Cortland Workshop C

Language Reference

Alpha Draft: May 26, 1986

Writer: Don Reed
Technical Publications, MS 22-K

Engineering Part Number: 030-3133
Marketing Part Number: A2L6003
Finance Number: PAP002-21

Cortland Workshop C

Language Reference

Contents

About This Manual

This manual contains the information about Cortland Workshop™ C that you need when writing C programs for the Cortland™. It assumes that most readers already know the C programming language, as defined in Kernighan and Ritchie's *The C Programming Language*. For this reason, it does not repeat their definition of the C language, but defines the differences between Cortland C and "K and R" C. However, this manual can also be used by those learning C for the first time. The introductory chapters tell how to write, compile, link, and run a simple C program. From there, one can follow K and R or any standard textbook on C.

The Cortland road map

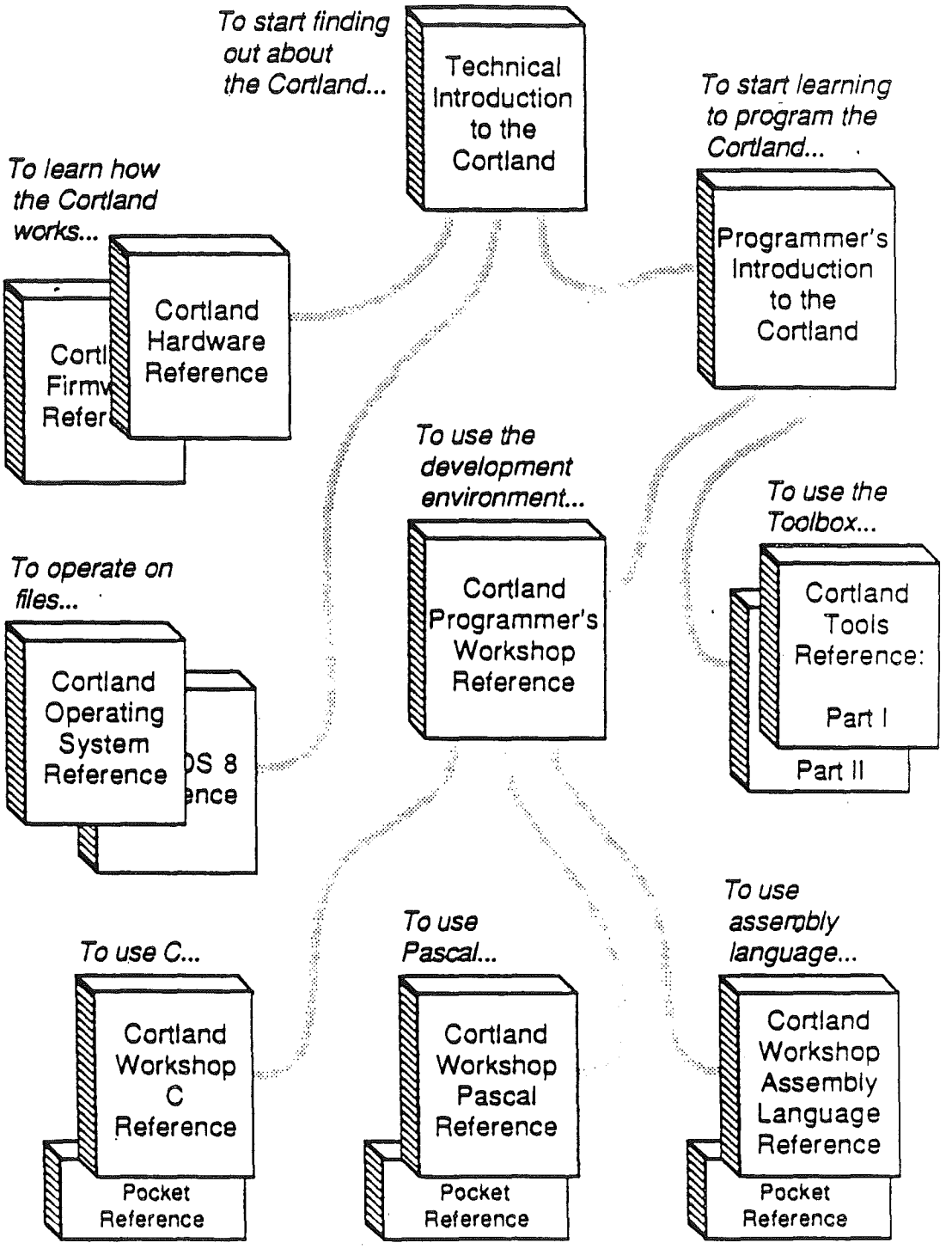
The Cortland has many advanced features, making it more complex than earlier models of the Apple II. To describe it fully, Apple has produced a whole suite of technical manuals. The manuals are listed in Table A-1. Figure A-1 is a diagram showing the relationships among the different manuals. Depending on the way you intend to use the Cortland, you may need to refer to a select few of the manuals, or you may need to refer to most of them.

Table A-1. The Cortland Technical Manuals

Title	Subject
Technical Introduction to the Cortland	what the Cortland is
Cortland Hardware Reference	machine internals—hardware
Cortland Firmware Reference	machine internals—firmware
Programmer's Introduction to the Cortland	sample program using the toolbox
Cortland Tools Reference: Part I	toolbox specifications
Cortland Tools Reference: Part II	more toolbox specifications
Cortland Function Summary	toolbox pocket guide
Cortland Programmer's Workshop	the development environment
Cortland Workshop Assembly Language Reference*	using assembly language
Cortland Workshop C Reference*	using C on the Cortland
Cortland Workshop Pascal Reference*	using Pascal on the Cortland
ProDOS/8 Technical Reference	ProDOS for Apple II programs
Cortland Operating System Reference	ProDOS and loader for Cortland
Human Interface Guidelines	for all Apple computers
Apple Numerics Manual	numerics for all Apple computers

*There is a Pocket Reference for each of these.

Figure A-1. Roadmap to the technical manuals



The Technical Introduction

The *Technical Introduction to the Cortland is an overview:* it tells a little about a lot of things, but it doesn't tell everything about anything. To find out all about any one aspect of the Cortland, you should read a specific technical manual. To find out which one, read on.

The Machine Reference Manuals

The *Cortland Hardware Reference* and the *Cortland Firmware Reference* contain information about the machine itself. You don't need to read these manuals to be able to develop applications for the Cortland, but they will give you a better understanding of the machine's features. They will also provide the reasons why some of those features work the way they do.

The Toolbox Manuals

Like the Macintosh, the Cortland has a built-in toolbox that can be called by applications. The toolbox serves two purposes: it makes developing new applications easier, and it supports the desktop user interface.

When you first start using the toolbox, the *Introduction for Programmers* provides the recommendations and guidelines you need. It is not a complete course in programming for the Cortland; rather, it is a starting point. It explains the Cortland tools and describes an event-driven program. It includes a simple example of such a program that uses the Cortland tools, and demonstrates the way you use the Cortland Programmer's Workshop to develop the program.

For detailed specifications of the tool calls, you'll need the two volumes making up the *Cortland Tools Reference*. The *Cortland Function Summary* is a pocket guide to the tools, including the name and parameters for each tool call.

The Cortland Programming Languages

The Cortland does not restrict developers to a single programming language. Apple is currently providing an assembler and compilers for C and Pascal. Other compilers can be used with the workshop, provided that they observe the standards Apple has set up.

There is a separate reference manual for each programming language on the Cortland. The manuals for the languages Apple provides are the *Cortland Assembler Reference*, the *Cortland C Compiler Reference*, and the *Cortland Pascal Compiler Reference*.

The Programmer's Workshop Manual

The core of the development environment on the Cortland is the Cortland Programmer's Workshop, also called CPW. CPW is a set of programs that enable developers to create and debug application programs on the Cortland. The manual that describes CPW is the *Cortland Programmer's Workshop* manual. It includes information about the parts of the

workshop that all developers will use, regardless which programming language they use: the shell, the editor, the linker, the debugger, and the utilities.

What About ProDOS?

ProDOS on the Cortland comes in two flavors: one for compatibility with the models of Apple II that use 8-bit CPUs, called ProDOS/8, and one that utilizes the full power of the Cortland, ProDOS/16. Those two versions of ProDOS are described in their own manuals, *ProDOS/8 Technical Reference* and *ProDOS/16 Technical Reference*

All-Apple Manuals

In addition to the Cortland manuals mentioned above, there are two manuals that apply to all Apple computers. Those are *Human Interface Guidelines* and *Apple Numerics Manual.*

How to use this book

If you are an experienced C programmer, Chapters 1 and 2 will give you enough information to get standard C programs running. (If you have used other Cortland programs, Chapter 1 will be redundant.) The remaining chapters tell you what you need to write C programs that use the capabilities of Cortland.

If you are new to C, Chapter 1 will tell you what you need to go through a C textbook like Kernighan and Ritchie. After that, you can learn about the capabilities of the compiler and this particular implementation.

What this manual contains

This manual contains the following chapters:

- About this Manual tells you about the manual.
- Chapter 1, Getting Started, describes Cortland Programmers Workshop C and takes you through the steps of writing, compiling, linking, and running a sample program.
- Chapter 2, Using the C Compiler, describes the compiler, lists the compiler options, and tells you which library files to compile and link with.
- Chapter 3, The Cortland Workshop C Language, describes Apple extensions to C and clarifies aspects of the language definition as they apply to this implementation.
- Chapter 4, The Standard C Library, documents functions for standard I/O, string manipulation, math routines, and other useful features not built into the language.
- Chapter 5, The Cortland Interface Libraries, lists the C interfaces to the Cortland ROM and other Cortland tool routines.
- Appendix A, Calling Conventions, tells how to write calls between C and Pascal.
- Appendix B, Files Supplied with Cortland Workshop C, contains a list of all the files that are supplied with this product.

- Appendix C, Library Index, is a combined index of identifiers in the Standard C Library and Cortland Interface Libraries.

Visual Cues

*** Boilerplate on warnings, gray boxes, etc., to be added when available. This has not yet been written for the Grand Design. ***

Other reference material you'll need

You'll need to be familiar with these additional reference materials:

- *Cortland Programmer's Workshop*, Apple Computer Inc. This book describes the CPW environment in which the C compiler operates, including the editor, linker, debugger, and other important tools.
- *The C Programming Language*, Kernighan and Ritchie, Prentice-Hall, 1978. This is a standard reference book for the C language. C is formally defined in Appendix A.
- *Cortland Tools*, Apple Computer Inc. This book contains everything you need to program using the Cortland ROM and associated RAM routines; it covers windows, alert boxes, menus, graphics, and much more.
- *Apple Numerics Manual*. Apple Computer, Inc. This book describes in detail the floating-point implementation used in the Cortland.

Language Notation

This manual uses certain conventions in common with other Cortland language manuals. The main purpose is to make sure you know which of three languages you're looking at:

- English is in Times Roman:

  C is a very nice language with a very short name

- C is in Courier:

```
int ndigit[10]
```

- Metalanguage expressions, used in syntax diagrams to indicate things that are replaced by C, are in Times Italic:

```
else if (condition)
        statement
```

  Here *condition* and *statement* are expressions that are replaced by actual C expressions. The `else if` and the parentheses are C code.

Chapter 1

Getting started

About Cortland Workshop C

Cortland Workshop C is a complete implementation of the C programming language. It consists of a C compiler developed by MegaMax, Inc.; the Standard C Library; and the Cortland Interface Libraries.

The C Programming Language by Kernighan and Ritchie is currently the most authoritative written definition of C. However, the language has changed in several ways since the book was written. In addition, numerous details of the language definition are open to interpretation. Therefore, the *de facto* standard definition of C differs in several ways from the language originally defined by Kernighan and Ritchie. This *de facto* standard is loosely defined by the most widely used implementation of C, the Portable C Compiler (PCC).

Standard C is our name for the *de facto* standard definition of C as defined and implemented by the Berkeley 4.2 BSD VAX implementation of PCC, including the documented Western Electric extensions. Cortland Workshop C is based on this *de facto* standard (not on the proposed ANSI standard currently under development).

Apple has extended Standard C to facilitate writing programs for the Cortland. Cortland Workshop C includes type void, enumeration data types, structure function parameters and results, enumeration data types, and a function modifier that allows calls to and from Pascal programs and the Cortland Interface Libraries.

Cortland Workshop C supports the Standard Apple Numeric Environment (SANE). It supports all SANE data types and operations, and gives the C programmer full control of the numeric environment. Cortland Workshop C together with the SANE library compose a conforming implementation of extended-precision binary floating-point arithmetic as specified by IEEE Standard 754. Furthermore, source programs written using only `float` and `double` types and standard C operations compile and run without modification.

System Requirements

You need a Cortland with at least one megabyte of RAM, two 800K disk drives or one 800K drive and a hard disk, and CPW.

Writing and running a sample program

Here is how to write, compile, link, and run a trivial sample program.

Entering the sample program

First choose Current Language from the Options menu, then select C from the list of languages and click the Change button. Next open a command file by choosing New Command from the File Menu. It's named `untitled1`*n*, where *n* is some unique number.

Now create a new file by choosing New from the File menu; name it `mice`

Then type a program:. for example:

```
main()
{
      printf(She sells C shells by the C shore.\n");
}
```

Now save the program by choosing Save from the File menu.

Compiling and linking the sample program

To compile your program, enter the `compile` command from the command window.

For example, to compile and link mice, creating an object file `she.root`, enter the following from the command window, then press RETURN:

```
compl she.keep = she
```

Running the sample program

Since you are running under the shell, if you type

```
she
```

you will get

```
She sells C shells by the C shore.
```

immediately below it in the window.

A longer sample program

A more interesting sample program is in the file xxx.c on your CPW disk. It is reprinted in Appendix N.

Chapter 2

Using the Cortland Workshop C Compiler

About the Cortland Workshop C compiler

You can invoke the compiler with any of three commands:

COMP	compile
COMPL	compile and link
COMPLG	compile, link, and go

The last two commands also invoke the linker. The third also executes the program.

In its simplest form, the `comp` command compiles the source file, but saves no object file: it simply verifies its correctness. To create an object file, use the `keep` option, described below.

Command descriptions

The following notation is used to describe commands:

UPPERCASE	Uppercase letters indicate a command or option name
italics	Italics indicate a variable, such as a filename or address
prefix	This parameter the pathname of a directory. It does not include a file name. The pathname must begin with a slash (/). For example, if you are copying a file to the subdirectory `subdirectory` on the volume `volume`, then the *prefix* parameter would be: `/volume/subdirectory/`.
filename	A *filename* may be preceded by any valid *prefix*. For example, if a file is named `file` in the subdirectory `directory` on the volume `volume`, the *filename* parameter would be `/volume/directory/file` .The unit names .CONSOLE, .PRINTER, .PRINTER1, .PRINTER2, and .PRINTER3 can be used as filenames.

|	A vertical bar separates alternative choices. For example, LIST ON|OFF indicates that the command can be entered as either LIST ON or LIST OFF.
A|B	An underlined choice is the default value.
[ ]	Parameters enclosed in square brackets are optional.

You can type commands into the command file whenever the cursor appears in the left margin. You must separate the command from its parameters by one or more spaces. You can use the right-arrow key to expand command names; you can use the up- and down-arrow keys to scroll through commands. Command names cannot be abbreviated, and are case-insensitive. If you omit a required parameter, you are prompted for it. Any of these commands can be placed in an EXEC command file for automatic execution.

Compiler commands

The Workshop C compiler recognizes the following commands:

C

This language command sets the Shell default language to Cortland Workshop C.

COMPILE

COMPILE [+LI-L] [+SI-S]*sourcefile* [KEEP=*outfile*] [NAMES=(*seg1*[,*seg2*[,...]])]
[*language1*=(*option* ...) [*language2*=(*option* ...) ...]]

This internal command compiles (or assembles) a source file. Its function is identical to that of the ASML command, except that it does not call the Linker to link edit the object modules it creates; therefore, no load module is generated. See the ASML command for a description of the parameters. See your compiler manual for the default values of the parameters.

CMPL

CMPL [+LI-L] [+SI-S]*sourcefile* [KEEP=*outfile*] [NAMES=(*seg1*[,*seg2*[,...]])]
[*language1*=(*option* ...) [*language2*=(*option* ...) ...]]

This internal command compiles (or assembles) and links a source file. Its function and parameters are identical to those of the ASML command. See your compiler manual for the default values of the parameters and the language-specific options available.

CMPLG

CMPLG [+L|-L] [+S|-S]*sourcefile* [KEEP=*outfile*] [NAMES=(*seg1*[,*seg2*[,...]])]
[*language1*=(*option* ...) [*language2*=(*option* ...) ...]]

This internal command compiles (or assembles), links, and runs a source file. Its function is identical to that of the ASMLG command. See the ASML command for a description of the parameters. See your compiler manual for the default values of the parameters and the language-specific options available.

Compiler options

The Workshop C compiler recognizes the following options:

Table 1-1. Compiler Options

Option	Description
+L|-L	If you specify +L, the compiler generates a source listing; if you specify -L, the listing is not produced. +L is the default unless you specify the LIST OFF directive in the source file. The L parameter overrides the LIST directive in the source file. *** Is this true? ***
+S|-S	If you specify +S, the compiler produces a symbol table; the linker (if it has been invoked) also produces an alphabetical listing of all global references in the object module. The CPW Assembler, for example, produces an alphabetical listing of all local symbols following each END directive. If you specify –S, these symbol tables are not produced. Each language has its own default for this parameter; the CPW Assembler defaults to +S unless you specify the SYMBOL OFF directive in the source file. The S parameter in this command overrides the SYMBOL directive in the source file. *** ?? Is that true?? *** *** What are CPW defaults for other languages??? ***
sourcefile	The full pathname and filename of the source file.
KEEP=*outfile*	This parameter specifies the filename of the output file. For a one-segment program, the output module is named *outfile*.root. If the program contains more than one segment, the first segment is placed in *outfile*.root and the other segments are placed in *outfile*.a, *outfile*.b, and so forth. If this is a partial compilation, other filename extensions may be used; see the section "Partial Compilation" in this chapter. If the assembly is followed by a successful link edit, then the load file is named *outfile*.

This parameter has the same effect as placing a KEEP directive in your source file. If you have a KEEP directive in the source file and you also use the KEEP parameter, then the filename in the KEEP directive takes precedence. In this case, two object modules are produced with the extension .ROOT; one corresponding to the parameter and one to the directive. However, other files with .A or other extensions are created only with the filename used in the directive, and the Link Editor uses only the filename given in the KEEP directive.

Important: Keep the following points in mind regarding the KEEP parameter:

- If you use neither the KEEP parameter nor the KEEP directive, then the object modules are not saved at all. In this case, the link edit cannot be performed, because there is no object module to link.
- The filename you specify as *outfile* must not be over 10 characters long. This is because the extension .ROOT is appended to the name, and ProDOS does not allow filenames longer than 15 characters.
- If a file named *outfile* already exists, it is overwritten without a warning when this command is executed.

NAMES=(*seg1*,*seg2*,...) This parameter causes the compiler to perform a partial compilation; the operands *seg1*, *seg2*, ... specify the names of the segments to becompiled. The CPW Linker automatically selects the latest version of each segment when the program is link edited.

You assign names to segments with START or DATA directives. The object file created when you use the NAMES parameter contains only the specified segments. When you link a program, the Linker scans all the files whose filenames are identical except for their extensions, and takes the latest version of each segment. Therefore, you must use the same output filename for every partial compilation of a program. For example, if you specify the output filename as OUTFILE for the original compilation of a program, then the compiler creates object modules named OUTFILE.ROOT and OUTFILE.A. In this case you must also specify the output filename as OUTFILE for the partial compilation . The new output file is named OUTFILE.B, and contains only the segments listed with the NAMES parameter.

Note: No blanks are permitted immediately before or after the equal sign in this parameter.

See the section "Partial Assemblies or Compiles" in Chapter 2 of the *CPW Manual* for a complete discussion of partial assemblies.

language1=(option ...)

This parameter allows you to pass parameters directly to specific CPW compilers or assemblers. For each compiler or assembler for which you want to specify options, type the name of the language (exactly as defined in the Command Table), an equal sign (=), and the string of options enclosed in parentheses. The contents and syntax of the options string is specified in the compiler or assembler reference manual; the CPW Shell does no error checking on this string, but passes it through to nthe compiler or assembler. You can include option strings in the command line for as many languages as you wish; if that language compiler is not called, then the string is ignored.

Note: No blanks are permitted immediately before or after the equal sign in this parameter.

Listings and error messages are sent to the command window unless you include a PRINTER ON directive (or equivalent) in the source file; or redirect output to another window, disk file, or the printer in the command line. Output redirection is described in the section "Redirecting Input and Output" in this chapter.

Important: If you are using a LinkEd file to take advantage of the advanced link-edit capabilities it provides, do *not* use the ASML command. Instead, use either the ASSEMBLE or COMPILE command to assemble or compile your program. You can process the LinkEd file automatically by appending it to the end of your program with an APPEND directive (or the equivalent), or you can process it independently with the ALINK command. The Linker is described in detail in Chapter 8.

Source Files, Object Files, and Listing Files

The compiler writes error and warning messages to the standard error file. The message contains source file name, line number, and error or warning text. If no errors or warnings are detected, the compiler runs silently.

If you specify the –p option, the compiler writes progress information and summary information to the standard error file.

C source-file names end with the suffix ".c" by convention. C object-file names consist of the source file name followed by ".o" by default.

Include-file search rules

If the include-file name is a full pathname, the compiler uses that name. A full pathname does not begin with a colon (:) and contains at least one embedded colon. A partial pathname either begins with a colon or does not contain a colon. (For more information about pathname syntax, refer to *Cortland Programmer's Workshop*.)

If the include-file name is a partial pathname, the compiler searches for include files using the rules shown in Table 1-2. The first file successfully opened using these rules is included.

Table 1-2. Include-file search rules

Include-File Name	Example	Search for Partial Pathname
In double quotes.	":Constants.h"	Look in the following directories: (1) The directory of the source file that contains the include statement. (2) Directories specified by the –i option, in the order given. (3) Directories specified by the environment variable *CIncludes*.
In angle brackets.	<CType.h>	Look in the directories described under (2) and (3) above.

Library files for compiling and linking

Appendix B, "Files Supplied with Cortland Workshop C," contains a list of include files and object files to be used with C. Specify the include files when compiling and the object files when linking. For more information on linking C programs, refer to the Linker chapter of *Cortland Programmer's Workshop*.

In general, you will want to specify

- all of the Standard C Library files listed in Appendix B
- only the particular Cortland Interface Libraries files you refer to in your program.

About ProDOS/16

ProDOS/16 is a new operating system for the Cortland. It is a superset of the ProDOS used on earlier Apple II computers. It supports all features of ProDOS but is more powerful, both in additional features and in improved performance

ProDOS/16 has a new system call structure that takes advantage of the 65SC816 processor.

New ProDOS/16 features

ProDOS/16 is designed to take advantage of certain Cortland capabilities and to provide additional programming convenience over ProDOS. For example:

- You can make ProDOS/16 system calls from anywhere in memory, using parameter lists located anywhere in memory. By comparison, ProDOS calls and lists must be in the lowest 64K of memory.
- You can make I/O data transfers under ProDOS/12 to or from anywhere in memory. ProDOS can perform I/O only with the lowest 64K bytes of memory.

- ProDOS/16 provides extensive support for named devices, which can be block or character devices. ProDOS supports only block devices and requires you to refer to a device by its volume name or its slot and drive numbers.
- ProDOS/16 supports up to four system prefixes; ProDOS supports only one.
- ProDOS allows any number of online devices; ProDOS allows only two devices per slot.
- ProDOS/16 supports at least three block device protocols, allowing an application to transparently use so-called "guest file systems", such as the Macintosh hierarchical file system [or MS/DOS???]
- ProDOS/16 supports up to 16interrupt handlers; ProDOS supports only four. Furthermore, ProDOS/16 allows for more than one method of handling unclaimed interrupts.
- ProDOS/16 assigns each caller a unique identification number. ProDOS does not.
- ProDOS has a volume mount function, which prompts the user to mount a needed volume; ProDOS does not.

ProDOS/16 has the following system calls that are not in ProDOS:

CLEAR_BACKUP BIT	Clears a file access bit
CHANGE_PATH	Moves a file's directory within a volume
SET_LEVEL	Sets the system file level
GET_LEVEL	Returns the system file level
GET_DEV_NUM	Returns the reference number for a named device
START_PRODOS	Returns a caller identification number
END_PRODOS	Releases a caller identification number
GET_PATHNAME	Returns pathname of current system program
GET_BOOT_VOL	Returns name of volume that contains ProDOS/16
GET_VERSION	Returns the current ProDOS/16 version
GET_ENTRY	Returns ASCII string with directory information
WRITE_PROTECT	Determines the write-protect status of a volume
GET_DIB	Returns a device information block
SAVE_STATE	Saves system state when leaving an application
RESTORE_STATE	Restores system state when an application returns
SET_INT_MODE	Sets method of handling unclaimed interrupts

Compatibilities

ProDOS/16 is functionally upward-compatible with ProDOS. While a program requires modification to run under ProDOS/16, ProDOS/16 supports all of ProDOS's capabilities:

- The set of ProDOS/16 system calls is a superset of the ProDOS system calls. For nearly every ProDOS system call, there is a functionally equivalent ProDOS/16 call, usually with the same name.

- The calls are made in nearly identical ways in both systems, and the parameter lists are laid out simalarly.
- ProDOS/16 uses exactly the same file system as ProDOS. It can read from and write to any disk volume produced by ProDOS, and *vice versa*. Both physical and logical file and volume formats are the same.
- The ProDOS/16 interrupt-handling procedures and QUIT protocol are functionally compatible with ProDOS.

Using ProDOS/16 from C

ProDOS/16 is fully accessible from C. All ProDOS/16 calls are available through the Cortland Interface Library for C, which provides interface ("glue") code to handle parameter passing and routine calling. The interface code is listed in Chapter 5; the calls are described in more detail in the *Cortland Tool Reference* and the *ProDOS/16 Reference*

For example, if your program's caller ID is 1 and you wished to change a file's pathname from /carmakers/ford/iaccocca to /carmakers/chrysler/iaccocca, you would use the call

```
change_path(/carmakers/ford/iaccocca, /carmakers/chrysler/iaccocca)
```

About Cortland tools

The Cortland User Interface Toolbox is designed so that you don't have to reinvent the menu. All the routines you need to handle mice and menus, windows and files, and other aspects of the human-machine interface, are in the Cortland Interface Toolbox. It consists of of nearly 600 routines, grouped into the following tools:

- Tool Locater
- Memory Manager
- Event Manager
- QuickDraw II
- SANE
- Desk Manager
- Sound Manager
- Control Manager
- Dialog Manager
- Menu Manager
- Window Manager
- File Operations
- Scrap Manager
- Print Manager
- Line Edit
- Miscellaneous Tools

Assembly-language programs call toolbox routines by means of call names. This is the sequence:

1. Push space for the result (if any) onto the stack.

2. Push the input parameters onto the stack.

3. Invoke the call macro.

4. Pull the result (if any) from the stack.

C programs call toolbox routines by calling functions in the Cortland Interface Library for C. These functions take care of the parameter passing. The interface library is listed in Chapter 5; the calls are described in more detail in the *Cortland Tool Reference*.

Appendix N is an exemplary event-driven application in C showing the use of the Cortland tools. It is akin to the application in the *Cortland Tool Reference*. This can be used as a model or plundered for useful code.

About libraries

The following libraries are provided with Cortland Workshop C:

- The Standard C Library (Chapter 4) is a collection of basic routines, not part of the C language, that let you read and write files, examine and manipulate strings, perform data conversion, acquire and release memory, and perform some mathematical procedures.
- The Cortland Interface Libraries (Chapter 5) are a set of interfaces from C to the Cortland Toolbox. They enable you to write C programs that access the routines described in *Cortland Tool Reference*.
- The SANE Library (Chapter 5) in the Cortland Interface Libraries provides mathematical functions and supports floating-point arithmetic. Its routines are documented in the *Apple Numerics Manual*.

*** Should the SANE Library be a separate chapter, or part of Chapter 5? ***

Within Chapters 4 and 5, the material is alphabetical by function or library name. All of the identifiers defined in the libraries are listed in a combined index in Appendix C. The files associated with these libraries are discussed under "Library Files for Compiling and Linking" in Chapter 2.

Chapter 3

The Cortland Workshop C Language

Language Definition

The information provided in this chapter supplements *The C Programming Language* by Kernighan and Ritchie. Where their language definition leaves choices to the implementers, this chapter describes how these aspects of C have been implemented on the Cortland. Where Apple has modified or extended their language definition, this chapter documents the changes.

Data Types

Table 3-1 lists the arithmetic and pointer types available in Cortland Workshop C and shows the number of bits allocated for variables of these types. Types `short` and `long` represent 16-bit and 32-bit integers, respectively. The machine type `int` is a 16-bit integer on Cortland: it is the type the 65SC816 uses most efficiently. Pointers require 32 bits. Enumeration types are allocated either 8, 16, or 32 bits, depending on the range of the enumeration literal values. Types `char`, `short`, `int`, and `long` use two's-complement representation.

Table 3-1. Size and Range of Data Types

Data Type	Bits	Description
`char`	8	range –128 to 127
`unsigned char`	8	range 0 to 255
`short`	16	range –32,768 to 32,767
`unsigned short`	16	range 0 to 65,535
`int`	16	range –32,768 to 32,767
`unsigned int`	16	range 0 to 65,535
`long`	32	range –2,147,483,648 to 2,147,483,647
`unsigned long`	32	range 0 to 4,294,967,295
`enum`	8, 16, or 32	depends on the range of the enumeration literals
*	32	pointer types
`float`	32	IEEE single-precision floating point
`double`	64	IEEE double-precision floating point
`comp`	64	SANE signed integral values
`extended`	80	IEEE extended-precision floating point

Type void

Type `void` has no values and no operators. Type void may be used as a type specifier in function declarations to indicate that the function has no meaningful return value. Specifying type void in Pascal-compatible function declarations reduces the number of instructions generated in calling the function. (See "Pascal-Compatible Functions" later in this chapter.)

Type enum

Type `enum` is a type analogous to the enumeration types of Pascal. Its syntax is similar to that of the `struct` and `union` declarations:

enum-specifier:
enum { *enum-list* }
enum *identifier* { *enum-list* }
enum *identifier*

enum-list:
enumeration-declaration
enumeration-declaration , *enum-list*

enumeration-declaration:
identifier

identifier = *constant-expression*

The first identifier in *enum-specifier*, like the structure tag in a *struct-specifier*, names a particular enumeration. For example,

```
enum color {chartreuse, burgundy, claret, winedark};
. . .
enum color *cp, col;
```

This enumeration makes `color` the enumeration tag of a type describing various colors and then declares `cp` as a pointer to an object of that type and `col` as an object of that type.

The identifiers in *enum-list* are declared as constants and may appear wherever constants are required. If no enumerators with a *constant-expression* appear, the values of the constants begin at 0 and increase by 1 as the declaration is read from left to right. An enumerator with a *constant-expression* gives the associated identifier the value indicated; subsequent identifiers continue the progression by 1 from the assigned value.

Enumeration tags and constants must be unique. They are drawn from the set of ordinary identifiers, unlike structure tags and members. Objects of a given enumeration type have a type distinct from objects of all other types.

Enumeration types are allocated the amount of space required by the smallest predefined type that allows representation of all of the literal values specified by the enumeration. The predefined types considered are `char` and `unsigned char` (8 bits), `short` and `unsigned short` (16 bits), and `int` and `unsigned int` (32 bits).

Register Variables

Most versions of C support register variables. Their function is undefined in Cortland as a result of the small number of registers available on the 65SC816 microprocessor. Use of the `register` declaration causes the compiler to generate code at least as efficient as that generated by the same program without `register` declarations.

Structures

Structures may be assigned, passed as parameters, and returned as function results. The left and right sides of a structure assignment must have identical types. Similiarly, actual and formal parameters must have identical types. Equality comparison for structures has been implemented, provided the structures have the same type.

> **Warning:** In functions that return structures, if an interrupt occurs during the return sequence and the same function is called reentrantly during the interrupt, the value returned from the first call may be corrupted. The problem can occur only in the presence of interrupts. Recursive calls are quite safe.

Return, Newline, and Vertical Tab

Return is the usual line-termination character on the Cortland and is represented by \r (a backslash character followed by a lowercase *r*). The newline character is represented by \n. Vertical tab is represented by \v.

Predefined Symbols

`_LINE_` is a predefined preprocessor symbol whose value is the current line number within the current source file. `_FILE_` is a predefined preprocessor symbol whose value is a character string consisting of the current file name. Each symbol begins and ends with an underscore character.

The symbol `Cortland` is predefined for use in conditional compilation. *** Any others? *** The symbol has the value 1, as if a statement of this form had appeared at the beginning of the source code:

```
#define Cortland 1
```

Standard Apple Numeric Environment Extensions

Cortland Workshop C has built-in support for the Standard Apple Numeric Environment (SANE). The language together with the SANE library support compose a scrupulously conforming extended-precision implementation of the IEEE Standard for Binary Floating-Point Arithmetic (754). SANE provides an extra data type and basic functions for application development. Our C recognizes the SANE data types, uses SANE for all C floating-point operations and conversions, and correctly handles NaNs (Not-a-Number) and infinities in comparisons and in ASCII-binary conversions. Furthermore, source programs from other C implementations, if they were written using only `float` and `double` types and standard C operations, will compile and run in Cortland Workshop C without modification.

Much of SANE is provided through the run-time library `sanelib` and its include file `sane.h`. However, to use extended-precision arithmetic efficiently and effectively, and to handle IEEE NaNs (Not-a-Number) and infinities, some extensions to standard C are required including use of the extended data type.

A change from `double` to `extended` as the basic floating-point type is the most salient change to standard C. Since C was originally developed on the DEC PDP-11, the PDP-11 architecture is reflected in standard C in the use of `float` and `double` as floating-point types, with `double` as the basic type: floating-point expressions are evaluated to `double`, anonymous variables are `double`, and floating-point parameters and function results are passed as `doubles`. However, the low-level SANE arithmetic (as well as the floating-point chips Intel 8087, Motorola 68881, and Zilog Z8070) evaluates arithmetic operations to the range and precision of an 80-bit `extended` type. Thus, `extended` naturally replaces PDP-11 `double` as the basic arithmetic type for computing purposes. The types `float` (IEEE single), `double`, and `comp` serve as space-saving storage types, just as `float` does in standard C.

The IEEE Standard specifies two kinds of special representations for its floating-point formats: NaNs (Not-a-Number) and infinities. Cortland Workshop C expands the syntax for I/O to accommodate NaNs and infinities, and includes the treatment of NaNs in relationals as required by the IEEE Standard.

The SANE extensions to standard C are backward compatible: programs written using only `float` and `double` floating-point types and standard C operations compile and run without modification. SANE does not affect integer arithmetic.

*** Does the term `long double`, used in the proposed ANSI C Standard, have any meaning here? It might be useful to make `long double` mean `extended`, and *vice versa*. ***

The *Apple Numerics Manual* contains detailed documentation of the Standard Apple Numeric Environment.

Constants

Numeric constants that include floating-point syntax—a point (.) or an exponent field—or that lie outside the range of `long` are of type `extended`. Decimal-to-binary conversion for numeric constants is done at compile time (and hence is governed by the default numeric environment; see "Numeric Environment" in this chapter).

Expressions

The SANE types—`float`, `double`, `comp`, and `extended`—can be mixed in expressions with each other and with integer types in the same manner that `float` and `double` can in standard C. An expression consisting solely of a SANE-type variable, constant, or function is of type `extended`. An expression formed by subexpressions and an arithmetic operation is of type `extended` if either of its subexpressions is. Expressions of type `extended` are evaluated using extended-precision SANE arithmetic, with conversions to type `extended` generated automatically as needed. Parentheses in `extended`-type expressions are honored: the compiler will not rearrange terms in violation of parentheses. Initialization of external and static variables, which may include expression evaluation, is done at compile time; all other evaluation of `extended`-type expressions is done at run time.

Comparison Involving a NaN

The result of a comparison involving a NaN operand is unordered. The usual trichotomy of comparisons is expanded to less (<), greater (>), equal (==), and unordered. For example, the negation of "a less than b" is not "a greater than or equal to b" but "(a greater than or equal to b) OR (a and b unordered)". The `sanelib` function `relation` tests all four alternatives.

Functions

A numeric actual parameter passed by value is an expression and hence is of `extended` or `integer` type. All `extended`-type arguments are passed as `extendeds`. Similarly, all results of functions declared `float`, `double`, `comp`, or `extended` are returned as `extendeds`.

Numeric Input/Output

In addition to the usual syntax accepted for numeric input, the Standard C Library function `scanf` recognizes "INF" as infinity and "NAN" as a NaN. NAN may be followed by parentheses, which may contain an integer (a code indicating the NaN's origin). INF and NAN are optionally preceded by a sign and are case insensitive. The `scanf` specifiers for SANE types extend standard C as follows: conversion characters `f`, `e`, and `g` indicate type `float`; `lf`, `le`, and `lg` indicate type `double`; `mf`, `me`, and `mg` indicate type `comp`; and `ne`, `nf`, and `ng` indicate type `extended`.

The Standard C Library function `printf` writes infinities as `[-]INF` and NaNs as `[-]NAN(`*ddd*`)`, where `[-]` is the optional minus sign and *ddd* is the NaN code.

Numeric Environment

The numeric environment refers to rounding direction, rounding precision, halt enables, and exception flags. IEEE Standard defaults—rounding to nearest, rounding to extended precision, and all halts disabled—are in effect for compile-time arithmetic (including decimal-to-binary conversion). Each program begins with these defaults and with all exception flags clear. Functions for managing the environment are included in the library `sanelib`. The compiler, in optimizing, will not change any part of the numeric environment, including the exception-flag setting, which is a side effect of arithmetic operations.

About the SANE Library

The SANE library rounds out the IEEE Standard implementation and provides the basic tools for developing a wide range of applications. The SANE library includes the following:

- logarithmic, exponential, and trigonometric functions
- financial functions
- random number generation
- binary-decimal conversion
- numeric scanning and formatting
- environment control
- other functions required or recommended by the IEEE Standard

Additional information can be found under the SANE entry in Chapter 4, "Cortland Interface Libraries."

Programming with IEEE Arithmetic

Cortland Workshop C's automatic use of the `extended` type produces results that are generally better than those of other C systems. Extended precision yields more accuracy and extended range avoids unnecessary underflow and overflow of intermediate results. The programmer can further exploit the `extended` type by declaring all floating-point temporary variables to be type `extended`. This is both time- and space-efficient, since it reduces the number of automatic conversions between types. External data should be stored in one of the three smaller SANE types (`float`, `double`, or `comp`), not only for economy but also because the `extended` format may vary between SANE implementations. As a general rule, use `float`, `double`, or `comp` data as program input; `extended` arithmetic for computations; and `float`, `double`, or `comp` data as program output.

In many instances, IEEE arithmetic allows simpler algorithms than were possible without IEEE arithmetic. The handling of infinities enlarges the domain of some formulas. For example, $1+1/x^2$ computes correctly even if x^2 overflows. Running with halts disabled (the default), a program will never crash due to a floating-point exception. Hence by monitoring exception flags a program can test for exceptional cases after the fact. The alternative of screening out bad input is often infeasible, sometimes impossible.

Pascal-Compatible Functions

The function-calling conventions used by Cortland Workshop C and Pascal differ radically in the order of parameters on the stack, the type coercions applied to parameters, the location of the return result, and the number of scratch registers. C has been extended to allow function calls between these languages. The specifier `pascal` in a function declaration or definition indicates a Pascal-compatible function.

Pascal-Compatible Function Declarations

A function or procedure written in Pascal (or written in assembly language following Pascal calling conventions) can be called from Cortland Workshop C. For example, the DrawText procedure is defined in Pascal as:

```
PROCEDURE DrawText (textBuf: Ptr;
      firstByte, byteCount: INTEGER);
```

The CPW syntax for declaring this procedure as a C function is:

```
extern pascal void DrawText();
```

To make the code more readable, we can list the parameters in a comment:

```
extern pascal void DrawText();
      /* Ptr textBuf;
```

```
        short firstByte, byteCount;
        extern; */
```

Pascal-Compatible Function Definitions

A function definition (the actual function), like a function declaration, can also be preceded by the `pascal` specifier. The function then adheres to Pascal-compatible calling conventions and can be called from Pascal. For example, the following C function can be called from Pascal:

```
pascal void MyText(byteCount,textAddr,numer,denom)
        short byteCount;
        Ptr textAddr;
        Point numer,demon;
{
        ...
}
```

The corresponding Pascal function declaration is

```
PROCEDURE MyText(bytecount: INTEGER; textAddr: Ptr;
        numer,denom: Point);
```

For compatibility with Pascal and assembly language, the compiler converts the names of Pascal-compatible functions to uppercase before writing them to the object file. When they are called in C programs, these routines should be capitalized exactly as they were declared in C. Pascal-compatible functions whose names differ only in their capitalization will become duplicate declarations when their names are converted to uppercase by the compiler; therefore such names should be avoided.

Parameter and Result Data Types

C and Pascal support different data types. Therefore when writing a Pascal-compatible function declaration in C, a translation of the parameter types and function-result type (from Pascal to C) is required. Often this translation is trivial, but other cases are surprising.

Table 3-2 below summarizes this translation. Find the Pascal parameter or result type in the first column. Use the equivalent C type found in the second column when declaring the function in C. Comments in the table point out unusual cases which may require special attention.

Table 3-2. Parameter and Result Data Types

Pascal Data Type	C Equivalent	Comments
`boolean`	`Boolean`	`Boolean` is defined in file `Types.h` as `enum {false,true}`.
`var boolean`	`Boolean *`	In C, `false` is zero and `true` is often considered nonzero.
`boolean result`	`Boolean`	In Pascal, `false` is zero and `true` is one.
`enumeration`	`enum`	Use identical ordering of the

(<128 or >255 literals)		enumeration literals.
enumeration (128 to 255 literals)	short	Pascal passes enumerations with 128 or more literals as words.
var enumeration (<128 or >255 literals)	enum *	
var enumeration (128 to 255 literals)	short *	
enumeration result (<128 or >255 literals)	enum	
enumeration result (128 to 255 literals)	short	
char	short	Surprise! Pascal passes chars as 16-bit values.
var char	char *	
char result	short	
integer	short	16-bit signed values.
var integer	short *	
short result	short	
longint	int or long	32-bit signed values.
var longint	int * or long	* *** *long only???* ***
longint result	int or long	*** *long only???* ***
real	extended *	Pascal passes real parameters as extended by address.
var real	float *	
real result	float	Pascal returns real results by value.
double	extended *	Pascal passes double parameters as extended by address.
var double	double *	
double result	double	The caller supplies the address of the double result.
comp	extended *	Pascal passes comp parameters as extended by address.
var comp	comp *	
comp result	comp	The caller supplies the address of the comp result.
extended	extended *	Pascal passes extended parameters by address.
var extended	extended *	
extended result	extended	The caller supplies the address of the extended result.
pointer	pointer	32-bit addresses.

`var pointer`	`pointer *`	
`pointer` result	`pointer`	
`array` (1 or 2 bytes)	`short`	Pascal passes small `arrays` by value.
`array` (3 or 4 bytes)	`int` or `long`	*** *long only???* ***
`array` (5 or more bytes)	`array`	Pascal passes larger `arrays` by address.
`var array`	`array`	
`array` result	---	C does not allow `array` results.
`record` (1 to 4 bytes)	`struct`	Pascal passes small `records` by value.
`record` (5 or more bytes)	`struct *`	Pascal passes larger `records` by address.
`var record` (any size)	`struct *`	
`record` result (1 or 2 bytes)	`short`	Pascal returns small `records` by value.
`record` result (3 or 4 bytes)	`int` or `long`	*** *long only???* ***
`record` result (1 or 2 bytes)	`struct`	The caller supplies the address of the `record` result.
`set` (1 to 7 elements)	`char`	Pascal passes `sets` with 1 to 7 elements as bytes.
`set` (8 to 16 elements)	`short`	Pascal passes `sets` with 8 to 16 elements as words.
`set` (≥17 elements)	`struct`	Pascal also passes larger `sets` by value.
`var set` (1 to 7 elements)	`char *`	
`var set` (8 to 16 elements)	`short *`	
`var set` (≥17 elements)	`struct *`	
`set` result (1 to 7 elements)	`char`	Pascal returns small `sets` by value.
`set` result (8 to 16 elements)	`short`	
`set` result (≥17 elements)	`struct`	The caller supplies the address of the `set` result.

Implementation Notes

A number of details in any language definition are left to the discretion of its individual implementations. Most programs do not rely on these details and therefore yield the same results on the various implementations. However, knowledge of the major differences between implementations can help you avoid reliance on language semantics that vary from implementation to implementation. This section explains several areas of the language definition that are specific to Workshop C.

Byte Ordering

On the 65SC816, the microprocessor used in the Cortland, the least significant byte of a short or long integer has the lowest memory address. This byte ordering is also used on

IBM/370 and Z8000 processors. The PDP-11 family, VAX, 8086, and NS16000 use a different ordering. Programs that rely on the order of the bytes within words and longs will not work correctly on both classes of machines.

Memory-Allocation Characteristics

The Workshop C compiler optimizes memory allocation in various ways. Static and global variables are not necessarily allocated in the order in which they are specified. (However, the order of fields within records is preserved.) Static variables may be allocated as if they were automatic if their values are always set before being referenced. Automatic and static variables that are never used may not be allocated at all. Programs should not rely on the compiler's allocation algorithms.

Types `unsigned char` and `unsigned short`

Types `unsigned char` and `unsigned short` are supported by the Cortland C compiler and by many implementations of PCC, although they are not required by the basic C language definition. The VAX implementation of PCC and the Cortland C compiler differ in the way they evaluate expressions involving these types. For example, the negation operator subtracts an `unsigned short` from 2^{16} under PCC (this seems like a bug), and from 2^{32} under Cortland Workshop C.

Bit Fields

Workshop C provides bit fields that are unsigned, as do all MC68000 versions of PCC of which we are aware. However, VAX implementations of C may support signed bit fields. In the following example, implementations using unsigned bit fields will set i to 3; implementations using signed bit fields will set i to -1:

```
struct {int  field:2;}  x;
   x.field  =  3;
   i  =  x.field;
```

Evaluation Order

Cortland Workshop C does not define the evaluation order of certain expressions. Expressions with side effects, such as function calls and the "++" and "--" operators, may yield different results on different machines or with different compilers. Specifically, when a variable is modified as a side effect of an expression's evaluation and the variable is also used at another point in the same expression, the value used may be either the value before modification or the value after modification.

Programs that rely on the order of evaluation in these situations are in error. The function call

```
f(i,i++)
```

is an example of an expression whose value is undefined.

Case Statements

Some implementations of C, including PCC, allow cases of a `switch` statement to be nested within compound statements. Cortland Workshop C considers this an error. The following `switch` statement compiles using PCC but generates an error message using the Cortland Workshop C compiler. The error is that "case 2:" is within the `if` statement.

```
switch  (i)  {
   case 1:
      if  (j)  {
         case  2:
               i  = 3;
      }
   }
```

Language Anachronisms

Several constructs formerly considered part of the C language are now considered anachronisms. When you specify the -z84 compiler option, anachronistic constructs are compiled and flagged with a warning message. Otherwise they are considered invalid. The anachronisms are described below.

Assignment Operators: The *=op* form of assignment operators is not supported. Alternate interpretations are accepted without warning. In particular,

`x =- 5;`	is interpreted as	`x = (-5);`
`x =* 5;`	is interpreted as	`x = (*5);`
`x =& p;`	is interpreted as	`x = (&p);`

Initialization: The equal sign that introduces an initializer must be present. The anachronism

```
int i 1;
```

is considered an error.

Structures and Unions: References to members of structures and unions must be to the appropriate structure or union. For example, the reference *a.b* is illegal if *b* is not a member of *a*. References to components of nested structures and unions must be fully qualified (i.e. all intermediate levels of the reference must be specified).

The names of structure and union members do not conflict with the names of ordinary variables in the same scope. Furthermore, a particular member name may be used in several structures and unions in the same scope.

Compiler Limitations

On the Cortland, the total size of all declared global variables, static variables, and string constants cannot exceed 32K bytes. Allocate large global arrays on the heap *** correct? *** in order to avoid exceeding this limit.

Automatic variables are limited to 32K bytes.

It is impossible to compile very large functions on the Cortland because the compiler's internal data structures cannot fit in memory. As functions approach this limit, compilation time increases noticeably. This problem can be alleviated by eliminating unnecessary include files, reducing the number of global declarations, compiling large functions separately, and rewriting large functions as two or more smaller functions.

Chapter 4

The Standard C Library

Introduction to the Standard C Library

This chapter describes the Standard C Library provided with Cortland Workshop C. After an introductory discussion of error-number conventions, the chapter is arranged alphabetically by library header. Several library routines—functions and macros—may be grouped under a single header. For example, a number of trigonometric functions are documented under the header *"trig."*

All of the identifiers in the Standard C Library are listed in the Library Index, Appendix C.

> **Note:** Remember that identifiers in C are case sensitive and should be spelled *exactly as shown in the synopsis.*

The library routines under each header are documented as follows:

- NAME. Lists the names of the library routines, followed by a descriptive phrase.
- SYNOPSIS. Shows the code you need to add to your program when using these library routines. Indicates libraries you need to include at compile time.
- DESCRIPTION. Discusses the library routines and their input and output.
- DIAGNOSTICS. Describes error conditions.
- RETURN VALUE. Describes the values returned by the routines.
- NOTE. Contains remarks.
- WARNING. Gives cautions.
- SEE ALSO. Provides the names of other library routines related to the ones described in the current document.

NAME

Introduction to error numbers similar to those in UNIX™ operating systems.

SYNOPSIS

```
#include <errno.h>

extern int errno;
```

DESCRIPTION

Many of the Standard C Library functions have one or more error returns. An error condition is indicated by an otherwise impossible returned value. This is almost always –1; see descriptions of individual functions for details. An error number is also made available in the external variable *errno*. The variable *errno* is not cleared on successful calls, so it should be tested only after an error has been indicated.

The error name appears in brackets following the text in a library function description; for example,

> "The next attempt to write a nonzero number of bytes will signal an error. [ENOSPC]"

Not all possible error numbers are listed for each library function because many errors are possible for most of the calls. Some UNIX operating system error numbers do not apply to Cortland and are not documented in this manual.

Here is a list of the error numbers and their names as defined in the <errno.h> file:

1 [EPERM] Not owner.
Typically this error indicates an attempt to modify a file in a way that is not permitted.

2 [ENOENT] No such file or directory
This error occurs when a file whose filename is specified does not exist or when one of the directories in a pathname does not exist.

5 [EIO] I/O error
Some physical I/O error has occurred. This error may in some cases be signaled on a call following the one to which it actually applies.

6 [ENXIO] No such device or address
I/O on a special file refers to a subdevice that does not exist, or the I/O is beyond the limits of the device. This error may also occur when, for example, no disk is present in a drive.

9 [EBADF] Bad file number
Either a file descriptor does not refer to an open file, or a read (or write) request is made to a file that is open only for writing (or reading).

12 [ENOMEM] Not enough space
The system ran out of memory while the library call was executing.

13 [EACCES] Permission denied
An attempt was made to access a file in a way forbidden by the protection system.

17 [EEXIST] File exists
An existing file was mentioned in an inappropriate context—e.g., open(file, O_CREAT+O_EXCL).

19 [ENODEV] No such device
An attempt was made to apply an inappropriate system call to a device—e.g., read a write-only device.

20 [ENOTDIR] Not a directory
An object that is not a directory was specified where a directory is required—e.g., in a path prefix.

21 [EISDIR] Is a directory
An attempt was made to write on a directory.

22 [EINVAL]] Invalid argument
Some invalid argument was provided to a library function.

23 [ENFILE] File table overflow
The system's table of open files is full, so temporarily a call to open cannot be accepted.

24 [EMFILE] Too many open files
No program may have more than 20 file descriptors open at a time.

28 [ENOSPC] No space left on device
During a write to an ordinary file, there is no free space left on the device.

29 [ESPIPE] Illegal seek
An lseek was issued incorrectly.

30 [EROFS] Read-only file system
An attempt to modify a file or directory was made on a device mounted for read-only access.

NOTE

Calls that interface to the Cortland I/O system—e.g., *open*, *close*, *read*, *write*, *ioctl*, and others—set the external variable *MacOSErr* as well as *errno*. This manual

documents only *errno* values. The equivalent Cortland ROM error-return values set in **** variable name? **** are documented in *Cortland Tools*. The appropriate include file for most values of **** variable name? **** is <files.h>.

NAME

abs—return integer absolute value

SYNOPSIS

```
int abs (i)
int i;
```

DESCRIPTION

Function *abs* returns the absolute value of its integer operand.

NOTE

The absolute value of the negative integer with largest magnitude is undefined.

SEE ALSO

floor.

NAME

atof—convert ASCII string to floating-point number

SYNOPSIS

```
extended atof (nptr)
char *nptr;
```

DESCRIPTION

Function *atof* converts a character string pointed to by *nptr* to an extended-precision floating-point number. The first unrecognized character ends the conversion. Function *atof* recognizes an optional string of white-space characters (blanks or tabs), then an optional sign, then a string of digits optionally containing a decimal point, then an optional "e" or "E" followed by an optionally signed integer. If the string begins with an unrecognized character, *atof* returns a NaN.

DIAGNOSTICS

Function *atof* honors the floating-point exception flags—invalid operation, underflow, overflow, divide by zero, and inexact—as prescribed by the Standard Apple Numeric Environment (SANE).

SEE ALSO

scanf, str2dec, dec2num.
Apple Numerics Manual.

NAME

atoi, atol—convert string to integer

SYNOPSIS

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

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

DESCRIPTION

The character string *str* is scanned up to the first nondigit character other than an optional leading minus sign (–). Leading white-space characters (blanks and tabs) are ignored.

Function *atoi* returns as an integer the decimal value represented by *str*.

Function *atol* returns as a long integer the decimal value represented by *str*.

NOTE

Overflow conditions are ignored.

SEE ALSO

atof, scanf, strtol.

NAME

close—close a file descriptor

SYNOPSIS

```
int close (fildes)
int fildes;
```

DESCRIPTION

Variable *fildes* is a file descriptor obtained from a *creat* or *open* call. Function *close* closes the file descriptor indicated by *fildes*.

Function *close* fails if *fildes* is not a valid open file descriptor. [EBADF]

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

NOTE

This routine provides facilities used in the Integrated Environment; for more information, refer to *Cortland Programmer's Workshop*.

SEE ALSO

creat, open.

NAME

toupper, tolower, _toupper, _tolower, toascii—translate characters

SYNOPSIS

```
#include <ctype.h>

int toupper (c)
int c;

int tolower (c)
int c;

int _toupper (c)
int c;

int _tolower (c)
int c;

int toascii (c)
int c;
```

DESCRIPTION

Functions *toupper* and *tolower* have as domain the range of *getc*: the integers from −1 through 255. If the argument of *toupper* represents a lowercase letter, the result is the corresponding uppercase letter. If the argument of *tolower* represents an uppercase letter, the result is the corresponding lowercase letter. All other arguments in the domain are returned unchanged.

Macros *_toupper* and *_tolower* produce the same results as functions *toupper* and *tolower* but have restricted domains and are faster. Macro *_toupper* requires a lowercase letter as its argument; its result is the corresponding uppercase letter. Macro *_tolower* requires an uppercase letter as its argument; its result is the corresponding lowercase letter. Arguments outside the domain cause undefined results.

Function *toascii* yields its argument by turning off all bits that are not part of a standard ASCII character; it is intended for compatibility with other systems.

SEE ALSO

ctype, getc.

NAME

creat—create a new file or rewrite an existing file

SYNOPSIS

```
int creat (path)
char *path;
```

DESCRIPTION

Function *creat* creates a new file or prepares to rewrite an existing file named by the pathname pointed to by *path*. If the file exists, the length of its data fork is truncated to 0.

Function *creat(path)* is equivalent to

```
open(path,O_WRONLY|O_TRUNC).
```

Upon successful completion, a nonnegative integer (the file descriptor) is returned and the file is open for writing. The file pointer is set to the beginning of the file. A maximum of about 30 files may be open at a given time; the actual maximum depends upon the current system environment.

RETURN VALUE

Upon successful completion, a nonnegative integer (the file descriptor) is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

NOTE

This routine provides facilities used in the Integrated Environment; for more information, refer to *Cortland Programmer's Workshop*.

SEE ALSO

close, lseek, open, read, write.

NAME

isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, isascii
—classify characters

SYNOPSIS

```
#include <ctype.h>

int isalpha (c)
int c;

. . .
```

DESCRIPTION

These macros classify character-coded integer values by table lookup, returning nonzero for true, zero for false. Macro *isascii* is defined on all integer values; the rest are defined only where *isascii* is true and on the single non-ASCII value EOF (–1).

Macro	Returns TRUE if...
isalpha	*c* is a letter.
isupper	*c* is an uppercase letter.
islower	*c* is a lowercase letter.
isdigit	*c* is a digit [0-9].
isxdigit	*c* is a hexadecimal digit [0-9], [A-F], or [a-f].
isalnum	*c* is alphanumeric (letter or digit).
isspace	*c* is a space, tab, return, new line, vertical tab, or form feed.
ispunct	*c* is a punctuation character (neither control nor alphanumeric).
isprint	*c* is a printing character, code 040 (space) through 0176 (tilde).
isgraph	*c* is a printing character, similar to *isprint* except false for space.
iscntrl	*c* is a delete character (0177) or an ordinary control character (less than 040).
isascii	*c* is an ASCII character, code less than 0200.

DIAGNOSTICS

If the argument to any of these macros is not in the domain of the function, the result is undefined.

NOTE

These macros do not support the Cortland extended character set.

NAME

dup—duplicate an open file descriptor

SYNOPSIS

```
int dup (fildes)
int fildes;
```

DESCRIPTION

Variable *fildes* is a file descriptor that has been obtained from a *creat*, *dup*, or *fcntl* call. The new file descriptor returned by *dup* is the lowest one available. It has the following in common with *fildes*:

- Same open file.
- Same file pointer.
- Same access mode: read, write, or read/write.

Because the new file descriptor and *fildes* share the same file pointer, a seek on *fildes* affects a subsequent read or write on the new file descriptor, and vice versa.

The function call *dup(fildes)* is equivalent to

```
fcntl (fildes, F_DUPFD, 0)
```

Function *dup* fails if one or more of the following are true:

- Variable *fildes* is not a valid open file descriptor. [EBADF]
- Too many file descriptors are currently open. [EMFILE]

RETURN VALUE

Upon successful completion a nonnegative integer, the file descriptor, is returned. Otherwise, a value of –1 is returned and *errno* is set to indicate the error.

SEE ALSO

creat, close, fcntl, open.

NAME

ecvt, fcvt—convert floating-point number to string

SYNOPSIS

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

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

DESCRIPTION

Function *ecvt* converts *value* to a null-terminated string of *ndigit* digits and returns a pointer to this string as the function result. The low-order digit is rounded.

The decimal point is not included in the returned string. The position of the decimal point is indicated by *decpt*, which indirectly stores the position of the decimal point relative to the returned string. If the int pointed to by *decpt* is negative, the decimal point lies to the left of the returned string. For example, if the string is "12345" and *decpt* points to an int of 3, the value of the string is 123.45; if *decpt* points to –3, the value of the string is .00012345 .

If the sign of the converted value is negative, the word pointed to by *sign* is nonzero; otherwise it is zero.

Functions *fcvt* and *ecvt* provide fixed-point output in the style of Fortran F-format output, with the following difference in the interpretation of *ndigit*:

- In *fcvt*, *ndigit* specifies the number of digits to the right of the decimal point.
- In *ecvt*, *ndigit* specifies the number of digits in the string.

NOTE

The string pointed to by the function result is static data whose content is overwritten by each call.

SEE ALSO

printf, num2dec, dec2str.
Apple Numerics Manual.

NAME

exit, _exit—terminate the current application

SYNOPSIS

```
void exit(status)
int status;

void _exit(status)
int status;
```

DESCRIPTION

Function *exit* terminates the current application, closing all of the open file descriptors. It also causes *stdio* cleanup actions before the application terminates.

Function *_exit* circumvents all cleanup.

RETURN VALUE

Variable *status* is returned to the Cortland Workshop Shell: zero for normal execution and nonzero for errors.

SEE ALSO

onexit.

NAME

exp, log, log10, pow, sqrt—exponential, logarithm, power, square-root functions

SYNOPSIS

```
#include <math.h>

extended exp(x)
extended x;

extended log(x)
extended x;

extended log10(x)
extended x;

extended pow(x, y)
extended x, y;

extended sqrt(x)
extended x;
```

DESCRIPTION

Function *exp* returns e^x, where e is the natural logarithm base.

Function *log* returns the natural logarithm (base e) of x.

Function *log10* returns the logarithm base ten of x.

Function *pow* returns x^y.

Function *sqrt* returns the square root of x.

For special cases, these functions return a NaN or signed infinity as appropriate.

DIAGNOSTICS

These functions honor the floating-point exception flags—invalid operation, underflow, overflow, divide by zero, and inexact—as prescribed by the Standard Apple Numeric Environment (SANE).

SEE ALSO

hypot, sinh.
Apple Numerics Manual.

NAME

faccess—*** ? ***

SYNOPSIS

```
int faccess (char *fileName, usigned int cmd, ...);
```

DESCRIPTION

F_DELETE, F_RENAME, F_OPEN (internal use only)

Extended CPW file information: FGTABINFO, F_STABINFO, F_GFONTINFO, F_SFONTINFO

NAME

fclose, fflush—close or flush a stream

SYNOPSIS

```
#include <stdio.h>

int fclose (stream)
FILE *stream;

int fflush (stream)
FILE *stream;
```

DESCRIPTION

Function *fclose* causes any buffered data for *stream* to be written out; *stream* is then closed.

Function *fclose* is performed automatically for all open files upon calling *exit*.

Function *fflush* causes any buffered data for *stream* to be written out; *stream* remains open.

DIAGNOSTICS

These functions return 0 for success or EOF if an error was detected (such as trying to write to a file that has not been opened for writing).

SEE ALSO

close, exit, fopen, setbuf.

NAME

fcntl—file control

SYNOPSIS

```
#include <fcntl.h>

int fcntl (fildes, cmd, arg)
int fildes;
unsigned int cmd;
int arg;
```

DESCRIPTION

Function *fcntl* provides control over open files. Variable *fildes* is an open file descriptor obtained from a *creat*, *open*, or *fcntl* call. Variable *cmd* is one of the following values:

F_DUPFD (return a new file descriptor):

Lowest numbered available file descriptor greater than or equal to *arg*.

Same open file as the original file.

Same file pointer as the original file (i.e., both file descriptors share one file pointer).

Same access mode (read, write, or read/write).

F_GETFD, F_SETFD, F_GETFL, F_SETFL

These commands are not supported on Cortland.

Function *fcntl* fails if one or more of the following are true:

Variable *fildes* is not a valid open file descriptor. [EBADF]

More than about 30 file descriptors are currently open; the exact number permissible depends upon the current system environment. [EMFILE]

Variable *arg* is negative or greater than the highest allowable file descriptor. [EINVAL]

RETURN VALUE

Upon successful completion, the value returned is a new file descriptor. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

SEE ALSO

close, dup, open.

NAME

ferror, feof, clearerr, fileno—stream status inquiries

SYNOPSIS

```
#include <stdio.h>

int feof (stream)
FILE *stream;

int ferror (stream)
FILE *stream;

void clearerr (stream)
FILE *stream;

int fileno (stream)
FILE *stream;
```

DESCRIPTION

Function *feof* returns nonzero when end-of-file has previously been detected reading the named input stream; otherwise, it returns zero.

Function *ferror* returns nonzero when an I/O error has previously occurred reading from or writing to the named stream; otherwise, it returns zero.

Function *clearerr* resets the error indicator and end-of-file indicator to zero on the named stream.

Function *fileno* returns the integer file descriptor associated with the named stream; see *open*.

NOTE

All these functions are implemented as macros; they cannot be declared or redeclared.

SEE ALSO

open, fopen.

NAME

`floor, ceil, fmod, fabs`—floor, ceiling, mod, absolute value functions

SYNOPSIS

```
#include <math.h>

extended floor(x)
extended x;

extended ceil(x)
extended x;

extended fmod(x, y)
extended x, y;

extended fabs(x)
extended x;
```

DESCRIPTION

Function `floor` returns the largest integer (as an extended-precision number) not greater than *x*.

Function `ceil` returns the smallest integer not less than *x*.

Whenever possible, `fmod` returns the number *f* with the same sign as *x*, such that

$x = iy + f$ for some integer i, and $|f| < |y|$. If *y* is zero, `fmod` returns a NaN.

Function `fabs` returns $|x|$.

SEE ALSO

`abs, rint, setround.`
Apple Numerics Manual.

NAME

fopen, freopen, fdopen—open a stream

SYNOPSIS

```
#include <stdio.h>

FILE *fopen (filename, type)
char *filename, *type;

FILE *freopen (filename, type, stream)
char *filename, *type;
FILE *stream;

FILE *fdopen (fildes, type)
int fildes;
char *type;
```

DESCRIPTION

Function *fopen* opens the file named by *filename* and associates a stream with it. Function *fopen* returns a pointer to the FILE structure associated with the stream.

Variable *filename* points to a character string that contains the name of the file to be opened.

Character string *type* is has one of the following values:

r	open for reading.
w	truncate or create for writing.
a	append: open for writing at end-of-file, or create for writing.
r+	open for update (reading and writing).
w+	truncate or create for update.
a+	append: open or create for update at end-of-file.

Function *freopen* substitutes the named file for the open stream. The original stream is closed, regardless of whether the open ultimately succeeds. Function *freopen* returns a pointer to the FILE structure associated with *stream*.

Function *freopen* is typically used to attach the previously opened streams associated with *stdin*, *stdout*, and *stderr* to other files.

Function *fdopen* associates a stream with a file descriptor by formatting a file structure from the file descriptor. Thus, *fdopen* can be used to access the file descriptors returned by *open* or *creat*. (These calls open files but do not return pointers to a FILE structure.) The type of stream must agree with the mode of the open file.

When a file is opened for update, both input and output may be done on the resulting stream. However, output may not be directly followed by input without an intervening *fseek* or *rewind*, and input may not be directly followed by output

without an intervening *fseek*, *rewind*, or an input operation that encounters end-of-file.

When a file is opened for append (i.e., when *type* is "a" or "a+"), it is impossible to overwrite information already in the file. Function *fseek* may be used to reposition the file pointer to any position in the file, but when output is written to the file the current file pointer is disregarded. All output is written at the end of the file and causes the file pointer to be repositioned at the end of the output.

DIAGNOSTICS

Functions *fopen* and *freopen* return a null pointer on failure.

SEE ALSO

open, fclose.

NAME

fread, fwrite—binary input/output

SYNOPSIS

```
#include <stdio.h>

int fread(ptr, size, nitems, stream)
char *ptr;
int size, nitems;
FILE *stream;

int fwrite(ptr, size, nitems, stream)
char *ptr;
int size, nitems;
FILE *stream;
```

DESCRIPTION

Function *fread* copies *nitems* items of data from the named input stream into an array beginning at *ptr*. An item of data is a sequence of bytes (not necessarily terminated by a null byte) of length *size*. Function *fread* stops appending bytes if an end-of-file or error condition is encountered while reading *stream* or if *nitems* items have been read. Function *fread* leaves the file pointer in *stream*, if defined, pointing to the byte following the last byte read if there is one. Function *fread* does not change the contents of *stream*.

Function *fwrite* appends at most *nitems* items of data to the named output stream from the array pointed to by *ptr*. Function *fwrite* stops appending when it has appended *nitems* items of data or if an error condition is encountered on *stream*. Function *fwrite* does not change the contents of the array pointed to by *ptr*.

The variable *size* is typically

```
sizeof(*ptr)
```

where the pseudo-function *sizeof* specifies the length of an item pointed to by *ptr*. If *ptr* points to a data type other than char it should be cast into a pointer to char.

DIAGNOSTICS

Functions *fread* and *fwrite* return the number of items read or written. If *nitems* is zero or negative, no characters are read or written and zero is returned by both *fread* and *fwrite*.

SEE ALSO

fopen, getc, gets, printf, putc, puts, read, scanf, stdio, write.

NAME

frexp, ldexp, modf—manipulate parts of floating-point numbers

SYNOPSIS

```
extended frexp(value, eptr)
extended value;
int *eptr;

extended ldexp(value, exp)
extended value;
int exp;

extended modf(value, iptr)
extended value, *iptr;
```

DESCRIPTION

Every nonzero number can be written uniquely as $x^* 2^n$, where the mantissa (fraction) x is in the range $0.5 \le |x| < 1.0$, and the exponent n is an integer. Function *frexp* returns the mantissa of an extended value and stores the exponent indirectly in the location pointed to by *eptr*. Note that the mantissa here differs from the significand described in the *Apple Numerics Manual*, whose normal values are in the range $1.0 \le |x| < 2.0$.

Function *ldexp* returns the quantity $value^* 2^{exp}$.

Function *modf* returns the signed fractional part of *value* and stores the integral part indirectly in the location pointed to by *iptr*.

DIAGNOSTICS

Function *ldexp* honors the floating-point exception flags—invalid operation, underflow, overflow, divide by zero, and inexact—as prescribed by the Standard Apple Numeric Environment (SANE).

SEE ALSO

logb, scalb.
Apple Numerics Manual.

NAME

fseek, rewind, ftell—reposition a file pointer in a stream

SYNOPSIS

```
#include <stdio.h>

int fseek (stream, offset, ptrname)
FILE *stream;
long offset;
int ptrname;

void rewind (stream)
FILE *stream;

long ftell (stream)
FILE *stream;
```

DESCRIPTION

Function *fseek* sets the position of the next input or output operation on the stream. The new position is at the signed distance *offset* bytes from the beginning, the current position, or the end of the file, when the value of *ptrname* is 0, 1, or 2, respectively.

The call

```
rewind(stream)
```

is equivalent to

```
fseek(stream, 0L, 0)
```

except that no value is returned.

Functions *fseek* and *rewind* undo any effects of *ungetc*.

After *fseek* or *rewind*, the next operation on a file opened for update may be either input or output.

Function *ftell* returns the offset of the current byte relative to the beginning of the file associated with the named stream.

DIAGNOSTICS

Function *fseek* returns nonzero for improper seeks; otherwise it returns 0. An example of an improper seek is an *fseek* on a file that has not been opened via *fopen*.

SEE ALSO

lseek, fopen.

NAME

getc, getchar, fgetc, getw—get character or word from stream

SYNOPSIS

```
#include <stdio.h>

int getc(stream)
FILE *stream;

int getchar()

int fgetc(stream)
FILE *stream;

int getw(stream)
FILE *stream;
```

DESCRIPTION

Macro *getc* returns the next character (i.e., byte) from the named input stream. It also moves the file pointer, if defined, ahead one character in *stream*. Macro *getc* cannot be used if a function is necessary; for example, you cannot have a function pointer point to it.

Macro *getchar* returns the next character from the standard input stream, *stdin*.

Function *fgetc* produces the same result as macro *getc*; *fgetc* runs more slowly than *getc* but takes less space per invocation.

Function *getw* returns the next "word" (i.e., four bytes) from the named input stream so that the order of bytes in the stream corresponds to the order of byes in memory. Function *getw* returns the constant EOF upon end-of-file or error. Since EOF is a valid integer value, *feof* and *ferror* should be used to check the success of *getw*. Function *getw* increments the associated file pointer, if defined, to point to the next word. Function *getw* assumes no special alignment in the file.

DIAGNOSTICS

These calls return the integer constant EOF at end-of-file or upon an error.

NOTE

Because it is implemented as a macro, *getc* treats incorrectly a stream argument with side effects. In particular, getc(*f++) doesn't work as you would expect. Use fgetc(*f++) instead.

SEE ALSO

fclose, ferror, fopen, fread, gets, putc, scanf, stdio.

NAME

gets, fgets—get a string from a stream

SYNOPSIS

```
#include <stdio.h>

char *gets(s)
char *s;

char *fgets(s, n, stream)
char *s;
int n;
FILE *stream;
```

DESCRIPTION

Function *gets* reads characters from the standard input stream *stdin* into the array pointed to by *s* until a newline character is read or an end-of-file condition is encountered. The newline character is discarded and the string is terminated with a null character.

Function *fgets* reads characters from *stream* into the array pointed to by *s* until *n*–1 characters are read, or a newline character is read and transferred to *s*, or an end-of-file condition is encountered. The string is then terminated with a null character.

DIAGNOSTICS

If end-of-file is encountered and no characters have been read, no characters are transferred to *s* and a null pointer is returned. If a read error occurs, a null pointer is returned. Otherwise *s* is returned. (A read error will occur, for example, if you attempt to use these functions on a file that has not been opened for reading.)

SEE ALSO

ferror, fopen, fread, getc, scanf, stdio.

NAME

hypot—Euclidean distance function

SYNOPSIS

```
#include <math.h>

extended hypot (x, y)
extended x, y;
```

DESCRIPTION

Function *hypot* returns

sqrt (x * x + y * y)

taking precautions against unwarranted overflows.

DIAGNOSTICS

Function *hypot* honors the floating-point exception flags—invalid operation, underflow, overflow, divide by zero, and inexact—as prescribed by the Standard Apple Numeric Environment (SANE).

SEE ALSO

sqrt.
Apple Numerics Manual.

NAME

ioctl—control a device

SYNOPSIS

```
#include <ioctl.h>

int ioctl (fildes, cmd, arg)
int filedes;
unsigned int cmd;
long *arg;
```

DESCRIPTION

Function *ioctl* communicates with a file's device handler by sending control information and/or requesting status information. Variable *cmd* indicates which device-specific operations *ioctl* must perform. Here are the control values:

- FIOINTERACTIVE returns 0 if the device is interactive, –1 if not.
- FIOBUFSIZE returns, in bytes, the optimal buffer size for this device; the buffer size is in a long integer pointed to by *arg*.
- TIOFLUSH tells the device handler to throw away terminal input.
- FIOLSEEK and FIODUPFD are reserved for operating system use.

Function *ioctl* fails if one or more of the following are true:

- File descriptor *fildes* is not valid or is not open . [EBADF]
- Variables *request* or *arg* are not valid for the device handler associated with *fildes*. [EINVAL]

RETURN VALUE

If an error has occurred, a value of –1 is returned and *errno* is set to indicate the error.

NAME

lseek—move read/write file pointer

SYNOPSIS

```
long lseek (fildes, offset, whence)
int fildes;
long offset;
int whence;
```

DESCRIPTION

A file descriptor, *fildes*, is returned from a *creat* or *open* call. Function *lseek* sets the file pointer associated with *fildes* as follows:

If *whence* is 0, the pointer is set to *offset* bytes.

If *whence* is 1, the pointer is set to its current location plus *offset*.

If *whence* is 2, the pointer is set to the size of the file plus *offset*.

Upon successful completion, the resulting pointer location as measured in bytes from the beginning of the file is returned.

The file pointer remains unchanged and *lseek* fails if one or more of the following are true:

File descriptor*fildes* is not open. [EBADF]

Variable*whence* is not 0, 1, or 2. [EINVAL]

The resulting file pointer would be negative. [EINVAL]

Some devices are incapable of seeking. The value of the file pointer associated with such a device is undefined.

RETURN VALUE

Upon successful completion, a nonnegative integer indicating the file pointer value is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NOTE

In previous versions of the Standard C Library, *tell(filedes)* was a function that returned the current file position. It is equivalent to the call

lseek(fildes,0L,1).

SEE ALSO

creat, open.

NAME

malloc, free, realloc, calloc, cfree—main memory allocator

SYNOPSIS

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

void free(ptr)
char *ptr;

char *realloc(ptr, size)
char *ptr;
unsigned size;

char *calloc(nelem, elsize)
unsigned nelem, elsize;

cfree *** ? ***
```

DESCRIPTION

Functions *malloc* and *free* provide a simple general-purpose memory allocation package. The memory that is allocated for a program's use is known as the "storage arena." The storage arena expands as *malloc* is called.

Function *malloc* returns a pointer to a block of at least *size* bytes suitably aligned for any use. The argument to *free* is a pointer to a block previously allocated by *malloc*; after *free* is performed this space is made available for further allocation.

Undefined results occur if the space assigned by *malloc* is overrun or if some random value is handed to *free*.

Function *malloc* allocates the first sufficiently large contiguous reach of free space it finds. It calls **** procedure name? **** (see *Cortland Tools*) to get more memory from the system when there is no suitable space already free.

Function *realloc* changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents are unchanged up to the lesser of the new and old sizes. If no free block of *size* bytes is available in the storage arena, *realloc* asks *malloc* to enlarge the storage arena by *size* bytes and then moves the data to the new space. If *ptr* is null, *realloc* is equivalent to *malloc*.

Function *calloc* allocates space for an array of *nelem* elements of size *elsize*. The space is initialized to zeros.

Function *cfree* .... **** ? ****

DIAGNOSTICS

Functions *malloc* , *realloc*, and *calloc* return a null pointer if there is no available memory or if the storage arena has been detectably corrupted by storing outside the

bounds of a block. When this happens the block pointed to by *ptr* may have been destroyed.

NAME

memccpy, memchr, memcmp, memcpy, memset—memory operations

SYNOPSIS

```
char *memccpy(s1, s2, c, n)
char *s1, *s2;
int c, n;

char *memchr(s, c, n)
char *s;
int c, n;

int memcmp(s1, s2, n)
char *s1, *s2;
int n;

char *memcpy(s1, s2, n)
char *s1, *s2;
int n;

char *memset(s, c, n)
char *s;
int c, n;
```

DESCRIPTION

These functions operate efficiently on memory areas (arrays of characters bounded by a count, not terminated by a null character). They do not check for the overflow of any receiving memory area.

Function *memccpy* copies characters from memory area *s2* into *s1*, stopping after the first occurrence of character *c* has been copied or after *n* characters have been copied, whichever comes first. It returns either a pointer to the character after the copy of *c* in *s1* or a null pointer if *c* was not found in the first *n* characters of *s2*.

Function *memchr* returns either a pointer to the first occurrence of character *c* in the first *n* characters of memory area *s* or a null pointer if *c* does not occur.

Function *memcmp* compares its arguments, looking at the first *n* characters only. It returns an integer less than, equal to, or greater than 0, depending on whether *s1* is less than, equal to, or greater than *s2* in the ASCII collating sequence.

Function *memcpy* copies *n* characters from memory area *s2* to *s1*. It returns *s1*.

Function *memset* sets the first *n* characters in memory area *s* to the value of character *c*. It returns *s*.

WARNING

Overlapping moves may yield unexpected results.

NAME

onexit—register a function for program termination

SYNOPSIS

```
#include <stdio.h>

int onexit (func);
void (*func)();
```

DESCRIPTION

Function *onexit* registers the exit function pointed to by *func*, to be called without arguments at program termination.

The number of exit functions is limited to *** ? ***.

RETURN VALUES

The function returns a nonzero value if the registration succeeds.

WARNING

If any function is registered more than once, the behavior is undefined.

SEE ALSO

exit.

NAME

open—open for reading or writing

SYNOPSIS

```
#include <fcntl.h>

int open(path, oflag)
char *path;
int oflag;
```

DESCRIPTION

Variable *path* points to a pathname naming a file. Function *open* opens a file descriptor for the named file and sets the file status flags according to the value of *oflag*. The value of *oflag* is constructed by or-ing flag settings; for example,

```
open("MyFile", O_WRONLY|O_CREAT|O_TRUNC);
```

To construct *oflag*, first select one of the following settings:

O_RDONLY	Open for reading only.
O_WRONLY	Open for writing only.
O_RDWR	Open for reading and writing.

Then optionally add one or more of these additional settings:

O_APPEND	The file pointer is set to the end of the file prior to each write.
O_CREAT	If the file does not exist, it is created.
O_TRUNC	If the file exists, its length is truncated to 0; the mode and owner are unchanged.
O_RSRC	The file's resource fork is opened. (Normally the data fork is opened.)
O_SELECT	I/O is restricted to a subset of the file (currently, the selection in a window).

The following setting is valid only if O_CREAT is also specified:

O_EXCL	Function *open* fails if the file exists.

Upon successful completion a nonnegative integer, the file descriptor, is returned. The file pointer used to mark the current position within the file is set to the beginning of the file.

The named file is opened unless one or more of the following are true:

- O_CREAT is not set and the named file does not exist. [ENOENT]
- More than about 30 file descriptors are currently open. The actual limit varies according to runtime conditions. [EMFILE]
- O_CREAT and O_EXCL are set and the named file exists. [EEXIST]

RETURN VALUE

Upon successful completion, a nonnegative integer (the file descriptor) is returned; otherwise, a value of −1 is returned and *errno* is set to indicate the error.

SEE ALSO

close, creat, lseek, read, write.

NAME

printf, fprintf, sprintf—print formatted output

SYNOPSIS

```
#include <stdio.h>

int printf(format [ , arg ] ... )
char *format;

int fprintf(stream, format [ , arg ] ... )
FILE *stream;
char *format;

int sprintf(str, format [ , arg ] ... )
char *str, format;
```

DESCRIPTION

Function *printf* places formatted output on the standard output stream *stdout*. Function *fprintf* places formatted output on the named output stream. Function *sprintf* places formatted output, followed by the null character (\0), into the character array pointed to by *str*; it's your responsibility to ensure that enough storage is available. Each function returns the number of characters transmitted (not including the \0 in the case of *sprintf*), or a negative value if an output error was encountered.

Each of these functions converts, formats, and prints its args under control of the format. The format is a character string that contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which results in fetching zero or more args. The results are undefined if there are insufficient args for the format. If the format is exhausted while args remain, the extra args are ignored.

Each conversion specification is introduced by the character %. After %, the following appear in sequence:

- Zero or more flag characters, which modify the meaning of the conversion specification.
- An optional decimal digit string specifying a minimum field width. If the converted value has fewer characters than the field width, it will be padded to the field width on the left (default) or right (if the left-adjustment flag has been given); see below for flag specification.
- A precision that gives the minimum number of digits to appear for the d, o, u, x, or X conversions, the number of digits to appear after the decimal point for the e, E, and f conversions, the maximum number of significant digits for the g and G conversions, or the maximum number of characters to be printed from a string in s conversion. The format of the precision is a period (.) followed by a decimal digit string; a null digit string is treated as zero.

- An optional l specifying that a following d, o, u, x, or X conversion character applies to a long-integer arg. The l option is ignored in this implementation since integers and long integers both require 32 bits.
- A character that indicates the type of conversion to be applied.

A field width or precision may be indicated by an asterisk (*) instead of a digit string. In this case, an integer arg supplies the field width or precision. The arg that is actually converted is not fetched until the conversion letter is seen; therefore, the args specifying field width or precision must appear immediately before the arg (if any) to be converted.

The flag characters and their meanings are:

–	The result of the conversion will be left-justified within the field.
+	The result of a signed conversion always begins with a sign (+ or –).
blank	If the first character of a signed conversion is not a sign, a blank will be prefixed to the result. This implies that if the blank and + flags both appear, the blank flag will be ignored.
#	This flag specifies that the value is to be converted to an "alternate form." For c, d, s, and u conversions, the flag has no effect. For o conversion, it increases the precision to force the first digit of the result to be a zero. For x (X) conversion, a nonzero result will have "0x" ("0X") prefixed to it. For e, E, f, g, and G conversions, the result will always contain a decimal point, even if no digits follow the point. (Normally, a decimal point appears in the result of these conversions only if a digit follows it.) For g and G conversions, trailing zeros in the fractional part will not be removed from the result (as they normally are).

The conversion characters and their meanings are:

d,o,u,x,X	The integer arg is converted to signed decimal (d), unsigned octal (o), decimal (u), or hexadecimal notation (x and X), respectively; the letters "abcdef" are used for x conversion and the letters "ABCDEF" for X conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is a null string.
f	The float, double, comp, or extended arg is converted to decimal notation in the style "[–]ddd.ddd", where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, it is assumed to be 6; if the precision is explicitly 0, no decimal point appears. Infinities are printed as "[–]INF" and NaNs are printed as "[–]NAN(ddd)" where ddd is a code indicating why the result is not a number.
e,E	The float, double, comp, or extended arg is converted in the style "[–]d.dddE±dd", where there is one digit before the decimal point and the number of digits after it is equal to the precision; when the precision is missing, it is assumed to be 6; if the precision is zero,

no decimal point appears. The E format code produces a number with "E" instead of "e" introducing the exponent. The exponent always contains at least two digits. Infinities are printed as "INF" and NaNs are printed as "[-]NAN(ddd)", where ddd is a code indicating why the result is not a number.

g,G — The float, double, comp, or extended arg is printed in style f or e (or in style E in the case of a G format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style e is used only if the exponent resulting from the conversion is less than –4 or greater than the precision. Trailing zeros are removed from the result. A decimal point appears only if it is followed by a digit.

c — The character arg is printed.

s — The arg is taken to be a string (character pointer) and characters from the string are printed until a NULL character (\0) is encountered or the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first null character are printed. If the string pointer arg has the value zero, the result is undefined. A null arg yields undefined results.

% — Print a %; no argument is converted. In no case does a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by *printf* and *fprintf* are printed as if *putc* had been called.

EXAMPLES

To print a date and time in the form "Sunday, July 3, 10:02", where weekday and month are pointers to null-terminated strings:

```
printf("%s, %s %d, %.2d:%.2d", weekday, month, day, hour, min);
```

To print pi to five decimal places:

```
printf("pi = %.5f", pi());
```

SEE ALSO

dec2str, ecvt, num2dec, putc, scanf, stdio.

NAME

putc, putchar, fputc, putw—put character or word on a stream

SYNOPSIS

```
#include <stdio.h>

int putc(c, stream)
char c;
FILE *stream;

int putchar(c)
char c;

int fputc(c, stream)
char c;
FILE *stream;

int putw(w, stream)
int w;
FILE *stream;
```

DESCRIPTION

Macro *putc* writes the character *c* to the output stream at the position pointed to by the file pointer, if one is defined. Macro *putchar(c)* is equivalent to

```
putc(c, stdout).
```

Function *fputc* behaves like macro *putc*. Function *fputc* runs more slowly than *putc* but takes less space per invocation.
Function *putw* writes the "word" *w* (i.e., four bytes) to the output stream at the position pointed to by the file pointer, if one is defined. This function neither assumes nor causes special alignment in the file.

Output streams, with the exception of the standard error stream *stderr*, are by default buffered if the output refers to a file and line-buffered if the output refers to a window. File *stderr* is by default unbuffered, but use of *freopen* causes it to become buffered or line-buffered. When an output stream is unbuffered information, it is queued for writing on the destination file or window as soon as written; when it is buffered, many characters are saved up and written as a block; when it is line-buffered, each line of output is queued for writing on the destination window as soon as the line is completed (i.e., as soon as a newline character is written or terminal input is requested). Function *setbuf* may be used to change the stream's buffering strategy.

DIAGNOSTICS

On success, these functions each return the value they have written. On failure, they return the constant EOF. This occurs if the file stream is not open for writing or if the output file cannot be grown. Because EOF is a valid integer, *ferror* should be used to detect *putw* errors.

NOTE

Because it is implemented as a macro, *putc* treats incorrectly a stream argument with side effects. In particular, `putc(c, *f++);` doesn't work as you would expect. Function *fputc* should be used instead.

SEE ALSO

fclose, ferror, fopen, fread, getc, printf, puts, setbuf, stdio.

NAME

puts, fputs—write a string to a stream

SYNOPSIS

```
#include <stdio.h>

int puts(s)
char *s;

int fputs(s, stream)
char *s;
FILE *stream;
```

DESCRIPTION

Function *puts* writes the null-terminated string pointed to by *s*, followed by a newline character, to the standard output stream stdout.

Function *fputs* writes the null-terminated string pointed to by *s* to the named output stream.

Neither function writes the terminating null character.

DIAGNOSTICS

Both routines return EOF on error. This occurs if the routines try to write on a file that has not been opened for writing.

NOTE

Function *puts* appends a newline character while *fputs* does not.

SEE ALSO

ferror, fopen, fread, printf, putc, stdio.

NAME

qsort—quicker sort

SYNOPSIS

```
void qsort ((char *) base, nel, sizeof (*base), compar)
unsigned int nel;
int (*compar ( );
```

DESCRIPTION

Function *qsort* is an implementation of the quicker-sort algorithm. It sorts a table of data in place.

Variable *base* points to the element at the base of the table. Variable *nel* is the number of elements in the table. Variable *compar* is the name of the comparison function, which is called with two arguments that point to the elements being compared. The function returns an integer value as follows:

Function Result	Meaning
< 0	The first argument is less than the second argument.
0	The first argument is equal to the second argument.
> 0	The first argument is greater than the second argument.

NOTE

The pointer to the base of the table should be of type pointer-to-element, and cast to type pointer-to-character. The value returned should be cast into type pointer-to-element.

The comparison function ignores data in the table that is not part of the elements being compared.

NAME

rand, srand—simple random-number generator

SYNOPSIS

```
int rand( )

void srand(seed)
unsigned seed;
```

DESCRIPTION

Function *rand* uses a multiplicative congruential random-number generator with period 2^{32} that returns successive pseudorandom numbers in the range from 0 to $2^{15}-1$.

Function *srand* can be called at any time to reset the random-number generator to a specific seed. The generator is initially seeded with a value of 1.

SEE ALSO

randomx.

NAME

read—read from file

SYNOPSIS

```
int read(fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;
```

DESCRIPTION

File descriptor *fildes* is obtained from a *creat* or *open* call.

Function *read* attempts to read *nbyte* bytes from the file associated with *fildes* into the buffer pointed to by *buf*.

On devices capable of seeking, the read starts at a position in the file given by the file pointer associated with *fildes*. Upon return from *read*, the file pointer is incremented by the number of bytes actually read.

Nonseeking devices always read from the current position. The value of a file pointer associated with such a file is undefined.

Upon successful completion, *read* returns the number of bytes actually read and placed in the buffer; this number may be less than *nbyte* if the file is associated with a window or if the number of bytes left in the file is less than *nbyte* bytes. A value of 0 is returned when an end-of-file has been reached.

Function *read* fails if *fildes* is not a valid file descriptor open for reading. [EBADF]

RETURN VALUE

Upon successful completion a nonnegative integer is returned indicating the number of bytes actually read. Otherwise, −1 is returned and *errno* is set to indicate the error.

SEE ALSO

creat, open, stdio.

NAME

scanf, fscanf, sscanf—convert formatted input

SYNOPSIS

```
#include <stdio.h>

int scanf(format [ , pointer ] ... )
char *format;

int fscanf(stream, format [ , pointer ] ... )
FILE *stream;
char *format;

int sscanf(s, format [ , pointer ] ... )
char *s, *format;
```

DESCRIPTION

Function *scanf* reads characters from the standard input stream *stdin*. Function *fscanf* reads characters from the named input stream, *stream*. Function *sscanf* reads characters from the character string *s*. Each function converts the input according to a control string *(format)* and stores the results according to a set of *pointer* arguments that indicate where the converted output should be stored.

Function *format*, the control string, contains specifications that control the interpretation of input sequences. The control string consists of characters to be matched in the input stream and/or conversion specifications that start with %. The control string may contain:

- White-space characters (blanks and tabs) that cause input to be read up to the next non-white-space character, except as described below.
- A character (any except %) that must match the next character of the input stream. To match a % character in the input stream, use "%%".
- Conversion specifications beginning with the character % and followed by an optional assignment suppression character *, an optional numeric maximum field width, an optional l, m, n, or h indicating the size of the receiving variable, and a conversion code.

An input field is defined relative to its conversion specification. The input field ends when the first character inappropriate for conversion is encountered or when the specified field width is exhausted. After conversion, the input pointer points to the inappropriate character.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, which is a pointer to a basic C type such as int or float.

Assignment can be suppressed by preceding a format character with *. Assignment suppression causes an input field to be skipped; the field is read and

converted but not assigned. Therefore *pointer* should be omitted when assignment of the corresponding input field is suppressed.

The format character dictates the interpretation of the input field. The following format characters are legal in a conversion specification, after %:

% A single % is expected in the input at this point; no assignment is done.

The conversion characters d, u, o, and x may be preceded by l or h to indicate that a pointer to long or short, rather than int, is in the argument list. The l is ignored in this implementation because ints and longints are both 32 bits.

d A decimal integer is expected; the corresponding argument should be an integer pointer.

u An unsigned decimal integer is expected; the corresponding argument should be an unsigned integer pointer.

o An octal integer is expected; the corresponding argument should be an integer pointer.

x A hexadecimal integer is expected; the corresponding argument should be an integer pointer.

The conversion characters e, f, and g may be preceded by l, m, or n to indicate that a pointer to double, comp, or extended, rather than float, is in the argument list.

e,f,g A floating-point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a float, double, comp, or extended, depending on the size specification. The input format for floating-point numbers is an optionally signed string of digits, possibly containing a decimal point, followed by an optional exponent field consisting of "E" or "e" followed by an optionally signed integer. In addition, infinity is represented by the string "INF", and NaNs are represented by the string "NAN", optionally followed by parentheses which may contain a string of digits (the NaN code). Case is ignored in the infinity and NaN strings.

s A character string is expected; the corresponding argument should be a character pointer to an array of characters large enough to accept the string; a terminating null character (\0) is added automatically. The input field is terminated by a white-space (blank or tab) character.

c A character is expected; the corresponding argument should be a character pointer. The normal skip over white space is suppressed in this case; to read the next non-white-space character, use "%1s". If a field width is given, the corresponding argument should refer to a character array; the indicated number of characters is read.

[The left bracket introduces a scanset format. The input field is the maximal sequence of input characters consisting entirely of characters in the scanset. When reading the input field, string data and the normal skip over leading white space are suppressed. The corresponding pointer argument must point to a character array large enough to hold the input field and the terminating null character (\0), which will be added automatically. The left bracket is followed by a set of characters (the scanset) and a terminating right bracket.

The circumflex (^), when it appears as the first character in the scanset, serves as a complement operator and redefines the scanset as the set of all characters not contained in the remainder of the scanset string.

The right bracket (]) ends the scanset. To include the right bracket as an element of the scanset, it must appear as the first character (possibly preceded by a circumflex) of the scanset; otherwise it will be interpreted syntactically as the closing bracket.

A range of characters may be represented by the construct *first-last*; thus the scanset [0123456789] may be expressed [0-9]. To use this convention, *first* must be less than or equal to *last* in the ASCII collating sequence; otherwise the minus (–) will stand for itself in the scanset. The minus will also stand for itself whenever it is the first or the last character in the scanset.

Function *scanf* conversion terminates at EOF, at the end of the control string, or when an input character doesn't match the control string. In the latter case, the unmatched character is left unread in the input stream.

Function *scanf* returns the number of successfully matched and assigned input items; this number can be zero when an early mismatch between an input character and the control string occurs. If the input ends before the first mismatch or conversion, EOF is returned.

EXAMPLES

Example 1. The call

```
int i; float x; char name[50];
scanf("%d%f%s", &i, &x, name);
```

with input

```
25 54.32E-1 hartwell
```

will assign the value 25 to *i*, and the value 5.432 to *x*; *name* will contain "hartwell\0".

Example 2. The call

```
int i; extended x; char name[50];
scanf("%2d%nf%*d %[0-9]", &i, &x, name);
```

with input

```
56789 0123 56a72
```

will assign 56 to *i*, 789.0 to *x*, skip 0123, and place the string "56\0" in *name*. The next call to *getchar* will return "a".

Example 3. The call

```
int i;
scanf("answer1=%d", &i);
```

with input

```
answer1=51 answer2=45
```

will assign the value 51 to *i* since "answer1" is matched explicitly in the input stream; the input pointer will be left at the space before "answer2".

DIAGNOSTICS

These functions return EOF on end of input and a short count for missing or illegal data items.

NOTE

Trailing white space is left unread unless matched in the control string.

The success of literal matches and suppressed assignments is not directly determinable.

SEE ALSO

atof, dec2num, getc, printf, stdio, str2dec, strtol.
Apple Numerics Manual.

NAME

setbuf, setvbuf—assign buffering to a stream

SYNOPSIS

```
#include <stdio.h>

void setbuf (stream, buf)
FILE *stream;
char *buf;

int setvbuf (stream, buf, type, size)
FILE *stream;
char *buf;
int type;
int size;
```

DESCRIPTION

Function *setbuf* is used after a stream has been opened but before it is read or written. Function *setbuf* causes the character array pointed to by *buf* to be used instead of an automatically allocated buffer. If *buf* is a null character pointer, input/output will be completely unbuffered.

BUFSIZ, a constant defined in the <stdio.h> header file, indicates how big an array is needed:

```
char buf[BUFSIZ];
```

A buffer is normally obtained from *malloc* at the time of the first *getc* or *putc* on the file, except that the standard error stream *stderr* is normally not buffered. Output streams directed to windows are either line buffered or unbuffered.

Function *setvbuf* is used after a stream has been associated with an open file but before it is read or written. If *buf* is not a null pointer, the array it points to is used instead of an automatically allocated buffer. Variable *size* specifies the size in bytes of the array to be used; *setvbuf* works most efficiently when *size* is a multiple of BUFSIZE.

Variable *type* determines how *stream* is buffered:

- _IOFBF causes input/output to be file buffered.
- _IOLBF causes output to be line buffered. The buffer is flushed when a newline character is written, when the buffer is full, or when input is requested.
- _IONBF causes input/output to be completely unbuffered. Variables *buf* and *size* are ignored.

RETURN VALUE

Function *setvbuf* returns nonzero if an invalid value is given for *type* or *size*.

NOTE

The buffer must have a lifetime at least as great as the open stream. Be sure to close the stream before the buffer is deallocated.

If you allocate buffer space as an automatic variable in a code block, be sure to close the stream in the same block.

If *buf* is null and the system cannot allocate *size* bytes, a smaller buffer will be allocated.

SEE ALSO

fopen, getc, malloc, putc.

NAME

sigset, sighold, sigrelease, sigpause—signal handling

SYNOPSIS

```
#include <Signal.h>

SignalHandler * sigset (sigMap, newHandler)
SignalMap sigMap;
SignalHandler *newHandler;

void _sig_dfl (sigNo, sigState, sigEnabled)
SignalMap sigNo;
SignalMap sigState;
SignalMap sigEnabled;

SignalMap sighold (sigMap)
SignalMap sigMap;

void sigrelease (sigMap, prevEnabled)
SignalMap sigMap;
SignalMap prevEnabled;

void sigpause (sigMap)
SignalMap sigMap;
```

DESCRIPTION

*** Of all the routines in the Standard C Library, these are the most likely to differ from their Macintosh counterparts. Please let me know of any differences so that I can change this description—DR ***

Introduction to Signal Handling: C programs that provide procedures to handle software interrupts—known as signals—should use these procedures, which support signal handling under the Cortland Programmer's Workshop. A signal is similar to a hardware interrupt in that its invocation can cause program control to be temporarily diverted from its normal execution sequence; the difference is that the events that raise a signal reflect a change in program state rather than hardware state. Examples of signal events are stack overflow, heap overflow, software floating point errors, and Command-period interrupts.

The <Signal.h> include file defines the constants, types, and procedure definitions for handling signals.

Signals can be caught, held and released, and/or ignored. The default action of any signal raised is to close all open files, execute any exit procedures installed with *onexit*, and terminate the program. No signal-handling calls are required to execute a normal termination on receipt of a signal. If a program requires special handling of a signal, or chooses to ignore it, *sigset* lets you replace the default procedure with a user procedure. You can also temporarily "hold" (that is, suspend) action on a signal by calling *sighold*. You may want to do this before entering a critical section of code. The signal can then be restored by calling the procedure *sigrelease*, whereupon its signal-handling procedure will take effect if the signal was raised

since the preceding call to *sighold*. Your program may also wait until one or more signals are raised by calling the *sigpause* procedure.

A signal is represented by a bit in the integer *SigMap*. The signal-handling procedures accept a *SigMap* which can specify several signals as a group. Refer to several signals at once by adding or or-ing their bits together. Refer to all signals at once by using the value SIGALLSIGS.

Currently, the only software interrupt provided by the Cortland Programmer's Workshop Integrated Environment is Command-period, which is represented by the value SIGINT. As additional software interrupts are provided by the Integrated Environment, new values will be added to this unit to represent them; the signal-handling procedures will then accept these new signals.

The sigset Function: Function *sigset* replaces the current signal handler (the procedure to be executed upon receipt of the signals specified in *sigMap*) with a user-supplied signal handler. The default signal handler may be set or restored by specifying SIG_DFL to the current signal handler. The signals may be ignored entirely by specifying SIG_IGN to the current signal handler.

Function *sigset* returns the previous *SignalHandler* pointer. If this pointer must be restored in another part of the program, save the return value and restore it with another call to *sigset*. Multiple signals may be set with one call to *sigset* by or-ing signal values together in *sigMap*, but in this case *sigset* cannot, of course, return all previous values and its return value is meaningless. To correctly save multiple previous signal handlers, call *sigset* separately for each signal.

The _sig_dfl Function: This is the default procedure SIG_DFL; it is not intended for use by the program directly. It is documented here as an example of a user-supplied signal handler that uses standard C calling conventions.

The first parameter, *sigNo*, is the signal that is being raised. Although it is declared as a *SigMap*, its value contains at most one signal bit; it can therefore be compared for equality against a signal name, for example, SIGINT. The same signal handler may trap several signals with common code and then inspect *sigNo* if special handling of particular signals is required.

The parameters *sigState* and *sigEnabled* provide runtime information about current active signals. Bitmap *sigState* describes all raised signals, including signals held by calls to *sighold*. Bitmap *sigEnabled* describes all signals currently enabled. By default, all signals are enabled, but they may be disabled by holding them.

Upon entry to a user-supplied signal handler, all signals are temporarily suspended; therefore the handler is not required to lock out recursive or nested calls to signal handlers. The signal state is restored upon normal return from the signal handler.

Signals cannot be raised while executing in ROM or in the Cortland Programmer's Workshop shell. If a signal event occurs while executing outside the user application, the signal state is set and the signal handler is executed as soon as program control returns to the application code. Since a signal can interrupt the application program at any point, there is no protection against heap corruption if a signal handler executes calls that modify the state of the heap. Since most buffered I/O potentially modifies the heap, *printf* and similar calls are not recommended in signal handlers unless they call *exit* to avoid returning to the application program.

Even then, the caller must be careful of interaction between *exit* and *onexit* procedures.

The sighold Function: The *sighold* function, along with *sigrelease*, permits temporary suspension and restoration of signals. Before a program enters a critical section of code, it should call *sighold* with a signal map of signals to suspend or with the identifier SIGALLSIGS, which represents all signals. Function *sighold* returns a *SignalMap* representing the list of signals already being held; this value should be saved for use as the *prevEnabled* parameter in the subsequent call to *sigrelease*. If the signal event (such as Command-period) occurs after a call to *sighold* is made, the event is recorded in the signal state but the signal handler is not executed.

The sigrelease Function: Funcion *sigrelease* lets you reenable signals that were held by a previous call to *sighold* by specifying their corresponding bits in *sigMap*. Signals that were already on hold when you called *sighold* should be specified to *sigrelease* in the *prevEnabled* parameter to permit correct handling of nested calls to *sighold*. If any of the signal events occured while they were held, their signal handling routines will take effect immediately after the return from *sigrelease*. Signal events do not stack; multiple occurrences of signal events which are being held do not yield multiple invocations of the signal handler when the signal is released.

The sigpause Function: A call to *sigpause* suspends program activity until a signal event is recorded for any signal not currently held. It is intended for signal synchronization, though in the current implementation its application is limited; it is included here in order to provide a complete signal environment model.

NAME

sinh, cosh, tanh—hyperbolic functions

SYNOPSIS

```
#include <math.h>

extended sinh (x)
extended x;

extended cosh (x)
extended x;

extended tanh (x)
extended x;
```

DESCRIPTION

Functions *sinh*, *cosh*, and *tanh* return, respectively, the hyberbolic sine, cosine, and tangent of their argument.

DIAGNOSTICS

Functions *sinh*, *cosh*, and *tanh* honor the floating-point exception flags—invalid operation, underflow, overflow, divide by zero, and inexact—as prescribed by the Standard Apple Numeric Environment (SANE).

SEE ALSO

Apple Numerics Manual.

NAME

stdio—standard buffered input/output package

SYNOPSIS

```
#include <stdio.h>

FILE *stdin, *stdout, *stderr;
```

DESCRIPTION

The standard I/O package constitutes an efficient user-level I/O buffering scheme. The inline macros *getc* and *putc* handle characters quickly. Macros *getchar* and *putchar*, and the higher-level routines *fgetc*, *fgets*, *fprintf*, *fputc*, *fputs*, *fread*, *fscanf*, *fwrite*, *gets*, *getw*, *printf*, *puts*, *putw*, and *scanf* all use *getc* and *putc*; they can be freely intermixed.

Any program that uses the standard I/O package must include the header file of pertinent macro definitions. The functions and constants mentioned in the standard I/O package are declared in the header file and need no further declaration. The header file is included as follows:

```
#include <stdio.h>
```

A file with associated buffering is called a *stream* and is declared to be a pointer to a defined type FILE. Function *fopen* creates certain descriptive data for a stream and returns a pointer to designate the stream in all further transactions. Normally, there are three open streams with constant pointers declared in the <stdio.h> header file and associated with the standard open files:

stdin	(standard input file)
stdout	(standard output file)
stderr	(standard error file)

A constant NULL (0) designates a nonexistent pointer.

An integer constant EOF (–1) is returned upon end-of-file or error by most integer functions that deal with streams. See the descriptions of the individual functions for details.

The constants and the following functions are implemented as macros: *getc*, *getchar*, *putc*, *putchar*, *feof*, *ferror*, *clearerr*, and *fileno*. Redeclaration of these names should be avoided.

NOTE

File <stdio.h> includes definitions other than those described above, but their use is not recommended.

DIAGNOSTICS

Invalid stream pointers cause serious errors, possibly including program termination. Individual function descriptions describe the possible error conditions.

SEE ALSO

open, close, lseek, read, write, fclose, ferror, fopen, fread, fseek, getc, gets, printf, putc, puts, scanf, setbuf, ungetc.

NAME

strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen,strchr, strrchr, strpbrk, strspn, strcspn, strtok
—string operations

SYNOPSIS

```
char *strcat (s1, s2)
char *s1, *s2;

char *strncat (s1, s2, n)
char *s1, *s2;
int n;

int strcmp (s1, s2)
char *s1, *s2;

int strncmp (s1, s2, n)
char *s1, *s2;
int n;

char *strcpy (s1, s2)
char *s1, *s2;

char *strncpy (s1, s2, n)
char *s1, *s2;
int n;

int strlen (s)
char *s;

char *strchr (s, c)
char *s, c;

char *strrchr (s, c)
char *s, c;

char *strpbrk (s1, s2)
char *s1, *s2;

int strspn (s1, s2)
char *s1, *s2;

int strcspn (s1, s2)
char *s1, *s2;

char *strtok (s1, s2)
char *s1, *s2;
```

DESCRIPTION

The arguments *s1*, *s2*, and *s* point to strings (arrays of characters terminated by a null character). The functions *strcat*, *strncat*, *strcpy*, and *strncpy* all alter *s1*. These functions do not check for overflow of the array pointed to by *s1*.

Function *strcat* appends a copy of string *s2* to the end of string *s1*. Function *strncat* appends at most *n* characters. Each function returns a pointer to the null-terminated result.

Function *strcmp* performs a comparison of its arguments according to the ASCII collating sequence and returns an integer less than, equal to, or greater than 0 when *s1* is less than, equal to, or greater than *s2*, respectively. Function *strncmp* makes the same comparison but looks at a maximum of *n* characters.

Function *strcpy* copies string *s2* to string *s1*, stopping after the null character has been copied. Function *strncpy* copies exactly *n* characters, truncating *s2* or adding null characters to *s1* if necessary. The result is not null-terminated if the length of *s2* is *n* or more. Each function returns *s1*.

Function *strlen* returns the number of characters in *s*, not including the terminating null character.

Function *strchr* (*strrchr*) returns a pointer to the first (last) occurrence of character *c* in string *s*; it returns a null pointer if *c* does not occur in the string. The null character terminating a string is considered to be part of the string. In previous versions of the Standard C Library, *strchr* was known as *index* and *strrchr* was known as *rindex*.

Function *strpbrk* returns a pointer to the first occurrence in string *s1* of any character from string *s2*, or a null pointer if no character from *s2* exists in *s1*.

Function *strspn* returns the length of the initial segment of string *s1* that consists entirely of characters from string *s2*.

Function *strcspn* returns the length of the initial segment of string *s1* that consists entirely of characters not from string *s2*.

Function *strtok* considers the string *s1* as a sequence of zero or more text tokens separated by spans of one or more characters from the separator string *s2*. The first call (with pointer *s1* specified) returns a pointer to the first character of the first token and writes a null character into *s1* immediately folloing the returned token. The function keeps track of its position in the string between calls. Subsequent calls for the same string must be made with a null pointer as the first argument. The separator string *s2* may be different from call to call. When no token remains in *s1*, a null pointer is returned.

WARNING

Overlapping moves may yield unexpected results.

NAME

strtol—convert string to integer

SYNOPSIS

```
long strtol (str, ptr, base)
char *str;
char **ptr;
int base;
```

DESCRIPTION

Function *strtol* returns a long integer containing the value represented by the character string *str*. The string is scanned up to the first character inconsistent with the base (decimal, hexadecimal, or octal). Leading white-space characters are ignored.

If the value of *ptr* is not null, a pointer to the character terminating the scan is returned in **ptr*. If no integer can be formed, **ptr* is set to *str* and zero is returned.

If *base* is zero, the base is determined from the string. If the first character after an optional leading sign is not 0, decimal conversion is done; if the 0 is followed by x or X, hexadecimal conversion is done; otherwise octal conversion is done.

The function call *atol(str)* is equivalent to

```
strtol (str, (char **)NULL, 10)
```

The function call *atoi(str)* is equivalent to

```
(int) strtol (str, (char **)NULL, 10)
```

NOTE

Overflow conditions are ignored.

Apple base conventions ($ for hexadecimal, % for binary) are not supported.

SEE ALSO

atof, atoi, scanf.

NAME

sin, cos, tan, asin, acos, atan, atan2—trigonometric functions

SYNOPSIS

```
#include <math.h>

extended sin (x)
extended x;

extended cos (x)
extended x;

extended tan (x)
extended x;

extended asin (x)
extended x;

extended acos (x)
extended x;

extended atan (x)
extended x;

extended atan2 (y, x)
extended x, y;
```

DESCRIPTION

Functions *sin*, *cos*, and *tan* return, respectively, the sine, cosine, and tangent of their argument, which is in radians.

Function *asin* returns the arcsine of x, in the range $-\pi/2$ to $\pi/2$.

Function *acos* returns the arccosine of x, in the range 0 to π.

Function *atan* returns the arctangent of x, in the range $-\pi/2$ to $\pi/2$.

Function *atan2* returns the arctangent of y/x, in the range $-\pi$ to π, using the signs of both arguments to determine the quadrant of the return value.

For special cases, these functions return a NaN or infinity as appropriate.

DIAGNOSTICS

These functions honor the floating-point exception flags—invalid operation, underflow, overflow, divide by zero, and inexact—as prescribed by the Standard Apple Numeric Environment (SANE).

NOTE

Functions *sin*, *cos*, and *tan* have periods based on the nearest extended-precision representation of mathematical π. Hence these functions diverge from their mathematical counterparts as their argument becomes far from zero.

SEE ALSO

Apple Numerics Manual.

NAME

ungetc—push character back into input stream

SYNOPSIS

```
#include <stdio.h>

int ungetc (c, stream)
char c;
FILE *stream;
```

DESCRIPTION

Function *ungetc* inserts the character *c* into the buffer associated with an input stream. That character, *c*, will be returned by the next *getc* call on that stream. Function *ungetc* returns *c* and leaves the file stream unchanged.

One character of pushback is guaranteed provided something has been read from the stream and the stream is actually buffered.

If *c* equals EOF, *ungetc* does nothing to the buffer and returns EOF.

Function *fseek* erases all memory of inserted characters.

DIAGNOSTICS

For *ungetc* to perform correctly, a read must have been performed prior to the call of the *ungetc* function. Function *ungetc* returns EOF if it can't insert the character. If *stream* is *stdin*, *ungetc* allows exactly one character to be pushed back onto the buffer without a previous read statement.

SEE ALSO

fseek, getc, setbuf.

NAME

unlink—remove a file

SYNOPSIS

```
int unlink (path)
char *path;
```

DESCRIPTION

Function *unlink* deletes the file named by the pathname pointed to by *path*.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of −1 is returned and *errno* is set to indicate the error.

NAME

write—write on a file

SYNOPSIS

```
int write (fildes, buf, nbyte)
int fildes;
char *buf;
unsigned nbyte;
```

DESCRIPTION

File descriptor *fildes* is obtained from a *creat* or *open* call.

Function *write* attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the *fildes*. Internal limitations may cause *write* to write fewer bytes than requested; the number of bytes actually written is indicated by the return value. Several calls to *write* may therefore be necessary to write out the contents of *buf*.

On devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. Upon return from *write*, the file pointer is incremented by the number of bytes actually written.

On nonseeking devices, writing always starts at the current position. The value of a file pointer associated with such a device is undefined.

If the O_APPEND file status flag is set—see *open*—the file pointer is set to end-of-file prior to each write.

The file pointer remains unchanged and *write* fails if *fildes* is not a valid file descriptor open for writing. [EBADF]

If you try to write more bytes than there is room for on the device, *write* writes as many bytes as possible. For example, if *nbyte* is 512 and there is room for 20 bytes more on the device, *write* writes 20 bytes and returns a value of 20. The next attempt to write a nonzero number of bytes will signal an error. [ENOSPC]

RETURN VALUE

Upon successful completion the number of bytes actually written is returned. Otherwise, –1 is returned and *errno* is set to indicate the error.

SEE ALSO

creat, lseek, open.

Chapter 5

The Cortland Interface Libraries

Introduction to the Cortland Interface Libraries

This chapter contains the C definition of the Cortland Interface Libraries. For complete documentation of these libraries, see *Cortland Tools*.

After an introductory description of the interface, the chapter is arranged alphabetically by library header. All of the identifiers in the Cortland Interface Libraries are listed in the Library Index, Appendix C.

NAME

C interface to the Cortland Tools Reference

SYNOPSIS

*** This list of #include files is approximate. Some files need to be added below.***

```
#include <Controls.h>          /* Control Manager              */
#include <Desk.h>              /* Desktop Environment          */
#include <Dialogs.h>           /* Dialog Manager               */
#include <Events.h>            /* Event Manager                */
#include <Files.h>             /* File Operations Manager      */
#include <QuickDrawII.h>       /* Graphics Core Tools          */
#include <Input.h>             /* Input Tools                  */
#include <Memory.h>            /* Memory Manager               */
#include <Menu.h>              /* Menu Manager                 */
#include <SANE.h>              /* Standard Apple Numerics      */
#include <Sound.h>             /* Sound Tools                  */
#include <Text.h>              /* Text Editor                  */
#include <Tools.h>             /* Tool Locator                 */
#include <Window.h>            /* Window Manager               */
#include <Misc.h>              /* Miscellaneous Tools          */
```

*** Any of these? ***

```
#include <types.h>             ?* common defines and types    *?
#include <strings.h>           ?* string conversions          *?
#include <quickdraw.h>         ?* QuickDraw                   *?
#include <fonts.h>             ?* Font Manager                *?
#include <scrap.h>             ?* Scrap Manager               *?
#include <printing.h>          ?* Printing Manager            *?
#include <segload.h>           ?* Segment Loader              *?
#include <devices.h>           ?* Device Manager              *?
#include <disks.h>             ?* Disk Driver                 *?
#include <sound.h>             ?* Sound Driver                *?
#include <serial.h>            ?* Serial Drivers              *?
#include <error.h>             ?* System Error Handler        *?
```

*** I am not sure the following belongs here: the MPW C Reference doesn't include this.
***

```
/*
   APWtypes.h  (version 1.0, 13 May 1986)

   Standard types, macros and constants for Apple II xx C
   Copyright 1986 Apple Computer Inc.
*/


#define nil 0
#define NULL 0
```

```
typedef enum {false, true} Boolean;
typedef char *Ptr;
typedef Ptr *Handle;
typedef long (*ProcPtr) ();
typedef ProcPtr *ProcHandle;
typedef short OSErr;
typedef long Fixed;
typedef long Frac;

#define String(size) struct{\
      unsigned char length; unsigned char text[size];}
typedef String(255) Str255, *StringPtr, **StringHandle;

struct Rect{
      short top,left,bottom,right;
};

#define bit0    0X0001
#define bit1    0X0002
#define bit2    0X0004
#define bit3    0X0008
#define bit4    0X0010
#define bit5    0X0020
#define bit6    0X0040
#define bit7    0X0080
#define bit8    0X0100
#define bit9    0X0200
#define bit10   0X0400
#define bit11   0X0800
#define bit12   0X1000
#define bit13   0X2000
#define bit14   0X4000
#define bit15   0X8000
```

DESCRIPTION

The C Interface provides C programs with access to all of the libraries defined in *Cortland Tools Reference*. Constants, types, and library routines are provided. The list of libraries appears above.

Header Files: Include the ".h" files in C programs to declare the defines, types, and functions provided by these libraries. Each library definition lists the includes necessary for use of that library. Functions whose declarations can be inferred from calls have been omitted from the header files. List the includes in the order in which the libraries are listed above.

Object File: The interface code is contained in file *** filename? ***. Link this file with the C program and other libraries. Not all functions require interface code. The linker includes interface code for only those routines that are called.

Interface Implementation: Most library routines are declared as external Pascal routines with trap numbers, and are trapped to directly by

compiled code. *** True? *** Other routines are declared to be C routines and are called through interface glue.

Parameter Types: The C interfaces expect structures (including Points) to be passed by address. String parameters are null-terminated C strings unless otherwise indicated. ResTypes and OSTypes can be expressed as character literals; for example, 'MENU'.

Spelling and Capitalization: The spelling and capitalization of identifiers is exactly as specified in *Cortland Tools Reference*. Constants, variables, parameter names, fields within structures, and enumeration-type elements begin with a lowercase letter. Routines and data types begin with an uppercase letter. Letters that begin new words in English are capitalized. All other letters are lowercase. When a name includes an acronym, the case of the entire acronym is determined by the case of the first letter (e.g., GetOSEvent, teJustLeft).

NAME

controls—Control Manager

SYNOPSIS

*** This is the Mac code. The Cortland code will resemble it in functionality but differ in form. Stay tuned. ***

```
#include <types.h>
#include <quickdraw.h>
#include <controls.h>

/* Control Definition Procedures IDs */

#define pushButProc      0
#define checkBoxProc     1
#define radioButProc     2
#define useWFont         8
#define scrollBarProc   16

/* FindControl Result Codes */

#define inButton        10
#define inCheckbox      11
#define inUpButton      20
#define inDownButton    21
#define inPageUp        22
#define inPageDown      23
#define inThumb        129

/* DragControl Axis Constraints */

#define noConstraint     0
#define hAxisOnly        1
#define vAxisOnly        2

/* Messages to Control definition function */

#define drawCntl         0
#define testCntl         1
#define calcCRgns        2
#define initCntl         3
#define dispCntl         4
#define posCntl          5
#define thumbCntl        6
#define dragCntl         7
#define autoTrack        8

typedef struct ControlRecord {
   struct ControlRecord **nextControl;
   struct GrafPort *contrlOwner;
   Rect            contrlRect;
   unsigned char   contrlVis;
   unsigned char   contrlHilite;
   short           contrlValue;
   short           contrlMin;
```

```
    short           contrlMax;
    ProcHandle      contrlDefProc;
    Handle          contrlData;
    ProcPtr         contrlAction;
    long            contrlrfCon;
    Str255          contrlTitle;
} ControlRecord, *ControlPtr, **ControlHandle;

/* Initialization and Allocation */

ControlHandle NewControl(theWindow,boundsRect,title,visible,value,
      min,max,procID,refCon)
   struct GrafPort *theWindow;
   Rect *boundsRect;
   char *title;
   Boolean visible;
   short value;
   short min;
   short max;
   short procID;
   long refCon;
pascal ControlHandle GetNewControl(controlID,theWindow)
   short controlID;
   struct GrafPort *theWindow;
pascal void DisposeControl(theControl)
   ControlHandle theControl;
pascal void KillControls(theWindow)
   struct GrafPort *theWindow;

/* Control Display */

void SetCTitle(theControl,title)
   ControlHandle theControl;
   char *title;
void GetCTitle(theControl,title)
   ControlHandle theControl;
   char *title;
pascal void HideControl(theControl)
   ControlHandle theControl;
pascal void ShowControl(theControl)
   ControlHandle theControl;
pascal void DrawControls(theWindow)
   struct GrafPort *theWindow;
pascal void HiliteControl(theControl,hiliteState)
   ControlHandle theControl;
   short hiliteState;
pascal void UpdateControls(theWindow,update)
   GrafPort *theWindow;
   RgnHandle update;

/* Mouse Location */

short TestControl(theControl,thePoint)
   ControlHandle theControl;
   Point *thePoint;
short FindControl(thePoint,theWindow,theControl)
   Point *thePoint;
   struct GrafPort *theWindow;
```

```
   ControlHandle *theControl;
short TrackControl(theControl,startPt,actionProc)
   ControlHandle theControl;
   Point *startPt;
   ProcPtr actionProc;

/* Control Movement and Sizing */

pascal void MoveControl(theControl,h,v)
   ControlHandle theControl;
   short h,v;
void DragControl(theControl,startPt,limitRect,slopRect,axis)
   ControlHandle theControl;
   Point *startPt;
   Rect *limitRect;
   Rect *slopRect;
   short axis;
pascal void SizeControl(theControl,w,h)
   ControlHandle theControl;
   short w,h;

/* Control Settings and Range */

pascal void SetCtlValue(theControl,theValue)
   ControlHandle theControl;
   short theValue;
pascal short GetCtlValue(theControl)
   ControlHandle theControl;
pascal void SetCtlMin(theControl,minValue)
   ControlHandle theControl;
   short minValue;
pascal short GetCtlMin(theControl)
   ControlHandle theControl;
pascal void SetCtlMax(theControl,maxValue)
   ControlHandle theControl;
   short maxValue;
pascal short GetCtlMax(theControl)
   ControlHandle theControl;

/* Miscellaneous Utilities */

pascal void SetCRefCon(theControl,data)
   ControlHandle theControl;
   long data;
pascal long GetCRefCon(theControl)
   ControlHandle theControl;
pascal void SetCtlAction(theControl,actionProc)
   ControlHandle theControl;
   ProcPtr actionProc;
pascal ProcPtr GetCtlAction(theControl)
   ControlHandle theControl;
```

USER ROUTINES

```
pascal void MyAction()
pascal void MyAction(theControl,partCode)
   ControlHandle theControl;
   short partCode;
```

```
pascal long MyControl(varCode,theControl,message,param)
   short varCode;
   ControlHandle theControl;
   short message;
   long param;
```

DESCRIPTION

The Control Manager provides routines for creating and manipulating controls (for example: buttons, scroll bars).

For more detailed information see the Control Manager chapter of the*Cortland Tools Reference*.

NAME

desk—Desk Accessory Manager

SYNOPSIS

*** Code for this will be added later. ***

DESCRIPTION

The Desk Accessory Manager supports small co-resident application programs like calculators, calendars, and such. (One of these can be a switcher.)

There are two kinds of desk accessories on the Cortland: classic desk accessories that can run with old-style applications (like Apple Works), and new desk accessories that run in the Cortland desktop environment. The Desk Accessory Manager checks to see which environment it is in and makes sure that a desk accessory can run in that environment before calling it.

One classic desk accesory is built in: the Control Panel.

For more detailed information see the Desk Accessory Manager chapter of *Cortland Tools Reference*.

NAME

dialogs—Dialog Manager

SYNOPSIS

*** Code for this will be added later. ***

DESCRIPTION

The Dialog Manager supports dialog boxes and the alert mechanism. It creates and displays dialog boxes, alerts the user by a sound, and finds out the user's responses to the boxes and sounds.

For more detailed information see the Dialog Manager chapter of *Cortland Tools Reference*.

NAME

events—Event Manager

SYNOPSIS

```
/*
   eventTypes.h -- type definitions used by the event manager (version
1.0, 13 May 1986)

   C Interface to the Apple II xx Libraries
   Copyright 1986 Apple Computer Inc.
*/


#define nullEvent  0
#define MouseDown 1
#define MouseUp 2
#define KeyDown 3
#define undefined4Event 4
#define autoKey 5
#define update 6
#define undefine7Event 7
#define activate 8
#define switch 9
#define DeskAccessory 10
#define deviceDriver 11
#define Application1 12
#define Application2 13
#define Application3 14
#define Application4 15

#define KeyPad        bit13
#define ControlKey    bit12
#define OptionKey     bit11
#define CapsLock      bit10
#define ShiftKey      bit9
#define AppleKey      bit8
#define Btn0State     bit7
#define Btn1State     bit6
#define ChangeFlag    bit1
#define ActiveFlag    bit0

typedef int Point; /* a structure? */
struct eventRecord{
      short int what;
      long int message;
      long int when;
      Point where;
      short int modifiers;
};

#define DupStartup  0X0601
#define ResetErr    0X0602
#define EMNotActive 0X0603
#define IllEvent    0X0604
#define IllButton   0X0605
#define LargeQueue  0X0606
```

```
#define NoMemory    0X0607

/*
   EventMgr.h -- Event Manager  (version 1.0, 13 May 1986)

   C Interface to the Apple II xx Libraries
   Copyright 1986 Apple Computer Inc.
*/



#include <APWtypes.h>
#include <eventTypes.h>


/* standard housekeeping functions - present in every manager */
extern pascal void EMBootInit();
extern pascal void EMStartUp(/*ZeroPage,QueueSize,XMinClamp,
      XMaxClamp,YMinClamp,YMaxClamp,ProgramID*/);
/* short
ZeroPage,QueueSize,XMinClamp,XMaxClamp,YMinClamp,YMaxClamp,ProgramID; */
extern pascal void EMShutDown();
extern pascal int EMVersion();
extern pascal void EMReset();
extern pascal Boolean EMActive();

/* More Housekeeping */

extern pascal short int DoWindows();

/*************************************************/
/*        Toolbox event manager routines         */
/*************************************************/

/* Accessing Events */

extern pascal Boolean GetNextvent(/*EventMask,EventPtr*/);
/* unsigned short EventMask; eventRecord *EventPtr; */

extern pascal Boolean EventAvail(/*EventMask,EventPtr*/);
/* unsigned short EventMask; eventRecord *EventPtr; */

/* Reading the Mouse */

extern pascal void GetMouse(/*MouseLocPtr*/);
/* Point *MouseLocPtr */

extern pascal Boolean Button(/*ButtonNum*/);
/* short int ButtonNum */

extern pascal Boolean StillDown(/*ButtonNum*/);
/* short int ButtonNum; */

extern pascal Boolean WaitMouseUp(/*ButtonNum*/);
/* short int ButtonNum; */

/* Miscellaneous Routines */
```

```
extern pascal long TickCount();

extern pascal long GetDblTime();

extern pascal long GetCaretTime();

extern pascal void SetSwitch();

/*****************************************/
/* Operating system event manager routines */
/*****************************************/

/* Posting and Removing Events */

#define EventPosted       0
#define EventNotDesiganted 1
extern pascal short PostEvent(/*EventCode,EventMsg*/);
/* short EventCode; long EventMsg; */

extern pascal short FlushEvents(/*EventMask,StopMask*/);
/* short EventMask,StopMask; */

/* Accessing events */

extern pascal Boolean GetOSEvent(/*EventMask,EventPtr*/);
/* short EventMask; EventRecord *EventPtr; */

extern pascal Boolean OSEventAvail(/*EventMask,EventPtr*/);
/* short EventMask; EventRecord *EventPtr; */

extern pascal void SetEventMask(/*TheMask*/);
/* short TheMask; */

/*************************************************/
/*          The Journaling Mechanism             */
/*************************************************/

/* TO BE DEFINED, IF ANY */
```

DESCRIPTION

The Event Manager provides access to the Cortland keyboard, keypad, and mouse. An application is organized as a loop containing a call to the Event Manager followed by a series of conditional (`switch`) statements. These conditional statements determyne the program's operations on the basis of the information returned by the Event Manager. The Event Manager also reports events within the application that may require a response: for example, changing one window may cause another window to become visible and need to be redrawn.

The Cortland Event Manager was designed to be as much like the event manager on the Macintosh as possible. The main difference is that the Macintosh has two event managers, one calling the other. The Cortland has only one.

NAME

files—File Operations Manager

SYNOPSIS

*** Code for this will be added later. ***

DESCRIPTION

The File Operations Manager controls the exchange of information between an application and files. It makes calls to ProDOS/16.

For more detailed information see the File Operations Manager chapter of *Cortland Tools Reference*.

NOTE

An I/O completion routine cannot reliably access any globals, strings, or other functions outside its segment.

*** This is true for Mac. What is true for Cortland? ***

WARNING

The low-level routines that use strings take as input and return as output pointers to Pascal-style strings (string length in first byte). However, the high-level routines use C-style strings (terminated by a null character) as input and output parameters.

NAME

- intmath—Integer Math

SYNOPSIS

```
/*
   FixMath.h -- Fixed Point Math  (version 1.0, 13 May 1986)

   C Interface to the Apple II xx Libraries
   Copyright 1986 Apple Computer Inc.
*/

#include <APWtypes.h>

/* standard housekeeping functions - present in every manager */
extern pascal void IMBootInit();
extern pascal void IMStartUp();
extern pascal void IMShutDown();
extern pascal int  IMVersion();
extern pascal void IMReset();
extern pascal Boolean IMActive();

#define BadParams   0X0B01
#define BadChar     0X0B02
#define IntOverflow 0X0B03
#define StrOverflow 0X0B04

struct SDivResult {
      short remainder, quotient;
};
struct UDivResult {
      unsigned short remainder, quotient;
};
struct LDivResult {
      long remainder, quotient;
};
struct LMulResult {
      long MostSig, LeastSig;
};

extern pascal long Multiply(/*i1,i2*/);
/*    int i1, i2; */

extern pascal LMulResult LMultiply(/*i1,i2*/);
/*    long i1, i2; */

extern pascal SDivResult SDivide(/*numerator,denominator*/);
/*    short numerator, denominator; */

extern pascal UDivResult UDivide(/*numerator,denominator*/);
/*    unsigned short numerator, denominator; */

extern pascal LDivResult LDivide(/*numerator,denominator*/);
/*    short numerator, denominator; */
```

```
extern pascal Fixed FixRatio(/*numerator,denominator*/);
/*    short numerator, denominator; */

extern pascal Fixed FixMul(/*f1,f2*/);
/*    Fixed f1,f2; */

extern pascal void Int2Hex(/*i,str,len*/);
/* unsigned short i; Ptr str; int len; */

extern pascal void Long2Hex(/*l,str,len*/);
/* unsigned long l; Ptr str; int len; */

extern pascal unsigned short Hex2Int(/*str,len*/);
/* Ptr str; int len; */

extern pascal unsigned long Hex2Long(/*str,len*/);
/* Ptr str; int len; */

#define UnsignedFlag 0
#define SignedFlag   1
extern pascal void Int2Dec(/*i, str, len, flag*/);
/* int i; Ptr str; int len, flag; */

extern pascal void Long2Dec(/*l, str, len, flag*/);
/* long l; Ptr str; int len, flag; */

extern pascal int Dec2Int(/*str, len, flag*/);
/* Ptr str; int len, flag; */

extern pascal long Dec2Long(/*str, len, flag*/);
/* Ptr str; int len, flag; */

typedef long HexString4;
extern pascal HexString4 Dec2Int(/*i*/);
/* unsigned short i; */
```

DESCRIPTION

The Integer Math toolset includes several routines for working on data of types `short`, `int`, `long`, `fixed`, and `frac` (that is, fractional part). It has routiens for multiplication, division, square root, some trigonometric functions, rounding, and conversions between data types.

For more detailed information see the Integer Math chapter of *Cortland Tools Reference*.

NAME

lined—Line Editor

SYNOPSIS

*** This is the Mac code. The Cortland code will resemble it in functionality but differ in form. Stay tuned. ***

```
#include <types.h>
#include <textedit.h>

#define teJustLeft      0

#define teJustCenter    1
#define teJustRight   (-1)

typedef char Chars[32001];
typedef Chars *CharsPtr, **CharsHandle;

typedef struct TERec {
   Rect     destRect;           /* destination rectangle */
   Rect     viewRect;           /* view rectangle */
   Rect     selRect;            /* select rectangle */
   short    lineHeight;         /* current font lineheight */
   short    fontAscent;         /* current font ascent */
   Point    selPoint;           /* selection point (mouseLoc) */
   short    selStart;           /* selection start */
   short    selEnd;             /* selection end */
   short    active;             /* != 0 if active */
   ProcPtr  wordBreak;          /* word break routine */
   ProcPtr  clikLoop;           /* click loop routine */
   long     clickTime;          /* time of first click */
   short    clickLoc;           /* char. location of click */
   long     caretTime;          /* time for next caret blink */
   short    caretState;         /* on/active booleans */
   short    just;               /* fill style */
   short    teLength;           /* length of text below */
   Handle   hText;              /* handle to actual text */
   short    recalBack;          /* != 0 if recal in background */
   short    recalLines;         /* line being recalulated */
   short    clikStuff;          /* click stuff (internal) */
   short    crOnly;             /* set to -1 if CR Line breaks only */
   short    txFont;             /* text Font */
   Style    txFace;             /* text Face */
   short    txMode;             /* text Mode */
   short    txSize;             /* text Size */
   struct GrafPort *inPort;     /* GrafPort */
   ProcPtr  highHook;           /* highlighting hook */
   ProcPtr  caretHook;          /* highlighting hook */
   short    nLines;             /* number of lines */
   short    lineStarts[16001];  /* line starts */
} TERec, *TEPtr, **TEHandle;

/* Initialization and Allocation */

pascal void TEInit()
```

```
pascal TEHandle TENew(destRect,viewRect)
   Rect *destRect, *viewRect;
pascal void TEDispose(h)
   TEHandle h;

/* Accessing Text */

pascal void TESetText(text,length,hTE)
   Ptr text;
   long length;
   TEHandle hTE;
pascal CharsHandle TEGetText(hTE)
   TEHandle hTE;

/* Insertion Point and Selection Range */

pascal void TEIdle(hTE)
   TEHandle hTE;
void TEClick(pt,extend,hTE)
   Point *pt;
   Boolean extend;
   TEHandle hTE;
pascal void TESetSelect(selStart,selEnd,hTE)
   long selStart;
   long selEnd;
   TEHandle hTE;
pascal void TEActivate(hTE)
   TEHandle hTE;
pascal void TEDeactivate(hTE)
   TEHandle hTE;

/* Editing */

pascal void TEKey(key,hTE)
   short key;
   TEHandle hTE;
pascal void TECut(hTE)
   TEHandle hTE;
pascal void TECopy(hTE)
   TEHandle hTE;
pascal void TEPaste(hTE)
   TEHandle hTE;
pascal void TEDelete(hTE)
   TEHandle hTE;
pascal void TEInsert(text,length,hTE)
   Ptr text;
   long length;
   TEHandle hTE;

/* Text Display and Scrolling */

pascal void TESetJust(just,hTE)
   short just;
   TEHandle hTE;
pascal void TEUpdate(rUpdate,hTE)
   Rect *rUpdate;
   TEHandle hTE;
pascal void TextBox(text,length,box,just)
```

```
    Ptr text;
    long length;
    Rect *box;
    short just;
pascal void TEScroll(dh,dv,hTE)
    short dh;
    short dv;
    TEHandle hTE;
pascal void TESelView(hTE)
    TEHandle hTE;
pascal void TEPinScroll(dh,dv,hTE)
    short dh;
    short dv;
    TEHandle hTE;
pascal void TEAutoView(auto,hTE)
    Boolean auto;
    TEHandle hTE;

/* Scrap Information */

OSErr TEFromScrap()
OSErr TEToScrap()
Handle TEScrapHandle()
long TEGetScrapLen()
void TESetScrapLen(length)
    long length;

/* Advanced Routines */

void SetWordBreak(wBrkProc,hTE)
    ProcPtr wBrkProc;
    TEHandle hTE;
void SetClikLoop(clikProc,hTE)
    ProcPtr clikProc;
    TEHandle hTE;
pascal void TECalText(hTE)
    TEHandle hTE;
```

USER ROUTINES

```
Pascal Boolean MyWordBreak(text,charPos)
    Ptr text;
    short charPos;
Pascal Boolean MyClikLoop()
```

DESCRIPTION

The Line Editor accepts text typed by the user and performs standard editing functions in response to calls from applications. Its functions include

- inserting and deleting text
- using the mouse to select text
- cutting and pasting text

For more detailed information see the "Line Editor" chapter of the *Cortland Tools Reference.*

NOTE

The user routines *highHook* and *caretHook* are called with register conventions and therefore can't be C routines.

*** True for Cortland??? ***

NAME

memory—Memory Manager

SYNOPSIS

```
/*
   MMtypes.h -- general types for Memory Manager (version 1.0, 13
May 1986)

   C Interface to the Apple II xx Libraries
   Copyright 1986 Apple Computer Inc.
*/


typedef unsigned short MMUserID;
typedef long int MMSize;
typedef unsigned short PurgeLevel;

#define MMnoError    0
#define MMFullErr    0x0201
#define MMNilErr     0x0202
#define MMNotNilErr  0x0203
#define MMLockErr    0x0204
#define MMPurgeErr   0x0205
#define MMHandleErr  0x0206
#define MMIDErr      0x0207

#define MMFixedBank    bit0
#define MMFixedAddr    bit1
#define MMPageAlign    bit2
#define MMNoSpecMem    bit3
#define MMWithinBank   bit4
#define MMFixed        bit14

/*
   MemoryMgr.h -- Memory Manager  (version 1.0, 13 May 1986)

   C Interface to the Apple II xx Libraries
   Copyright 1986 Apple Computer Inc.
*/


#include <APWtypes.h>
#include <MMtypes.h>


/* standard housekeeping functions - present in every manager */
extern pascal void MMBootInit();
extern pascal int MMApplInit();
extern pascal void MMAppQuit();
extern pascal int MMVersion();
extern pascal void MMReset();
extern pascal Boolean MMStatus();

/* Allocating Memory */
```

```
extern pascal Handle
NewHandle(/*BlockSize,Owner,Attributes,Location*/);
/* MMsize BlockSize; MMUserID Owner; short Attributes; Ptr
Location; */

extern pascal void ReallocHandle(/*TheHandle,BlockSize,

Owner,Attributes,Location*/);
/* Handle TheHandle; MMSize BlockSize; MMUserID Owner;
   short Attributes; Ptr Location */

/* Freeing Memory */

extern pascal void DisposHandle(/*theHandle*/);
/* Handle theHandle; */

extern pascal void DisposAll(/*Owner*/);
/* MMUserID Owner; */

extern pascal void PurgeHandle(/*theHandle*/);
/* Handle theHandle; */

extern pascal void PurgeAll(/*Owner*/);
/* MMUserID Owner */

/* Information on blocks */

extern pascal MMSize GetHandleSize(/*theHandle*/);
/* Handle theHandle; */

extern pascal void SetHandleSize(/*newSize,theHandle*/);
/* MMSize newSize; Handle theHandle; */

extern pascal Handle FindHandle(/*location*/);
/* Ptr location; */

extern pascal MMSize FreeMem();

extern pascal MMSize MaxBlock();

extern pascal MMSize TotalMem();

/* Other properties of block */

extern pascal void HLock(/*theHandle*/);
/* Handle theHandle; */

extern pascal void HLockAll(/*Owner*/);
/* MMUserID Owner; */

extern pascal void HUnlock(/*theHandle*/);
/* Handle theHandle; */

extern pascal void HUnlockAll(/*Owner*/);
/* MMUserID Owner; */

extern pascal void SetPurge(/*newPlevel,theHandle*/);
/* PurgeLevel newPlevel; Handle theHandle; */
```

```
extern pascal void SetPurgeAll(/*newPlevel,Owner*/);
/* PurgeLevel newPlevel; MMUserID Owner; */

/* Copying Data */

extern pascal void BlockMove(/*Source,Dest,Count*/);
/* Ptr Source,Dest; MMSize Count; */
```

DESCRIPTION

The Memory Manager controls use of memory for by application programs. Keeping memory usage under control of the Memory Manager makes it possible to have co-resident applications like desk accessories. Programs call the memory Manager to request (allocate) memory, release (deallocate) memory, and find out how much memory is free.

For more detailed information see the Memory Manager chapter of *Cortland Tools Reference*.

NAME

menus—Menu Manager

SYNOPSIS

```
/*
   MenuMgr.h -- Menu Manager  (version 1.0, 13 May 1986)

   C Interface to the Apple II xx Libraries
   Copyright 1986 Apple Computer Inc.
*/



#include <APWtypes.h>
#include <eventTypes.h>

#define TheOneWithID    0
#define HeadOfList    1
#define LastInList 2
#define TheOneBeforeID  -1

/* standard housekeeping functions - present in every manager */
extern pascal void BootMmgr();
extern pascal void TermMenus();
extern pascal int MmgrVersion();
extern pascal void MmgrReset();

struct ItemRecord{
      struct ItemRecord *NextItem;
      unsigned short ItemId;
      unsigned long ItemName;
      unsigned char ItemChar;
      unsigned char ItemCheck;
      unsigned char ItemFlag;
};
struct MenuRecord{
      struct MenuRecord *NextMenu;
      unsigned short MenuId;
      unsigned short MenuWidth;
      unsigned short MenuHeight;
      ProcPtr MenuProc;
      unsigned short TitleWidth;
      StringPtr TitleName;
      unsigned char MenuFlag;
      struct ItemRecord *ItemList;
};
struct MenuBar{
      long *NextCtrl; /* pointer to next control - !!!!!!! */
      unsigned char CtrlType;
      Rect Bar;
      unsigned char FallDown;
      unsigned char BarColor;
      unsigned char InvertColor;
      unsigned char Outline;
      unsigned char BarFlag;
      struct MenuRecord *MenuList;
};
```

```
struct TwoShort {
      unsigned short Divide, XOR;
};

extern pascal MenuRecord
*NewMenu(/*textColor,background,MenuString*/);
/* short textColor, background; StringPtr MenuString; */

extern pascal void DisposeMenu(/*MenuList*/);
/* MenuRecord *MenuList; */

extern pascal unsigned short FixMenuBar(/*theBar*/);
/* MenuBar *theBar; */

extern pascal void CalcMenuSize(/*newWidth,newHeight,MenuPtr*/);
/* unsigned short newWidth,newHeight; MenuPtr ???; */

/* drawing and user interaction routines */

extern pascal void MenuSelect(/*eventRec,theBar*/);
/* eventRec ???; MenuBar *theBar; */

extern pascal void MenuKey(/*eventRec,theBar*/);
/* eventRec ???; MenuBar *theBar; */

extern pascal void CheckFallDown(/*eventRec,theBar*/);
/* eventRec ???; MenuBar *theBar; */

extern pascal void MenuRefresh(/*RedrawRoutine*/);
/* ProcPtr RedrawRoutine; */

/* Drawing */

extern pascal void DrawMenuBar();

extern pascal void HiliteMenu(/*Hilite,MenuNum*/);
/* Boolean Hilite; unsigned short MenuNum; */

extern pascal void FlashMenuBar();

/* Menu and Item Shuffling */

extern pascal void InsertMenu(/*AddMenu,InsertAfter*/);
/* MenuRecord *AddMenu; unsigned short InsertAfter; */

extern pascal void DeleteMenu(/*MenuNum*/);
/* unsigned short MenuNum; */

extern pascal void InsertItem(/*AddItem,InsertAfter,MenuNum*/);
/* ItemRecord *AddItem; unsigned short InsertAfter, MenuNum; */

extern pascal void DeleteItem(/*ItemNum,MenuNum*/);
/* unsigned short ItemNum, MenuNum; */

/* Menu Bar Access */

extern pascal void SetSysBar(/*NewBar*/);
/* MenuBar *NewBar; */
```

```
extern pascal MenuBar *GetSysBar();

extern pascal void SetMenuBar(/*TheBar*/);
/* MenuBar *TheBar; */

extern pascal MenuBar *GetMenuBar();

extern pascal short CountMitems(/*MenuNum*/);
/* unsigned MenuNum; */

extern pascal void SetFalArea(/*FallHeight*/);
/* short FallHeight */

extern pascal short GetFallArea();

extern pascal void
SetBarColors(/*NewBarColor,NewInvertColor,NewOutColor*/);
/* unsigned short NewBarColor,NewInvertColor,NewOutColor; */

extern pascal unsigned long GetBarColors();

extern pascal void SetTileStart(/*XStart*/);
/* unsigned short XStart; */

extern pascal unsigned short GetTileStart();

/* Menu Record Access Routines */

extern pascal MenuRecord *GetMenuPtr(/*LookFor,MenuNum*/);
/* short LookFor; unsigned short MenuNum; */

extern pascal void SetTileWidth(/*NewWidth,MenuNum*/);
/* unsigned short NewWidth, MenuNum; */

extern pascal unsigned short GetTileWidth(/*MenuNum*/);
/* unsigned short MenuNum */

#define MenuFlag  0XFF7F
#define MenuTileFlag 0XFFBF
#define HighlightFlag 0XFFDF
#define MenuKindFlag 0XFFE7
#define EnableMenu 0X0000
#define DisableMenu 0X0080
#define NormalTile 0X0000
#define InvertTile 0X0040
#define RedrawHighlight 0X0000
#define XORHighlight 0X0020
#define TextMenu 0X0000
#define ColorMenu 0X0008
#define ApplicationMenu 0X0010
extern pascal void SetMenuFlag(/*NewState,FlagMask,MenuNum*/);
/* unsigned short NewState,FlagMask,MenuNum */

extern pascal unsigned short GetMenuFlag(/*MenuNum*/);
/* unsigned short MenuNum; */

extern pascal void SetMenuTile(/*NewStrg,MenuNum*/);
```

```
/* StringPtr NewStrg; unsigned short MenuNum; */

extern pascal StringPtr GetMenuTile(/*MenuNum*/);
/* unsigned short MenuNum; */

extern pascal void SetMenuID(/*NewID,MenuNum*/);
/* unsigned short NewID,MenuNum; */

/* Item Record Access */

extern pascal ItemRecord *GetItemPtr(/*LookFor,ItemNum*/);
/* unsigned short LookFor,ItemNum; */

extern pascal void SetItem(/*NewStrg,ItemNum*/);
/* StringPtr NewStrg; unsigned short ItemNum; */

extern pascal StringPtr *GetItem(/*ItemNum*/);
/* unsigned short ItemNum; */

extern pascal void EnableItem(/*ItemNum*/);
/* unsigned short ItemNum; */

extern pascal void DisableItem(/*ItemNum*/);
/* unsigned short ItemNum; */

extern pascal void CheckItem(/*Checked,ItemNum*/);
/* Boolean Checked; unsigned short ItemNum; */

extern pascal void SetItemMark(/*Mark,ItemNum*/);
/* unsigned short Mark, ItemNum; */

extern pascal unsigned short GetItemMark(/*ItemNum*/);
/* unsigned short ItemNum; */

#define Bold       0X0001
#define Italic     0X0002
#define Underscore 0X0004
extern pascal void SetItemStyle(/*ChStyle,ItemNum*/);
/* unsigned short ChStyle,ItemNum; */

extern pascal unsigned short GetItemStyle(/*ItemNum*/);
/* unsigned short ItemNum; */

#define ItemUnderline          0X0040
#define ItemNoUnderline        0XFFBF
#define ItemXORHighlight       0X0020
#define ItemRedrawHiglight     0XFFDF
extern pascal void SetItemFlag(/*NewValue,ItemNum*/);
/* unsigned short NewValue, ItemNum */

extern pascal TwoShort GetItemFlag(/*ItemNum*/);
/* unsigned short ItemNum; */

extern pascal void SetItemID(/*NewID,ItemNum*/);
/* unsigned short NewID, ItemNum; */

extern pascal void SetItemBlink(/*Count*/);
/* unsigned short Count; */
```

```
/* Miscellaneous routines */

extern pascal void MNewRes();

extern pascal void

extern pascal void
```

DESCRIPTION

The Menu Manager provides routines for creating and using menus. The application calls the Menu Manager whenever the user gives a command, whether from the menu by using the mouse or by typing a command key, to find out which command it is. For more detailed information see the Menu Manager chapter of *Cortland Tools Reference*.

WARNING

The names of desk accessories start with a null byte. The output parameter from GetMenuItem will return a string that begins with a null byte when a desk accessory is selected from the Apple menu. OpenDeskAcc skips over the null byte when interpreting its parameter.

*** True for Cortland? ***

NAME

misc—Miscellaneous Tools for talking to hardware

SYNOPSIS

This code will be added later.

DESCRIPTION

The Miscellaneous Tools include

- Routines to access battery-backed-up RAM
- Clock routines
- Routines to access peripheral cards
- Routines to change firmware vectors
- Routines to manage the heartbeat interrupt queue
- Routines for directly accessing the mouse
- Interrupt-control routines

For more detailed information see the "Miscellaneous Tools" chapter of the *Cortland Tools Reference.*

NAME

printing—Printing Manager

SYNOPSIS

*** This is the Mac code. The Cortland code will resemble it in functionality but differ in form. Stay tuned. ***

```
#include <types.h>
#include <quickdraw.h>
#include <printing.h>

/* Printing Methods */

#define  bDraftLoop            0  /* draft printing */
#define  bSpoolLoop            1  /* spooling */

/* Printer specification in prStl field of print record */

#define     bDevCItoh   1      /* ImageWriter printer */
#define  bDevLaser             3  /* LaserWriter printer */

/* Maximum number of pages in a spool file */

#define     iPFMaxPgs   128    /* max pages in a spool file */
#define  iPrPgFract         120   /* paper units per inch */

/* Result codes */

#define     noErr 0      /* no error */
#define  iPrSavPFil         (-1)   /* saving spool file */
#define  iIOAbort          (-27)   /* I/O abort error */
#define  iMemFullErr       (-108)  /* not enough room in heap zone */
#define  iPrAbort            128   /* application or user requested abort */

/* Printer Driver Control Call Parameters */

#define  iPrDevCtl             7  /* device control */
#define  lPrReset    0x00010000  /* reset printer */
#define  lPrLineFeed 0x00030000  /* start new line */
#define  lPrLFSixth  0x0003FFFF  /* standard 1/6" line feed */
#define  lPrPageEnd  0x00020000  /* start new page */
#define  iPrBitsCtl            4  /* bit map printing */
#define  lScreenBits           0  /* configurable */
#define  lPaintBits            1  /* 72 x 72 dots */
#define  iPrIOCtl              5  /* text streaming */

/* Printing Resources */

#define  sPrDrvr         ".Print" /* Printer Driver resource name */
#define  iPrDrvrRef          (-3) /* Printer Driver reference number */

/* Type definitions */

typedef Rect *TPRect;
```

```
typedef struct TPrPort {
   GrafPort gPort;       /* graph port to draw in */
   QDProcs  gProcs;      /* pointers to drawing routines */
   long     lGParam1;    /* internal */
   long     lGParam2;    /* internal */
   long     lGParam3;    /* internal */
   long     lGParam4;    /* internal */
   Boolean  fOurPtr;     /* internal */
   Boolean  fOurBits;    /* internal */
} TPrPort, *TPPrPort;

typedef struct TPrInfo {
   short    iDev;        /* printer information */
   short    iVRes;       /* printer vertical resolution */
   short    iHRes;       /* printer horizontal resolution */
   Rect     rPage;       /* page rectangle */
} TPrInfo;

typedef enum {feedCut,feedFanfold,feedMechCut,feedOther} TFeed;

typedef struct TPrStl {
   short    wDev;        /* high byte specifies device */
   short    iPageV;      /* paper height */
   short    iPageH;      /* paper width */
   char     bPort;       /* printer or modem port - ignored */
   TFeed    feed;        /* paper type */
} TPrStl;

typedef enum {scanTB,scanBT,scanLR,scanRL} TScan;

typedef struct TPrXInfo {
   short    iRowBytes;   /* bytes per row */
   short    iBandV;      /* vertical dots */
   short    iBandH;      /* horizontal dots */
   short    iDevBytes;   /* size of bit image */
   short    iBands;      /* bands per page */
   char     bPatScale;   /* used by QuickDraw */
   char     bUlThick;    /* underline thickness */
   char     bUlOffset;   /* underline offset */
   char     bUlShadow;   /* underline descender */
   TScan    scan;        /* scan direction */
   char     bXInfoX;     /* not used */
} TPrXInfo;

typedef struct TPrJob {
   short    iFstPage;    /* first page to print */
   short    iLstPage;    /* last page to print */
   short    iCopies;     /* number of copies */
   char     bJDocLoop;   /* printing method */
   Boolean  fFromUsr;    /* true if called from application */
   ProcPtr  pIdleProc;   /* background procedure */
   StringPtr pFileName;  /* spool file name */  ???
   short    iFileVol;    /* volume reference number */
   char     bFileVers;   /* version number of spool file */
   char     bJobX;       /* not used */
} TPrJob;

typedef struct TPrint {
```

```
    short     iPrVersion; /* Printing Manager version */
    TPrInfo   prInfo;     /* printing information */
    Rect      rPaper;     /* paper rectangle */
    TPrStl    prStl;      /* style information */
    TPrInfo   prInfoPT;   /* copy of prInfo */
    TPrXInfo  prXInfo;    /* band information */
    TPrJob    prJob;      /* job information */
    short     printX[19]  /* internal */
} TPrint, *TPPrint, **THPrint;

typedef struct TPrStatus {
    short     iTotPages;  /* total number of pages */
    short     iCurPage;   /* page being printed */
    short     iTotCopies; /* number of copies */
    short     iCurCopy;   /* copy begin printed */
    short     iTotBands;  /* bands per page */
    short     iCurBand;   /* band being printed */
    Boolean   fPgDirty;   /* true if started printing page */
    Boolean   fImaging;   /* true if imaging */
    THPrint   hPrint;     /* print record */
    TPPrPort  pPrPort;    /* printing port */
    PicHandle hPic;       /* internal */
} TPrStatus;

/* Initialization and Termination */

pascal void PrOpen()
pascal void PrClose()

/* Print Records and Dialogs */

pascal void PrintDefault(hPrint)
    THPrint hPrint;
pascal Boolean PrValidate(hPrint)
    THPrint hPrint;
pascal Boolean PrStlDialog(hPrint)
    THPrint hPrint;
pascal Boolean PrJobDialog(hPrint)
    THPrint hPrint;
pascal void PrJobMerge(hPrintSrc,hPrintDst)
    THPrint hPrintSrc,hPrintDst;

/* Document Printing */

pascal TPPrPort PrOpenDoc(hPrint,pPrPort,pIOBuf)
    THPrint hPrint;
    TPPrPort pPrPort;
    Ptr pIOBuf;
pascal void PrCloseDoc(pPrPort)
    TPPrPort pPrPort;
pascal void PrOpenPage(pPrPort,pPageFrame)
    TPPrPort pPrPort;
    TPRect pPageFrame;
pascal void PrClosePage(pPrPort)
    TPPrPort pPrPort;

/* Spool Printing */
```

```
pascal void PrPicFile(hPrint,pPrPort,pIOBuf,pDevBuf,prStatus)
   THPrint hPrint;
   TPPrPort pPrPort;
   Ptr pIOBuf,pDevBuf;
   TPrStatus *prStatus;

/* Handling Errors */

pascal short PrError()
pascal void PrSetError(iErr)
   short iErr;

/* Low Level Driver Access */

pascal void PrDrvrOpen()
pascal void PrDrvrClose()
pascal void PrCtlCall(iWhichCtl,lParam1,lParam2,lParam3)
   short iWhichCtl;
   long lParam1,lParam2,lParam3;
pascal Handle PrDrvrDCE()
pascal short PrDrvrVers()
```

DESCRIPTION

The Printing Manager supports printing on a variety of devices. Programs that call Printing Manager routines should be linked with file PrintCalls.o.

*** True? ***

For more detailed information see the Printing Manager chapter of *Cortland Tools Reference*.

NOTE

The current Pascal implementation has additional constants and data types that aren't documented in *Cortland Tools Reference* because they're not generally used.

*** True? ***

This interface follows the *Cortland Tools Reference* .

NAME

quickdraw2—QuickDraw II

SYNOPSIS

*** This is the Mac code. The Cortland code will resemble it in functionality but differ in form. Stay tuned. ***

```
#include <types.h>
#include <quickdraw.h>

/* 16 Transfer Modes */

#define  srcCopy        0
#define  srcOr          1
#define  srcXor         2
#define  srcBic         3
#define  notSrcCopy     4
#define  notSrcOr       5
#define  notSrcXor      6
#define  notSrcBic      7
#define  patCopy        8
#define  patOr          9
#define  patXor         10
#define  patBic         11
#define  notPatCopy     12
#define  notPatOr       13
#define  notPatXor      14
#define  notPatBic      15

/* QuickDraw Color Separation Constants */

#define  normalBit      0
#define  inverseBit     1
#define  redBit         4       /* RGB Additive Mapping */
#define  greenBit       3
#define  blueBit        2
#define  cyanBit        8       /* CMYBk Subtractive Mapping */
#define  magentaBit     7
#define  yellowBit      6
#define  blackBit       5
#define  blackColor     33      /* Colors Expressed in these Mappings *
#define  whiteColor     30
#define  redColor       205
#define  greenColor     341
#define  blueColor      409
#define  cyanColor      273
#define  magentaColor   137
#define  yellowColor    69

/* Picture Comments */

#define  picLParen      0
#define  picRParen      1

/* Type Style Constants */
```

```
#define  normal         0x00
#define  bold           0x01
#define  italic         0x02
#define  underline      0x04
#define  outline        0x08
#define  shadow         0x10
#define  condense       0x20
#define  expand         0x40

/* Types */

typedef  unsigned char Pattern[8];
typedef  short Bits16[16];
typedef  enum {frame,paint,erase,invert,fill} GrafVerb;

/* typedefs Style, Point, and Rect appear in file TYPES */

typedef struct FontInfo {
   short     ascent;
   short     descent;
   short     widMax;
   short     leading;
} FontInfo;

typedef struct BitMap {
   Ptr       baseAddr;
   short     rowBytes;
   Rect      bounds;
} BitMap;

typedef struct Cursor {
   Bits16    data;
   Bits16    mask;
   Point     hotSpot;
} Cursor;

typedef struct PenState {
   Point     pnLoc;
   Point     pnSize;
   short     pnMode;
   Pattern   pnPat;
} PenState;

typedef struct Region {
   short     rgnSize;
   Rect      rgnBBox;
   short     rgnData[0];
} Region, *RgnPtr, **RgnHandle;

typedef struct Picture {
   short     picSize;
   Rect      picFrame;
   short     picData[0];
} Picture, *PicPtr, **PicHandle;

typedef struct Polygon {
   short     polySize;
   Rect      polyBBox;
```

```
   Point    polyPoints[0];
} Polygon, *PolyPtr, **PolyHandle;

typedef struct QDProcs {
   ProcPtr  textProc;
   ProcPtr  lineProc;
   ProcPtr  rectProc;
   ProcPtr  rRectProc;
   ProcPtr  ovalProc;
   ProcPtr  arcProc;
   ProcPtr  polyProc;
   ProcPtr  rgnProc;
   ProcPtr  bitsProc;
   ProcPtr  commentProc;
   ProcPtr  txMeasProc;
   ProcPtr  getPicProc;
   ProcPtr  putPicProc;
} QDProcs, *QDProcsPtr;

typedef struct GrafPort {
   short       device;
   BitMap      portBits;
   Rect        portRect;
   RgnHandle   visRgn;
   RgnHandle   clipRgn;
   Pattern     bkPat;
   Pattern     fillPat;
   Point       pnLoc;
   Point       pnSize;
   short       pnMode;
   Pattern     pnPat;
   short       pnVis;
   short       txFont;
   Style       txFace;
   short       txMode;
   short       txSize;
   long        spExtra;
   long        fgColor;
   long        bkColor;
   short       colrBit;
   short       patStretch;
   PicHandle   picSave;
   RgnHandle   rgnSave;
   PolyHandle  polySave;
   QDProcsPtr  grafProcs;
} GrafPort, *GrafPtr;

/* External Variable Declarations */

extern struct qd {
   char      private[78];
   long      randSeed;
   BitMap    screenBits;
   Cursor    arrow;
   Pattern   dkGray;
   Pattern   ltGray;
   Pattern   gray;
   Pattern   black;
```

```
   Pattern   white;
   GrafPtr   thePort;
} qd;

/* GrafPort Routines */

pascal void InitGraf(globalPtr)
   Ptr globalPtr;
pascal void OpenPort(port)
   GrafPtr port;
pascal void InitPort(port)
   GrafPtr port;
pascal void ClosePort(port)
   GrafPtr port;
pascal void SetPort(port)
   GrafPtr port;
pascal void GetPort(port)
   GrafPtr *port;
pascal void GrafDevice(device)
   short device;
pascal void SetPortBits(bm)
   BitMap *bm;
pascal void PortSize(width,height)
   short width,height;
pascal void MovePortTo(leftGlobal,rightGlobal)
   short leftGlobal,rightGlobal;
pascal void SetOrigin(h,v)
   short h,v;
pascal void SetClip(rgn)
   RgnHandle rgn;
pascal void GetClip(rgn)
   RgnHandle rgn;
pascal void ClipRect(r)
   Rect *r;
pascal void BackPat(pat)
   Pattern *pat;

/* Cursor Routines */

pascal void InitCursor()
pascal void SetCursor(crsr)
   Cursor *crsr;
pascal void HideCursor()
pascal void ShowCursor()
pascal void ObscureCursor()

/* Line Routines */

pascal void HidePen()
pascal void ShowPen()
pascal void GetPen(pt)
   Point *pt;
pascal void GetPenState(pnState)
   PenState *pnState;
pascal void SetPenState(pnState)
   PenState *pnState;
pascal void PenSize(width,height)
   short width,height;
```

```
pascal void PenMode(mode)
   short mode;
pascal void PenPat(pat)
   Pattern *pat;
pascal void PenNormal()
pascal void MoveTo(h,v)
   short h,v;
pascal void Move(dh,dv)
   short dh,dv;
pascal void LineTo(h,v)
   short h,v;
pascal void Line(dh,dv)
   short dh,dv;

/* Text Routines */

pascal void TextFont(font)
   short font;
pascal void TextFace(face)
   Style face;
pascal void TextMode(mode)
   short mode;
pascal void TextSize(size)
   short size;
pascal void SpaceExtra(extra)
   long extra;
pascal void DrawChar(ch)
   short ch;
void DrawString(s)
   char *s;
pascal void DrawText(textBuf,firstByte,byteCount)
   Ptr textBuf;
   short firstByte,byteCount;
pascal short CharWidth(ch)
   short ch;
short StringWidth(s)
   char *s;
pascal short TextWidth(textBuf,firstByte,byteCount)
   Ptr textBuf;
   short firstByte,byteCount;
pascal void MeasureText(count,textAddr,charLocs)
   short count;
   Ptr textAddr,charLocs;
pascal void GetFontInfo(info)
   FontInfo *info;

/* Drawing in Color */

pascal void ForeColor(color)
   long color;
pascal void BackColor(color)
   long color;
pascal void ColorBit(whichBit)
   short whichBit;

/* Rectangle Calculations */

pascal void SetRect(r,left,top,right,bottom)
```

```
   Rect *r;
   short left,top,right,bottom;
pascal void OffsetRect(r,dh,dv)
   Rect *r;
   short dh,dv;
pascal void InsetRect(r,dh,dv)
   Rect *r;
   short dh,dv;
pascal Boolean SectRect(srcRect1,srcRect2,dstRect)
   Rect *srcRect1,*srcRect2,*dstRect;
pascal void UnionRect(srcRect1,srcRect2,dstRect)
   Rect *srcRect1,*srcRect2,*dstRect;
Boolean PtInRect(pt,r)
   Point *pt;
   Rect *r;
void Pt2Rect(pt1,pt2,dstRect)
   Point *pt1, *pt2;
   Rect *dstRect;
void PtToAngle(r,pt,angle)
   Rect *r;
   Point *pt;
   short *angle;
pascal Boolean EqualRect(rect1,rect2)
   Rect *rect1,*rect2;
pascal Boolean EmptyRect(r)
   Rect *r;

/* Graphical Operations on Rectangles */

pascal void FrameRect(r)
   Rect *r;
pascal void PaintRect(r)
   Rect *r;
pascal void EraseRect(r)
   Rect *r;
pascal void InvertRect(r)
   Rect *r;
pascal void FillRect(r,pat)
   Rect *r;
   Pattern *pat;

/* Oval Routines */

pascal void FrameOval(r)
   Rect *r;
pascal void PaintOval(r)
   Rect *r;
pascal void EraseOval(r)
   Rect *r;
pascal void InvertOval(r)
   Rect *r;
pascal void FillOval(r,pat)
   Rect *r;
   Pattern *pat;

/* RoundRect Routines */

pascal void FrameRoundRect(r,ovalWidth,ovalHeight)
```

```
   Rect *r;
   short ovalWidth,ovalHeight;
pascal void PaintRoundRect(r,ovalWidth,ovalHeight)
   Rect *r;
   short ovalWidth,ovalHeight;
pascal void EraseRoundRect(r,ovalWidth,ovalHeight)
   Rect *r;
   short ovalWidth,ovalHeight;
pascal void InvertRoundRect(r,ovalWidth,ovalHeight)
   Rect *r;
   short ovalWidth,ovalHeight;
pascal void FillRoundRect(r,ovalWidth,ovalHeight,pat)
   Rect *r;
   short ovalWidth,ovalHeight;
   Pattern *pat;

/* Arc Routines */

pascal void FrameArc(r,startAngle,arcAngle)
   Rect *r;
   short startAngle,arcAngle;
pascal void PaintArc(r,startAngle,arcAngle)
   Rect *r;
   short startAngle,arcAngle;
pascal void EraseArc(r,startAngle,arcAngle)
   Rect *r;
   short startAngle,arcAngle;
pascal void InvertArc(r,startAngle,arcAngle)
   Rect *r;
   short startAngle,arcAngle;
pascal void FillArc(r,startAngle,arcAngle,pat)
   Rect *r;
   short startAngle,arcAngle;
   Pattern *pat;

/* Region Calculations */

pascal RgnHandle NewRgn()
pascal void DisposeRgn(rgn)
   RgnHandle rgn;
pascal void CopyRgn(srcRgn,dstRgn)
   RgnHandle srcRgn,dstRgn;
pascal void SetEmptyRgn(rgn)
   RgnHandle rgn;
pascal void SetRectRgn(rgn,left,top,right,bottom)
   RgnHandle rgn;
   short left,top,right,bottom;
pascal void RectRgn(rgn,r)
   RgnHandle rgn;
   Rect *r;
pascal void OpenRgn()
pascal void CloseRgn(dstRgn)
   RgnHandle dstRgn;
pascal void OffsetRgn(rgn,dh,dv)
   RgnHandle rgn;
   short dh,dv;
pascal void InsetRgn(rgn,dh,dv)
   RgnHandle rgn;
```

```
   short dh,dv;
pascal void SectRgn(srcRgnA,srcRgnB,dstRgn)
   RgnHandle srcRgnA,srcRgnB,dstRgn;
pascal void UnionRgn(srcRgnA,srcRgnB,dstRgn)
   RgnHandle srcRgnA,srcRgnB,dstRgn;
pascal void DiffRgn(srcRgnA,srcRgnB,dstRgn)
   RgnHandle srcRgnA,srcRgnB,dstRgn;
pascal void XorRgn(srcRgnA,srcRgnB,dstRgn)
   RgnHandle srcRgnA,srcRgnB,dstRgn;
Boolean PtInRgn(pt,rgn)
   Point *pt;
   RgnHandle rgn;
pascal Boolean RectInRgn(r,rgn)
   Rect *r;
   RgnHandle rgn;
pascal Boolean EqualRgn(rgnA,rgnB)
   RgnHandle rgnA,rgnB;
pascal Boolean EmptyRgn(rgn)
   RgnHandle rgn;

/* Graphical Operations on Regions */

pascal void FrameRgn(rgn)
   RgnHandle rgn;
pascal void PaintRgn(rgn)
   RgnHandle rgn;
pascal void EraseRgn(rgn)
   RgnHandle rgn;
pascal void InvertRgn(rgn)
   RgnHandle rgn;
pascal void FillRgn(rgn,pat)
   RgnHandle rgn;
   Pattern *pat;

/* Graphical Operations on Bit Maps */

pascal void ScrollRect(r,dh,dv,updateRgn)
   Rect *r;
   short dh,dv;
   RgnHandle updateRgn;
pascal void CopyBits(srcBits,dstBits,srcRect,dstRect,mode,maskRgn)
   BitMap *srcBits,*dstBits;
   Rect *srcRect,*dstRect;
   short mode;
   RgnHandle maskRgn;
pascal void SeedFill(srcPtr,dstPtr,srcRow,dstRow,height,words,seedH,seedV)
   Ptr srcPtr,dstPtr;
   short srcRow,dstRow,height,words;
   short seedH,seedV;
pascal void CalcMask(srcPtr,dstPtr,srcRow,dstRow,height,words)
   Ptr srcPtr,dstPtr;
   short srcRow,dstRow,height,words;
pascal void CopyMask(srcBits,maskBits,dstBits,srcRect,maskRect,dstRect)
   BitMap srcBits,maskBits,dstBits;
   Rect *srcRect, *maskRect, *dstRect;

/* Picture Routines */
```

```
pascal PicHandle OpenPicture(picFrame)
   Rect *picFrame;
pascal void PicComment(kind,dataSize,dataHandle)
   short kind,dataSize;
   Handle dataHandle;
pascal void ClosePicture()
pascal void DrawPicture(myPicture,dstRect)
   PicHandle myPicture;
   Rect *dstRect;
pascal void KillPicture(myPicture)
   PicHandle myPicture;

/* Polygon Calculations */

pascal PolyHandle OpenPoly()
pascal void ClosePoly()
pascal void KillPoly(poly)
   PolyHandle poly;
pascal void OffsetPoly(poly,dh,dv)
   PolyHandle poly;
   short dh,dv;

/* Graphical Operations on Polygons */

pascal void FramePoly(poly)
   PolyHandle poly;
pascal void PaintPoly(poly)
   PolyHandle poly;
pascal void ErasePoly(poly)
   PolyHandle poly;
pascal void InvertPoly(poly)
   PolyHandle poly;
pascal void FillPoly(poly,pat)
   PolyHandle poly;
   Pattern *pat;

/* Point Calculations */

void AddPt(srcPt,dstPt)
   Point *srcPt,*dstPt;
void SubPt(srcPt,dstPt)
   Point *srcPt,*dstPt;
pascal void SetPt(pt,h,v)
   Point *pt;
   short h,v;
Boolean EqualPt(pt1,pt2)
   Point *pt1, *pt2;
pascal void LocalToGlobal(pt)
   Point *pt;
pascal void GlobalToLocal(pt)
   Point *pt;

/* Miscellaneous Utility Routines */

pascal short Random()
pascal Boolean GetPixel(h,v)
   short h,v;
void StuffHex(thingPtr,s)
```

```
   Ptr thingPtr;
   char *s;
pascal void ScalePt(pt,srcRect,dstRect)
   Point *pt;
   Rect *srcRect,*dstRect;
pascal void MapPt(pt,srcRect,dstRect)
   Point *pt;
   Rect *srcRect,*dstRect;
pascal void MapRect(r,srcRect,dstRect)
   Rect *r,*srcRect,*dstRect;
pascal void MapRgn(rgn,srcRect,dstRect)
   RgnHandle rgn;
   Rect *srcRect,*dstRect;
pascal void MapPoly(poly,srcRect,dstRect)
   PolyHandle poly;
   Rect *srcRect,*dstRect;

/* Bottleneck Interface */

pascal void SetStdProcs(procs)
   QDProcsPtr procs;
void StdText(byteCount,textAddr,numer,denom)
   short byteCount;
   Ptr textAddr;
   Point *numer, *denom;
void StdLine(newPt)
   Point *newPt;
pascal void StdRect(verb,r)
   GrafVerb verb;
   Rect *r;
pascal void StdRRect(verb,r,ovalWidth,ovalHeight)
   GrafVerb verb;
   Rect *r;
   short ovalWidth,ovalHeight;
pascal void StdOval(verb,r)
   GrafVerb verb;
   Rect *r;
pascal void StdArc(verb,r,startAngle,arcAngle)
   GrafVerb verb;
   Rect *r;
   short startAngle,arcAngle;
pascal void StdPoly(verb,poly)
   GrafVerb verb;
   PolyHandle poly;
pascal void StdRgn(verb,rgn)
   GrafVerb verb;
   RgnHandle rgn;
pascal void StdBits(srcBits,srcRect,dstRect,mode,maskRgn)
   BitMap *srcBits;
   Rect *srcRect,*dstRect;
   short mode;
   RgnHandle maskRgn;
pascal void StdComment(kind,dataSize,dataHandle)
   short kind,dataSize;
   Handle dataHandle;
pascal short StdTxMeas(byteCount,textAddr,numer,denom,info)
   short byteCount;
   Ptr textAddr;
```

```
    Point *numer,*denom;
    FontInfo *info;
pascal void StdGetPic(dataPtr,byteCount)
    Ptr dataPtr;
    short byteCount;
pascal void StdPutPic(dataPtr,byteCount)
    Ptr dataPtr;
    short byteCount;
```

USER ROUTINES

```
pascal void MyText(byteCount,textAddr,numer,denom)
    short byteCount;
    Ptr textAddr;
    Point numer,denom;
pascal void MyLine(newPt)
    Point newPt;
pascal void MyRect(verb,r)
    GrafVerb verb;
    Rect *r;
pascal void MyRRect(verb,r,ovWd,ovHt)
    GrafVerb verb;
    Rect *r;
    short ovWd,ovHt;
pascal void MyOval(verb,r)
    GrafVerb verb;
    Rect *r;
pascal void MyArc(verb,r,startAngle,arcAngle)
    GrafVerb verb;
    Rect *r;
    short startAngle,arcAngle;
pascal void MyPoly(verb,poly)
    GrafVerb verb;
    PolyHandle poly;
pascal void MyRgn(verb,rgn)
    GrafVerb verb;
    RgnHandle rgn;
pascal void MyBits(srcBits,srcRect,dstRect,mode,maskRgn)
    BitMap *srcBits;
    Rect *srcRect,*dstRect;
    short mode;
    RgnHandle maskRgn;
pascal void MyComment(kind,dataSize,dataHandle)
    short kind,dataSize;
    Handle dataHandle;
pascal short MyTxMeas(byteCount,textAddr,numer,denom,info)
    short byteCount;
    Ptr textAddr;
    Point *numer,*denom;
    FontInfo *info;
pascal void MyGetPic(dataPtr,byteCount)
    Ptr dataPtr;
    short byteCount;
pascal void MyPutPic(dataPtr,byteCount)
    Ptr dataPtr;
    short byteCount;
```

DESCRIPTION

QuickDraw II is the Cortland graphics package. It is based on a subset of the Macintosh QuickDraw subroutines, which support the operations useful in menus and windews, such as drawing lines, drawing text characters, and filling areas. In addition, QuickDraw II supports Cortland's standard display mode, the new color super Hi-Res Graphics.

For more detailed information see the QuickDraw chapter of *Cortland Tools Reference*.

WARNING

User routines MyText and MyLine are not identical to their counterparts StdText and StdLine. Point parameters to MyText and MyLine are passed by value; the corresponding parameters to StdText and StdLine are passed by reference.

*** True for Cortland? ***

NAME

sane—SANE Numerics

SYNOPSIS

*** This should be exactly like the Macintosh SANE C library. Is it? ***

```
#include <sane.h>

/* Decimal Representation Constants */

#define  SIGDIGLEN           20     /* significant decimal digits     */
#define  DECSTROUTLEN        80     /* max length for decimal string  */

/* Decimal Formatting Styles */

#define  FLOATDECIMAL         0

#define  FIXEDDECIMAL         1

/* Exceptions */

#define  INVALID              1
#define  UNDERFLOW            2
#define  OVERFLOW             4
#define  DIVBYZERO            8
#define  INEXACT             16

/* Ordering Relations */

#define  GREATERTHAN          0

#define  LESSTHAN             1
#define  EQUALTO              2
#define  UNORDERED            3

/* Inquiry Classes */

#define  SNAN                 0
#define  QNAN                 1
#define  INFINITE             2
#define  ZERONUM              3
#define  NORMALNUM            4
#define  DENORMALNUM          5

/* Rounding Directions */

#define  TONEAREST            0
#define  UPWARD               1
#define  DOWNWARD             2
#define  TOWARDZERO           3

/* Rounding Precisions */
```

```
#define  EXTPRECISION          0
#define  DBLPRECISION          1
#define  FLOATPRECISION        2

typedef short exception;       /* sum of INVALID...INEXACT              */

typedef short relop;           /* relational operator                   */

typedef short numclass;        /* inquiry class                         */

typedef short rounddir;        /* rounding direction                    */

typedef short roundpre;        /* rounding precision                    */

typedef short environment;

typedef struct decimal {
   char sgn, unused;           /* sign 0 for +, 1 for -                 */
   short exp;                  /* decimal exponent                      */
   struct {unsigned char length, text[SIGDIGLEN], unused} sig;
                               /* significant digits                    */

} decimal;

typedef struct decform {
   char style, unused;         /* FLOATDECIMAL or FIXEDDECIMAL          */
   short digits;
} decform;

typedef void (*haltvector)();

/* Conversions between Binary and Decimal Records */

void num2dec(f,x,d)            /* d <-- x, according to format f        */
   decform *f;
   extended x;
   decimal *d;
extended dec2num(d)            /* returns d as extended                 */
   decimal *d;

/* Conversions between Decimal Records and ASCII Strings */

void dec2str(f,d,s)            /* s <-- d, according to format f        */
   decform *f;
   decimal *d;
   char *s;

void str2dec(s,ix,d,vp) /* on input ix is starting index into s, on     */
   char *s;             /* output ix is one greater than index of last */
   short *ix,*vp;       /* character of longest numeric substring;     */
   decimal *d;          /* boolean vp = "s begining at given ix is a   */
                        /* valid numeric string or a valid prefix of   */
                        /* some numeric string"                        */

/* Arithmetic, Auxiliary, and Elementary Functions */

extended fabs(x)               /* absolute value                        */
```

```
    extended x;
extended remainder(x,y,quo)     /* IEEE remainder; quo <-- 7 low order  */
    extended x,y;               /*   bits of integer quotient x/y,     */
    short *quo;                 /*   -127 <= quo <= 127                */
extended sqrt(x)                /* square root                         */
    extended x;
extended rint(x)                /* round to integral value             */
    extended x;
extended scalb(n,x)             /* binary scale: x * 2^n;              */
    short n;
    extended x;
extended logb(x)                /* binary log:                         */
    extended x;                 /*   binary exponent of normalized x   */
extended copysign(x,y)          /* y with sign of x                    */
    extended x,y;
extended nextfloat(x,y)         /* next float representation after     */
    extended x,y;               /*   (float) x in direction of (float) y*/
extended nextdouble(x,y)        /* next double representation after    */
    extended x,y;               /*  (double) x in direction of (double) y */
extended nextextended(x,y)      /* next extended representation after x */
    extended x,y;               /*   in direction of y                 */
extended log2(x)                /* base-2 log                          */
    extended x;
extended log(x)                 /* base-e log                          */
    extended x;
extended log1(x)                /* log(1 + x)                          */
    extended x;
extended exp2(x)                /* base-2 exponential                  */
    extended x;
extended exp(x)                 /* base-e exponential                  */
    extended x;
extended expl(x)                /* exp(x) - 1                          */
    extended x;
extended power(x,y)             /* general exponential: x ^ y          */
    extended x,y;
extended ipower(x,i)            /* integer exponential: x ^ i          */
    extended x;
    short i;
extended compound(r,n)          /* compound: (1 + r) ^ n               */
    extended r,n;
extended annuity(r,n)           /* annuity: (1 - (1 + r) ^ (-n)) / r   */
    extended r,n;
extended tan(x)                 /* tangent                             */
    extended x;
extended sin(x)                 /* sine                                */
    extended x;
extended cos(x)                 /* cosine                              */
    extended x;
extended atan(x)                /* arctangent                          */
    extended x;
extended randomx(x)             /* returns next random number; updates x; */
    extended *x;                /*   x integral, 1 <= x <= 2^31 - 2    */

/* Inquiry Routines */

numclass classfloat(x)          /* class of (float) x                  */
    extended x;
numclass classdouble(x)         /* class of (double) x                 */
```

```
   extended x;
numclass classcomp(x)          /* class of (comp) x                    */
   extended x;
numclass classextended(x)      /* class of x                           */
   extended x;
long signnum(x)                /* returns 0 for +, 1 for -             */
   extended x;

/* Environment Access Routines */
/*  An exception variable encodes the exceptions
          whose sum is its value */

void setexception(e,s)         /* clrs e flags if s is 0, sets e flags */
   exception e;                /*   otherwise; may cause halt          */
   long s;
long testexception(e)          /* returns 1 if any e flag is set,      */
   exception e;                /*   returns 0 otherwise                */
void sethalt(e,s)              /* disables e halts if s is 0,          */
   exception e;                /*   enables e halts otherwise          */
   long s;
long testhalt(e)               /* returns 1 if any e halt is enabled,  */
   exception e;                /*   returns 0 otherwise                */
void setround(r)               /* sets rounding direction to r         */
   rounddir r;
rounddir getround()            /* returns rounding direction           */
void setprecison(p)            /* sets rounding precision to p         */
   roundpre p;
roundpre getprecsion()         /* returns rounding precision           */
void setenvironment(e)         /* sets environment to e                */
   environment e;
void getenvironment(e)         /* e <-- environment                    */
   environment *e;
void procentry(e)              /* e <-- environment;                   */
   environment *e;             /*   environment <-- IEEE default       */
void procexit(e)               /* temp <--exceptions; environment <-- e; */
   environment e;              /*   signals exceptions in temp         */
haltvector gethaltvector()     /* returns halt vector                  */
void sethaltvector(v)          /* halt vector <-- v                    */
   haltvector v;

/* Comparision Routine */

relop relation(x,y)            /* returns relation such that           */
   extended x,y;               /*   "x relation y" is true             */

/* NaNs and Special Constants */

extended nan(c)                /* returns NaN with code c              */
   unsigned char c;
extended inf()                 /* infinity                             */
extended pi()                  /* pi                                   */
```

DESCRIPTION

These routines together with Apple's C language fully support the Standard Apple Numeric Environment (SANE). They provide a scrupulously conforming implementation of extended-precision floating-point arithmetic as specified by IEEE

Standard 754. The SANE Tool Set contains the same routines as Pack 4, Pack 5, and Pack 7 of the Macintosh Toolkit.

The Standard Apple Numeric Environment is documented in the *Apple Numerics Manual*.

NAME

scheduler—Scheduler

SYNOPSIS

This code will be added later.

DESCRIPTION

The Scheduler makes it possible to delay the execution of tasks that require non-reentrant system code whenever that code is already in use. Non-reentrant resources indicate that they are in use by modifying a flag called the Busy Word. The Scheduler maintains a queue of processes waiting to use non-reentrant resources. By keeping track of the Busy Word, the Scheduler determines when to activate the next process in the queue.

For more detailed information, see the "Scheduler " chapter of the *Cortland Tools Reference*.

NAME

scrap—Scrap Manager

SYNOPSIS

*** This is the Mac code. The Cortland code will resemble it in functionality but differ in form. Stay tuned. ***

```
#include <types.h>
#include <scrap.h>

#define noScrapErr    (-100)    /* desk scrap isn't initialized  */
#define noTypeErr     (-102)    /* no data of the requested type */

typedef struct ScrapStuff {
   long       scrapSize;
   Handle     scrapHandle;
   short      scrapCount;
   short      scrapState;
   StringPtr  scrapName;
} ScrapStuff, *PScrapStuff;

/* Getting Desk Scrap Information */

pascal PScrapStuff InfoScrap()

/* Keeping the Desk Scrap on the Disk */

pascal long UnloadScrap()
pascal long LoadScrap()

/* Reading from the Desk Scrap */

pascal long GetScrap(hDest,theType,offset)
   Handle hDest;
   ResType theType;
   long *offset;

/* Writing to the Desk Scrap */

pascal long ZeroScrap()
pascal long PutScrap(length,theType,source)
   long length;
   ResType theType;
   Ptr source;
```

DESCRIPTION

The Scrap Manager provides a mechanism for cutting and pasting between applications and desk accessories.

For more detailed information see the Scrap Manager chapter of the *Cortland Tools Reference*.

NAME

segload—Segment Loader

SYNOPSIS

*** This is the Mac code. The Cortland code will resemble it in functionality but differ in form. Stay tuned. ***

```
#include <types.h>
#include <segload.h>

/* Message returned by CountAppFiles */

#define  appOpen          0        /* Open the document(s) */
#define  appPrint         1        /* Print the document(s) */

typedef struct AppFile {
   short       vRefNum;            /* volume reference number */
   OSType      fType;              /* file type */
   short       versNum;            /* version number */
   Str255      fName;              /* file name */
} AppFile;

pascal void UnloadSeg(routineAddr)
   Ptr routineAddr;
void CountAppFiles(message,count)
   short *message,*count;
void GetAppFiles(index,theFile)
   short index;
   AppFile *theFile;
void ClrAppFiles(index)
   short index;
void GetAppParms(apName,apRefNum,apParam)
   char *apName;
   short *apRefNum;
   Handle *apParam;
pascal void ExitToShell()
```

DESCRIPTION

The Segment Loader is the part of the Cortland Toolbox that lets you divide your application into several parts and have only some of them in memory at a time. When an application starts up, the Segment Loader also provides it with a list of files to open or print.

*** True? ***

For more detailed information see the Segment Loader chapter of the *Cortland Tools Reference*.

NAME

sound—Sound Manager

SYNOPSIS

*** This is the Mac code. The Cortland code will resemble it in functionality but differ greatly in form. Stay tuned. ***

```
#include <types.h>
#include <sound.h>


/* Mode values for synthesizers */

#define  swMode          (-1)     /* square-wave synthesizer */
#define  ftMode            1      /* four-tone synthesizer */
#define  ffMode            0      /* free-form synthesizer */

/* Free-Form synthesizer */

typedef unsigned char FreeWave[30001];

typedef struct FFSynthRec {
   short       mode;              /* always ffMode */
   Fixed       count;             /* "sizing" factor */
   FreeWave    waveBytes;         /* waveform description */
} FFSynthRec, *FFSynthPtr;

/* Square-Wave synthesizer */

typedef struct Tone {
   short       count;             /* frequency */
   short       amplitude;         /* amplitude, 0-255 */
   short       duration;          /* duration in ticks */
} Tone;

typedef Tone Tones[5001];

typedef struct SWSynthRec {
   short       mode;              /* always swMode */
   Tones       triplets;          /* sounds */
} SWSynthRec, *SWSynthPtr;

/* Four-Tone Synthesizer */

typedef unsigned char Wave[256];

typedef Wave *WavePtr;

typedef struct FTSoundRec {
   short       duration;          /* duration in ticks */
   Fixed       sound1Rate;        /* tone 1 cycle rate */
   long        sound1Phase;       /* tone 1 byte offset */
   Fixed       sound2Rate;        /* tone 2 cycle rate */
   long        sound2Phase;       /* tone 2 byte offset */
```

```
   Fixed       sound3Rate;     /* tone 3 cycle rate */
   long        sound3Phase;    /* tone 3 byte offset */
   Fixed       sound4Rate;     /* tone 4 cycle rate */
   long        sound4Phase;    /* tone 4 byte offset */
   WavePtr     sound1Wave;     /* tone 1 wave form */
   WavePtr     sound2Wave;     /* tone 2 wave form */
   WavePtr     sound3Wave;     /* tone 3 wave form */
   WavePtr     sound4Wave;     /* tone 4 wave form */
} FTSoundRec, *FTSndRecPtr;

typedef struct FTSynthRec {
   short       mode;           /* always ftMode */
   FTSndRecPtr sndRec;         /* tones to play */
} FTSynthRec, *FTSynthPtr;

void StartSound(synthRec,numBytes,completionRtn)
   Ptr synthRec;
   long numBytes;
   ProcPtr completionRtn;
void StopSound()
Boolean SoundDone()
void GetSoundVol(level)
   short *level;
void SetSoundVol(level)
   short level
```

DESCRIPTION

The Sound Manager is a Cortland device driver for handling sound and music generation in a Cortland application. It provides access to the Ensoniq chip.

For more detailed information see the Sound Manager chapter of *Cortland Tools Reference*.

NAME

text—Text Screen Tools

SYNOPSIS

*** The code for this will be added later. ***

DESCRIPTION

The Text Screen Tools make it possible for applications to use the text maodes without switching modes and moving to bank zero.

For more detailed information see the TextEdit chapter of *Cortland Tools Reference*.

NAME

toolloc—Tool Locator

SYNOPSIS

This code will be added later.

DESCRIPTION

The Tool Locator provides the mechanism for dispatching tool calls. It allows tool sets to reside either in ROM or in RAM, transparently to an application.

For more detailed information see the Tool Locator chapter of *Cortland Tools Reference*.

NAME

types—common defines and types

SYNOPSIS

*** This is the Mac code. It is not known how the corresponding Cortland code will resemble it. Stay tuned. ***

```
#include <types.h>

#define nil 0
#define NULL 0

typedef enum {false,true} Boolean;
typedef char *Ptr;
typedef Ptr *Handle;
typedef long (*ProcPtr)();
typedef ProcPtr *ProcHandle;
typedef long Fixed;
typedef unsigned long ResType;
typedef long OSType;
typedef short OSErr;
typedef short Style;
typedef struct Point {
   short    v;
   short    h;
} Point;
typedef struct Rect {
   short    top;
   short    left;
   short    bottom;
   short    right;
} Rect;

#define String(size) struct {\
   unsigned char length; unsigned char text[size];}
typedef String(255) Str255, *StringPtr, **StringHandle;
```

DESCRIPTION

These defines and types are shared by several Cortland libraries.

The define *String* approximates Pascal strings. It creates a struct, not an array. Remember to use & when passing structs as parameters.

NAME

windows—Window Manager

SYNOPSIS

*** This is the Mac code. The Cortland code will resemble it in functionality but differ in form. Stay tuned. ***

```
#include <types.h>
#include <quickdraw.h>
#include <windows.h>

/* Window Definition Procedure IDs */

#define documentProc    0
#define dBoxProc        1
#define plainDBox       2
#define altDBoxProc     3
#define noGrowDocProc   4
#define rDocProc       16

/* Types of Windows */

#define dialogKind      2
#define userKind        8

/* FindWindow Result Codes */

#define inDesk          0
#define inMenuBar       1
#define inSysWindow     2
#define inContent       3
#define inDrag          4
#define inGrow          5
#define inGoAway        6
#define inZoomIn        7
#define inZoomOut       8

/* Axis Constraints for DragGrayRgn */

#define noConstraint    0
#define hAxisOnly       1
#define vAxisOnly       2

/* Messages to window definition functions */

#define wDraw           0
#define wHit            1
#define wCalcRgns       2
#define wNew            3
#define wDispose        4
#define wGrow           5
#define wDrawGIcon      6

/* defProc Hit Test Codes */
```

```
#define wNoHit          0
#define wInContent      1
#define wInDrag         2
#define wInGrow         3
#define wInGoAway       4
#define wInZoomIn       5
#define wInZoomOut      6

#define deskPatID      16

typedef GrafPtr WindowPtr;

typedef struct WindowRecord {
   GrafPort       port;
   short          windowKind;
   Boolean        visible;
   Boolean        hilited;
   Boolean        goAwayFlag;
   Boolean        spareFlag;
   RgnHandle      strucRgn;
   RgnHandle      contRgn;
   RgnHandle      updateRgn;
   ProcHandle     windowDefProc;
   Handle         dataHandle;
   StringHandle   titleHandle;
   short          titleWidth;
   struct ControlRecord **controlList;
   struct WindowRecord *nextWindow;
   PicHandle      windowPic;
   long           refCon;
} WindowRecord, *WindowPeek;

/* Initialization and Allocation */

pascal void InitWindows()
pascal void GetWMgrPort(wPort)
   GrafPtr *wPort;
WindowPtr NewWindow(wStorage,boundsRect,title,visible,procID,behind,
      goAwayFlag,refCon)
   Ptr wStorage;
   Rect *boundsRect;
   char *title;
   Boolean visible;
   short procID;
   WindowPtr behind;
   Boolean goAwayFlag;
   long refCon;
pascal WindowPtr GetNewWindow(windowID,wStorage,behind)
   short windowID;
   Ptr wStorage;
   WindowPtr behind;
pascal void CloseWindow(theWindow)
   WindowPtr theWindow;
pascal void DisposeWindow(theWindow)
   WindowPtr theWindow;

/* Window Display */
```

```
void SetWTitle(theWindow,title)
   WindowPtr theWindow;
   char *title;
void GetWTitle(theWindow,title)
   WindowPtr theWindow;
   char *title;
pascal void SelectWindow(theWindow)
   WindowPtr theWindow;
pascal void HideWindow(theWindow)
   WindowPtr theWindow;
pascal void ShowWindow(theWindow)
   WindowPtr theWindow;
pascal void ShowHide(theWindow,showFlag)
   WindowPtr theWindow;
   Boolean showFlag;
pascal void HiliteWindow(theWindow,fHiLite)
   WindowPtr theWindow;
   Boolean fHiLite;
pascal void BringToFront(theWindow)
   WindowPtr theWindow;
pascal void SendBehind(theWindow,behindWindow)
   WindowPtr theWindow;
   WindowPtr behindWindow;
pascal WindowPtr FrontWindow()
pascal void DrawGrowIcon(theWindow)
   WindowPtr theWindow;

/* Mouse Location */

short FindWindow(thePt,theWindow)
   Point *thePt;
   WindowPtr *theWindow;
Boolean TrackGoAway(theWindow,thePt)
   WindowPtr theWindow;
   Point *thePt;
pascal Boolean TrackBox(theWindow,thePt,partCode)
   WindowPTr theWindow;
   Point *thePt;
   short partCode;

/* Window Movement and Sizing */

pascal void MoveWindow(theWindow,hGlobal,vGlobal,front)
   WindowPtr theWindow;
   short hGlobal,vGlobal;
   Boolean front;
void DragWindow(theWindow,startPt,boundsRect)
   WindowPtr theWindow;
   Point *startPt;
   Rect *boundsRect;
long GrowWindow(theWindow,startPt,sizeRect)
   WindowPtr theWindow;
   Point *startPt;
   Rect *sizeRect;
pascal void SizeWindow(theWindow,w,h,fUpdate)
   WindowPtr theWindow;
   short w,h;
   Boolean fUpdate;
```

```
pascal void ZoomWindow(theWindow,partCode,front)
   WindowPtr theWindow;
   short partCode;
   Boolean front;

/* Update Region Maintenance */

pascal void InvalRect(badRect)
   Rect *badRect;
pascal void InvalRgn(badRgn)
   RgnHandle badRgn;
pascal void ValidRect(goodRect)
   Rect *goodRect;
pascal void ValidRgn(goodRgn)
   RgnHandle goodRgn;
pascal void BeginUpdate(theWindow)
   WindowPtr theWindow;
pascal void EndUpdate(theWindow)
   WindowPtr theWindow;

/* Miscellaneous Utilities */

pascal void SetWRefCon(theWindow,data)
   WindowPtr theWindow;
   long data;
pascal long GetWRefCon(theWindow)
   WindowPtr theWindow;
pascal void SetWindowPic(theWindow,pic)
   WindowPtr theWindow;
   PicHandle pic;
pascal PicHandle GetWindowPic(theWindow)
   WindowPtr theWindow;
long PinRect(theRect,thePt)
   Rect *theRect;
   Point *thePt;
long DragGrayRgn(theRgn,startPt,limitRect,slopRect,axis,actionProc)
   RgnHandle theRgn;
   Point *startPt;
   Rect *limitRect;
   Rect *slopRect;
   short axis;
   ProcPtr actionProc;

/* Low-Level Routines */

pascal Boolean CheckUpdate(theEvent)
   struct EventRecord *theEvent;
pascal void ClipAbove(window)
   WindowPeek window;
pascal void SaveOld(window)
   WindowPeek window;
pascal void DrawNew(window,update)
   WindowPeek window;
   Boolean update;
pascal void PaintOne(window,clobberedRgn)
   WindowPeek window;
   RgnHandle clobberedRgn;
pascal void PaintBehind(startWindow,clobberedRgn)
```

```
    WindowPeek startWindow;
    RgnHandle clobberedRgn;
pascal void CalcVis(window)
    WindowPeek window;
pascal void CalcVisBehind(startWindow,clobberedRgn)
    WindowPeek startWindow;
    RgnHandle clobberedRgn;
```

USER ROUTINES

```
pascal MyAction()

pascal long MyWindow(varCode,theWindow,message,param)
    short varCode;
    WindowPtr theWindow;
    short message;
    long param;
```

DESCRIPTION

The Window Manager provides routines for creating and manipulating windows. It creates them, activates them, moves them, resizes them, and closes them in response to calls from an application. It keeps track of overlapping windows and posts an event so the application can redraw newly uncovered windows. When the user presses the mouse button, the Window Manager tells the application which part of which window the cursor was in.

For more detailed information see the Window Manager chapter of *Cortland Tools Reference*.

Chapter 6

SANE and the C SANE Library

This chapter describes the Standard Apple Numeric Environment (SANE) and the routines contained in the SANE library CSANELib.o. SANE is the basis for all floating-point mathematical calculations performed by Cortland Workshop C. It meets all requirements for extended-precision floating-point arithmetic as prescribed by IEEE Standard 754. The chapter contains three parts:

- A discussion of the floating-point data types provided by SANE.
- A description of the constants and types used in the C SANE library.
- A description of the functions contained in the C SANE library.

SANE ensures that all floating-point operations are performed consistently and that they return the most accurate results possible.

SANE provides an easy-to-use, flexible environment for floating-point calculations. It gives you extremely accurate results without extra coding. You can write C programs that use only the standard C float type and be confident that your results are as accurate as possible within that format.

Programmers who are interested in precision beyond that possible using only the float type can use the other floating-point types provided as an extension to C by SANE. In addition, the SANE library contains numerical functions not found in standard C and routines for controlling the environment in which floating-point calculations are performed.

If you are using CPW C for advanced numerical programming, you might be interested in the complete and detailed description of SANE, which is contained in the *Apple Numerics Manual*, available from your Apple dealer.

The SANE Data Types

Cortland Workshop C supplements the `float` type with three others: `double`, `extended`, and `comp`.

A Note on Terminology

SANE is designed to be a generic system that can be used with a variety of high-level languages. SANE provides the three floating-point types specified by the IEEE Standard (where they are called `single`, `double`, and `extended`). CPW C uses the SANE type single as the C type `float`.

Descriptions of the Types

The `float` type is the smallest format for use with floating-point numbers. It stores floating-point numbers using 32 bits of storage.

The `double` type is twice the size of the `float` type. It uses 64 bits for storage.

The `extended` type is larger yet—it uses an 80-bit format. All arithmetic involving real-type values is done using the `extended` type.

The `comp` type stores integral values in a 64-bit format. Arithmetic done with operands of type `comp` uses the extended type. Results assigned to a variable of type `comp` are converted from extended.

Choosing a Data Type

Typically, picking a data type requires that you determine the trade-offs between

- fixed- or floating-point form
- precision
- range
- memory usage

The precision, range, and memory usage for each SANE data type are shown in Table 5-1, SANE Data Types.

Many programs require a counting type that counts things (pennies, dollars, widgets) exactly. Using SANE, you can write a program that deals with monetary values by representing these values as integral numbers of cents or mils, which can be stored exactly in the `comp` type. The sum, difference, or product of any two `comp` values is exact if the magnitude of the result does not exceed $2^{63}-1$ (that is, 9,223,372,036,854,775,807). This number is larger than the U.S. national debt expressed in Argentine pesos. In addition, `comp` values (for example, the results of accounting computations) can be mixed with `extended` values in floating-point computations (such as compound interest).

Arithmetic with `comp`-type variables, like all SANE arithmetic, is done internally using the `extended` type for arithmetic. There is no loss of precision, as conversion from `comp` to `extended` is always exact. You can save by storing numbers in the `comp` type, which is 20 percent shorter than `extended` (64 versus 80 bits).

Values Represented

The floating-point types (`float`, `double`, and `extended`) store binary representations of a **sign** (+ or –), an **exponent**, and a **significand**. A represented number has the value

$$\pm\, significand * 2^{exponent}$$

where the significand has a float bit to the left of the binary point (that is, $0 \le significand < 2$).

Range and Precision of SANE Types

The range and precision of the floating-point types supported by SANE and CPW C are shown in Table 5–1. Decimal ranges are expressed as chopped two-digit decimal representations of the exact binary values.

Table 5-1. SANE Data Types

Type identifier	`float`	`double`	`comp`	`extended`
Size (bytes:bits)	4:32	8:64	8:64	10:80
Binary exponent range				
Minimum	–126	–1022	----	16383
Maximum	127	1023	----	16383
Significand precision				
Bits	24	53	63	64
Decimal digits	7–8	15–16	18–19	19–20
Decimal range (approximate)				
Min negative	–3.4E+38	–1.7E+308	≈9.2E18	–1.1E+4932
Max neg norm	–1.2E–38	–2.3E–308		–1.7E–4932
Max neg denorm	–1.5E–45	–5.0E–324		–1.9E–4951
Min pos denorm	1.5E–45	5.0E–324		1.9E–4951
Min pos norm	1.2E–38	2.3E–308		1.7E–4932
Max positive	3.4E+38	1.7E+308	≈9.2E18	1.1E+4932
Infinities	Yes	Yes	No	Yes
NaNs	Yes	Yes	Yes	Yes

Example

Using the `float` type, the largest representable number has

significand	$= 2 - 2^{-23}$
	$= 1.11111111111111111111111_2$
exponent	$=127$
value	$(2 - 2^{-23}) * 2^{127}$
	$\approx 3.403 * 10^{38}$

the smallest representable positive normalized number has

significand	$=1$
	$=1.00000000000000000000000_2$
exponent	$=-126$
value	$=1 * 2^{-126}$

$$\approx 1.175 * 10^{-38}$$

and the smallest representable positive denormalized number has

significand	$=2^{-23}$
	$=0.00000000000000000000001_2$
exponent	$=-126$
value	$=2^{-23} * 2^{-126}$
	$\approx 1.401 * 10^{-45}$

The `float` Type

A 32-bit `float` number is divided into three fields as shown below.

*** copy of figure on page 16 of *Apple Numerics Manual* ***

Figure 6-1: float type

The value *v* of the number is determined by these fields:

If $0 < e < 255$,	then $v = (-1)^s * 2^{(e-127)} * (1.f)$.
If $e = 0$ and $f \neq 0$,	then $v = (-1)^s * 2^{(-126)} * (0.f)$.
If $e = 0$ and $f = 0$,	then $v = (-1)^s * 0$.
If $e = 255$ and $f = 0$,	then $v = (-1)^s * \infty$.
If $e = 255$ and $f \neq 0$,	then v is a NaN.

The `double` Type

A 64-bit double number is divided into three fields as shown below.

*** copy of top figure on page 17 of *Apple Numerics Manual* ***

Figure 6-2: double type

The value *v* of the number is determined by these fields:

If $0 < e < 2047$,	then $v = (-1)^s * 2^{(e-1023)} * (1.f)$.
If $e = 0$ and $f \neq 0$,	then $v = (-1)^s * 2^{(-1022)} * (0.f)$.
If $e = 0$ and $f = 0$,	then $v = (-1)^s * 0$.
If $e = 2047$ and $f = 0$,	then $v = (-1)^s * \infty$.
If $e = 2047$ and $f \neq 0$,	then v is a NaN.

The `comp` Type

A 64-bit comp number is divided into two fields as shown below.

*** copy of bottom figure on page 17 of *Apple Numerics Manual* ***

Figure 6-3: Comp type

The value *v* of the number is determined by these fields:

If $s = 1$ and $d = 0$,	then *v* is the unique comp NaN.
Otherwise,	*v* is the two's-complement value of the 64-bit representation.

The `extended` Type

An 80-bit extended format number is divided into four fields as shown below.

*** copy of figure on page 18 of *Apple Numerics Manual* ***

Figure: 6-4: Extended Type

The value *v* of the number is determined by these fields:

If $0 <= e < 32767$,	then $v = (-1)^s * 2^{(e-16383)} * (i.f)$.
If $e = 32767$ and $f = 0$,	then $v = (-1)^s * \infty$, regardless of *i*.
If $e = 32767$ and $f \neq 0$,	then *v* is a NaN, regardless of *i*.

Extended Arithmetic

While the CPW C types `float`, `double`, and `comp` are intended for economical data storage, the `extended` type is the foundation for all arithmetic computation. As specified by the IEEE Standard, all basic arithmetic operations, including addition, subtract, multiply, divide, and square root, yield the best possible results. In CPW C these operations produce `extended` results, so they are accurate to a precision of 19 decimal digits, throughout a range exceeding 10^{-4900} to 10^{+4900}.

CPW C takes advantage of `extended` arithmetic by storing all non-integer numeric constants in the extended format, and by evaluating all non-integer numeric expressions to `extended`, regardless of the types involved. For example, the entire right side of the assignment below will be computed in `extended` before being converted to the type of the left side:

```
float x, a, b, c;

{
    ...
    x = ( b + sqrt(b * b - a * c) ) / a;
```

With no special effort by the programmer, CPW C performs computations using extended precision and range. Extra precision means smaller roundoff errors, so that results are more accurate, more often. Extra range means overflow and underflow are rarer, so that programs work more often.

By following a few simple programming practices you can exploit the `extended` type, beyond what CPW C does for you automatically.

Declare variables used for intermediate results to be of type `extended`. This practice is illustrated in the following example, which computes a sum of products.

```
float    sum;
float    x[N], y[N];
int      i;
extended t;
{
...
t = 0.0;
for (i = 0; i < N; i++)
    t = t + x[i] * y[i];
sum = t;
```

Had `t` been declared as `float`, like the input arrays `x` and `y` and the result `sum`, each time through the loop the assignment to `t` would have caused a roundoff error at the limit of float precision. In the example, all roundoff errors are at the limit of extended precision, except for the one caused by the assignment of `t` to sum. This means roundoff errors will be less likely to accumulate to produce an inaccurate result.

Declare formal value parameters and function results to be of type `extended`, rather than `float`, `double`, or `comp`. This saves CPW C from having to do unnecessary conversions between numeric types, which may result in loss of accuracy. The example below illustrates this practice.

```
#include <SANE.h>
extended area(radius)
    extended radius;
{
    return pi() * radius * radius;
}
```

Number Classes

Representations in the SANE data formats fall into five classes:

- Normalized numbers—like `3.0`, `75.8`, `-2.3e78` and all others that can be represented with a leading significand bit of 1.
- Zero—`+0.0` and `-0.0`.
- Infinities—positive and negative infinity.
- NaNs—short for Not-a-Number.
- Denormalized numbers—nonzero numbers that are too small for normalized representation.

Infinities: Infinities are special SANE representations that can arise in two ways from operations on finite values:

- When a operation should produce an exact mathematical infinity (such as `1.0/0.0`), the result is an infinity.

- When an operation produces a number with magnitude too great for the number's intended floating-point format, the result may (depending on the current rounding direction) be an infinity.

Library CSANELib.o contains a function `inf` that returns the constant INF, which has the value positive infinity. `INF` also represents infinity for input and output of floating-point values. Infinities behave like mathematical infinities. For example, `1-INF = -INF`. Infinities can be helpful even when "infinity arithmetic" is not the goal. For example, if `X*X` is too large for the `extended` format, the expression `1 + 1/(X*X)` still computes to the correct value of `1.0` (assuming overflow halts are off).

Try this:

```
main()
{
    extended x;

    x = 1e4000;
    printf("x * x  =  %f \n", x * x);
    printf("1 / (x * x)  =  %f \n", 1 / (x * x));
    printf("1 + 1 / (x * x)  =  %f \n", 1 + 1 / (x * x));
}
```

NaNs: Another special SANE representation is a NaN (Not-a-Number). A NaN is produced whenever an operation cannot produce a meaningful numeric result. For example, `0.0/0.0` and `sqrt(-1.0)` yield NaNs.

Each time a NaN is generated, an associated NaN code is returned as part of the NaN's representation. This code tells you what kind of operation caused the NaN to be created. NaN codes, shown in Table 5-2, can help with debugging.

Table 5-2. NaN Codes

Code	Meaning
1	Invalid square root, such as `sqrt(-1.0)`
2	Invalid addition, such as `(+INF) - (+INF)`
4	Invalid division, such as `0.0/0.0`
8	Invalid multiplication, such as `0.0 * INF`
9	Invalid remainder, such as X REM 0
17	Attempt to convert invalid ASCII string
20	Result of converting the comp NaN to floating-point format
21	Attempt to create a NaN with a zero code
33	Invalid argument to trig routine
34	Invalid argument to inverse trig routine
36	Invalid argument to log routine
37	Invalid argument to x^i or x^y routine
38	Invalid argument to financial function

The statement `x = 0.0/0.0` will produce the result NAN(004) provided the invalid operation halt is off. `NAN(004)`, `nan(4)`, and `NaN` are examples of acceptable input for reading a NaN into a SANE variable at execution time. At compile time, you specify a

NaN by means of the `nan` function provided in CSANELib.o. See the *fconstants* page in the C SANE Library section of this chapter for more information about the `nan` function.

Denormalized Numbers: Whenever possible, SANE stores values in normalized form: the most significant bit of the significand is a one, rather than a zero.

However, when a very small number is being stored, and the exponent is the smallest possible negative value, it is possible to store still smaller values by storing leading zeroes. For example,

$1.0..0_2 * 2^{-126}$ — smallest normalized float

$0.1..0_2 * 2^{-126}$ — still smaller denormalized float

Because of denormalized numbers, IEEE arithmetic has the desirable property that `A!=B` if and only if `A - B != 0`. In most non-IEEE arithmetics, `A - B` will "flush to zero" if `A-B` is too small for normalized representation, even though `A` and `B` may be different values.

Exceptional Conditions

Exceptional conditions can arise from floating-point calculations in a number of cases. For example, multiplying two very large values can result in a value too large to be represented in one of the CPW C data formats. Or an operation such as `0.0/0.0` can be performed.

SANE provides a way for a program to determine when a floating-point calculation has resulted in one of these exceptional conditions by setting a flag when an exception occurs.

The SANE environment includes a **halt setting** for each of the five exceptions. The halt setting determines whether the occurrence of the exception halts the program. The CPW C default setting is the IEEE Standard default, which calls for all halts clear (off). You can access the halt settings by using the `testhalt` function and the `sethalt` function.

Exceptional conditions fall into five categories:

- invalid operation
- underflow
- overflow
- divide-by-zero
- inexact

Invalid Operation: The invalid operation exception arises when operands for an operation are invalid, so that a meaningful numeric result is impossible. For example, `0.0/0.0` and `sqrt(-1.0)` are invalid operations.

Underflow: Underflow occurs when a result is both denormalized and has lost significant digits through rounding. For example, to return the result of:

$(1.00000000000000000000001_2 * 2^{-126}) / 2$

to the `float` format, a leading zero would be introduced and the last significant bit would be lost in rounding. This result:

$0.10000000000000000000000_2 * 2^{-126}$

would be returned and underflow would be signaled.

Overflow: The condition of calculating a value that is too large to fit in the format of its designated type is called overflow. The destination format must be one of the floating-point types; if the destination format is an integer type, the invalid exception occurs.

Divide-by-Zero: The divide-by-zero exception occurs when a finite nonzero number is divided by zero. It also occurs when an operation on finite operands produces an exact infinite result. For example, the operation `1.0/0.0` (which results in INF) and the operation `log(0.0)` (which results in –INF) both signal divide-by-zero.

Inexact: The inexact exception occurs if the rounded result of an operation is not identical to the mathematical (exact) result. (Thus, any time overflow or underflow occurs, the inexact exception is signaled.) For example, the operation `2.0/3.0` signals inexact, regardless of the floating-point format used.

The Environment

The SANE environment consists of :

- rounding direction
- rounding precision
- exception flags
- halt settings

The C SANE library includes functions that allow you to determine the current status of the environment. These functions can be used to flag exceptional conditions and to control optional environment settings. For example, you may be working with very small values and need to know exactly when underflow occurs. Or you might want to have floating-point conversions rounded downward.

The standard rounding direction is `TONEAREST`. You can find out the current rounding direction by using the `getround` function. You can change the rounding direction by using the `setround` function.

The following routine saves the current rounding direction, computes a function using `TOWARDZERO` rounding, and finally restores the saved rounding direction.

```
rounddir r;
extended x, y;
```

```
{
    ...
    r = getround();
    setround(TOWARDZERO);
    y = f(x);
    setround(r);
```

Normally, all CPW C floating-point calculations return results that are rounded to extended precision and range. However, the **rounding precision** can be set to `float` or `double` precision and range. Results will still be returned in the `extended` format. There is no performance benefit in setting `float` or `double` rounding precision. You can access the rounding precision by using the `setprecision` function and the `getprecision` function. These functions are useful if you want to use SANE to perform calculations and then simulate the results you would get if you used a system that did not provide extended-precision arithmetic.

The entire SANE environment (rounding direction, rounding precision, exception flags, and halt settings) can be encoded in a value of type environment. The procedures described below access the current SANE environment as a whole. They are useful for managing the environment so that routines run with the environments they require and for controling the exception information passed between routines.

When your program begins, the environment will reflect the **IEEE standard environment defaults:**

- Rounding direction—TONEAREST
- Rounding Precision—EXTENDED
- All exception flags cleared
- Halt on INVALID, UNDERFLOW, and DIVBYZERO

To reinstall the IEEE standard defaults, use the statement

```
setenvironment(0);
```

The following routine runs under the IEEE default environment, while not affecting its caller's environment:

```
***TRANSLATE FROM PASCAL TO C***
PROCEDURE P;
VAR
 SaveEnv:Environment;
BEGIN
 GetEnvironment(SaveEnv);
 SetEnvironment(0);
 .
 .
 .
 SetEnvironment(SaveEnv);
END;
```

The statement

```
procentry(&e);
```

is equivalent to

```
getenvironment(&e);
setenvironment(0);
```

The `procentry` and `procexit` functions can be used in routines to selectively hide spurious exceptions from the routine's caller. For example:

```
extended arccos(x)
    extended x;
{
    environment e;

    procentry(&e);
    x = atan( sqrt( (1.0-x) / (1.0+x) ) );
    setexception(DIVBYZERO, 0);
    procexit(e);
    return x;
}
```

The statement

```
procentry(&e)
```

saves the caller's environment in e and sets IEEE defaults, so exceptions cannot halt the routine. If `x=-1`, the computation of the right side of the assignment to `acos` will signal `DIVBYZERO`, even though `acos` will be assigned the correct value, `pi()/2`. The function call

```
setexception(DIVBYZERO, false)
```

clears the `DIVBYZERO` flag, so the caller never sees it. If `x > 1` or `x <-1`, the computation of `acos` will appropriately signal `INVALID`. The `procexit` function will resignal `INVALID` after restoring the caller's environment, so if the caller's environment calls for halts on invalid, the halt will occur.

C SANE Library Constants and Types

This section explains each of the constants and types used in the C SANE library.

Exception Condition Constants

Table 5–3 defines the exception condition constants:

Table 5-3. Exception Condition Constants

Exception	Constant Value	Event Causing Exception	Example
INVALID	1	Operation not meaningful—NaN result	`sqrt(-1.0)`
UNDERFLOW	2	Accuracy lost—result too small	`(exp2(16383.0)) 3.0`
OVERFLOW	4	Result too large for number representation	`exp2(16384.0)`
DIVBYZERO	8	Division of nonzero number by zero	`1.0/0.0`
INEXACT	16	Rounded result not same as exact math result	`1.0/3.0`

The exception condition constants are used to define the value of a variable of type `exception`.

For example, if `e` is a variable of type `exception`, then

```
e = INVALID + OVERFLOW + DIVBYZERO
```

gives `e` a value that represents these three exceptions collectively.

The `setexception` and `sethalt` procedures each take arguments of type `exception`.

The `testexception` and `testhalt` functions each return a value of type `exception`.

The DECSTROUTLEN Constant

DECSTROUTLEN defines the maximum output length of a decimal string; it is defined by this declaration:

```
#define  DECSTROUTLEN        80
```

The SIGDIGLEN Constant

The SIGDIGLEN constant represents the number of significant digits in a floating-point value; it is defined by this declaration:

```
#define  SIGDIGLEN          20
```

The FLOATDECIMAL and FIXEDDECIMAL Constants

These constants represents the style of decimal representation in a number of type decform:

```
#define  FLOATDECIMAL       0     /* floating point  */
#define  FIXEDDECIMAL       1     /* fixed point     */
```

The decform Structure Type

A struct of type decform (decimal format) is defined by this declaration:

```
typedef struct decform {
      char style, unused;
      short digits;
} decform;
```

A decform structure holds the specifications for the format of a decimal number.

- The style variable specifies the decimal representation as either FLOATDECIMAL or FIXEDDECIMAL.
- The digits variable holds the number of significant digits for FLOATDECIMAL style or the number of digits to the right of the decimal point for FIXEDDECIMAL style.

The num2dec function takes a decform argument. It uses the information in decform to determine the format for the string returned in the function result.

The decimal Structure Type

A struct of type decimal is defined by this declaration:

```
typedef struct decimal {
   char sgn, unused;              /* sign 0 for +, 1 for -          *
   short exp;                     /* decimal exponent              *
   struct {unsigned char length, text[SIGDIGLEN], unused} sig;
                                  /* significant digits            *
} decimal;
```

The relop Type

The `relop` (relational operator) type is defined by this declaration:

```
typedef short relop;          /* relational operator */
```

A result of this type is returned by the `relation` function, described later.

The numclass Type

The `numclass` type is defined by this declaration:

```
typedef short numclass;       /* inquiry class */
```

Table 14-4. Number Class Descriptions

Number Class	Value	Meaning
SNAN	0	Signaling NaN
QNAN	1	Quiet NaN
INFINITE	2	Infinity or –Infinity
ZERONUM	3	0.0 or –0.0
NORMALNUM	4	Normalized number
DENORMALNUM	5	Denormalized number

Quiet NaNs are the usual kind produced by floating-point operations. Signaling NaNs, potentially useful for flagging uninitialized varaibles, are discussed in the *Apple Numerics Manual*.

The `numclass` type is used to return results from the inquiry functions, described below.

The exception Type

A variable of type `exception` holds an integer value that corresponds to the value of one of the exception constants, or to a sum of two or more of the exception constants. The `exception` type is defined by this declaration:

```
typedef short exception;      /* sum of INVALID...INEXACT */
```

The `setexception`, `testexception`, `sethalt`, and `testhalt` functions all take arguments of type `exception`.

The haltvector Pointer Type

A variable of type `haltvector` points to the address to which control is transferred when a halt occurs. The `haltvector` type is defined by this declaration:

```
typedef void (*haltvector)();
```

The `gethaltvector` and `sethaltvector` functions take arguments of type `haltvector`.

The rounddir Type

The `rounddir` (rounding direction) type is defined by this declaration:

```
typedef short rounddir;        /* rounding direction */
```

The `rounddir` type is used to determine how values are to be rounded, when rounding becomes necessary during arithmetic operations or conversions. The `setround` function takes an argument of type `rounddir` . The `getround` function returns a value of type `rounddir`.

The roundpre Type

The `roundpre` (rounding precision) type is defined by this declaration:

```
typedef short roundpre;        /* rounding precision */);
```

Rounding precision can be used to simulate arithmetic with only `float` or `double` precision. The `setprecision` function takes an argument of type `roundpre` . The `getprecision` function returns a value of type `roundpre` .

The environment Type

A variable of type `environment` holds a value that represents the settings of the SANE environment. For example, a setting of 0 represents the default IEEE setting (including no halts set). The `environment` type is defined with this declaration:

```
typedef short environment;
```

You use a variable of type `environment` with these environmental access routines: `setenvironment`, `getenvironment`, `procentry`, and `procexit`.

C SANE Library Functions

This section includes a description of each of the functions in the C SANE Library. These include

- SANE arithmetic functions
- conversions between decimal, string, and binary representation
- elementary transcendental functions
- functions that save and restore environmental settings
- functions that handle exceptional conditions
- functions that provide constants for NANs, INF, and π
- financial functions
- IEEE recommended functions
- functions that determine the class of a numeric value
- a random number function
- a relationship function
- functions that set rounding direction and precision

Trigonometric functions are provided in the Standard C Library: the `tan`, `sin`, `cos`, and `atan` functions are implemented with SANE arithmetic and conform to the IEEE Standard. The Standard C Library also provides `asin`, `acos`, and `atan2` functions. All of these functions are documented in Chapter 3.

More information on SANE functions can be found in the *Apple Numerics Manual*.

> **Note:** Any function with a formal parameter of any of the floating-point types can be passed a value of any floating-point type.

NAME

`remainder, rint`—SANE arithmetic functions

SYNOPSIS

```
#include <SANE.h>

extended remainder(x,y,quo)     /* IEEE remainder; quo <-- 7 low order  *
   extended x,y;                /*   bits of integer quotient x/y,      *
   short *quo;                  /*   -127 <= quo <= 127                 *

extended rint(x)                /* round to integral value              *
   extended x;
```

DESCRIPTION

The `remainder` function returns the remainder of the division of its two extended arguments *x/y*, as specified by the IEEE Standard. This function returns an exact remainder of the smallest possible magnitude. The result is computed as

$x-n*y$

where *n* is a nearest integral approximation to the quotient *x/y*. For example, `remainder(9,0,5.0,q)` returns `-1.0`, because $-1 = 9-2*5$.

The integer variable argument *quo* receives the seven low-order bits of n as a value between –127 and 127; this is useful for programming functions, like the trigonometric functions, that require argument reduction.

The `rint` function takes an `extended` argument and rounds it to an integral value in the `extended` format. Note that all sufficiently large floating-point values are integral. The result depends upon the rounding direction, which can be changed using the `setround` function.

SEE ALSO

`fabs, sqrt.`

NAME

num2dec, dec2num, dec2str, str2dec
—conversions between decimal, string, and binary representation

SYNOPSIS

```
#include <SANE.h>

void num2dec(f,x,d)              /* d <-- x, according to format f      *
   decform *f;
   extended x;
   decimal *d;
extended dec2num(d)              /* returns d as extended               *
   decimal *d;

void dec2str(f,d,s)              /* s <-- d, according to format f      *
   decform *f;
   decimal *d;
   char *s;

void str2dec(s,ix,d,vp) /* on input ix is starting index into s, on    *
   char *s;             /* output ix is one greater than index of last*
   short *ix,*vp;       /* character of longest numeric substring;    *
   decimal *d;          /* boolean vp = "s begining at given ix is a  *
                        /* valid numeric string or a valid prefix of  *
                        /* some numeric string"                       *
```

DESCRIPTION

The num2dec function converts a numeric value *x* to a decimal struct *d*. Here are some examples; the headings represent the effects of different decform parameters for *x* = 123.45 and *sgn* = 0:

style	*digits*	*exp*	*sig*
FLOATDECIMAL	6	–3	6, "123450"
FLOATDECIMAL	2	1	2, "12"
FIXEDDECIMAL	6	–6	9, "123450000"
FIXEDDECIMAL	2	–2	5, "12345"

The dec2num function takes a decimal argument and converts it to type extended.

The dec2str function converts a struct of type decimal to a string value using the specifications in the decform struct.

The str2dec function takes a string argument and converts it to a struct of type decimal. It scans the string in s and returns the result in *d*. On input, the index variable *ix* is the starting index into the string; on output, the value of *ix* is one greater than the index of the last character in the numeric substring just parsed. The longest possible numeric substring is parsed; if no numeric substring is recognized, *ix* remains unchanged. If the entire input string, beginning at *ix*, is a

valid numeric string or a valid prefix of a numeric string, the function sets *vp* to 1 to indicate successful completion.

NAME

`log1, log2, expl, exp2, ipower, power`—elementary transcendental functions

SYNOPSIS

```
#include <SANE.h>

extended log1(x)                /* log(1 + x)
   extended x;

extended log2(x)                /* base-2 log
   extended x;

extended expl(x)                /* exp(x) - 1
   extended x;

extended exp2(x)                /* base-2 exponential
   extended x;

extended ipower(x,i)            /* integer exponential: x ^ i
   extended x;
   short i;

extended power(x,y)             /* general exponential: x ^ y
   extended x,y;
```

DESCRIPTION

The `log1` function returns the base-*e* logarithm of 1 plus *x*. For *x* near 0, log1(*x*) is more accurate than log(1.0+*x*).

The `log2` function returns the base-2 logarithm of *x*.

The `expl` function returns e^x-1. For *x* near 0, `expl (x)` is more accurate than `exp(x) - 1.0`.

The `exp2` function returns 2 raised to the power of *x*: 2^x.

The `ipower` function returns the value of *x*, raised to the integer power of *i*: x^i.

The `power` function returns the value of *x*, raised to the floating-point power of *y*: x^y.

SEE ALSO

`atan, cos, exp, log, sin, tan.`

NAME

`getenvironment, setenvironment, procentry, procexit, gethaltvector, sethaltvector`
—save and restore SANE environmental settings

SYNOPSIS

```
#include <SANE.h>

void getenvironment(e)          /* e <-- environment                    *
   environment *e;

void setenvironment(e)          /* sets environment to e                *
   environment e;

void procentry(e)               /* e <-- environment;                   *
   environment *e;              /*   environment <-- IEEE default       *

void procexit(e)                /* temp <--exceptions; environment <-- e; *
   environment e;               /*   signals exceptions in temp         *

haltvector gethaltvector()      /* returns halt vector                  *

void sethaltvector(v)           /* halt vector <-- v                    *
   haltvector v;
```

DESCRIPTION

The `getenvironment` function assigns the current settings of the environment to variable *e*.

The `setenvironment` function sets the effective environment to the one specified in *e*.

The `procentry` function saves the current environment (the rounding direction, rounding precision, exception flags, and halt settings) in *e* and then sets the environment to the IEEE defaults.

The `procexit` function temporarily saves the current exception flags, sets the effective environment as encoded in *e*, and then signals the temporarily saved exceptions.

The `gethaltvector` function returns as its function result the address of a halt vector.

The `sethaltvector` function sets in *v* the address of a halt vector.

NAME

`setexception, testexception, testhalt, sethalt`—exceptional conditions

SYNOPSIS

```
#include <SANE.h>

#define  INVALID              1
#define  UNDERFLOW            2
#define  OVERFLOW             4
#define  DIVBYZERO            8
#define  INEXACT             16

void setexception(e,s)          /* clrs e flags if s is 0, sets e flags */
   exception e;                 /*   otherwise; may cause halt          */
   long s;

long testexception(e)           /* returns 1 if any e flag is set,      */
   exception e;                 /*   returns 0 otherwise                */

long testhalt(e)                /* returns 1 if any e halt is enabled,  */
   exception e;                 /*   returns 0 otherwise                */

void sethalt(e,s)               /* disables e halts if s is 0,          */
   exception e;                 /*   enables e halts otherwise          */
   long s;
```

DESCRIPTION

The C SANE library defines a constant for each kind of exception: invalid, underflow, overflow, divide-by-zero, and inexact.

If parameter *s* is 0, `setexception` signals the exceptions encoded in *e*; otherwise it clears the exception flags specified by *e*. For example,

```
setexception(OVERFLOW + INEXACT, 0);
```

This statement signals the overflow and inexact exceptions. If halt on overflow or inexact were set, this statement would halt the program.

The `testexception` function takes an argument of type exception and returns 1 if any of the exceptions encoded in *e* are set.

Following the `setexception` function call above, the call

```
testexception(OVERFLOW + INVALID);
```

would return a value of 1.

The `testhalt` function returns 1 if any of the flags indicated by *e* is set; otherwise it returns 0.

The `sethalt` function lets you enable or disable exceptions. Enabled exceptions cause your program to halt when they occur; disabled exceptions allow your program to continue processing when they occur. If *s* is 0, the exceptions in *e* are enabled; otherwise they're disabled.

NAME

`inf, nan, pi`—functions that return a constant value

SYNOPSIS

```
#include <SANE.h>

extended inf()                  /* returns INF

extended nan(c)                 /* returns NAN with code c
   unsigned char c;

extended pi()                   /* returns the value of pi
```

DESCRIPTION

The `inf` function returns the constant INF.

The `nan` function returns a NaN associated with the code given as an argument. The SANE NaN error codes are shown in Table 5-2 in the "Number Classes" section earlier in this chapter.

The `pi` function returns the nearest `extended` approximation to the mathematical value of π.

NAME

compound, annuity—financial functions

SYNOPSIS

```
#include <SANE.h>

extended compound(r,n)          /* compound: (1 + r) ^ n
   extended r,n;

extended annuity(r,n)           /* annuity: (1 - (1 + r) ^ (-n)) / r
   extended r,n;
```

DESCRIPTION

In the compound function, *r* specifies the interest rate per period as a decimal (.1075), not as a percent (10.75%); *n* specifies the number of periods. The function returns $(1+r)^n$, which is the principal plus accrued compound interest on an original investment of one unit.

In the annuity function, *r* specifies the interest rate; *n* specifies the number of periods. The function returns $(1-(1+r)^{-n})/r$, which is the present-value factor of an ordinary annuity.

Here is an example of how the annuity function can be used:

```
main()
{
    extended loan, payment, interest, periods;

    printf("Loan amount: ");
    scanf("%nf", &loan);
    printf("Annual interest rate  (E.g. enter 10% as 0.1):  ");
    scanf("%nf", &interest);
    printf("Number of years:  ");
    scanf("%nf", &periods);
    payment = loan / annuity(interest/12, periods*12);
    printf("Your payment is:  %8.2f \n", payment);
}
```

In this example, given a loan amount of $120,000 and an interest rate of .1075 per year for 30 years, the payment will be $1120.18.

NAME

`scalb, logb, copysign, nextfloat, nextdouble, nextextended`
—recommended IEEE functions

SYNOPSIS

```
#include <SANE.h>

extended scalb(n,x)             /* binary scale: x * 2^n;
   short n;
   extended x;

extended logb(x)                /* binary log:
   extended x;                  /*   binary exponent of normalized x

extended copysign(x,y)          /* y with sign of x
   extended x,y;

extended nextfloat(x,y)         /* next float representation after
   extended x,y;                /*   (float) x in direction of (float) y

extended nextdouble(x,y)        /* next double representation after
   extended x,y;                /*  (double) x in direction of (double) y

extended nextextended(x,y)      /* next extended representation after x
   extended x,y;                /*   in direction of y
```

DESCRIPTION

The `scalb` function scales x by the power of two specified by n. The value $2^n x$ is returned in extended format.

The `logb` function returns the largest power of two that does not exceed the magnitude of x. For example,

```
logb(-65535.0)
```

yields 15 because $2^{15} \leq 65535 < 2^{16}$.

The `copysign` function returns the value of y with the sign of x. For example, copysign(2.0, –3.0) yields 3.0.

The `nextfloat` function returns the next value that can be represented in float format after x, in the direction of y.

The `nextdouble` function returns the next value that can be represented in double format after x, in the direction y.

The `nextextended` function returns the next value that can be represented in `extended` format after x, in the direction of y.

NOTE

Additional IEEE recommended functions are described on the *inquiry* page.

NAME

`classfloat, classdouble, classextended, classcomp, signnum`
—determine the class of a numeric value

SYNOPSIS

```
#include <SANE.h>

numclass classfloat(x)          /* class of (float) x
   extended x;

numclass classdouble(x)         /* class of (double) x
   extended x;

numclass classextended(x)       /* class of x
   extended x;

numclass classcomp(x)           /* class of (comp) x
   extended x;

long signnum(x)                 /* returns 0 for +, 1 for -
   extended x;
```

DESCRIPTION

These functions are IEEE recommended functions (in addition to those on the *IEEE* page). The result of each of these functions is of type numclass.

The `classfloat` function determines the number class of its `extended` argument as if it were type `float`. For example,

```
classfloat(1.0)
classfloat(1e-310)
```

The first function call returns NORMALNUM, the code for a normalized number. The second call returns ZERONUM, the code for zero (because 1e–310 rounds to +0 in the extended format).

The `classdouble` function determines the number class of its `extended` argument as if it were type `double`. For example,

```
classdouble(0.0/0.0)
classdouble(1e-310)
```

The first example returns QNAN, the code for a quiet NaN. The second example returns DENORMALNUM, the code for a denormalized number (because 1e–310 is denormalized in the `double` format).

The `classextended` function determines the number class of its extended argument. For example,

```
classextended(1.0/0.0)
classextended(1e-310)
```

The first example returns `INFINITE`, the code for infinities. The second example returns `NORMALNUM`, the code for a normalized number.

The `classcomp` function determines the number class of its extended argument as if it were type `comp`. For example,

```
classcomp(1.0)
classcomp(0.1)
```

The first example returns `NORMALNUM`, the code for a normal number. The second example returns `ZERONUM`, the code for zero. (Remember that `comp` stores integral values.)

The `signnum` function indicates the sign of x: it returns 1 if x is negative, 0 if x is positive.

NAME

randomx—next extended random number

SYNOPSIS

```
#include <SANE.h>

extended randomx(x)              /* returns next random num; updates x; *
   extended *x;                  /*    x integral, 1 <= x <= 2^31 - 2    *
```

DESCRIPTION

The randomx function takes a variable argument of type extended which contains an integral value in the range $1 \le r \le 2^{31}-2$. It returns the next random number in sequence within the same range. Variable *x* is updated to the value returned. The randomx function uses this algorithm:

$$NewX = (7^5 * OldX) \bmod (2^{31}-1)$$

SEE ALSO

rand.

NAME

`relation`—specify relationship between two arguments

SYNOPSIS

```
#include <SANE.h>

#define     GREATERTHAN          0
#define     LESSTHAN             1
#define     EQUALTO              2
#define     UNORDERED            3

relop relation(x,y)          /* returns relation such that
   extended x,y;             /*   "x relation y" is true
```

DESCRIPTION

The `relation` function returns a value that specifies the relationship between the two arguments as greater than, less than, equal to, or unordered.

For example,

```
relation(0.1, nan(0))
```

returns `UNORDERED`, since all comparisons involving NaNs are unordered.

NAME

`getround, setround, getprecision, setprecision`—rounding direction and precision

SYNOPSIS

```
#include <SANE.h>

#define  TONEAREST            0
#define  UPWARD               1
#define  DOWNWARD             2
#define  TOWARDZERO           3

#define  EXTPRECISION         0     /*  extended  */
#define  DBLPRECISION         1     /*  double    */
#define  FLOATPRECISION       2     /*  float     */


rounddir getround()           /* returns rounding direction

void setround(r)              /* sets rounding direction to r
   rounddir r;

roundpre getprecision()        /* returns rounding precision

void setprecision(p)           /* sets rounding precision to p
   roundpre p;
```

DESCRIPTION

The **rounding direction** can be set to nearest, upward, downward, or toward zero. The default rounding direction is to nearest. The **rounding precision** may be set to extended, double, or float. The default rounding precision is extended.

The `getround` function returns the current rounding direction as a value of type `rounddir`.

The `setround` function sets the effective rounding direction to the one indicated by *p*.

The `getprecision` function returns the current rounding precision.

The `setprecision` function sets the desired rounding precision.

Appendix A

Calling Conventions

*** Needs engineering edit!! ***

Cortland Workshop C uses two different function-calling conventions: C calling conventions and Pascal-compatible calling conventions.

C Calling Conventions

This section describes the normal C calling conventions. It explains how function parameters are passed, hew function results are returned, and how registers are saved across function calls. This information is useful when writing calls between C and assembly language.

Parameters

Parameters to C functions are evaluated from right to left and are pushed onto the stack in the order they are evaluated. Characters, integers, and enumeration types are passed as sign-extended 32-bit values. Pointers and arrays are passed as 32-bit addresses. Types float, double, comp, and extended are passed as extended 80-bit values. Structures are also passed on the stack. Their size is rounded up to a multiple of 16 bits (2 bytes). If rounding occurs, the unused storage has the highest memory address. The caller removes the parameters from the stack.

Function Results

*** On Cortland, function results are returned in a global variable, not on the stack. The conventions for returning function results are still being defined. ***

Register Conventions

No registers are preserved across function calls. Tool calls have their own conventions for returning error codes in the A register.

Pascal-Compatible Calling Conventions

This section describes the conventions used for calling Pascal functions from C and for functions written in C that use Pascal-compatible calling conventions. These conventions differ from the usual C calling conventions defined in Chapter 2; they also differ from the calling conventions used by the Pascal compiler.

Parameters

Parameters to Pascal-compatible functions are evaluated left to right and are pushed onto the stack in the order they are evaluated. Characters and enumeration types whose literal values fall in the range of types char or unsigned char are pushed as bytes. (This requires a 16-bit word on the stack. The value is in the high-order 8 bits; the low-order 8 bits are unused.) Short ints and enumeration types whose literal values fall in the range of types short or unsigned short are passed as 16-bit values. Ints, long ints, and the remaining enumeration types are passed as 32-bit values. Pointers and arrays are passed as 32-bit addresses. SANE types float, double, comp, and extended are passed as extended 80-bit values; however this doesn't correspond to the Pascal compiler's calling conventions, so a compiler warning is given. Table 2-2 shows the recommended way to pass SANE-type values to Pascal. Structures are also passed by value on the stack, and also yield a compiler warning. Their size is rounded up to a multiple of 16 bits (2 bytes). If rounding occurs, the unused storage has the highest memory address. The function being called removes the parameters from the stack.

Function Results

*** On Cortland, function results are returned in a global variable, not on the stack. The conventions for returning function results are still being defined. ***

Register Conventions

No registers are preserved across function calls. Tool calls have their own conventions for returning error codes in the A register.

Appendix B

Files Supplied with Cortland Workshop C

Cortland Workshop C is intended for use with the Cortland Programmer's Workshop. The files listed below are on the Cortland Workshop C release disk, which contains the C compiler, the Standard C Library, and the Cortland Interface Library. These files may be used directly from the release disk or copied to a hard disk.

C Compiler Files

File Name	Size	Comments
C	?K	MegaMax C compiler
Instruction	2K	instructions for building sample programs
MakeFile	2K	sample program makefile
Sample.c	10K	sample C program
Sample.r	2K	sample resource maker input

Standard C Library Include Files

File Name	Size	Comments
CType.h	2K	character types
ErrNo.h	3K	C library error numbers
FCntl.h	2K	file control
IOCtl.h	3K	input/output control
Math.h	1K	math functions
StdIO.h	2K	standard input/output
Values.h	2K	numeric parameters
VarArgs.h	1K	variable argument list processing
Signal.h	1K	signal handling

Cortland Interface Library Include Files

File Name	Size	Comments
AppleTalk.h	2K	AppleTalk Manager
Controls.h	3K	Control Manager
Desk.h	1K	Desk Manager
Devices.h	1K	Device Manager
Dialogs.h	3K	Dialog Manager
Disks.h	2K	Disk Manager

Error.h	1K	System Error Handler
Events.h	2K	Event Manager
Files.h	7K	File Manager <HFS>
Fonts.h	2K	Font Manager
Graf3D.h	2K	Graf3D interface
Memory.h	1K	Memory Manager
Menus.h	3K	Menu Manager
OSEvents.h	1K	Operating System Event Manager
OSUtils.h	2K	Operating System Utilities
Packages.h	3K	Package Manager and packages
Printing.h	4K	Printing Manager
QuickDraw.h	13K	Quickdraw
Resources.h	3K	Resource Manager
Retrace.h	1K	Vertical Retrace Manager
SANE.h	2K	SANE Numerics
Scrap.h	1K	Scrap Manager
SegLoad.h	1K	Segment Loader
Serial.h	2K	Serial Drivers
Sound.h	2K	Sound Driver
Strings.h	1K	string conversions
TextEdit.h	3K	TextEdit
ToolUtils.h	3K	Toolbox Utilities
Types.h	1K	common defines and types
Windows.h	4K	Window Manager

Standard C Library Object Files

File Name	Size	Comments
CRuntime.o	7K	execution starting point for use with C libraries
Math.o	5K	C Library math functions
StdCLib.o	26K	Standard C library

Cortland Interface Library Object Files

File Name	Size	Comments
CInterface.o	21K	Cortland Interface Libraries
CSANELib.o	5K	SANE library
PrintCalls.o		Printing Manager routines

Appendix C

Comparison with Macintosh Workshop C

Data Types

The following data types are implemented differently in CPW and MPW C.

Data Type	Size in bits	
	CPW	MPW
int	16	32
unsigned int	16	32
enum	8 or 16	8, 16 or 32

Register Variables

Register variables are not supported in Cortland Workshop C due to the small number of registers available on the 65816. Use of the *register* declaration will cause the compiler to generate code at least as efficient as that generated by the same program without *register* declarations.

Structured Variables

Structures may be assigned, passed as parameters, and returned as function results in both versions of C. Cortland Workshop C allows equality comparison for structures; MPW C does not.

Pascal-Compatible Function Declarations

A function or procedure written in Pascal (or written in assembly language following Pascal calling conventions) can be called from either MPW C or Cortland Workshop C. For example, the DrawText procedure is defined in Pascal as:

```
PROCEDURE DrawText (textBuf: Ptr;
      firstByte, byteCount: INTEGER);
```

The MPW syntax for such a declaration is:

```
pascal void DrawText(textBuf, firstByte, byteCount)
      Ptr textBuf;
      short firstByte, byteCount;
      extern;
```

The CPW syntax for this declaration is:

```
extern pascal void DrawText();
```

To make the CPW form more readable, we can list the parameters in a comment:

```
extern pascal void DrawText();
      /* Ptr textBuf;
      short firstByte, byteCount;
      extern; */
```

In addition, in MPW C the word *extern* may be followed by a constant, which is interpreted as a 16-bit 68000 instruction that replaces the usual subroutine call (JSR) instruction in the calling sequence. This allows direct traps to the Macintosh ROM. For example:

```
pascal void OpenPort(port)
      GrafPtr port;
      extern 0xA86F;
```

*** Issues for further investigation ***

*** How do the C's implement byte-sized elements of structures? Are they padded to word length? ***

Appendix D

Library Index

About the Library Index

The Library Index contains an index entry for all the defines, types, enumeration literals, global variables, and functions defined in the Standard C Library and the Cortland Interface Libraries.

- Column 1 contains an alphabetical list of the index entries.
- Column 2 specifies the type of declaration (for example, "literal") for the index entry.
- Column 3 contains the library header under which documentation for the index entry can be found. If column 3 contains "(C)" following the library header—for example, "abs(C)"—look in Chapter 3, The Standard C Library. Otherwise look in Chapter 4, The Cortland Interface Library. These chapters are organized alphabetically by library header except for the first entry in each, which contains introductory material.

Cortland Workshop

C Pocket Reference

Writer: Don Reed, Apple Technical Publications Department
Alpha Draft: 26 May 1986
Part Number: PAP002-21

Contents

The Standard C Library

ASCII Table

The compiler

Compiler commands

COMPILE [+L|-L] [+S|-S]*sourcefile* [KEEP=*outfile*] [NAMES=(*seg1*[,*seg2*[,...]])]
[*language1*=(*option* ...) [*language2*=(*option* ...) ...]]

CMPL [+L|-L] [+S|-S]*sourcefile* [KEEP=*outfile*] [NAMES=(*seg1*[,*seg2*[,...]])]
[*language1*=(*option* ...) [*language2*=(*option* ...) ...]]

CMPLG [+L|-L] [+S|-S]*sourcefile* [KEEP=*outfile*] [NAMES=(*seg1*[,*seg2*[,...]])]

[*language1*=(*option* ...) [*language2*=(*option* ...) ...]]

Compiler options

Option	Description
±L|-L	+L produces source listing.
±S|-S	+S produces symbol table
sourcefile	The full pathname and filename of the source file.
KEEP=*outfile*	Filename of output file.

NAMES=(*seg1*,*seg2*,...) Partial compilation of segments *seg1*, *seg2*,...
language1=(*option* ...) Options for *language1*.

Language specification

Size and range of data types

Type	Bits	Range
`char`	8	-128 to 127
`unsigned char`	8	0 to 255

`short`	16	–32,768 to 32,767
`unsigned short`	16	0 to 65,535
`int`	16	–32,768 to 32,767
`unsigned int`	16	0 to 65,535
`long`	32	–2,147,483,648 to 2,147,483,647
`unsigned long`	32	0 to 4,294,967,295
`enum`	8, 16, 32	enumerated types
`*`	32	pointer types
`float`	32	±1.5E–45 to ±3.4E38
`double`	64	±5.0E–324 to ±1.7E308
`comp`	64	–9.2E18 to +9.2E18
`extended`	80	±1.9E–4951 to ±1.1E4932

Reserved words

int	extern	else
char	register	for
float	typedef	do
double	static	while
struct	goto	switch
union	return	case
long	sizeof	default
short	break	entry
unsigned	continue	
auto	if	

Operator precedence

Operator	Associativity
`()  []  ->  .`	left to right
`!  ~  ++  --  -  (type)  *  &  sizeof`	right to left
`*  /  %`	left to right
`+  -`	left to right
`<<  >>`	left to right
`<  <=  >  >=`	left to right
`==  !=`	left to right
`&`	left to right
`^`	left to right

!	left to right
&&	left to right
||	left to right
?:	right to left
= += -=	right to left
,	left to right

Character constants

new-line	nl (lf)	\n
horizontal tab	ht	\t
vertical tab	vt	\v
backspace	bs	\b
carriage return	cr	\r
form feed	ff	\f
backslash	\	\\
single quote	'	\'
bit pattern	\0[0-7][0-7]	\0[0-7][0-7]

Equivalent data types

Pascal Data Type	C Equivalent	Comments
boolean	Boolean	Defined in file Types.h as enum {false,true}.
var boolean	Boolean *	In C, *false* is zero and *true* is often considered nonzero.
boolean result	Boolean	In Pascal, *false* is zero and *true* is one.enumeration

(<128 or >255 literals)	enum	Use identical ordering of the enumeration literals.
enumeration (128 to 255 literals)	short	Pascal passes enumerations with 128 or more literals as words.
var enumeration (<128 or >255 literals)	enum *	
var enumeration (128 to 255 literals)	short *	
enumeration result (<128 or >255 literals)	enum	
enumeration result		

(128 to 255 literals)	short	
char	short	Pascal passes chars as 16-bit values.
var char	char *	
char result	short	
integer	short	16-bit signed values.
var integer	short *	
short result	short	
longint	int or long	32-bit signed values.
var longint	int * or long *	*** *long only???* ***
longint result	int or long	*** *long only???* ***

real	extended *	Pascal passes real parameters as extended by address.
var real	float *	
real result	float	Pascal returns real results by value.
double	extended *	Pascal passes double parameters as extended by address.
var double	double *	
double result	double	The caller supplies the address of the double result.
comp	extended *	Pascal passes comp parameters as extended by address.

var comp	comp *	
comp result	comp	The caller supplies the address of the comp result.
extended	extended *	Pascal passes extended parameters by address.
var extended	extended *	
extended result	extended	The caller supplies the address of the extended result.
pointer	pointer	32-bit addresses.
var pointer	pointer *	
pointer result	pointer	

array (1 or 2 bytes)	short	Pascal passes small arrays by value.
array (3 or 4 bytes)	int or long	*** *long only???* ***
array (5 or more bytes)	array	Pascal passes larger arrays by address.
var array	array	
array result	---	C does not allow array results.
record (1 to 4 bytes)	struct	Pascal passes small records by value.
record (5 or more bytes)	struct *	Pascal passes larger records by address.
var record (any size)	struct *	
record result (1 or 2 bytes)	short	Pascal returns small records by value.
record result (3 or 4 bytes)	int or long	*** *long only???* ***

record result (1 or 2 bytes)	struct	The caller supplies the address of the record result.
set (1 to 7 elements)	char	Pascal passes sets with 1 to 7 elements as bytes.
set (8 to 16 elements)	short	Pascal passes sets with 8 to 16 elements as words.
set (≥17 elements)	struct	Pascal also passes larger sets by value.
var set (1 to 7 elements)	char *	
var set (8 to 16 elements)	short *	
var set (≥17 elements)	struct *	
set result (1 to 7 elements)	char	Pascal returns small sets by value.

set result (8 to 16 elements)	short	
set result (≥17 elements)	struct	The caller supplies the address of the set result.

Error numbers

Number	Name	Meaning
1	[EPERM]	Not owner
2	[ENOENT]	No such file or directory

5	[EIO]	I/O error
6	[ENXIO]	No such device or address
9	[EBADF]	Bad file number
12	[ENOMEM]	Not enough space
13	[EACCES]	Permission denied
17	[EEXIST]	File exists
19	[ENODEV]	No such device
20	[ENOTDIR]	Not a directory
21	[EISDIR]	Is a directory
22	[EINVAL]]	Invalid argument
23	[ENFILE]	File table overflow
24	[EMFILE]	Too many open files

28	[ENOSPC]	No space left on device
29	[ESPIPE]	Illegal seek
30	[EROFS]	Read-only file system

The Standard C library

errno *** ??? ***

```
#include <errno.h>
extern int errno;
```

abs

```
int abs (i)
int i;
```

atof

```
extended atof (nptr)
char *nptr;
```

atoi, atol

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

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

close

```
int close (fildes)
int fildes;
```

toupper, tolower, _toupper, _tolower, toascii

```
#include <ctype.h>

int toupper (c)
int c;

int tolower (c)
int c;

int _toupper (c)
int c;

int _tolower (c)
int c;
```

```
int toascii (c)
int c;
```

creat

```
int creat (path)
char *path;
```

isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, isascii

```
#include <ctype.h>

int isalpha (c)
```

```
int c;

. . .

int isascii (c)
int c;
```

dup

```
int dup (fildes)
int fildes;
```

exit, _exit

```
void exit(status)
int status;
void _exit(status)
int status;
```

exp, log, log10, pow, sqrt

```
#include <math.h>

extended exp(x)
extended x;
```

```
extended log(x)
extended x;

extended log10(x)
extended x;

extended pow(x, y)
extended x, y;

extended sqrt(x)
extended x;
```

faccess

```
int faccess (char *fileName, usigned int cmd, ...);
```

fclose, fflush

```
#include <stdio.h>

int fclose (stream)
FILE *stream;

int fflush (stream)
FILE *stream;
```

fcntl

```
#include <fcntl.h>

int fcntl (fildes, cmd, arg)
int fildes;
unsigned int cmd;
int arg;
```

ferror, feof, clearerr, fileno

```
#include <stdio.h>

int feof (stream)
```

```
FILE *stream;

int ferror (stream)
FILE *stream;

void clearerr (stream)
FILE *stream;

int fileno (stream)
FILE *stream;
```

floor, ceil, fmod, fabs

```
#include <math.h>

extended floor(x)
extended x;

extended ceil(x)
extended x;

extended fmod(x, y)
extended x, y;
```

```
extended fabs(x)
extended x;
```

fopen, freopen, fdopen

```
#include <stdio.h>

FILE *fopen (filename, type)
char *filename, *type;

FILE *freopen (filename, type, stream)
char *filename, *type;
FILE *stream;
```

```
FILE *fdopen (fildes, type)
int fildes;
char *type;
```

fread, fwrite

```
#include <stdio.h>

int fread(ptr, size, nitems, stream)
char *ptr;
int size, nitems;
FILE *stream;
```

```
int fwrite(ptr, size, nitems, stream)
char *ptr;
int size, nitems;
FILE *stream;
```

frexp, ldexp, modf

```
extended frexp(value, eptr)
extended value;
int *eptr;

extended ldexp(value, exp)
extended value;
```

```
int exp;

extended modf(value, iptr)
extended value, *iptr;
```

fseek, rewind, ftell

```
#include <stdio.h>

int fseek (stream, offset, ptrname)
FILE *stream;
long offset;
```

```
int ptrname;

void rewind (stream)
FILE *stream;

long ftell (stream)
FILE *stream;
```

getc, getchar, fgetc, getw

```
#include <stdio.h>

int getc(stream)
```

```
FILE *stream;

int getchar()

int fgetc(stream)
FILE *stream;

int getw(stream)
FILE *stream;
```

gets, fgets

```
#include <stdio.h>
```

```
char *gets(s)
char *s;

char *fgets(s, n, stream)
char *s;
int n;
FILE *stream;
```

hypot

```
#include <math.h>

extended hypot (x, y)
```

```
extended x, y;
```

ioctl

```
#include <ioctl.h>

int ioctl (fildes, cmd, arg)
int filedes;
unsigned int cmd;
long *arg;
```

lseek

```
long lseek (fildes, offset, whence)
```

```
int fildes;
long offset;
int whence;
```

malloc, free, realloc, calloc, cfree

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

void free(ptr)
char *ptr;

char *realloc(ptr, size)
```

```
char *ptr;
unsigned size;

char *calloc(nelem, elsize)
unsigned nelem, elsize;

cfree *** ? ***
```

memccpy, memchr, memcmp, memcpy, memset

```
char *memccpy(s1, s2, c, n)
char *s1, *s2;
int c, n;
```

```
char *memchr(s, c, n)
char *s;
int c, n;

int memcmp(s1, s2, n)
char *s1, *s2;
int n;

char *memcpy(s1, s2, n)
char *s1, *s2;
int n;
```

```
char *memset(s, c, n)
char *s;
int c, n;
```

onexit

```
#include <stdio.h>

int onexit (func);
void (*func)();
```

open

```
#include <fcntl.h>
```

```
int open(path, oflag)
char *path;
int oflag;
```

printf, fprintf, sprintf

```
#include <stdio.h>

int printf(format [ , arg ] ... )
char *format;

int fprintf(stream, format [ , arg ] ... )
FILE *stream;
```

```
char *format;

int sprintf(str, format [ , arg ] ... )
char *str, format;
```

putc, putchar, fputc, putw

```
#include <stdio.h>

int putc(c, stream)
char c;
FILE *stream;
```

```
int putchar(c)
char c;

int fputc(c, stream)
char c;
FILE *stream;

int putw(w, stream)
int w;
FILE *stream;
```

puts, fputs

```
#include <stdio.h>

int puts(s)
char *s;

int fputs(s, stream)
char *s;
FILE *stream;
```

qsort

```
void qsort ((char *) base, nel, sizeof (*base), compar)
```

```
        unsigned int nel;
        int (*compar ( );
```

rand, srand

```
        int rand( )

        void srand(seed)
        unsigned seed;
```

read

```
        int read(fildes, buf, nbyte)
        int fildes;
```

```
char *buf;
unsigned nbyte;
```

scanf, fscanf, sscanf

```
#include <stdio.h>

int scanf(format [ , pointer ] ... )
char *format;

int fscanf(stream, format [ , pointer ] ... )
FILE *stream;
char *format;
```

```
int sscanf(s, format [ , pointer ] ... )
char *s, *format;
```

setbuf, setvbuf

```
#include <stdio.h>

void setbuf (stream, buf)
FILE *stream;
char *buf;

int setvbuf (stream, buf, type, size)
FILE *stream;
```

```
char *buf;
int type;
int size;
```

sigset, sighold, sigrelease, sigpause

```
#include <Signal.h>

SignalHandler * sigset (sigMap, newHandler)
SignalMap sigMap;
SignalHandler *newHandler;

void _sig_dfl (sigNo, sigState, sigEnabled)
```

```
SignalMap sigNo;
SignalMap sigState;
SignalMap sigEnabled;

SignalMap sighold (sigMap)
SignalMap sigMap;

void sigrelease (sigMap, prevEnabled)
SignalMap sigMap;
SignalMap prevEnabled;

void sigpause (sigMap)
```

```
SignalMap sigMap;
```

sinh, cosh, tanh

```
#include <math.h>

extended sinh (x)
extended x;

extended cosh (x)
extended x;

extended tanh (x)
```

```
extended x;
```

stdio

```
#include <stdio.h>

FILE *stdin, *stdout, *stderr;
```

strcat, strncat, strcmp, strncmp, strcpy, strncpy, strlen, strchr, strrchr, strpbrk, strspn, strcspn, strtok

```
char *strcat (s1, s2)
char *s1, *s2;
```

```
char *strncat (s1, s2, n)
char *s1, *s2;
int n;

int strcmp (s1, s2)
char *s1, *s2;

int strncmp (s1, s2, n)
char *s1, *s2;
int n;

char *strcpy (s1, s2)
```

```
char *s1, *s2;

char *strncpy (s1, s2, n)
char *s1, *s2;
int n;

int strlen (s)
char *s;

char *strchr (s, c)
char *s, c;
```

```
char *strrchr (s, c)
char *s, c;

char *strpbrk (s1, s2)
char *s1, *s2;

int strspn (s1, s2)
char *s1, *s2;

int strcspn (s1, s2)
char *s1, *s2;
```

```
char *strtok (s1, s2)
char *s1, *s2;
```

strtol

```
long strtol (str, ptr, base)
char *str;
char **ptr;
int base;
```

sin, cos, tan, asin, acos, atan, atan2

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

```
extended sin (x)
extended x;

extended cos (x)
extended x;

extended tan (x)
extended x;

extended asin (x)
extended x;
```

```
extended acos (x)
extended x;

extended atan (x)
extended x;

extended atan2 (y, x)
extended x, y;
```

ungetc

```
#include <stdio.h>
```

```
int ungetc (c, stream)
char c;
FILE *stream;
```

unlink

```
int unlink (path)
char *path;
```

write

```
int write (fildes, buf, nbyte)
int fildes;
char *buf;
```

```
unsigned nbyte;
```

ASCII Table

Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex
nul	0	0	0	sp	32	40	20	@	64	100	40	`	96	140	60
soh	1	1	1	!	33	41	21	A	65	101	41	a	97	141	61
stx	2	2	2	"	34	42	22	B	66	102	42	b	98	142	62
etx	3	3	3	#	35	43	23	C	67	103	43	c	99	143	63
eot	4	4	4	$	36	44	24	D	68	104	44	d	100	144	64
enq	5	5	5	%	37	45	25	E	69	105	45	e	101	145	65
ack	6	6	6	&	38	46	26	F	70	106	46	f	102	146	66
bel	7	7	7	'	39	47	27	G	71	107	47	g	103	147	67
bs	8	10	8	(	40	50	28	H	72	110	48	h	104	150	68
ht	9	11	9	)	41	51	29	I	73	111	49	i	105	151	69
lf	10	12	A	*	42	52	2A	J	74	112	4A	j	106	152	6A
vt	11	13	B	+	43	53	2B	K	75	113	4B	k	107	153	6B
ff	12	14	C	,	44	54	2C	L	76	114	4C	l	108	154	6C
cr	13	15	D	-	45	55	2D	M	77	115	4D	m	109	155	6D

so	14	16	E	.	46	56	2E	N	78	116	4E	n	110	156	6E
si	15	17	F	/	47	57	2F	O	79	117	4F	o	111	157	6F
dle	16	20	10	0	48	60	30	P	80	120	50	p	112	160	70
dc1	17	21	11	1	49	61	31	Q	81	121	51	q	113	161	71
dc2	18	22	12	2	50	62	32	R	82	122	52	r	114	162	72
dc3	19	23	13	3	51	63	33	S	83	123	53	s	115	163	73
dc4	20	24	14	4	52	64	34	T	84	124	54	t	116	164	74
nak	21	25	15	5	53	65	35	U	85	125	55	u	117	165	75
syn	22	26	16	6	54	66	36	V	86	126	56	v	118	166	76
etb	23	27	17	7	55	67	37	W	87	127	57	w	119	167	77
can	24	30	18	8	56	70	38	X	88	130	58	x	120	170	78
em	25	31	19	9	57	71	39	Y	89	131	59	y	121	171	79
sub	26	32	1A	:	58	72	3A	Z	90	132	5A	z	122	172	7A
esc	27	33	1B	;	59	73	3B	[	91	133	5B	{	123	173	7B
fs	28	34	1C	<	60	74	3C	\	92	134	5C	|	124	174	7C

Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex
gs	29	35	1D	=	61	75	3D	]	93	135	5D	}	125	175	7D
rs	30	36	1E	>	62	76	3E	^	94	136	5E	~	126	176	7E
us	31	37	1F	?	63	77	3F	_	95	137	5F	del	127	177	7F