strcoll( ) *	Compares two strings using the LC_COLLATE category of the current locale.
strcpy( ) *	Copies one argument string into another argument string.
strcspn( ) *	Searches a string for a character in a specified set of characters.
strerror( ) *	Maps an error code to a message that is displayed at your terminal.
strlen( ) *	Returns the length of a string of ASCII characters. The returned length does not include the terminating NULL character (\0).
strncat( ) *	Concatenates two strings up through a maximum number of characters.
strncmp( ) *	Compares two ASCII strings up through a maximum number of characters.
strncpy( ) *	Copies a maximum number of characters from one string to another string.
strpbrk( ) *	Searches a string for a specified set of characters.

TABLE D-17. String Handling Library Functions (continued)

Function	*Returned Value*
strrchr( ) *	Returns the rightmost position of a specified character in a string of characters (synonymous with the rindex( ) library function).
strspn( ) *	Searches for a character that is not in a specified set of characters.
strstr( ) *	Locates the first occurrence of a sequence of characters in a string.
strtok( ) *	Breaks a string into a sequence of tokens.
strxfrm( ) *	Transforms a string.

TABLE D-18. Date and Time Library Functions

Function	*Description*
asctime( ) *	Converts local time into a string.
clock( ) *	Returns an estimate in seconds of processor time used.
ctime( ) *	Converts a numerical time to a ASCII string.
difftime( ) *	Returns the difference in seconds between two times.
ftime( )	Returns the elapsed time in a timeb structure.
gmtime( ) *	Converts time in terms of seconds into Coordinated Universal Time.
localtime( ) *	Converts a time to a time structure.
mktime( ) *	Converts local time into time in terms of seconds.
strftime( ) *	Performs formatted output of time/date information to a character string in memory.
time( ) *	Returns in seconds the time elapsed since 00:00:00, Jan. 1, 1970.
timer( )	Raises the PRIMOS ALARM$ condition after a specified number of minutes have elapsed.

TABLE D-19. Miscellaneous Library Functions

Function	*Description*
access( )	Checks for access rights.
chdir( )	Changes the current working directory.
fexists( )	Checks for the existence of a specified PRIMOS pathname.
getmod( )	Returns the access rights of a specified file.
lsdir( )	Returns a pointer to a static character array containing the next filename in an open directory.
mkdir( )	Creates a specified directory.
primospath( )	Converts a UNIX operating system pathname to a PRIMOS pathname.
sleep( )	Suspends the execution of the current process.
stat( )	Fills a stat structure with information concerning a specified file.

E

C DATA FORMATS

The data formats in this appendix are used by PRIMOS C. For comparisons with other compilers and CPUs, please consult the appendix entitled C Reference Manual in the Kernighan and Ritchie text.

In the PRIMOS implementation of C, **int** is the same as **long int**.

DATA FORMATS

Although the character boundary on a 50 Series machine is a 16-bit halfword, arrays of characters are packed with two characters per halfword. Each member of a structure or union (except the bit fields) starts on a 16-bit boundary. Figure E-1 shows the C data formats.

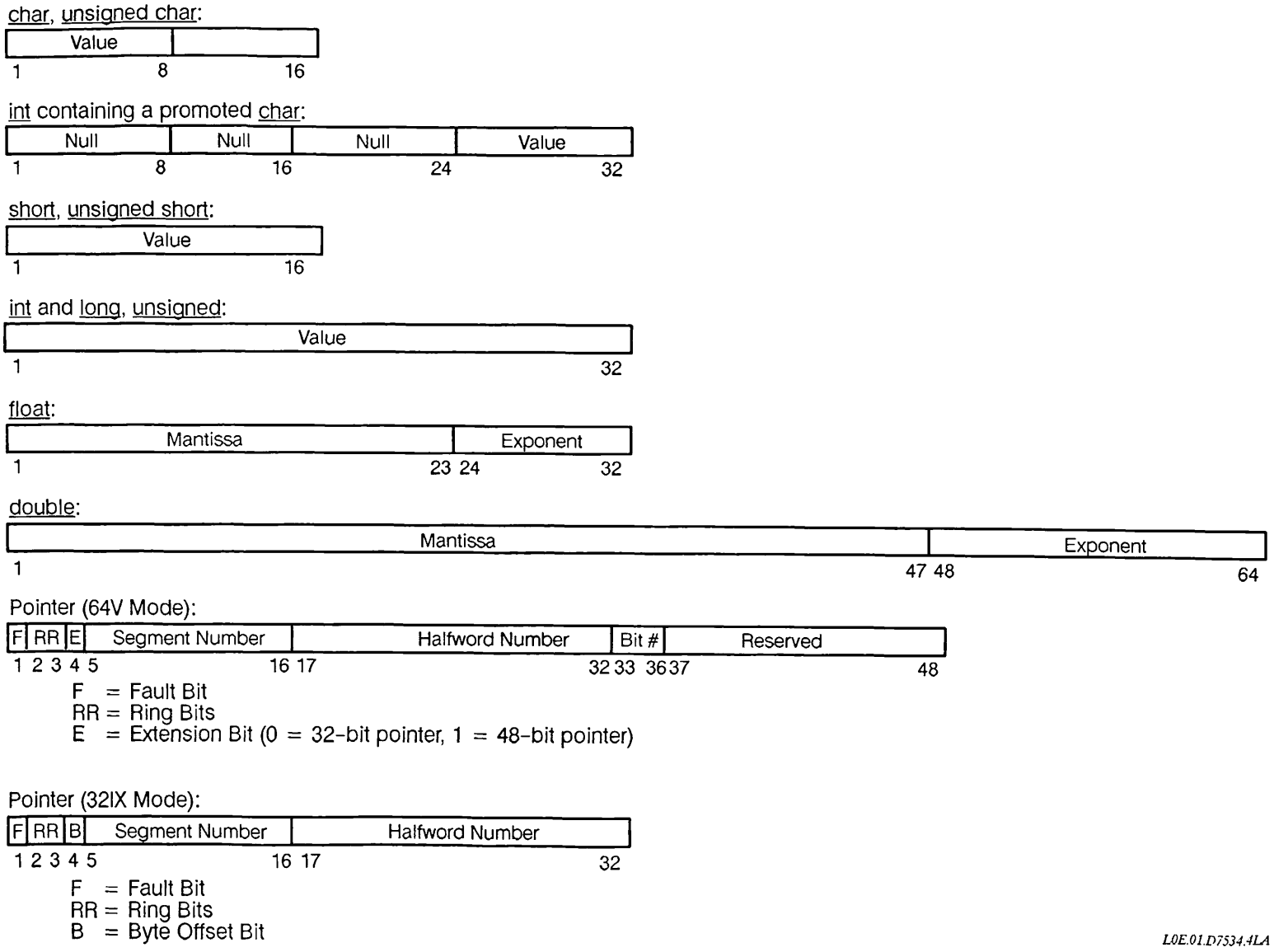


FIGURE E-1. C Data Formats

F

THE PRIME EXTENDED CHARACTER SET

As of Rev. 21.0, Prime expanded its character set. The basic character set, known as Prime ASCII, remains the same as it was before Rev. 21.0; it is the ANSI ASCII 7-bit set (called ASCII-7), with the 8th bit turned on. However, the 8th bit is now significant; when it is turned off, it signifies a different character. Thus, the size of the character set has doubled, from 128 to 256 characters. This expanded character set is called the Prime Extended Character Set (Prime ECS).

Prime ASCII, the pre-Rev. 21.0 character set, is a proper subset of Prime ECS. These characters have not changed. Software written before Rev. 21.0 continues to run exactly as it did before. Software written at Rev. 21.0 that does not use the new characters needs no special coding to use the old ones.

Prime ECS support is automatic at Rev. 21.0. You may begin to use characters that have the 8th bit turned off. However, the extra characters are not available on most printers and terminals. Check with your System Administrator to find out whether you can take advantage of the new characters in Prime ECS.

Table F-1 shows the Prime Extended Character Set. The pre-Rev. 21.0 character set consists of the characters with decimal values 128 through 255 (octal values 200 through 377). The characters added at Rev. 21.0 all have decimal values less than 128 (octal values less than 200).

SPECIFYING PRIME ECS CHARACTERS

Direct Entry

On terminals that support Prime ECS, you can enter the printing characters directly; the characters appear on the screen as you type them. For information on how to do this, see the appropriate manual for your terminal.
A terminal supports Prime ECS if

- It uses ASCII-8 as its internal character set.
- The TTY8 protocol is configured on your asynchronous line.

If you do not know whether your terminal supports Prime ECS, ask your System Administrator.

On terminals that do not support Prime ECS, you can enter any of the ASCII-7 printing characters (characters with a decimal value of 160 or higher) directly by just typing them.

Octal Notation

If you use the Editor (ED), you can enter any Prime ECS character on any terminal by typing a caret (^), followed by the octal value of the character, as given in Table F-1. You must type all three digits, including leading zeroes.

Before you use this method to enter any of the ECS characters that have decimal values between 32 and 127, first specify the following ED command:

```
MODE CKPAR
```

This command permits ED to print as ^*nnn* any characters that have a first bit of 0.

SPECIAL MEANINGS OF PRIME ECS CHARACTERS

PRIMOS, or an applications program running on PRIMOS, may interpret some Prime ECS characters in a special way. For example, PRIMOS interprets ^P as a process interrupt. ED, the Editor, interprets the backslash (\) as a logical tab.

For a detailed description of how PRIMOS interprets the following Prime ECS characters, see the discussion in the *PRIMOS User's Guide* of special terminal keys and special characters:

^ \ " ? ^P ^S ^Q _ ;

C PROGRAMMING CONSIDERATIONS

At Rev 21.0, Prime ECS support was added to C through the use of the include file PRIME_ECS_CHARS.H.INS.CC, which is in the top-level directory SYSCOM on your system.

This file contains a number of **#define** statements of the form

```
#define BEL_CHAR     '\207'
#define BEL_STR      "\207"
```

Each character in the Prime character set is defined by its octal value, both as a character constant and as a string constant. The above example shows the definition for the character BEL_CHAR and for the string BEL_STR. In Prime ECS, both of these have the value 207 octal.

You may use these characters and strings in a program, as shown in the following example.

```
#include <stdio.h>
#include <prime_ecs_chars.h>
#include <string.h>
main( )
{
    char message[100], ch;

    message[0] = (char)NULL;        /* Clean out message[] string */
    strcat(message, "ATTENTION!\n");
    strcat(message, BEL_STR);       /* Use of ECS symbol */
    puts(message);
    .
    .
    .
    ch = BEL_CHAR;                  /* Use of ECS symbol */
    puts("ATTENTION!\n");
    putchar(ch);
    .
    .
    .
    puts("ATTENTION!"BEL_STR"\n");  /* Use of auto-concatenation */
}
```

Notes

All of the preprocessor symbols supplied in PRIME_ECS_CHARS.H.INS.CC are in uppercase.

PRIMOS C automatically concatenates two string literals. That is, the string

```
"Hello ""there.\n"
```

is functionally equivalent to the string

```
"Hello there.\n"
```

Automatic string concatenation is part of Prime ECS and is also part of the ANSI C standard. It is an extension to the 1978 C language.

PRIME EXTENDED CHARACTER SET TABLE

Table F-1 contains all of the Prime ECS characters, arranged in ascending order. This order provides both the collating sequence and the way that comparisons are done for character strings.

For each character, the table includes the graphic, the mnemonic, the description, and the binary, decimal, hexadecimal, and octal values. The preprocessor symbols defined in PRIME_ECS_CHARS.H.INS.CC consist of the mnemonic plus _CHAR and _STR. For example, the mnemonic BEL has the two definitions BEL_CHAR and BEL_STR. A blank entry indicates that the particular item does not apply to this character. The graphics for control characters are specified as ^*character*; for example, ^P represents the character produced when you type P while holding the control key down.

Characters with decimal values from 000 to 031 and from 128 to 159 are control characters.

Characters with decimal values from 032 to 127 and from 160 to 255 are printing characters.

TABLE F-1. The Prime Extended Character Set

[ISO 8859-1 Latin Alphabet 1]

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
	RES1	Reserved for future standardization	0000 0000	000	00	000
	RES2	Reserved for future standardization	0000 0001	001	01	001
	RES3	Reserved for future standardization	0000 0010	002	02	002
	RES4	Reserved for future standardization	0000 0011	003	03	003
	IND	Index	0000 0100	004	04	004
	NEL	Next line	0000 0101	005	05	005
	SSA	Start of selected area	0000 0110	006	06	006
	ESA	End of selected area	0000 0111	007	07	007
	HTS	Horizontal tabulation set	0000 1000	008	08	010
	HTJ	Horizontal tab with justify	0000 1001	009	09	011
	VTS	Vertical tabulation set	0000 1010	010	0A	012
	PLD	Partial line down	0000 1011	011	0B	013
	PLU	Partial line up	0000 1100	012	0C	014
	RI	Reverse index	0000 1101	013	0D	015
	SS2	Single shift 2	0000 1110	014	0E	016
	SS3	Single shift 3	0000 1111	015	0F	017
	DCS	Device control string	0001 0000	016	10	020
	PU1	Private use 1	0001 0001	017	11	021
	PU2	Private use 2	0001 0010	018	12	022
	STS	Set transmission state	0001 0011	019	13	023
	CCH	Cancel character	0001 0100	020	14	024
	MW	Message waiting	0001 0101	021	15	025
	SPA	Start of protected area	0001 0110	022	16	026
	EPA	End of protected area	0001 0111	023	17	027
	RES5	Reserved for future standardization	0001 1000	024	18	030
	RES6	Reserved for future standardization	0001 1001	025	19	031
	RES7	Reserved for future standardization	0001 1010	026	1A	032
	CSI	Control sequence introducer	0001 1011	027	1B	033
	ST	String terminator	0001 1100	028	1C	034
	OSC	Operating system command	0001 1101	029	1D	035
	PM	Privacy message	0001 1110	030	1E	036

TABLE F-1. The Prime Extended Character Set (Continued)

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
	APC	Application program command	0001 1111	031	1F	037
	NBSP	No-break space	0010 0000	032	20	040
¡	INVE	Inverted exclamation mark	0010 0001	033	21	041
¢	CENT	Cent sign	0010 0010	034	22	042
£	PND	Pound sign	0010 0011	035	23	043
¤	CURR	Currency sign	0010 0100	036	24	044
¥	YEN	Yen sign	0010 0101	037	25	045
¦	BBAR	Broken bar	0010 0110	038	26	046
§	SECT	Section sign	0010 0111	039	27	047
¨	DIA	Diaeresis, umlaut	0010 1000	040	28	050
©	COPY	Copyright sign	0010 1001	041	29	051
ª	FOI	Feminine ordinal indicator	0010 1010	042	2A	052
«	LAQM	Left angle quotation mark	0010 1011	043	2B	053
¬	NOT	Not sign	0010 1100	044	2C	054
	SHY	Soft hyphen	0010 1101	045	2D	055
®	TM	Registered trademark sign	0010 1110	046	2E	056
¯	MACN	Macron	0010 1111	047	2F	057
°	DEGR	Degree sign	0011 0000	048	30	060
±	PLMI	Plus/minus sign	0011 0001	049	31	061
²	SPS2	Superscript two	0011 0010	050	32	062
³	SPS3	Superscript three	0011 0011	051	33	063
´	AAC	Acute accent	0011 0100	052	34	064
µ	LCMU	Lowercase Greek letter µ, micro sign	0011 0101	053	35	065
¶	PARA	Paragraph sign, Pilgrow sign	0011 0110	054	36	066
·	MIDD	Middle dot	0011 0111	055	37	067
¸	CED	Cedilla	0011 1000	056	38	070
¹	SPS1	Superscript one	0011 1001	057	39	071
º	MOI	Masculine ordinal indicator	0011 1010	058	3A	072
»	RAQM	Right angle quotation mark	0011 1011	059	3B	073
¼	FR14	Common fraction one-quarter	0011 1100	060	3C	074

TABLE F-1. The Prime Extended Character Set (Continued)

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
½	FR12	Common fraction one-half	0011 1101	061	3D	075
¾	FR34	Common fraction three-quarters	0011 1110	062	3E	076
¿	INVQ	Inverted question mark	0011 1111	063	3F	077
À	UCAG	Uppercase A with grave accent	0100 0000	064	40	100
Á	UCAA	Uppercase A with acute accent	0100 0001	065	41	101
Â	UCAC	Uppercase A with circumflex	0100 0010	066	42	102
Ã	UCAT	Uppercase A with tilde	0100 0011	067	43	103
Ä	UCAD	Uppercase A with diaeresis	0100 0100	068	44	104
Å	UCAR	Uppercase A with ring above	0100 0101	069	45	105
Æ	UCAE	Uppercase diphthong Æ	0100 0110	070	46	106
Ç	UCCC	Uppercase C with cedilla	0100 0111	071	47	107
È	UCEG	Uppercase E with grave accent	0100 1000	072	48	110
É	UCEA	Uppercase E with acute accent	0100 1001	073	49	111
Ê	UCEC	Uppercase E with circumflex	0100 1010	074	4A	112
Ë	UCED	Uppercase E with diaeresis	0100 1011	075	4B	113
Ì	UCIG	Uppercase I with grave accent	0100 1100	076	4C	114
Í	UCIA	Uppercase I with acute accent	0100 1101	077	4D	115
Î	UCIC	Uppercase I with circumflex	0100 1110	078	4E	116
Ï	UCID	Uppercase I with diaeresis	0100 1111	079	4F	117
Ð	UETH	Uppercase Icelandic letter Eth	0101 0000	080	50	120
Ñ	UCNT	Uppercase N with tilde	0101 0001	081	51	121
Ò	UCOG	Uppercase O with grave accent	0101 0010	082	52	122
Ó	UCOA	Uppercase O with acute accent	0101 0011	083	53	123

TABLE F-1. The Prime Extended Character Set (Continued)

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
Ô	UCOC	Uppercase O with circumflex	0101 0100	084	54	124
Õ	UCOT	Uppercase O with tilde	0101 0101	085	55	125
Ö	UCOD	Uppercase O with diaeresis	0101 0110	086	56	126
×	MULT	Multiplication sign used in mathematics	0101 0111	087	57	127
Ø	UCOO	Uppercase O with oblique line	0101 1000	088	58	130
Ù	UCUG	Uppercase U with grave accent	0101 1001	089	59	131
Ú	UCUA	Uppercase U with acute accent	0101 1010	090	5A	132
Û	UCUC	Uppercase U with circumflex	0101 1011	091	5B	133
Ü	UCUD	Uppercase U with diaeresis	0101 1100	092	5C	134
Ý	UCYA	Uppercase Y with acute accent	0101 1101	093	5D	135
Þ	UTHN	Uppercase Icelandic letter Thorn	0101 1110	094	5E	136
ß	LGSS	Lowercase German letter double s	0101 1111	095	5F	137
à	LCAG	Lowercase a with grave accent	0110 0000	096	60	140
á	LCAA	Lowercase a with acute accent	0110 0001	097	61	141
â	LCAC	Lowercase a with circumflex	0110 0010	098	62	142
ã	LCAT	Lowercase a with tilde	0110 0011	099	63	143
ä	LCAD	Lowercase a with diaeresis	0110 0100	100	64	144
å	LCAR	Lowercase a with ring above	0110 0101	101	65	145
æ	LCAE	Lowercase diphthong ae	0110 0110	102	66	146
ç	LCCC	Lowercase c with cedilla	0110 0111	103	67	147
e	LCEG	Lowercase e with grave accent	0110 1000	104	68	150
é	LCEA	Lowercase e with acute accent	0110 1001	105	69	151
ê	LCEC	Lowercase e with circumflex	0110 1010	106	6A	152

TABLE F-1. The Prime Extended Character Set (Continued)

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
ë	LCED	Lowercase e with diaeresis	0110 1011	107	6B	153
ì	LCIG	Lowercase i with grave accent	0110 1100	108	6C	154
í	LCIA	Lowercase i with acute accent	0110 1101	109	6D	155
î	LCIC	Lowercase i with circumflex	0110 1110	110	6E	156
ï	LCID	Lowercase i with diaeresis	0110 1111	111	6F	157
ð	LETH	Lowercase Icelandic letter Eth	0111 0000	112	70	160
ñ	LCNT	Lowercase n with tilde	0111 0001	113	71	161
ò	LCOG	Lowercase o with grave accent	0111 0010	114	72	162
ó	LCOA	Lowercase o with acute accent	0111 0011	115	73	163
ô	LCOC	Lowercase o with circumflex	0111 0100	116	74	164
õ	LCOT	Lowercase o with tilde	0111 0101	117	75	165
ö	LCOD	Lowercase o with diaeresis	0111 0110	118	76	166
÷	DIV	Division sign used in mathematics	0111 0111	119	77	167
ø	LCOO	Lowercase o with oblique line	0111 1000	120	78	170
ù	LCUG	Lowercase u with grave accent	0111 1001	121	79	171
ú	LCUA	Lowercase u with acute accent	0111 1010	122	7A	172
û	LCUC	Lowercase u with circumflex	0111 1011	123	7B	173
ü	LCUD	Lowercase u with diaeresis	0111 1100	124	7C	174
ý	LCYA	Lowercase y with acute accent	0111 1101	125	7D	175
þ	LTHN	Lowercase Icelandic letter Thorn	0111 1110	126	7E	176
ÿ	LCYD	Lowercase y with diaeresis	0111 1111	127	7F	177

TABLE F-1. The Prime Extended Character Set (Continued)

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
	NUL	Null	1000 0000	128	80	200
^A	SOH/TC1	Start of heading	1000 0001	129	81	201
^B	STX/TC2	Start of text	1000 0010	130	82	202
^C	ETX/TC3	End of text	1000 0011	131	83	203
^D	EOT/TC4	End of transmission	1000 0100	132	84	204
^E	ENQ/TC5	Enquiry	1000 0101	133	85	205
^F	ACK/TC6	Acknowledge	1000 0110	134	86	206
^G	BEL	Bell	1000 0111	135	87	207
^H	BS/FE0	Backspace	1000 1000	136	88	210
^I	HT/FE1	Horizontal tab	1000 1001	137	89	211
^J	LF/NL/FE2	Line feed	1000 1010	138	8A	212
^K	VT/FE3	Vertical tab	1000 1011	139	8B	213
^L	FF/FE4	Form feed	1000 1100	140	8C	214
^M	CR/FE5	Carriage return	1000 1101	141	8D	215
^N	SO/LS1	Shift out	1000 1110	142	8E	216
^O	SI/LS0	Shift in	1000 1111	143	8F	217
^P	DLE/TC7	Data link escape	1001 0000	144	90	220
^Q	DC1/XON	Device control 1	1001 0001	145	91	221
^R	DC2	Device control 2	1001 0010	146	92	222
^S	DC3/XOFF	Device control 3	1001 0011	147	93	223
^T	DC4	Device control 4	1001 0100	148	94	224
^U	NAK/TC8	Negative acknowledge	1001 0101	149	95	225
^V	SYN/TC9	Synchronous idle	1001 0110	150	96	226
^W	ETB/TC10	End of transmission block	1001 0111	151	97	227
^X	CAN	Cancel	1001 1000	152	98	230
^Y	EM	End of medium	1001 1001	153	99	231
^Z	SUB	Substitute	1001 1010	154	9A	232
^[	ESC	Escape	1001 1011	155	9B	233
^\	FS/IS4	File separator	1001 1100	156	9C	234
^]	GS/IS3	Group separator	1001 1101	157	9D	235
^^	RS/IS2	Record separator	1001 1110	158	9E	236
^_	US/IS1	Unit separator	1001 1111	159	9F	237
	SP	Space	1010 0000	160	A0	240
!		Exclamation mark	1010 0001	161	A1	241
"		Quotation mark	1010 0010	162	A2	242
#	NUMB	Number sign	1010 0011	163	A3	243
$	DOLR	Dollar sign	1010 0100	164	A4	244
%		Percent sign	1010 0101	165	A5	245
&		Ampersand	1010 0110	166	A6	246

TABLE F-1. The Prime Extended Character Set (Continued)

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
'		Apostrophe	1010 0111	167	A7	247
(		Left parenthesis	1010 1000	168	A8	250
)		Right parenthesis	1010 1001	169	A9	251
*		Asterisk	1010 1010	170	AA	252
+		Plus sign	1010 1011	171	AB	253
,		Comma	1010 1100	172	AC	254
–		Minus sign	1010 1101	173	AD	255
.		Period	1010 1110	174	AE	256
/		Slash	1010 1111	175	AF	257
0		Zero	1011 0000	176	B0	260
1		One	1011 0001	177	B1	261
2		Two	1011 0010	178	B2	262
3		Three	1011 0011	179	B3	263
4		Four	1011 0100	180	B4	264
5		Five	1011 0101	181	B5	265
6		Six	1011 0110	182	B6	266
7		Seven	1011 0111	183	B7	267
8		Eight	1011 1000	184	B8	270
9		Nine	1011 1001	185	B9	271
:		Colon	1011 1010	186	BA	272
;		Semicolon	1011 1011	187	BB	273
<		Less than sign	1011 1100	188	BC	274
=		Equal sign	1011 1101	189	BD	275
>		Greater than sign	1011 1110	190	BE	276
?		Question mark	1011 1111	191	BF	277
@	AT	Commercial at sign	1100 0000	192	C0	300
A		Uppercase A	1100 0001	193	C1	301
B		Uppercase B	1100 0010	194	C2	302
C		Uppercase C	1100 0011	195	C3	303
D		Uppercase D	1100 0100	196	C4	304
E		Uppercase E	1100 0101	197	C5	305
F		Uppercase F	1100 0110	198	C6	306
G		Uppercase G	1100 0111	199	C7	307
H		Uppercase H	1100 1000	200	C8	310
I		Uppercase I	1100 1001	201	C9	311
J		Uppercase J	1100 1010	202	CA	312
K		Uppercase K	1100 1011	203	CB	313
L		Uppercase L	1100 1100	204	CC	314
M		Uppercase M	1100 1101	205	CD	315
N		Uppercase N	1100 1110	206	CE	316

TABLE F-1. The Prime Extended Character Set (Continued)

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
O		Uppercase O	1100 1111	207	CF	317
P		Uppercase P	1101 0000	208	D0	320
Q		Uppercase Q	1101 0001	209	D1	321
R		Uppercase R	1101 0010	210	D2	322
S		Uppercase S	1101 0011	211	D3	323
T		Uppercase T	1101 0100	212	D4	324
U		Uppercase U	1101 0101	213	D5	325
V		Uppercase V	1101 0110	214	D6	326
W		Uppercase W	1101 0111	215	D7	327
X		Uppercase X	1101 1000	216	D8	330
Y		Uppercase Y	1101 1001	217	D9	331
Z		Uppercase Z	1101 1010	218	DA	332
[	LBKT	Left bracket	1101 1011	219	DB	333
\	REVS	Reverse slash, backslash	1101 1100	220	DC	334
]	RBKT	Right bracket	1101 1101	221	DD	335
^	CFLX	Circumflex	1101 1110	222	DE	336
_		Underline, underscore	1101 1111	223	DF	337
`	GRAV	Left single quote, grave accent	1110 0000	224	E0	340
a		Lowercase a	1110 0001	225	E1	341
b		Lowercase b	1110 0010	226	E2	342
c		Lowercase c	1110 0011	227	E3	343
d		Lowercase d	1110 0100	228	E4	344
e		Lowercase e	1110 0101	229	E5	345
f		Lowercase f	1110 0110	230	E6	346
g		Lowercase g	1110 0111	231	E7	347
h		Lowercase h	1110 1000	232	E8	350
i		Lowercase i	1110 1001	233	E9	351
j		Lowercase j	1110 1010	234	EA	352
k		Lowercase k	1110 1011	235	EB	353
l		Lowercase l	1110 1100	236	EC	354
m		Lowercase m	1110 1101	237	ED	355
n		Lowercase n	1110 1110	238	EE	356
o		Lowercase o	1110 1111	239	EF	357
p		Lowercase p	1111 0000	240	F0	360
q		Lowercase q	1111 0001	241	F1	361
r		Lowercase r	1111 0010	242	F2	362
s		Lowercase s	1111 0011	243	F3	363
t		Lowercase t	1111 0100	244	F4	364

TABLE F-1. The Prime Extended Character Set (Continued)

Graphic	Mnemonic	Description	Binary	Decimal	Hex	Octal
u		Lowercase u	1111 0101	245	F5	365
v		Lowercase v	1111 0110	246	F6	366
w		Lowercase w	1111 0111	247	F7	367
x		Lowercase x	1111 1000	248	F8	370
y		Lowercase y	1111 1001	249	F9	371
z		Lowercase z	1111 1010	250	FA	372
{	LBCE	Left brace	1111 1011	251	FB	373
|	VERT	Vertical line	1111 1100	252	FC	374
}	RBCE	Right brace	1111 1101	253	FD	375
~	TIL	Tilde	1111 1110	254	FE	376
	DEL	Delete	1111 1111	255	FF	377

G

GLOSSARY

This appendix is a glossary of PRIMOS concepts and conventions.

64V mode
A segmented accumulator-based instruction set, standard on all 50 Series machines.

32IX mode
A segmented, general-register-based instruction set, available on all newer 50 Series processors (that is, all processors numbered 2350™ or above).

Access Control List (ACL)
A list of users and their respective access rights to a file, a directory, or another object. The *PRIMOS User's Guide* provides a detailed explanation of ACLs.

condition mechanism
A hardware feature of 50 Series machines. The condition mechanism directs a condition incapable of being handled by the software (for example, division by zero, or use of the BREAK signal) to a set of routines. These user-written routines, known as on-units, treat the condition differently from the way it is handled by the default routine, which normally aborts the process and returns you to PRIMOS command level. The condition mechanism and subroutines used to invoke it are described in the *Subroutines Reference III: Operating System.*

file system error code
A value returned by a file system subroutine, as described in Appendix D of the *Subroutines Reference II: File System.* For example, an error code of 1 always signals an end of file, and an error code of 15 indicates that a specified file does not exist. Many PRIMOS C routines use the same set of error codes as a convenience to programmers.

IX mode
See 32IX mode.

on-unit
A routine set up to handle certain conditions that might normally cause a process to abort. *See* condition mechanism.

pass by reference
To pass the address, rather than the value, of an argument from one routine to another. The called routine may then modify the actual data item that is to be referenced by the caller.

pass by value
To pass the value of an argument from one routine to another routine when a copy of the argument is made and passed. The called routine cannot modify the original data item.

pathname
A fully qualified PRIMOS pathname consists of a disk partition name, followed by the names of the top-level directory and subdirectories, followed by a filename. For example,

```
<PART>DIR>DIR>DIR>FILE
```

PRIMOS
The PRIMOS operating system.

standard error (stderr)
The standard error file for C. By default, this is your terminal.

standard input (stdin)
The standard input file for C. By default, this is your terminal.

standard output (stdout)
The standard output file for C. By default, this is your terminal.

V mode
See 64V mode.

INDEX

INDEX

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

SURVEYS

READER RESPONSE FORM

C User's Guide
DOC7534-4LA

Your feedback will help us continue to improve the quality, accuracy, and organization of our publications.

1. How do you rate this document for overall usefulness?

   ☐ *excellent* ☐ *very good* ☐ *good* ☐ *fair* ☐ *poor*

2. What features of this manual did you find most useful?

   _____
   _____
   _____
   _____
   _____
   _____

3. What faults or errors in this manual gave you problems?

   _____
   _____
   _____
   _____
   _____
   _____

4. How does this manual compare to equivalent manuals produced by other computer companies?

   ☐ *Much better* ☐ *Slightly better* ☐ *About the same*
   ☐ *Much worse* ☐ *Slightly worse* ☐ *Can't judge*

5. Which other companies' manuals have you read?

   _____
   _____

Name:_____ Position:_____

Company:_____

Address:_____

_____

_____ Postal Code:_____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

Postage will be paid by:

Prime™

Attention: Technical Publications
Bldg 10
Prime Park, Natick, Ma. 01760

READER RESPONSE FORM

C User's Guide
DOC7534-4LA

Your feedback will help us continue to improve the quality, accuracy, and organization of our publications.

1. How do you rate this document for overall usefulness?

   ☐ *excellent* ☐ *very good* ☐ *good* ☐ *fair* ☐ *poor*

2. What features of this manual did you find most useful?

   _____
   _____
   _____
   _____
   _____
   _____

3. What faults or errors in this manual gave you problems?

   _____
   _____
   _____
   _____
   _____
   _____

4. How does this manual compare to equivalent manuals produced by other computer companies?

   ☐ *Much better* ☐ *Slightly better* ☐ *About the same*
   ☐ *Much worse* ☐ *Slightly worse* ☐ *Can't judge*

5. Which other companies' manuals have you read?

   _____
   _____

Name:_____ Position:_____

Company:_____

Address:_____

_____

_____ Postal Code:_____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

First Class Permit #531 Natick, Massachusetts 01760

BUSINESS REPLY MAIL

Postage will be paid by:

Prime™

Attention: Technical Publications
Bldg 10
Prime Park, Natick, Ma. 01760