C REFERENCE GUIDE

XEROX

610E00280
September 1986

Xerox Corporation
Information Systems Division
475 Oakmead Parkway
Sunnyvale, CA 94086

TABLE OF CONTENTS

Appendices:

List of figures and tables

Figures

Tables

(This page intentionally blank.)

1. INTRODUCTION

This chapter introduces the C programming language tools and environment for XDE. It describes the features of the environment and how to use them. It also discusses other C, XDE, and ViewPoint documentation as well as the organization of this document.

You should be familiar with ViewPoint, XDE, and the C language. The last section of this chapter provides references to documents that more fully describe ViewPoint, XDE, and C.

1.1 System overview

The system consists of an environment and tools for developing and executing C programs on 8010 and 6085 processors. The tools include the C compiler and preprocessor, the assembler, the linker, and cc (a compilation driver program). The environment , which consists of a user interface, C libraries, and C runtime support, is designed to aid in porting C programs from other environments. There are separate versions of the C environment--one for XDE and one for ViewPoint. The two versions are functionally almost the same.

The C compiler can be run in either the ViewPoint or the XDE version of the C environment. There are not separate versions for each environment.

Many of the program development tools of XDE, including the debugger, DF software (for program source code and version control), and the release tools, are compatible with C programs.

1.1.1 Environment for C programs

The C environment for ViewPoint and XDE supports both porting applications from other programming environments to ViewPoint and XDE, and developing new applications specifically for ViewPoint and XDE. C programs can therefore assume either the ViewPoint/XDE program paradigm (see section 1.1.2) or the the traditional paradigm of most C programming environments. They can either use the standard user interfaces of ViewPoint and XDE or the simple TTY user interface provided by CTool (see section 1.1.3). In general, programs that assume the traditional C programming paradigm and a TTY user interface are run in CTool, while those that assume the program paradigm of ViewPoint and

XDE are loaded and started with the standard user interfaces of ViewPoint and XDE.

1.1.2 The ViewPoint/XDE program paradigm

The general program paradigm of ViewPoint and XDE differs from that of most other programming environments. In most other environments, programs perform their tasks when they are loaded and started. When the flow of control reaches the end of the program, the program is finished; it is unloaded and the resources it holds are reclaimed. In ViewPoint and XDE, a program is not considered to be finished when the flow of control reaches the end of the program. When the program is started, it performs only minimal intialization. However, it remains loaded and performs its main tasks in response to user actions (such as mouse clicking or menu selecting). The ViewPoint paradigm does not include a notion of programs terminating.

A further discussion of the ViewPoint/XDE paradigm is in the *Mesa Course*, chapter 11 (Introduction to Tajo). This chapter is written about XDE (Tajo is the name of the XDE user interface), but the general ideas about program paradigm and call-back procedures are the same for both ViewPoint and XDE.For a brief description of ViewPoint and its user interfaces, see the *ViewPoint Programmer's Manual*, chapter 2 (Overview).

1.1.3 CTool

CTool provides a subenvironment within ViewPoint or XDE for C programs that assume the program paradigm of most C programming environments. Unlike programs that assume the ViewPoint program paradigm, a program run in CTool is considered finished when the flow of control reaches the end of the program. Resources acquired through C library functions (see below) are automatically freed when programs that run in CTool finish.

CTool also provides a user interface that is similar to most C programming environments. It contains a TTY window for entering the names of programs to be run and *argv* parameters to be passed to a main function. The TTY window is also the default source and sink of the *stdin*, *stdout*, and *stderr* streams of programs run in CTool.

For more information on CTool, see chapter 2 (CTool and CExec).

1.1.4 C library

The C library contains many standard C library functions found in other C programming environments. These functions perform I/O operations, storage allocation, string and character operations, string-to-number conversions, math functions, and program aborting, as well as other operations. Programs using the C library can avoid directly calling ViewPoint, XDE, and Pilot to perform these operations.

If a program runs in CTool and uses the C library, its resources that are acquired through library procedures are automatically freed when the program finishes. Files opened through C library functions are automatically closed, and storage allocated through library functions is automatically freed.

Programs that do not run in CTool can also use the C library, but resources acquired through library functions are not automatically freed for such programs.

For complete documentation of the C library, see chapter 5 (Library).

1.1.5 Accessing Mesa interfaces

C programs can access procedures and variables in Mesa interfaces by using a C language extension described in chapter 3 (cc), section 3.4 (C language extensions). In many cases, C programs can avoid directly calling Mesa interfaces by calling functions in the C library.

1.2 Getting started

1.2.1 Files

The first step in preparing a workstation to use the C environment is fetching the environment and tools from the release directory. The files needed are:

BWSCenvironment.bcd	Provides the CTool user interfaces to the C support tools, along with the C library and runtime support. XDE users should use **CEnvironment.bcd** instead of **BWSCEnvironment.bcd**.
Library interfaces	Compiled Mesa interfaces to the C library and runtime support. These files must be present to compile C programs. The interfaces are: **CIOLib.bcd**, **CHeap.bcd**, **CFormat.bcd**, **CAbort.bcd**, **CWindowLib.bcd**, **CTypeArray.bcd**, **Libm.bcd**, **StringOps.bcd**, **VarArgs.bcd**, **CBasics.bcd**, and **CRuntime.bcd**.
Header files	Including **stdio.h**, **strings.h**, **stdlib.h**, **ctype.h**, **math.h**, **varargs.h**, and **sgtty.h**. These files provide the declarations necessary to use the C library.

Compiler tools Including the C preprocessor (**cpp.bcd**), the C compiler (**CComp.bcd**), the assembler (**Assembler.bcd**), the Linker (**Linker.bcd**), and the compiler driver program cc (**cc.bcd**).

For XDE users, a DF file (**Cuser.df**) on the release directory simplifies fetching these files. Executing the command

```
>BringOver Cuser.df
```

in the XDE executive fetches the released version of all these files. It fetches **CEnvironment.bcd** (the XDE version of the environment) rather than **BWSCEnvironment.bcd**.

ViewPoint users can use the DF file to ascertain the location of the files but cannot use the DF software to automatically bring the files to their local desktop.

1.2.2 Starting the C environment

The process of starting the C environment is different for ViewPoint and for XDE.

1.2.2.1 Starting the C environment in ViewPoint

To load and start the C environment, copy the icon for BWSCEnvironment.bcd to the ViewPoint loader. When the environment is started, a CTool icon appears in the Basic Icons folder of the directory Copy this icon to the desktop. When it is opened, it becomes a CTool instance. You can make additional copies of the icon for multiple instances of CTool.

1.2.2.2 Starting the C environment in XDE

To load and start the C environment, enter the command

```
>Run.~ CEnvironment
```

to the XDE executive. Once the environment is started, the commands *CTool.~* and *CExec.~* (see chapter 2) are registered with the executive. To create an instance of CTool, type

```
>CTool.~
```

in the executive.

To unload the C environment, use the executive command

```
> Unload.~CEnvironment.~
```

1.2.3 The first program

Suppose that the program

```
#include <stdio.h>
main ()
{
   printf("hello, world\n");
}
```

is in the file hello.c. To compile this program, you type into the CTool

```
>>>cc hello.c
```

When the compilation is complete, you can run the program by typing into the CTool

```
>>>foo.bcd
```

To gain some familiarity with compiling and running C programs in ViewPoint or XDE, compile and run this program before you go on.

1.3 Guide to documentation

1.3.1 Organization of this document

The rest of this document describes the user interface of the various tools, libraries, and utilities that are provided. chapter 2 describes the CTool and provides the information necessary for using input and output redirection. chapter 3 describes the compiler driver program, cc. chapter 4 describes the linker and explains the steps necessary for linking large programs and library files into executable modules. chapter 5 explains the library functions provided. chapter 6 describes the C runtime support and the Mesa interfaces it exports. chapter 7 gives an overview of debugging C programs in XDE. Appendix A, "Porting C Programs to ViewPoint and XDE," gives some hints on how to ease the job of porting C programs from other environments to ViewPoint or XDE. Appendix B discusses calling Mesa procedures from C programs.

1.3.2 Other documentation

1.3.2.1 Mesa language

To access procedures and variables in Mesa interfaces, you must know some Mesa constructs, especially Mesa data types. Besides Appendix B of this manual, chapters 4-10 of the *Mesa Course* contain an abridged description of the Mesa language, including all the data types. The *Mesa Language Manual* contains a full definition of the language.

1.3.2.2 Environment interfaces

The *ViewPoint Programmer's Manual* contains documentation for the public ViewPoint interfaces. chapter 3 of this manual gives general information about which interfaces are needed for various types of applications.

Documentation for the public XDE interfaces is in the *Mesa Programmer's Manual*. Appendix A of this manual contains an example tool demonstrating the use of many of these interfaces.

The public Pilot interfaces are documented in the *Pilot Programmer's Manual*. These interfaces include access to basic kernel facilities (such as processes), environment definitions, storage management, I/O devices, communication facilities, and formatting procedures.

The *Services Programmer's Guide* documents the public interfaces to the NS file system (the file system ViewPoint uses) and all the network services.

1.3.2.3 Using XDE

Although the C compiler runs in both ViewPoint and XDE, most program development tools, including the debugger, run only in XDE. C programmers, therefore, need some familiarity with XDE.

Complete documentation for XDE is in *XDE User Guide*. chapter 1, contains a general introduction to XDE and its user interfaces.

1.4 References

Several other documents may help you understand ViewPoint, XDE and the C Language. They include:

[1] *XDE User Guide*. Version 3.0 [November 1984].

[2] *The C Programming Language*. Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, Inc., Englewood Cliffs, NJ. 1978.

[3] *Mesa Assembler Reference Manual.*

[4] *A Tour Through the Portable C Compiler*, S.C. Johnson.

[5] *Mesa Programmer's Manual*. Version 3.0 [November 1984].

[6] *Pilot Programmer's Manual*. Version 3.0 [November 1984].

[7] *ViewPoint Programmer's Manual*. Version 4.0 [September, 1985].

[8] *Mesa Course*. Version 11.1 [February 1985].

[9] *Services Programmer's Guide*. Version 8.0 [November 1984].

[10] *Mesa Language Manual*. Version 11.0 [June 1984].

(This page intentionally blank)

2. CTOOL AND CEXEC

2.1 Overview

CTool and CExec provide a user interface for interactively running C programs in ViewPoint and the XDE. This chapter describes the function and operation of CTool and CExec.

CTool is a multi-instance window tool with a TTY subwindow for entering program names to be run. The TTY subwindow of the instance in which a client program is run is the default source and sink of the standard streams of that program. CTool also has a form subwindow for specifying redirection of the standard streams and for setting switches, and a message subwindow for posting error messages.

CExec exists only in the XDE version of the C environment. It provides the same functionality as CTool but uses the executive window. Because CExec does not contain a form subwindow, there is also command line syntax for setting switches and redirecting standard streams. CTool and CExec both recognize the same command line syntax

These tools also accept script files for running client programs in batch mode.

CTool and CExec are clients of StartState and CRuntime (see chapter. 6). They will restart a program whenever it is safe. They assume that a program is completed when it returns from its mainline flow of control. Upon completion, they free a program's resources (delete the heap used in library storage allocation functions and close open streams) and assume that the global frames can be reused. Programs that register procedures with the environment, such as procedures associated with menu items, should not be run in CTool or CExec.

2.2 Starting CTool and CExec

There are different procedures for creating instances of CTool and CExec in the two versions of the C environment.

2.2.1 Starting CTool in ViewPoint

The file **BWSCEnvironment.bcd** contains a configuration that includes CTool as well as all runtime support and library functions.

After you load and start CTool, its icon appears in the Basic Icons folder of the directory. When you copy this icon to the desktop and open it, it becomes a CTool instance. You can create multiple instances of CTool by making copies of the icon.

The **Destroy** command in CTool closes the icon. To delete a CTool instance, delete the icon.

2.2.2 Starting CTool and CExec in XDE

The file **CEnvironment.bcd** contains a configuration that includes CTool, CExec, and all runtime support and library functions. When it is started, it registers the commands *CTool.*~ and *CExec.*~. with the executive. Invoking the *CTool.*~ command creates an instance of CTool. Clicking over **Another!**, or reinvoking the executive command creates additional instances of CTool. Clicking over **Destroy!** destroys the CTool instance.

The command *CExec.*~ causes the executive window to function like the TTY subwindow of CTool. You can create multiple instances of CExec with multiple instances of the executive.

2.3 User interface

CTool has a message subwindow, a form subwindow (Figure 2.1), and a TTY subwindow. In support of script files and CExec, you can set all the data items in the form subwindow by entering commands in the TTY subwindow.

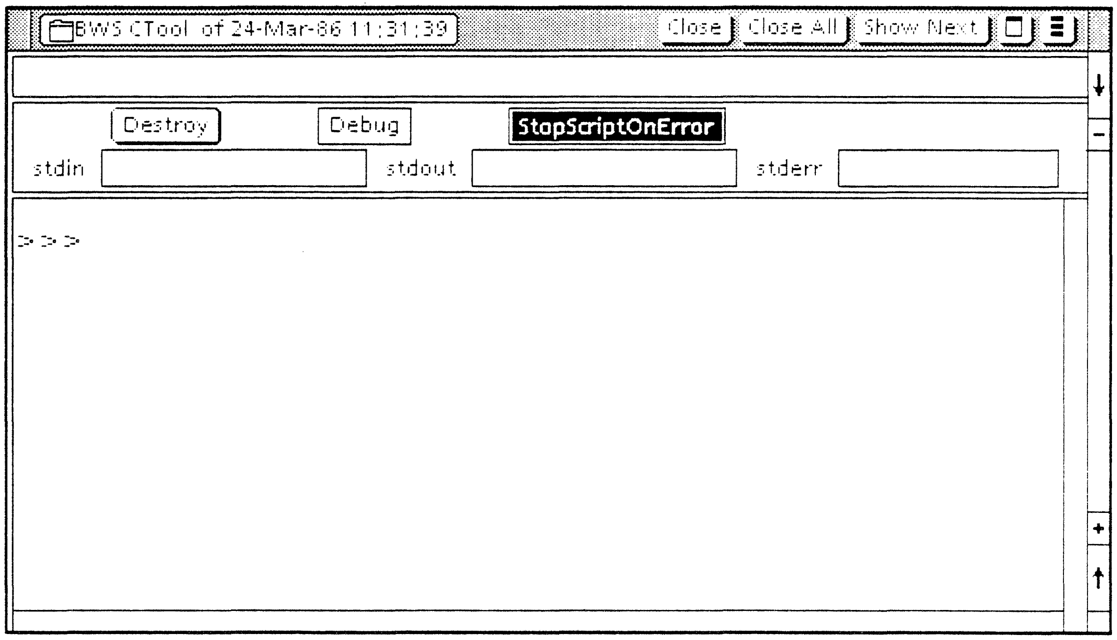


Figure 2-1: CTool Form Subwindow

2.3.1 Script files

Script files enable several client programs to be executed in a batch mode. When a script file is executed, each line is interpreted as if it were typed into the TTY subwindow. The line is echoed in the TTY subwindow, and the action taken is the same as if the line were typed in.

Input for *stdin* cannot be included in a script file. After a command to execute a cliênt program, the next line of the script file is not read until the program completes.

2.3.2 Form subwindow

The form subwindow of CTool has three text items--***stdin:***, ***stdout:***, and ***stderr:***--for redirecting the standard streams to files. If these fields are empty, the TTY subwindow is the source and sink of the standard streams. Otherwise, the contents of these fields are interpreted as the names of files to be used for the standard streams.

There are two boolean items: **Debug** and **StopScriptOnError**. If **Debug** is set, a world swap occurs just before a client program is started or restarted. If **StopScriptOnError** is set, execution of a script file ceases after the first program that cannot be successfully run. For this purpose, execution is unsuccessful if:

-- The program file or a standard stream file cannot be acquired.
-- The program cannot be loaded successfully.
-- The program calls **abort ( )**.
-- The program calls **exit ( )** with an argument other than zero.

Because CExec has no form subwindow, all of these data items must be set with commands in the TTY subwindow, as described in section 2.3.3.

The form subwindow also has command items for creating, destroying, and closing CTool instances. These are discussed in section 2.2.

2.3.3 TTY subwindow

The TTY subwindow is used for:

-- Entering the names of client programs to be run and the strings for the *argv* parameters.
-- Setting the data items of the form subwindow of CTool. In CTool this is an alternate way of setting them; in CExec it is the only way.
-- Providing a source and sink of the standard streams when they are not redirected

Setting data items with the TTY subwindow is equivalent to setting them in the form subwindow. The settings take effect the next time a client program is run and remain in effect until they are set again.

The interpretation of a line of input to the TTY subwindow depends on the first non-blank character of the line, as described below.

2.3.3.1 Running client programs

If the first non-blank character of a line is neither '!, '@, '-, '^, or '/, and the first word is not *Unload.~* or *Show.~*, then the line is interpreted as the name of a program to be run, followed by the strings for *argv*. The *argv* strings are separated from each other and from the program name by one or more spaces. The program name need only be the unique prefix of a filename. The filename extensions **.archivebcd** and **.bcd** are assumed if the name provided does not specify a unique filename by itself.

Pressing the **STOP** key causes a running program to abort execution. No exit status is returned, and a new prompt is printed. The program does not always abort at the moment the user presses the **STOP** key (see section 6.2.2.8).

2.3.3.2 Unloading programs

If the first word of a line is *Unload.~*, then the line is interpreted as a command to unload a program. Following the *Unload.~* is a list of either file names or load handles, **MLoader.Handles.** A load handle corresponds to a single loaded instance of a program. It is printed in CTool's message subwindow whenever a program is loaded or restarted (with CExec it is printed in the Herald window). When it is given as a parameter to an unload command, the corresponding program instance is unloaded. If a file name is given, all loaded instances of that program are unloaded.

The load handles for each load instance of a program can be displayed with the *Show.~* command. If the first word of a line is *Show.~* then the line is interpreted as a command to show the load handles for a list of programs. Following the *Show.~* is a list of the file names of the programs for which load handles are to be displayed.

Programs currently running in another instance of CTool or CExec will not be unloaded.

The descriptions in section 6.3.1.1 of the unloading procedures in the StartState and BWSStartState interfaces give a more detailed description of what occurs when unloading programs.

2.3.3.3 Changing the standard streams

If the first non-blank character of a line is '!, the line is interpreted as a command to change the standard stream string items. The names of the files follow the '! They can be separated by either commas or spaces, but commas are necessary to specify a name after an empty string.

Examples:

!foo, bar, baz	Changes *stdin* to "foo", *stdout* to "bar", and *stderr* to "baz".
! foo	Changes *stdin* to "foo", and *stdout* and *stderr* to empty.
!,bar	Changes *stdin* and *stderr* to empty, and *stdout* to "bar".
! foo, ,baz	Changes *stdin* to "foo", *stdout* to empty, and *stderr* to "baz".
!	Changes all three to empty.

The files are not acquired at the time these fields are set; they are acquired the next time a client program is run.

2.3.3.4 Changing the switches

If the first non-blank character of a line is '/, the line is interpreted as a command to change the **Debug** and **StopScriptOnError** switches. The switches follow the '/ . A **'d** sets the **Debug** switch and an **'s** sets the **stopScriptOnError** switch. To negate a switch, precede the letter with either '~ or '-. Switches other than **'d** or **'s** are ignored. The individual switches are not separated.

Examples:

/d-s	Sets **Debug** to TRUE and **StopScriptOnError** to FALSE
/ -d	Sets **Debug** to FALSE and leaves **StopScriptOnError** as it was.
/	Has no effect.

2.3.3.5 Displaying the current status

If the first non-blank character of a line is '?, the line is interpreted as a command to display the current values of the standard stream strings and the switches in the TTY subwindow. Text on the line after the '? is ignored. This feature is particularly useful for CExec because the values cannot be examined in a form subwindow.

2.3.3.6 Executing script files

If the first non-blank character on a line is a '@, the line is interpreted as a command to execute a script file The name of the script file follows the '@. The default extension for script files is **.cscript**.

2.3.3.7 Comment lines

If the first non-blank character of a line is a '- the line is ignored. Such lines are useful only in script files.

2.3.3.8 Exiting from CExec

If the first non-blank character of a line entered to CExec is a '^, the line is interpreted as a command to quit and return the executive to its normal state. In CTool such lines are ignored.

2.4 Sample script file

It is advisable to begin script files with commands to initialize the standard stream fields and switches, so that the result of the file's execution is not affected by commands that were previously entered.

```
-- Sample.cscript
-- Created 13-Sep-85 9:17:02

-- First initialize the switches and standard stream
-- fields
/-ds
!
!in1 out1
Test p1 p2 p3

! in2, out2
Test p4  p5  p6 p7

!errorIn,,errLog
Test p9  p10
```

2.5 Terminal emulation

2.5.1 Terminal emulation in the ViewPoint CTool

In the ViewPoint version of CTool, the TTY subwindow emulates a VT100 terminal. No special action is needed to get terminal emulation

2.5.2 Terminal emulation in the XDE CTool

If an **e** switch is appended to the *CTool.~* command, a CTool instance is created that has a VT100 terminal emulation subwindow in place of the ordinary TTY subwindow. Clicking **Another!** in the form subwindow of such an instance causes another instance with a terminal emulation subwindow to be created.

To create an instance with terminal emulation, the file **Emu.bcd** (or any configuration that includes **Emu.bcd** and exports **Emulators.bcd**) must be loaded. If it is not loaded, an error message is printed and no tool instance is created.

The terminal emulation version of the XDE CTool has an additional boolean item, **WriteLog**, in the form subwindow. Output to the terminal emulation subwindow is sent to a log file only when this switch is set. This switch allows users to avoid having a log file that partially consists of characters used for cursor control.

2.6 User Profile and User.cm

The ViewPoint CTool is affected by a [CTool] section of the User Profile, and in XDE both CTool and CExec are affected by a [CTool] section in the User.cm file. The prompt used in the TTY subwindow can be set as follows:

```
[CTool]
Prompt: <your prompt>
```

The default prompt is ">>>".

2.7 BWS searchpath

To facilitate program development in BWS, a searchpath mechanism similar to that of XDE has been provided in the BWS. In BWS, all file I/O operations in the C libraries, along with the C Tool command interpreter, use the searchpath when trying to open files. The BWS Searchpath Tool is provided to manipulate the searchpath.

2.7.1 Files

The BWS searchpath mechanism is part of the BWS CEnvironment. The searchpath tool is created when **BWSCEnvironment.bcd** is loaded. The searchpath mechanism and tool are also created when just running **CSupport.bcd** and **MonNS.bcd** (see section 6.1.1).

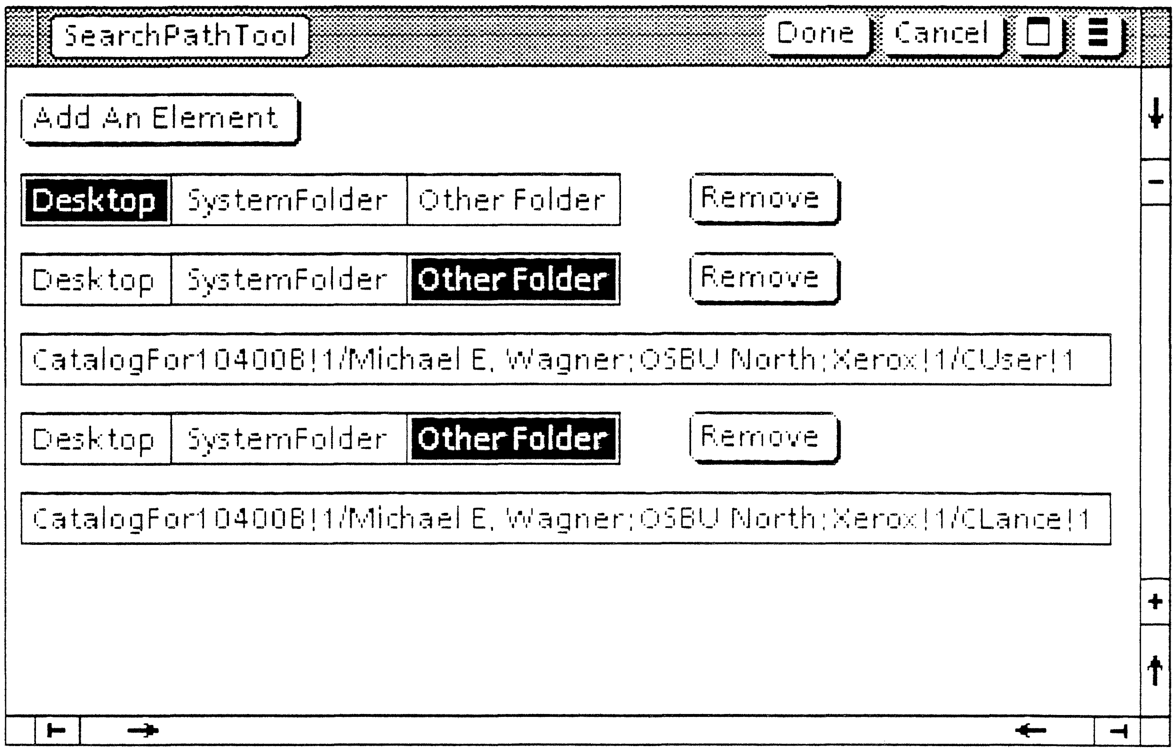
2.7.2 Searchpath concept

The BWS searchpath is similar to the XDE searchpath implemented by MFile. The searchpath is a sequence of directories. These directories are searched for files in the order in which they appear on the searchpath. The search occurs whenever a client attempts to open a file. When a file is created, if it is not found, a new file is created in the first directory on the searchpath.

The BWS searchpath is similar to the PATH facility in Unix. However, in Unix the PATH is searched only for commands. When a file is opened, only the current directory is searched. In BWS the searchpath is used whenever a file is opened. Also note that the BWS searchpath is used only for the C Tool command interpreter and the C I/O libraries. It has no effect on other ViewPoint applications.

2.7.3 User interface

The searchpath in BWS is manipulated via the BWS Searchpath Tool (Figure 2-2).



CatalogFor10400B!1/Michael E. Wagner:OSBU North:Xerox!1/CUser!1

CatalogFor10400B!1/Michael E. Wagner:OSBU North:Xerox!1/CLance!1

Figure 2-2: **BWS Searchpath Tool**

Three types of objects can be put on the searchpath. Two of them, the Desktop and the System Folder, are single-instance objects. Although they may be on the searchpath many times, only their first occurrence is useful. Subsequent occurrences only slow performance. The third object, the Other Folder, may refer to any folder in the local file system. When the BWS CEnvironment is loaded, the searchpath is initialized only to the desktop.

An instance of the BWS Searchpath Tool is created by selecting **SearchPathTool** from the attention menu. A form window displaying the searchpath's current setting is created. The searchpath reads top down, so in Figure 2-2, the desktop is the first item on the searchpath (the first directory searched for a file), and the last directory searched is a folder called CLance on the desktop. You can remove any element of the searchpath by clicking on the Remove button next to a particular element. You add elements to the end of the searchpath by clicking the Add An Element button at the top of the window. To change the value of a particular element, click on the type of element desired--Desktop, SystemFolder, or Other Folder. When you choose the Other Folder option, you must first select the folder you want to be entered onto the searchpath.

(This page intentionally blank.)

3. CC

The command *cc* invokes the C language compiler by issuing commands that perform the desired sequence of compilations. Command line arguments to *cc* consist of a list of files and a set of switches. Options allow the user to specify the level of optimization desired, rename output files, stop compilation at v arious points, or to pass other flags to the C preprocessor, compiler, assembler, or linker.

3.1 Files

Retrieve **cc.bcd** from the Release directory. There is one version of **cc.bcd**, **cpp.bcd**, **CComp.bcd**, **Assembler.bcd**, and **Linker.bcd** for both ViewPoint and XDE.

3.2 User interface

The command *cc* runs in the CTool or under CExec and takes arguments from the command line. The simplest form of cc is a list of file names, as in the following:

```
>>> cc  sourcefile1.c  sourcefile2.c  ....  sourcefilen.c
```

During execution, *cc* gives feedback about its operations in the form of the name of the file under process together with messages provided by the various steps. Any errors reported during the process are output to *stderr* (normally mapped to the CTool window).

The command *cc* produces an executable program out of a list of source (**.c**), assembler(**.as**) or object (**.bcd**) files. Each source file is preprocessed and compiled, producing a assembly file. Each assembly file (either specified on the command line or produced by a previous preprocess/compile step) is assembled, producing an object file. The object files (either specified on the command line or produced by previous assembler steps) are linked into the final executable file **foo.bcd**.

You can modify the behavior of *cc* by using various switches.

3.2.1 Options

CC accepts several switches that modify the command input. A command has the general form

>>> cc [switches] filelist

where [ ] indicates an optional part and the *switches* and *filelist* are as described.

Each switch specification is a a letter prefixed by a hyphen. You can specify multiple switches, but each switch must consist of a hyphen followed immediately by one of the letters on the following list.

c	Suppress the linking phase of the compilation and force a **.bcd** file to be produced even if only one program is compiled.
d	Have the compiler generate runtime stack error checks.
j	Have the assembler turn off cross-jumping optimizations.
h	Have the assembler turn off peephole optimizations.
p	Have the compiler pause after the first error is encountered.
w	Suppress warning diagnostics.
k	Retain intermediate files. Compiling with **cc -k testfile.c** produces the files **testfile.cpp** (preprocessor output), **testfile.as** (compiler output), **testfile.bcd** (assembler output), and **foo.bcd** (linker output).
s	Compile the named C programs and leave the assembly language output on corresponding files suffixed **.as**. (**-s** implies **-c**)
e	Execute only the preprocessor step on the named C source files, and send the output to *stdout*.
a	All arrays are guaranteed by the user to be less than 32K words long. This allows the compiler to generate more efficient code for array accesses. It should be used if possible.
g	Produce minimal debugging information. This shortens compile times and decreases object code size, but forces the user to use octal debugging.
r	Allocate string literals in read-only virtual memory. Any attempt to modify these literals will result in a write-protect fault. The default is to allocate string literals in read/write memory so that they may be modified.
o *output*	Name the final output file ***output*.bcd**. If this option is used, the file **foo.bcd** will be left undisturbed. A blank space may appear between the switch o and the argument ***output***.
m *mname*	Normally, the Linker assumes that the first routine to be called in a configuration is **main()**. Using the **-m** option instructs the Linker to generate code to call the routine **mname()** first,

instead of **main()**. A blank space must appear between the switch **m** and the argument ***mname***.

l *library* The given file is a library (a Mesa interface file). It is passed to the linker, and any unbound references to items in that library are correctly bound by the linker.

C Prevent the C preprocessor from eliding comments.

Dname = def Define the **name** to the preprocessor, as if by "**#define**". If no definition is given, the **name** is defined as "1". Blank spaces cannot appear after the switch **D**.

U*name* Remove any initial definition of **name**. Blank spaces cannot appear after the switch **U**.

The default for all the switches is off. For example, cc program.c compiles and links the source file into the executable image **foo.bcd**, with crossjumping and peephole optimization enabled.

Other arguments are taken to be members of the file list, typically C source programs or files produced by an earlier cc run. These files, together with the results of any compilations specified, are linked (in the order given) to produce an executable program with the name **foo.bcd**. C source files must have the extension **.c**, files with the extension **.as** are assumed to be C-compatible assembly language files, and files with the extension **.bcd** are assumed to be linkable object files.

3.2.2 Examples

```
>>> cc example.c
```

Compiles the C source in the file **example.c**. The output is a file, **foo.bcd**, which is executable in the CTool/CExec.

```
>>> cc -s example.c
```

Compiles the C source in the file **example.c** but stops after generating the assembly language file **example.as**.

```
>>> cc -w -o program ex.c lib1.bcd
```

Compiles the C source file **ex.c** and then links the assembler output (**ex.bcd**) with the library module **lib1.bcd**. The flag **-w** directs cc to suppress warning messages from the compiler. The output of the linker is placed in the file **program.bcd**.

```
>> cc -Ddebug ex.as impl.c demo.bcd mumble.c
```

Compiles the C source files **impl.c** and **mumble.c** as if the declaration "**#define debug 1**" appeared at the start of both files. The assembly language file **ex.as** is assembled and the respective **bcds** are linked with the file

demo.bcd. The output of the linker is place in the file **foo.bcd**.

3.3 Error messages

The C preprocessor, compiler, assembler, and linker all direct error messages to the *stderr* stream. (See chapter 2 for an explanation of the standard streams and stream redirection.) Error messages are reported in (approximate) source order by each of the compiler tools. A souce file character position is reported if possible, along with a description of the error condition. If an error is discovered by a compilation tool (for example, by the assembler), processing continues until that tool reaches completion. The compilation is then aborted before the next compilation tool (in this example, the linker) is invoked.

3.4 C Language extensions

A new storage class, the Mesa External class, has been added to this C implementation. Because references to external objects implemented in Mesa programs must use an interface in the reference, a different method of declaring such objects in C programs is needed. The syntax is similar to that of **extern** declarations. For instance, to use **Time.Current[]** in a C program, add the declaration:

mesa unsigned long Time_Current();

as if the procedure/constant werean external object. The object name must have an embedded underscore, which is used as the delimiter between the interface name and the object name. Similarly, to use **Heap.systemZone**, add the declaration:

mesa int *Heap_systemZone;.

Note that there is no **UNCOUNTED ZONE** type in C, so **int *** is used to get the size right. Table 3-1 summarizes the bit-size implementation specifics of the C compiler tools. Appendix A provides a simple example of the use of Mesa routines from C programs.

3.5 Current limitations

The following limits are built into the current implementation of C and are enforced by the compiler tools:

The include syntax **#include <file.h>** is equivalent to **#include "file.h"**. In both cases the search path is traversed until the first occurrence of **file.h** is located. In ViewPoint **file.h** cannot be a full pathname, and the header file must be on the searchpath. In XDE the header file need not be on the searchpath if **file.h** is a full pathname. If it is a full pathname, the quote syntax must be used because of XDE's directory syntax. For example: **#include "<Tajo>Directory>File.h"** will include **File.h** regardless of the setting of the search path.

The maximum local frame size is limited to 4K words.

The maximum length for a single assembly language file is 64 Kbytes.

Table 3-1 summarizes the bit-size implementation specifics of the C compiler tools.

	8010/6085 Workstation	VAX 11/780	PDP-11/70	IBM 370	Interdata 8/32
Character Set	Xerox Character Set	ASCII	ASCII	EBCDIC	ASCII
char	8 bits	8 bits	8 bits	8 bits	8 bits
int	16	32	16	32	32
short	16	16	16	16	16
long	32	32	32	32	32
float	32	32	32	32	32
double	64	64	64	64	64

Table 3-1: **Sizes of types**

(This page intentionally blank.)

4. LINKER

This section discusses the operation of the linker, including its switches and error messages. The linker is the counterpart of the Mesa Binder for non-Mesa languages. It takes a collection of separately compiled modules and creates a single output file. The linker resolves all external references among the modules that it can, linking implementors to importers. References that cannot be resolved are left unbound and may optionally be bound by subsequent applications of the linker, or, in the case of unbound references to Mesa programs, by the binder or the loader.

4.1 Files

Retrieve **Linker.bcd** from the Release directory. There is one version for both ViewPoint and XDE.

4.2 User interface

Although the linker runs in both ViewPoint and XDE, in ViewPoint it can only be invoked by cc. There is no direct user interface to the linker in ViewPoint.

In XDE the linker can be invoked directly. It runs in the executive and registers the command *Linker.~* with the executive.

A summary of the linker's commands, including errors and warnings, is written to the executive. No log file is produced.

4.2.1 Command line

To invoke the linker in XDE, type a command of the following form to the executive:

>Linker /global-switches [outputfile/o] file$_1$/local-switches file$_2$/local-switches...

The command is **Linker**, the **global-switches** apply to all files that will be linked, **file**$_i$ are the actual object files to be linked, and **local-switches** are switches that apply only to a single file. Local switches always supersede global switches. An optional output file may be given, followed by the **/o** switch. If no output file is given, output is written on Foo.bcd.

The input files of the linker may be the output of any compiler (including the Mesa compiler), the table compiler, or the linker itself.

4.2.2 Switches

The optional switches are a sequence of zero or more letters, preceded by a slash. Each letter is interpreted as a separate switch designator, and each may optionally be preceded by - or ~ to invert the sense of the switch.

The linker recognizes the following switches. Defaults are given in parentheses.

c	Link in the driver module for C. This module contains storage for standard stream variables and calls the **main** function. (**TRUE**)
d	File is a Mesa interface; warn about unbound references to this interface. (**FALSE**)
e	Name is the entry function for the configuration. If not given, a function named **main** is the default. (**FALSE**)
g	Pass unbound non-Mesa imports to the resulting object file for subsequent linking. Default is to generate a warning and suppress the import. (**FALSE**)
l	File is a library (may only be a local switch) .(**FALSE**)
m	Warn about unbound Mesa imports. (**FALSE**)
o	File is to be the output file. **.bcd** extension is appended if not already present. (**FALSE**)
s	Copy symbols into the file **output.symbols**, where **output** is the root of the output filename. (**FALSE**)
u	Warn about unbound non-Mesa imports. (**TRUE**)
v	Apply two-page uniform swap units to the virtual memory of the resulting object file's code. (**TRUE**)

4.3 Examples

```
>Linker Impl1 Impl2 Impl3
```

Links **Impl1.bcd**, **Impl2.bcd**, and **Impl3.bcd**, putting the output into **Foo.bcd**. Warnings will be given only for unbound non-Mesa references.

>Linker Prog/o Impl1 Impl2 CIOLib/l Heap/d Stream/d

Links **Impl1.bcd** and **Impl2.bcd**, putting the output in **Prog.bcd**. References to items in the library **CIOLib** will also be resolved. The file **CIOLib.bcd** must be on the local disk. Unbound references to the Mesa interfaces **Heap** and **Stream** will generate warnings.

>Linker/~u Out/o mymain/e MesaImpl CImpl FortImpl

Links **MesaImpl.bcd, CImpl.bcd,** and **FortImpl.bcd**, putting the output in **Out.bcd**. No warnings about unbound references will be given. The procedure **mymain** is called as the entry point to the configuration.

(This page intentionally blank.)

5. LIBRARY

5.1 I/O functions

I/O functions that involve access to files are implemented with calls to procedures in the NSFileStream, Stream, and NSFile interfaces in the ViewPoint version of the C environment, and with calls to procedures in the MStream, Stream, and MFile interfaces in the XDE version. The library procedures catch any signals raised by the procedures in these interfaces and return values that indicate an error occurred in the operation.

The type FILE, defined in **stdio.h**, is a replacement for a **Stream.Handle**. A pointer to a FILE is interpreted as a LONG POINTER TO **Stream.Handle** within the library procedures.

There are no restrictions on intermixing calls to C I/O library procedures and calls to Mesa I/O interfaces. Calls to the C I/O library do not affect the behavior of the Mesa interface procedures. However, calls to the function **ungetc** do temporarily change the state of the associated stream object (see the discussion of **ungetc** below).

Programs using these functions should include **stdio.h**.

5.1.1 File operations

fopen

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

This function attaches a stream to the file named **filename** and returns a pointer to that stream. The **type** parameter, whose acceptable values are listed below, specifies the type of file (text or binary), the type of access, and the initial stream position (beginning or end of the file). If the file cannot be opened with the specified access, or if **type** is not one of the strings listed below, then the value NULL is returned.

Values for the type parameter:

r or rb:	Read-only access. The type of the file does not matter.
r + :	Read and write access to a text file.
r + b:	Read and write access to a binary file.
w:	Write-only access to a text file.
wb:	Write-only access to a binary file.
w + :	Read and write access to a text file (same as r +)

w + b:	Read and write access to a binary file (same as **r + b**).
a:	Write-only access to a text file. The stream position is initialized to be the end of the file.
ab:	Write-only access to a binary file. The stream position is initialized to be the end of the file.
a +:	Read and write access to a text file. The stream position is initialized to be the end of the file.
a + b:	Read and write access to a binary file. The stream position is initialized to be the end of the file.

For values that begin with **r** or **w**, the stream position is initialized to the beginning of the file. For values that begin with **a**, the stream position is initialized to the end of the file, but there is no restriction on later setting it to any other position in the file.

fclose

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

This function deletes a stream and releases the associated file. The output buffer for the stream is flushed before it is deleted. The return value is zero if no errors arise during the operation, and non-zero otherwise.

freopen

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

The **freopen** function first deletes the stream specified by the **stream** parameter and then attaches a stream to the file named **filename**. The return value is the **stream** parameter (whose referent has been changed) if the file is opened successfully and **NULL** otherwise. The **type** parameter is the same as for the function **fopen**.

unlink

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

The **unlink** function deletes the file whose name is **path**. The return value is zero if it is successful, and **EOF** otherwise.

fflush

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

The **fflush** function forces all data written to the stream to be written to the file, thus flushing the output buffer. The value returned is zero if no errors arise during the operation, and non-zero otherwise.

tmpfile

```
FILE *tmpfile ( )
```

This function acquires a temporary file and attaches a stream to it, returning a pointer to that stream. The file is deleted when the stream is deleted. If the operation does not complete successfully, then the value **NULL** is returned.

rename

```
int rename (old, new)
char *old, *new;
```

The **rename** function changes the name of the file specified by **old** to **new**. The return value is zero if no errors occur, and non-zero otherwise. The file must be closed to rename it.

feof

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

The function **feof** returns a non-zero value if **stream** is positioned at the end of the stream, and zero otherwise.

5.1.2 Character I/O

fgetc

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

The **fgetc** function returns the next character, converted to an integer, from the stream specified by the parameter. If an error occurs during the read, or if the end of the file has been reached, then it returns **EOF**, a constant defined in **stdio.h**.

getc

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

The function **getc** is identical to the function **fgetc**. It is simply a macro, defined in **stdio.h**, that expands to **fgetc (stream)**.

getchar

```
int getchar ( )
```

The function **getchar** is a macro, defined in **stdio.h**, that expands to **fgetc (stdin)**.

fputc

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

The **putc** function interprets the integer parameter **c** as a character and writes it to the specified stream. If the write is successful, then it returns the parameter **c**. Otherwise, it returns the constant **EOF**.

putc

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

The function **putc** is identical to the function **fputc**. It is simply a macro, defined in **stdio.h**, that expands to **fputc (c, stream)**.

putchar

```
int putchar (c)
```

The function **putchar** is a macro, defined in **stdio.h**, that expands to **fputc (c, stdout)**.

ungetc

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

The function **ungetc** interprets the integer parameter **c** as a character and puts it back into the input stream. The position of the stream is set back by one. This function does not change the file associated with the stream and may be used even if there is only read access to the stream.

After **ungetc** is called, any input procedure called with the same stream, whether from a Mesa interface or from any library, returns the pushed-back character as the first character of input.

Any write procedure, and any procedure that changes the stream position (such as **fseek**), if successful, erases the memory of the pushed-back character. Because **ungetc** sets the stream position back by one, calling a write procedure after calling **ungetc** overwrites the pushed-back character.

The value returned by **ungetc** is the same as the parameter c if the operation is successful, and **EOF** otherwise. The operation is unsuccessful if the current stream position is the beginning of the file.

Only one character may be pushed back at any time. Attempts to push back several characters will yield undefined results.

The value **EOF** cannot be pushed back into the stream.

This function is implemented by retaining the pushed-back character and temporarily replacing the procedures of the stream object. The new procedures return the retained character as the first byte of input and then restore the original procedures to the stream object. The **clientData** field of the stream object is used to cache the pushed-back character and the original procedures. The original value of the **clientData** field is retained; it is restored when the rest of the stream object is restored. If clients need to access the original client data before the stream object is restored, it can be accessed by using one extra dereference. While the stream is in its altered state, the **clientData** field points to a record whose first field is the original **clientData** pointer.

5.1.3 String I/O

fgets

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

The **fgets** function reads up to n-1 characters into the string referenced by the parameter **s**. Input stops either after a carriage return or after n-1 characters have been read, whichever happens first. A null character is appended after the last character read.

Unlike the **gets** function (described below), if a carriage return is read, it is put in the string.

The value returned is either **NULL** if a read error occurs and no characters were read, or the parameter **s** otherwise.

gets

```
char *gets (s)
char *s;
```

The **gets** function reads into the string referenced by the parameter **s** from the *stdin* stream, stopping when a carriage return is read. The carriage return is not put in the string (unlike the **fgets** function) but is replaced by a null character.

The value returned is either **NULL** if a read error occurs and no characters were read, or the parameter **s** otherwise.

fputs

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

This function writes the string referenced by the parameter **s**, which must end with a null character, to the specified stream. The null character is not written to the stream. The value returned is the last character printed.

puts

```
int puts (s)
char *s;
```

This function writes the string referenced by the parameter **s**, which must end with a null character, to the *stdout* stream. It then writes a carriage return to *stdout*. The null character is not written. The value returned is a carriage return.

5.1.4 Block I/O

fread

```
int fread (ptr, sizeof (*ptr), count, stream)
unsigned count;
char *ptr;
FILE *stream;
```

This function reads into a block referenced by the parameter **ptr** from the specified stream. It reads **count** items, whose size in bytes is specified by the second parmaeter. The value returned is the number of items actually read.

fwrite

```
int fwrite (ptr, sizeof (*ptr), count, stream)
unsigned count;
char *ptr;
FILE *stream;
```

This function writes from the block referenced by the parameter **ptr** to the specified stream. It writes **count** items, whose size in bytes is specified by the second parameter. The value returned is the number of items actually written.

The pointer passed in to **fread** and **fwrite** is a pointer to a character and therefore should be a byte pointer. However, if an ordinary pointer is passed in, these functions convert it to a byte pointer before doing any I/O, so the results are correct anyway.

5.1.5 Random access functions

fseek

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

The **fseek** function changes the current position of the specified stream. The new position is **offset** bytes from a position specified by **ptrname**. The valid values for **ptrname** are the constants **SEEK_SET**, **SEEK_CUR**, and **SEEK_END**, defined in **stdio.h**. They specify whether the offset is from the beginning of the file, the current position, or the end of the file, respectively. The stream position is unchanged if any other values are passed for ptrname.

The value returned is zero if the operation is successful, and non-zero otherwise. It is also non-zero if an invalid value for **ptrname** is passed.

If **fseek** is called after **ungetc** is called, the original procedures and client data of the stream object are restored (see note on implementation of **ungetc** in section 5.1.2.).

rewind

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

This function sets the position of the specified stream to the beginning of the file. It is equivalent to **fseek (stream, 0, SEEK_SET)**.

ftell

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

The **ftell** function returns the current position of the stream as an offset, in bytes, from the beginning of the file. It returns -1 if an error occurs.

5.1.6 Formatted I/O

These functions, which read and write formatted input and output, can be called with a variable number of parameters. Each has a **format** parameter, which is a string specifying the number, type, and format of the parameters that follow it. If the number and type of the parameters passed do not match that of the format string passed, the results are undefined.

fprintf, printf, and sprintf

```
int fprintf (stream, format, . . .)
FILE *stream;
char *format;

int printf (format, . . .)
char *format;

char *sprintf (s, format, . . .)
char *s, *format;
```

These function produce formatted output. The **fprint** function writes to the stream specified, **printf** writes to the *stdout* stream, and **sprintf** writes to the string supplied and appends a null character. The value returned for **fprintf** and **printf** is the number of characters written. **sprintf** returns the string passed in.

The **format** string contains two types of objects: plain characters, which are simply copied to the output stream or string, and conversion specifications, which convert and print the next successive parameter.

A conversion specification begins with the % character. Following this character is one of the conversion characters described below. Between the % and the conversion character there may optionally be one or more of the following:

-- A minus sign (-) that indicates left justification (right justification is the default).

-- A digit string specifying a minimum field width. If the value requires fewer characters than the specified field width, blank padding is added. If the digit string begins with a zero, then zero padding is used instead of blanks. If more characters than the specified field width are needed, they are printed as needed. The number is not truncated.

-- A period (.) separating the field width string from a second digit string.

-- A second digit string if the value to be printed is a floatinq-point number (conversion characters **e**, f, or g) or a string (conversion character **s**). If it is a floating-point number, this string specifies the number of digits that appear after the decimal point. If it is a string, it specifies the maximum number of characters from the string that will be printed.

-- A lowercase ell (l) if the value to be printed is a fixed-point number (conversion characters **d,o,x**, and **u**). It specifies that the argument is a **long**.

The conversion characters and their meanings are:

-- **d**: The argument is a fixed-point number and is printed in decimal.

-- o: The argument is a fixed-point number and is printed in octal.

-- **x**: The argument is a fixed-point number and is printed in hexadecimal.

-- u: The argument is an unsigned fixed-point number and is printed in decimal.

-- c: The argument is a character.

-- **s**: The argument is a string.

-- e: The argument is a floating-point number and is printed as **[-]d.dddE ± dd**. There is one digit before the decimal point, and the number of digits after the decimal point is specified by the second digit string preceding the conversion character, or defaults to six.

-- f: The argument is a floating-point number and is printed as **[-]ddd.ddd**. The number of digits following the decimal point is specified by the second digit string preceding the conversion character, or defaults to six. If zero digits are specified to follow the decimal point, the decimal point is not printed.

-- g: The argument is a floating-point number and is printed as an e format if the exponent is less than -4 or greater than the precision.

If the characters that follow a % cannot be interpreted as a valid conversion specification, then they are simply output to the stream or string. A % can, therefore, be printed by putting "%%" in the **format** string.

fscanf, scanf, and sscanf

```
fscanf(stream, format, ...)
FILE *stream;
char *format;

scanf(format, ...)
char *format;

sscanf(s, format, ...)
char *s, *format;
```

Scanf reads from the standard input stream stdin. **Fscanf** reads from the named input stream. **Sscanf** reads from the character string **s**. Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects as arguments a control string format, described below, and a set of pointer arguments indicating where the converted input should be stored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. Blanks, tabs, or new lines that match optional white space in the input.

2. An ordinary character (not %) that must match the next character of the input stream.

3. Conversion specifications, consisting of the character %, an optional assignment-suppressing character *, an optional numerical maximum field width, and a conversion 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, unless assignment suppression was indicated by *. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted.

The conversion character indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. The following conversion characters are legal:

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

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

-- o: An octal integer is expected; the corresponding argument should be a integer pointer.

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

-- s: A character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating **\0**, which will be added. The input field is terminated by a space character or a new line.

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

-- f: 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. 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 an E or e followed by an optionally signed integer.

-- [: Indicates a string not to be delimited by space characters. The left bracket is followed by a set of characters and a right bracket; the characters between the brackets define a set of characters making up the string. If the first character is not circumflex (^), the input field is all characters until the first character not in the set between the brackets; if the first character after the left bracket is ^, the input field is all characters until the first character that is in the remaining set of characters between the brackets. The corresponding argument must point to a character array.

The conversion characters **d**, **o**, and **x** may be capitalized or preceded by **l** to indicate that a pointer to **long** rather than to **int** is in the argument list. Similarly, the conversion characters e or **f** may be capitalized or preceded by l to indicate a pointer to **double** rather than to **float**. The conversion characters **d**, **o**, and **x** may be preceded by **h** to indicate a pointer to **short** rather than to **int**.

The **scanf** functions return the number of successfully matched and assigned input items. This can be used to decide how many input items were found. The constant **EOF** is returned upon end of input; note that this is different from 0, which means that no conversion was done. If conversion was intended, it was frustrated by an inappropriate characer in the input.

For example, the call

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

with the input line

```
25 54.32E-1 thompson
```

assigns to i the value 25, **x** the value 5.432, and **name** will contain **thompson\0**. Or

```
int i; float x; char name[50];
scanf("%2d%f%*d%[1234567890]", &i, &x, name);
```

with input

```
56789 0123 56a72
```

assigns 56 to i, 789.0 to **x**, skips **0123**, and places the string `56\0' in **name**. The next call to **getchar** returns **a**.

5.1.7 Accessing standard streams

get_stdin, get_stdout, and get_stderr

```
FILE *get_stdin ()
FILE *get_stdout ()
FILE *get_stderr ()
```

These functions access the standard streams associated with the caller's configuration.

set_stdin, set_stdout, and set_stderr

```
int set_stdin (sH)
FILE *sH;

int set_stdout (sH)
FILE *sH;

int set_stderr (sH)
FILE *sH;
```

Programs can dynamically supply or change their standard streams with these functions. The value returned is zero.

5.2 Storage allocation functions

The library procedures for allocating and deallocating storage are implemented with calls to the Heap interface (see the *Pilot Programmer's Manual*). If the error **Heap.Error** is raised in these calls, it is caught by the library procedures, and the value NULL is returned.

A separate heap is maintained for each program loaded. A program's heap is created the first time it calls any of the procedures in this library. The allocation procedures all allocate from the caller's private heap, and the deallocation procedures assume that their parameters point to storage allocated from that heap.

The size parameters in the allocation procedures specify the size in bytes, not in words.

Programs using these functions should include **stdlib.h**.

5.2.1 Allocation/Deallocation operations

malloc

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

The **malloc** function allocates a block of storage at least **size** bytes large and returns a pointer to the beginning of the block. If **size** is zero, then this function returns the value **NIL**.

calloc

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

This function allocates space for an array of **nelem** elements whose size in bytes is **elsize**, and initializes the entire space with zeroes. The value returned is either **NULL** if either **nelem** or **elsize** is zero, or is a pointer to the storage allocated otherwise.

realloc

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

The **realloc** function changes the size of the block referenced by **ptr** to **size** bytes. The value returned is a pointer to the block, which might have been moved. The contents of the block up to the original size is unchanged (that is, the contents are copied if the block is moved). The results are undefined if **ptr** does not point to a previously allocated block.

free

```
free (ptr)
char *ptr;
```

This function deallocates the block referenced by **ptr**. The results are undefined if **ptr** does not point to a previously allocated block.

The **ptr** passed in to **realloc** or **free** is a pointer to a character, so it should be a byte pointer (See scection 6.2.3). However, these functions check if an ordinary pointer is passed in and the results are correct anyway.

5.2.2 Accessing the heap

In general, programs do not need to call these functions because the functions described in the previous section retrieve and set the heap as needed.

get_heap

```
char *get_heap ( );
```

This function returns the heap assigned to the caller's configuration. A heap is not created for a configuration until the first time it calls one of the allocation library procedures. However, calling **get_heap** causes a heap to be created for the calling configuration, if one has not already been created.

set_heap

```
set_heap (z);
char *z;
```

A program can supply the heap used by the procedures in this library by calling **set_heap**. If this function is called after a heap has already been created for a program (either by calling **malloc** or **calloc** or a previous call to **set_heap**), then the program must take responsibility for the old heap to avoid space leaks. Furthermore, storage allocated from the old heap cannot be freed or reallocated with functions in this library unless the old heap is restored.

5.3 String operations

Programs using these functions should include **strings.h**.

strcpy and strncpy

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

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

These functions copy the string **s2** to **s1** and return **s1**. **strcpy** copies as many characters as there are in **s2** (including the terminating null character). **strncpy** copies exactly **n** characters, truncating or padding with null characters if necessary. The results of these functions are undefined if **s1** is not large enough to hold the characters to be copied.

strcat and strncat

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

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

These functions append the string **s2** to the end of **s1** and return **s1**. **strcat** appends as many characters as there are in **s2** (including the terminating null character). **strncat** appends at most **n** characters. The results of these functions are undefined if **s1** is not large enough to hold the characters to be appended.

strcmp and strncmp

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

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

These functions compare the strings **s1** and **s2**. The value returned is 0 if the strings are the same (that is, the same characters), a negative value if **s1** is less than **s2** (that is, for the first position in the strings for which the characters are different, the character in **s1** is less than the character in **s2**), and a positive value if **s1** is greater than **s2**. The function **strncmp** compares only the first n characters.

strlen

```
int strlen (s)
char *s;
```

This function returns the number of characters in the string s, up to but excluding the terminating null character.

5.4 Character operations and predicates

The operations and predicates listed here are macros defined in **ctype.h**.

toupper, tolower and toascii

```
toupper (c)
tolower (c)
toascii (c)
```

The macros **toupper** and **tolower** subtract and add, respectively, the proper amount to a letter to change its case. If the argument is not a letter, they return the argument. **toascii** masks off all but the lower seven bits of a character.

predicates

Although characters are 16-bit values, the predicates listed here only consider the low-order byte to be significant.

The following macros return a non-zero value if the predicate is true and zero if it is false .

isalpha (c)	c is a letter.
isupper (c)	c is an uppercase letter.
islower (c)	c is a lowercase letter.
isdigit (c)	c is a decimal digit.
isxdigit (c)	c is a letter (upper- or lowercase) that is a hexadecimal digit.
isalnum (c)	c is an alphanumeric character.
isspace (c)	c is a space, tab, carriage return, new line, or form feed.
ispunct (c)	c is a punctuation character.
is print (c)	c is in the range 20_{16} (space) through $7E_{16}$ (tilde).
isgraph (c)	c is any printing character other than a space
iscntrl(c)	c is either a delete ($7F_{16}$) or a control character ($0 < c < 20_{16}$)

isascii (c) c is an Ascii character ($0<c<80_{16}$)

5.5 VarArgs functions

5.5.1 Operations Provided

Programs that implement functions that are called with varying numbers of parameters must use the functions described here. Such programs should include **varargs.h**.

va_start and va_end

```
int *va_start ( );

va_end(ap);
int *p;
```

A function that is to be called with varying numbers of parameters must be declared as having no parameters. The first executable statement in the function must be a call to **va_start**, which returns a pointer to a block containing all the parameters. Before returning, the function should call **va_end**. The pointer passed to **va_end** must be the same as that returned by **va_start**. The parameters cannot be accessed after **va_end** is called.

va_arg

```
int *va_arg (p, size);
int *p;
```

Accessing individual parameters can be facilitated by successive calls to **va_arg**, which is a macro that returns a pointer to the next parameter. The arguments to **va_arg** are a pointer and the size of the argument. The units of the **size** argument must be the same as that of the referent of the pointer. Thus, if the referent of the argument **p** is **int**, then **size** is in words (16-bits). For the first call to **va_arg**, the pointer passed should be the pointer returned by **va_start**. For subsequent calls, the pointer passed should be the pointer returned by the previous call.

If the **va_arg** macro is used, then the pointer returned by **va_start** must be saved with a separate pointer, because **va_arg** changes its pointer argument and the pointer passed to **va_end** must be the same as the pointer returned by **va_start**.

5.5.2 Sample function

The following example implements a function that can be called with varying numbers of parameters:

```
include "varargs.h"
include "stdio.h"

fprintf()
{
  char *p1, *p2;
  FILE *stream;
  char *format;
  int  *other_args;

  p1 = p2 = va_start();
  stream = *(FILE **)va_arg(p2, 2);
  format = *(char **)va_arg(p2, 2);
  other_args = va_arg(p2, 0);
  do_print(stream, format, other_args);
  va_end(p1);
}

do_print(iop, fmt, argptr)
FILE *iop;
char *fmt;
int  *argptr;
{...}
```

5.6 Functions provided by CTool

The functions described here are available to programs that are run in CTool or CExec.

5.6.1 ioctl

```
int ioctl(stream, request, argp);
FILE *stream;
sgttyb *argp;
```

Programs using this function should include **sgtty.h**.

This function is used to set and access the mode of the default *stdin* stream of a CTool or CExec instance. The mode affects whether the input is echoed, whether it is buffered, and how the control-C character is treated. The two valid values for the **request** parameter (defined in **sgtty.h**) are: **TIOCGETP** for getting the mode and **TIOCSETP** for setting the mode. If the value of **request** is not one of these two values, or if **stream** is not the default *stdin* stream, then the function returns -1 and has no effect. The return value is 0 if these parameters are valid.

The **argp** parameter points to an object that encodes the mode. Each feature of the mode (such as echoing) has a corresponding bit in the object that indicates whether that feature is on. If **request** is **TIOCSETP**, this object contains the mode to which the stream is to be set If **request** is **TIOCGETP**, the mode is copied to this object (the type **sgttyb** is defined in **sgtty.h**).

The following are the constants (defined in **sgtty.h**) for features that can be set or cleared. Each constant has one bit set and the others cleared.

-- **ECHO**: If set, each character typed in is echoed to the window instance.

-- **RAW**: If set, then the input is not buffered (that is, characters are available to a program as soon as they are typed, and control-C is treated as any other character.

-- **CBREAK**: If set, then input is not buffered, but if control-C is entered, the program aborts with an exit status of 1 the next time any input is read from the stream.

If both **RAW** and **CBREAK** are cleared, then the input is buffered (that is, a line of input is not available to a program until a carriage return is typed) and control-C causes an abort at the next read. The default mode is **ECHO** set, and **RAW** and **CBREAK** cleared.

Calls to the **ioctl** function affect neither command line input nor the mode of subsequent programs run in the tool instance. Each program starts with the default *stdin* stream in default mode.

The following code clears **ECHO** and set **CBREAK**:

```
include "sgtty.h"

{
   sgttyb ptr;

   ptr = CBREAK;
   ioctl (stdin, TIOCSETP, &ptr);
}
```

5.6.2 System

```
int system (string);
char *string;
```

This function is defined in **stdlib.h**.

The **system** function processes the string passed to it as if it were typed in to the window of a CTool or CExec instance. The string is not echoed, but any response to it is printed in the window instance in which the calling program was started. This window instance also becomes the default source and sink for the standard streams of programs invoked by calls to **system**.

The string passed to **system** may contain carriage returns that separate it into lines. Each line of the string is processed sequentially.

For each call to **system**, a temporary set of data items corresponding to the form subwindow items of CTool is created. Strings that would change these data items (such as "**!foo, bar**") thus do not change the data items for the program that calls **system** and have no effect on subsequent calls to **system**. The result is similar to creating a new temporary instance of CTool or CExec, except that it shares the window of the instance in which the calling program is started.

The defaults for the data items are: empty strings for the standard streams, Debug set to false, and **StopScriptOnError** set to true. To change these data items and invoke a program, it is necessary to pass a multi-line string to **system**. For example, the following call invokes **Prog**, with *stdin* set to the file **foo**:

```
system ("!foo \N Prog");
```

In contrast, in the following two calls, the first call has no net effect, and the second call invokes **Prog** with the default standard streams.

```
system ("!foo"); system ("Prog");
```

If the last line of the string is a command to run a program (the only type of last line that makes sense), then the return value of **system** is the exit status of the last program. However, if the last program cannot be acquired or loaded, or if the standard streams cannot be opened, then the return value is -1. If the last line of the string is not a command to run a program, the return value is 0.

5.7 Math functions

Programs using math library functions should include **math.h**.

sin, cos, tan, asin, acos, atan, atan2

```
double sin(x)
double x;

double cos(x)
double x;

double tan(x)
double x;

double asin(x)
double x;

double acos(x)
double x;

double atan(x)
double x;

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

sin, **cos** and **tan** return trigonometric functions of radian arguments x.

asin returns the arc sine in the range -pi/2 to pi/2.

acos returns the arc cosine in the range 0 to pi.

atan returns the arc tangent in the range -pi/2 to pi/2.

atan2 returns the arc tangent of **x/y** in the range -pi to pi.

sinh, cosh, tanh, asinh, acosh, atanh

```
double sinh(x)
double x;

double cosh(x)
double x;

double tanh(x)
double x;

double asinh(x)
double x;

double acosh(x)
double x;

double atanh(x)
double x;
```

These functions compute the hyperbolic functions for the floating-point value **x**.

exp, expm1, log, log10, log1p, lgamma, pow

```
double exp(x)
double x;

double expm1(x)
double x;

double log(x)
double x;

double log10(x)
double x;

double log1p(x)
double x;

double lgamma(x)
double x;

double pow(x,y)
double x,y;
```

exp returns the exponential function of **x**.

expm1 returns exp(x)-1 accurately even for tiny **x**.

log returns the natural logarithm of **x**.

log10 returns the logarithm of x to base 10

log1p returns log(1 + x) accurately even for tiny **x**.

lgamma returns the log of the absolute value of the gamma function applied to **x**.

pow(x,y) returns **x** raised to the power **y**.

frexp, ldexp, modf, modf

```
double frexp(value, iptr)
double value;
int * iptr;

double ldexp(value, exp)
double value;

double modf(value,iptr)
double value, * iptr;

double modf(x,y)
double x, y;
```

The function **frexp** returns a double value **x** and stores an integer as the referent of **iptr**, such that **value** = **x** * (2 raised to the ***iptr** power).

The return value of **ldexp** is **value** * (2 raised to the **exp** power).

The function **modf** separates a floating-point value into an integer and fractional part. The return value is the fractional part, and the the integer part is stored as the referent of **iptr**.

The function **fmod** returns the floating-point remainder of *x*/y.

ceil, **floor**, rint

```
double ceil(x)
double x;

double floor(x)
double x;

double rint(x)
double x;
```

The function **ceil**, **floor**, and **rint** return integer values in floating-point format. **ceil** returns the nearest integer no less than **x**. **floor** returns the nearest integer no greater than **x**. **rint** rounds **x** to the nearest integer. If **x** is exactly halfway between integers, it rounds toward the even integer.

abs, fabs

int abs(i)

```
double fabs (x)
double x;
```

The **abs** function returns the absolute value of the integer i, and the **fabs** function returns the absolute value of the floating- point value **x**.

sqrt, cbrt

```
double sqrt(x)
double x;

double cbrt (x)
double x;
```

The functions **sqrt** and **cbrt** return the square root and cube root, respectively, of the value **x**.

cabs, hypot

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

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

The functions **cabs** and **hypot** return **sqrt(x*x + y*y)** computed in such a way that underflow will not happen.

hypot(infinity,**v**) = **hypot**(**v**,infinity) = positive infinity for all **v**, including non-numbers.

j0, j1, jn

```
double j0(x)
double x;

double j1(x)
double x;

double jn(n, x)
double x;
```

These functions return Bessel's functions of the first kind for the value **x**. The order is 0 for j0, 1 for j1, and **n** for jn.

5.8 Miscellaneous functions

5.8.1 String-to-number conversions

atoi, atol, atof

```
double atof (nptr)
char nptr;

long atol (nptr)
char nptr;

int atoi (nptr)
char nptr;
```

The functions **atof**, **atol**, and **atoi** convert strings to **doubles**, **longs**, and **ints**, respectively.

5.8.2 Aborting programs

abort, exit

```
int abort ( );

int exit (status);
```

These functions terminate execution and invoke cleanup operations (see section 6.2). The value passed as a parameter to **exit** is the exit code of the program. When **abort** is called, the exit code is -1.

Although these are library functions, they are implemented in the Runtime Basics of the runtime support (see section 6.1). They can be called only by programs invoked with procedures in the **CRuntime** interface (see section 6.2.2) or the **BWSStartState** (**StartState** for the XDE version) interface (see section 6.3.1). Programs run in CTool or CExec (see chapter 2) are always invoked with procedures in the **BWSStartState** (**StartState**) interface, so **abort** and **exit** can be called by any program run in CTool or CExec.

Although **abort** and **exit** are declared as returning an integer, they never return.

6. RUNTIME SUPPORT

6.1 Overview

Figure 6-1 shows the general structure of the runtime support and its relation to the library and CTool. This chapter describes areas labeled in Figure 6-1 as Runtime Basics and Start State. It explains the functionality and exported interfaces of the runtime support. Most C programs do not directly use these interfaces. Their main clients are the C library and CTool/CExec (see chapter 2). Because most of these client programs are Mesa programs, the interfaces are described with Mesa syntax.

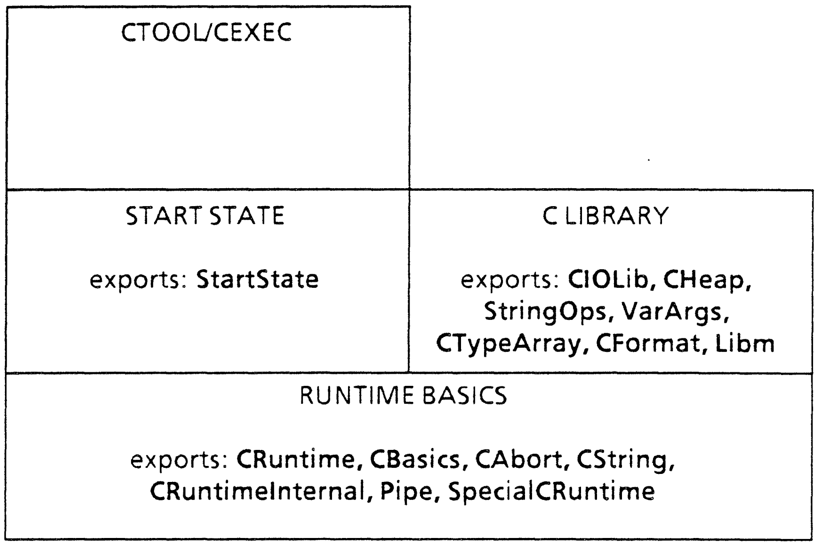

CTOOL/CEXEC
START STATE
exports: StartState
C LIBRARY
exports: CIOLib, CHeap, StringOps, VarArgs, CTypeArray, CFormat, Libm
RUNTIME BASICS
exports: CRuntime, CBasics, CAbort, CString, CRuntimeInternal, Pipe, SpecialCRuntime

Figure 6-1: **Structure of runtime support**

The lowest level, Runtime Basics, keeps track of the standard streams, open files, and storage heap of each C program. It provides facilities for accessing or altering this data, starting and restarting C programs, aborting C programs, supplying the arguments to the **Main** function (**argc** and **argv**), cleaning up resources (closing open files and deallocating storage), and performing pipes. It also provides routines for converting between C strings and Mesa strings.

The start state and the CTool/CExec are for programs that assume a traditional paradigm of performing their tasks when the program is started and having their resources automatically freed upon completion. This paradigm is in opposition with that of XDE and ViewPoint, in which programs perform only their intialization when they are started, register procedures that perform the main tasks with the environment, and have no notion of completion. The start state keeps track of programs loaded and which ones are currently running, so it can determine whether a program must be loaded or just restarted. CTool is a multiple-instance tool with a TTY window, from which users can run programs and supply the arguments to the **Main** function (**argc** and **argv**). The window instance from which a program is started or restarted is the default source and sink of the standard streams. CExec, which exists only in the XDE version of the C environment, provides the same functionality as CTool but uses the executive window. CTool and CExec are described in detail in chapter 2.

6.1.1 Files

The file **CSupport.bcd** includes Runtime Basics, the C Library, and the Start State. There is one version of **CSupport.bcd** for both XDE and ViewPoint. The file **CEnvironment.bcd** (**BWSCEnvironment.bcd** for the BWS version) includes everything in **CSupport.bcd** (**BWSCSupport.bcd**) plus CTool (and CExec for XDE).

If you don't need CTool, you can run **CSupport.bcd** instead of **CEnvironment.bcd** or **BWSCEnvironment.bcd**. ViewPoint users who are running **CSupport.bcd** instead of **BWSCEnvironment.bcd** must also run the file **MonNS.bcd**.

6.2 Runtime basics

This layer keeps track of resources used by each program that is started or restarted through the **CRuntime** or **StartState** interface. For each configuration, it records: the *stdin*, *stdout*, and *stderr* streams, a list of files opened through the library procedure **fopen** (or opened through some other means and then entered into this list), and a heap for the library procedures **malloc**, **calloc**, **realloc**, and **free**. It provides procedures for accessing and setting this data, for providing the arguments to the **Main** function, for starting and restarting programs, for aborting programs, and for performing clean-up operations to free resources.

In addition, it maintains a heap for global arrays for each module. Because ported C programs were written without the restrictions imposed by a 64-Kb main data space, they often contain large global arrays. To avoid filling up the main data space with global arrays, they are allocated from this heap rather than from global frames; only pointers to the arrays are stored in global frames. The C compiler generates the proper code for this added indirection, so the programmer can access global arrays as if they were actually in global frames.

To facilitate calling Mesa procedures from C programs and vice versa, the **CString** interface provides routines for converting between Mesa strings and C strings.

6.2.1 The CBasics Interface

The C compiler automatically generates calls to procedures in the **CBasics** interface. Although C programs do not need to explicitly call these procedures, the interface must be available to compile and link C programs. This interface also contains the type definition of **FilePtr**, which is how streams are represented in C programs. See chapter 5 for how streams are represented.

There should be no need for C or Mesa programmers to call functions in the **CBasics** interface.

6.2.2 The CRuntime Interface

The **CRuntime** interface provides procedures for setting and acquiring the resources associated with each program. Some of the procedures require a global frame handle as a parameter to identify a particular program. While a program is running, however, it is identified by its process.

6.2.2.1 Starting and restarting programs

Start: **PROCEDURE** [
gf: *PrincOps.GlobalFrameHandle*, *argc*: **CARDINAL**,*argv*: **LONG POINTER TO** *CString.CString*, *stdin*, *stdout*, *stderr*: *Stream.Handle*] **RETURNS** [*outcome*: **INTEGER**];

Restart: **PROCEDURE** [
gf: *PrincOps.GlobalFrameHandle*, *argc*: **CARDINAL**, *argv*: **LONG POINTER TO** *CString.CString*, *stdin*, *stdout*, *stderr*: *Stream.Handle*] **RETURNS** [*outcome*: **INTEGER**];

StartProgram: **PROCEDURE** [
filename: *CString.CString*, *argc*: **CARDINAL**, *argv*: **LONG POINTER TO** *CString.CString*, *stdin*, *stdout*, *stderr*: *Stream.Handle*] **RETURNS** [*outcome*: **INTEGER**];

normalOutcome: **INTEGER** = 0;

abortOutcome: **INTEGER** = -1;

Loaded programs can be started or restarted with the **Start** and **Restart** procedures, with the standard streams and arguments to **Main** supplied as parameters. The return value is **normalOutcome** unless the program started calls the library procedures **exit** or **abort** (see section 5.8.2). If **exit** is called, the value returned is the value passed to **exit**. If **abort** is called, the return value is **abortOutcome**.

The semantics of restarting a program are the same as that of starting a program for the first time.

The procedure **StartProgram** loads and starts a program specified by a filename. If the file cannot be acquired, or the program cannot be loaded successfully, the return value is

abortOutcome. Otherwise, the return value is the same as for **Start** and **Restart**.

Programs that use standard streams or need **argc** and **argv** parameters must be started with **Start**, **Restart**, **StartProgram**, or one of the procedures in the **StartState** interface.

Start, **Restart**, and **StartProgram** do not access or update the start state and do not invoke any automatic clean up operations. They are called by implementations of the start state and by programs that do not use the start state at all. Clients that start or restart programs with the assumptions of a traditional C paradigm (see section 6.1) should be invoked with the procedures in the **StartState** interface.

6.2.2.2 Removing configurations

RemoveConfig:PROCEDURE [*gf*: *PrincOps.GlobalFrameHandle*];

This procedure removes all information pertaining to a configuration from the runtime basics data structures and unmaps the global array space of every module in the configuration. This procedure should be called when programs are unloaded because programs loaded thereafter might have the same global frame handles as the program unloaded. The unloading procedures in the **StartState** interfaces call **RemoveConfig**.

6.2.2.3 Standard streams

GetStdin, GetStdout, GetStderr: PROCEDURE RETURNS [*FilePtr*];

SetStdin, SetStdout, SetStderr: PROCEDURE [*fp:FilePtr*];

These procedures store and retrieve the standard streams of the configuration associated with the caller's process.

The parameters of these procedures are pointers to stream handles rather than stream handles themselves because streams in C programs are implemented as pointers to stream handles. This representation facilitates library functions that change the referent of a stream and also allows the standard streams to be changed without changing the standard stream variables.

6.2.2.4 Heap used by library functions

GetHeap: PROCEDURE RETURNS [UNCOUNTED ZONE];

SetHeap: PROCEDURE [*h*: UNCOUNTED ZONE];

These procedures store and retrieve the heap C runtime records for the configuration associated with the caller's process configuration.

When a configuration is started or restarted, no heap is created. The procedures in the **CHeap** interface create a heap for a configuration the first time an allocation procedure is called. A call to **GetHeap** before a heap is created returns NIL.

6.2.2.5 Freeing resources

Cleanup: PROCEDURE;

EnterStream: PROCEDURE [*sH*: *Stream.Handle*];

RemoveStream: PROCEDURE[*sH*: *Stream.Handle*];

A call to **CleanUp** causes the resources for the caller's configuration to be freed. If there is a heap, it is deleted, and open files that were entered into the list of open files are closed and their associated streams are deleted. Space mapped for global arrays, however, is left mapped so that the program can be restarted more efficiently. This space is unmapped only when the procedure **RemoveConfig** is called for the configuration. All clean-up operations are logged in the file **CRuntime.log**.

The start and restart procedures of the **StartState** interface and the library functions **exit** and **abort** call **CleanUp**.

Streams on opened files are added to and removed from the list of open files by calls to **EnterStream** and **RemoveStream**, respectively. The library functions **fopen** and **fclose** (from the **CIOLib** interface) call **EnterStream** and **RemoveStream**, respectively, when they succeed. If the stream handle passed is not in the list, no action is taken. **EnterStream** and **RemoveStream** do not perform any stream operations, but simply add or remove a stream handle to or from the list of streams for the configuration.

6.2.2.6 Registering processes

RegisterProcess:PROCEDURE;

ProcessNotRegistered: SIGNAL;

While programs are executing, the runtime support maintains an association between their process and a record showing the resources they hold. For C programs, the association is made when any module in a configuration is started in the implicit call to **CBasics.RegisterFrame** (see section 6.2.1). An association can aso be made with a call to **RegisterProcess**. This procedure identifies the caller by its global frame handle and then associates its resources with its process. This procedure is used for the C library in procedures called back from the environment (such as menu procedures), which may not be running in the same process as when they were started.

If a procedure in the **CRuntime** interface is called that requires accessing the resources of the caller's configuration, and no association has been made with the caller's process, the signal **ProcessNotRegistered** is raised.

6.2.2.8 User aborts

processesAborted:*ProcessList*;

ProcessList:**TYPE** = **LONG POINTER TO** *ProcessEntry*;

ProcessEntry:**TYPE**;

StopIfUserAborted:**PROCEDURE** = **INLINE** {
 IF *processesAborted* # **NIL THEN** *StopIfCurrentProcessAborted*[];};

StopIfCurrentProcessAborted:**PROCEDURE**;

NoteAbortedProcess:**PROCEDURE**[*p*:**PROCESS**];

The runtime support maintains a list of processes for which user aborts have been noted. The variable **processesAborted** references this list. Processes can be added to this list with the procedure **NoteAbortedProcess**. Clients that provide environments for running C programs (such as CTool) should call **NoteAbortedProcess** whenever they detect a user abort.

The procedure **StopIfCurrentProcessAborted** checks if the current process is in the list of processes for which user aborts have been noted. If it finds the current process in the list, it removes it and then raises the error **ABORTED**. For efficiency, clients can check if the variable **processesAborted** is **NIL** before calling **StopIfCurrentProcesssAborted**. Mesa clients can make this check by calling the inline procedure **StopIfUserAborted** instead of **StopIfCurrentProcessAborted**. Programs running in environments that catch **ABORTED** can periodically call **StopIfUserAborted** or **StopIfCurrentProcessAborted** to stop if the user has aborted. Most of the C library functions call **StopIfUserAborted** before performing any operations, so C programs that use the C library need not explicity check for user aborts.

6.2.3 The CRuntime.log File

In the XDE version of the C environment, automatic clean-up operations are recorded in a file called **CRuntime.log**. If any clean-up operations are performed after a program finishes execution, the following information is appended to this file:

1. The name of the program for which the clean-up operations are being performed.

2. The names of all the files being closed, if there are any.

3. The size of the heap being deleted, if there is one.

This log file is rewritten each time **CSupport.bcd** or **CEnvironment.bcd** is started. It can be loaded into a window while **CSupport** or **CEnvironment** are running, but the window is not automatically updated as the file is writtten to. To see additions to the file, empty the window and reload it.

The ViewPoint version of the C environment does not create this log file.

6.2.4 The CString interface

CString:TYPE = *PrincOpsExtrasBP.BytePointer*;

CStringToString: PROCEDURE [
  cs: *CString*, *z*: MDSZone] RETURNS [STRING];

CStringToLongString: PROCEDURE [
  cs: *CString*, *z*: UNCOUNTED ZONE] RETURNS [LONG STRING];

StringToCString: PROCEDURE [
  s: STRING, *z*: UNCOUNTED ZONE] RETURNS [*CString*];

LongStringToCString: PROCEDURE [
  s: LONG STRING, *z*: UNCOUNTED ZONE] RETURNS [*CString*];

ReadChar: PROCEDURE [*CString*] RETURNS [CHARACTER] = MACHINE CODE ...;

ReadByte: PROCEDURE [*CString*] RETURNS [*Environment.Byte*] = MACHINE CODE ...;

WriteChar: PROCEDURE [CHARACTER, *CString*] = MACHINE CODE ...;

WriteByte: PROCEDURE [*Environment.Byte*, *CString*] = MACHINE CODE ...;

IncrBPointer: PROCEDURE [*p*: *CString*] RETURNS [*CString*] = INLINE ...;

DecrBPointer: PROCEDURE [*p*: *CString*] RETURNS [*CString*] = INLINE ...;

AddToBPointer: PROCEDURE [*p*: *CString*, *i*: INTEGER] RETURNS [CString] = INLINE ...;

ToBytePointer: PROCEDURE [LONG POINTER] RETURNS [*CString*] = MACHINE CODE...;

ToWordPointer: PROCEDURE [*CString*] RETURNS [LONG POINTER] = MACHINE CODE...;

A C string points to a packed array of characters that terminates with an Ascii null character. Thus, strings passed to **CStringToString** and **CStringToLongString** must be terminate with an Ascii null, and the C strings returned by **StringToCString** and **LongStringToCString** have an Ascii null appended at the end.

The characters in a C string are packed with one byte per character. Therefore, to be able to reference any character in a string, a **CString** is a byte pointer, which is a special kind of pointer that can reference either the high byte or the low byte of a word. C programs need not do anything special to dereference or add to C strings. Programs in other languages can do these operations on C strings with the procedures ReadChar, ReadByte, WriteChar, WriteByte, IncrBPointer,

DecrBPointer, and **AddToBPointer**. The procedures **ToBytePointer** and **ToWordPointer** are for conversions between byte pointers and ordinary pointers.

6.2.5 The Pipe interface

Handle: TYPE = LONG POINTER TO *Object*;

Object: TYPE;

NWords: TYPE = CARDINAL;

defaultBufferSize: *NWords* = *Environment.wordsPerPage* * 4;

Create: PROCEDURE[
  bufferSize: *NWords* ← *defaultBufferSize*] RETURNS [*h*:*Handle*,
  producer, *consumer*:*Stream.Handle*];

Delete: PROCEDURE[*h*:*Handle*];

GetProducer: PROCEDURE[*h*:*Handle*] RETURNS [*Stream.Handle*];

GetConsumer: PROCEDURE[*h*:*Handle*] RETURNS [*Stream.Handle*];

The **Pipe** interface supports tools such as a C Shell that allow users to have the output of one program become the input of another program.

The **Create** procedure creates a pipe and returns a **Handle** that can be passed to other procedures in this interface. It also returns the streams that write into and read out of the pipe. When the pipe is no longer needed, the **Handle** should be passed to **Delete**. This procedure deletes both streams and frees the pipe's resources. Clients should not delete the streams themselves.

The two streams associated with each pipe can be accessed with **GetProducer** and **GetConsumer**. However, the **Create** procedure also returns these two streams, so most applications need not call **GetProducer** or **GetConsumer**.

6.3 Start state

The start state is a record of loaded programs, identifying which are currently running, to determine whether a program can be safely restarted.

The start state is updated only when programs are loaded and started by procedures in the **StartState** interface. Clients that load and start programs with the intention of restarting them later, must use the procedures in these interfaces.

Programs started by procedures in the **StartState** interface are assumed to be finished after they return from their main line flow of control. Their resources will be freed (see section 6 2), and their global frames may be reused if the program is restarted. Therefore programs that register procedures with the environment (such as procedures associated with menu items) should not be invoked with the start state.

6.3.1 The StartState interface

Some of the parameters to procedures in the **StartState** interface are of type **MFile.Handle** or **MLoader.Handle**. Although the **MFile** and **MLoader** interfaces are not normally available to programs that run in ViewPoint, the ViewPoint C environment exports them. Therefore clients of **StartState** can acquire an **MFile.Handle** or an **MLoader.Handle** in the same way that they would be acquired in XDE.

6.3.1.1 Loading, unloading, starting, and restarting

```
StartOrRestart: PROCEDURE [
  file: MFile.Handle, argc: CARDINAL, argv: LONG POINTER TO CString.CString, stdin, stdout, stderr:
    Stream.Handle] RETURNS [outcome: INTEGER];

GetHandle: PROCEDURE [file: MFile.Handle] RETURNS [h:Handle, canRestart: BOOLEAN];

Load: PROCEDURE [h:Handle, fh: MFile.Handle] RETURNS [MLoader. Handle];

Start: PROCEDURE [
  h:Handle, argc: CARDINAL, argv: LONG POINTER TO CString.CString,
  stdin, stdout, stderr: Stream.Handle] RETURNS [outcome: INTEGER];

Resart: PROCEDURE [
  h:Handle, argc: CARDINAL, argv: LONG POINTER TO CString.CString,
  stdin, stdout, stderr: Stream.Handle] RETURNS [outcome: INTEGER];

Unload:PROCEDURE [h: MLoader.Handle];

UnloadFromFile:PROCEDURE[
  file:MFile.Handle]RETURNS[instancesUnloaded:CARDINAL];

UnloadUnstartedProgram:PROCEDURE [h:Handle];

Handle: TYPE = LONG POINTER TOObject;

Object: TYPE;

normalOutcome: INTEGER = 0;

abortOutcome: INTEGER = -1;

LoadError: ERROR [message: LONG STRING];

VersionMismatch: SIGNAL [module: LONG STRING];
```

UnloadError:**ERROR** [*message*:**LONG STRING**, *InstancesAlreadyUnloaded*:**CARDINAL**];

The simplest way to start or restart a program is to call **StartOrRestart**, supplying a file handle, the arguments to **Main**, and the standard streams. This procedure checks if there is a loaded instance of it not currently running. If there is, then it restarts that instance; otherwise, it loads a new instance and starts it. The return value is **normalOutcome** unless the started program calls **Exit** or **Abort**. If **Exit** is called, the return value is the value passed to **Exit**. If **Abort** is called, the return value is **abortOutcome**.

Clients can gain finer control over the loading and starting process by using **GetHandle**, **Load**, **Start**, and **Restart** instead of **StartOrRestart**. The procedure **GetHandle** acquires a **StartState.Handle** from a file handle and also returns a flag indicating whether that **Handle** can be passed to **Restart**. If the handle cannot be passed to **Restart**, it must be passed first to **Load** and then to **Start**. The return value of **Start** and **Restart** is the same as that of **StartOrRestart**.

Programs that expect their resources to be freed automatically (closing files and deallocating storage allocated through library procedures) must be started with **Start**, **Restart**, or **StartOrRestart**.

The procedures **StartOrRestart** and **Load** can raise the error **LoadError** or the signal **VersionMismatch**.

The **Unload** procedure unloads the program associated with the load handle passed and removes the program instance from the record of loaded programs. It also calls **CRuntime.RemoveConfig** (see section 6.2.2.2) to remove the configuration from the data structures of the runtime basics. If the program associated with the handle passed is running, then the error **UnloadError** is raised and the program is not unloaded.

All loaded instances of a program can be unloaded and removed from the start state and runtime basics data structures with the procedure **UnloadFromFile**. The return value is the number of instances that were unloaded. If it encounters a running instance of the program, it raises the error **UnloadError** with the **instancesAlreadyUnloaded** parameter, indicating how many instances it unloaded before it encountered the running instance.

Programs that have been loaded but not started should not be unloaded with **Unload** or **UnloadFromFile**, but instead should call **UnloadUnstartedProgram**. It does not call **CRuntime.RemoveConfig**, as do **Unload** and **UnloadFromFile**. This function is useful for unloading programs in which version mismatches were detected while loading.

Restarting a program has the same effect as starting a program for the first time.

6.3.1.2 Accessing load handles

GetLoadHandle: **PROCEDURE** [*h*:.*Handle*] **RETURNS** [*MLoader.Handle*];

SetLoadHandle: **PROCEDURE** [*h*:*Handle*, *lh*: *MLoader.Handle*];

EnumerateHandles:**PROCEDURE**[*file*:*MFile.Handle*,*proc*:*EnumerateProc*];

EnumerateProc: **TYPE** = **PROCEDURE** [
  ssh: *Hanaıe*,*mn*: *MLoader.Handle*] **RETURNS** [*continue*:**BOOLEAN** ← **TRUE**];

These procedures support accessing and setting the load handle associated with each **StartState.Handle**. Clients do not usually need to call these procedures because the procedures described in section 6.3.1.1. return and set the load handles as needed.

EnumerateHandles calls the procedure proc for every instance of **file** in the start state, passing in the **StartState.Handle** and load handle associated with the instance. The enumeration stops if **proc** returns **FALSE**.

(This page intentionally blank.)

7. C PROGRAM DEBUGGING

7.1 Introduction

The Xerox Development Environment provides a source-level debugger called Copilot,to aid in program development. This document describes how to use Copilot to develop and debug C programs. The document reviews the use of Copilot but does not describe the debugger commands in detail. For this information, see chapter 24 of the *XDE User Guide*.

Copilot is not yet fully multilingual in the programming language sense. In most cases, however, a C programmer can debug at a source-program level. For instance, the debugger can coordinate C source program locations with runtime program counter values. This coordination enables a user to set breakpoints by pointing at a source program statement,and enables the debugger to display the source location associated with the PC of any suspended procedure. The debugger is not multilingual in that it currently understands only Mesa expressions and data types (see the *Mesa Language Manual*). Fortunately, it is very easy to convert between the two.

The remaining sections of this document give:

- A review of basic Copilot operations and functionality
- A description of how types and values are used and displayed in the debugger
- A guide to converting C expressions to Mesa expressions
- A list of current limitations
- A guide to CPrint, a C debugging aid

The reader is assumed to be familiar with C programming and the use of the Xerox Development Environment.

7.2 Copilot review

This section provides a whirlwind review of the basic Copilot features and characteristics. It gives the reader an overview of Copilot's functionality and a model of the user-debugger interaction. For a complete description of the debugger, refer to the *XDE User Guide*, chapter 24.

7.2.1 Invoking the debugger

There are several ways to invoke the debugger. The simplest is to press the CALLDEBUG key. This action suspends all processing in the current XDE instance (called the client) and swaps the user to another instance of the XDE. This instance provides debugging support for the client instance. Once in this environment, you can execute debugger commands, evaluate expressions, browse source files, and perform any other XDE activity. After you have performed the desired debugging activities, you may return to the client instance in one of two ways:

1. You can swap back. This reactivates all of the suspended processes and no context is lost. This method is not always possible or desirable. Sometimes too much of the client world was damaged by the problem that invoked the debugger.

2. You can reboot the client instance and start again. In this case you lose the context of the previous boot session. For instance, you must reload all previously loaded tools.

The process of switching between instances of the XDE is called *world swapping.* A world swap can occur for several reasons other than the use of the CALLDEBUG key. When a debugging instance is invoked, the mouse cursor temporaily displays an indication of why the world swap occurred. The following list summarizes the possible swap reasons and the cursor icon displayed for each:

- Uncaught SIGNAL or ERROR. SIGNALs and ERRORS are Mesa language constructs that are used to specify exceptional conditions. For example, if the Pilot storage management package notices that a storage region has been trashed, it raises a SIGNAL saying so. If a SIGNAL is raised in the call stack of some process, and no procedure in that call stack catches the SIGNAL, you will end up in the debugger. Currently, a C program cannot raise a SIGNAL or ERROR, nor can it catch one. The debugger catches all SIGNALs and ERRORS. The mouse cursor contains "**Unc Sig**" for this swap reason.

- A program explicitly requested a trip to the debugger by calling a routine in the Pilot runtime package (see the *Pilot Programmer's Manual*, section 2.4.4). A program might enter the debugger this way if it decides that some important invariant no longer holds and that proceeding may cause serious problems. The cursor displayed is "**Call Debug**".

- A program executing in the client instance hits a previously set breakpoint. The cursor displayed is "**Brk pt**".

- A fault occurs. A program that address faults or write-protect faults causes the debugger to be invoked. The cursor displayed is "**Addr Fault**".

- To maintain a consistent and accurate map of the client's virtual memory, the debugger is invoked periodically to update internal data structures. This action, called *map logging*, requires no intervention from the user. The debugger automatically swaps back to the client when map logging is complete. The cursor displayed is "**Map Log**".

- In the event of a serious internal system error, the client world calls the debugger, indicating that a bug has been hit. The cursor displayed is "**Bug**".
- When CALLDEBUG is used, the cursor displays "**Int**" for interrupt.

7.2.2 Commands vs. expressions

The debugger has a command-line-oriented interpreter that accepts commands as well as language level expressions. Currently, the debugger only accepts Mesa language expressions. Programmers must manually convert C expressions to Mesa syntax to interpret them using by Copilot. Section 7.4 gives the details of converting from C to Mesa.

All debugger input falls into one of three categories:

1. A debugger command: The debugger is usually waiting for this type of input. The debugger prompts for command input with a greater-than sign (>).
2. A response to a debugger prompt: Some debugger commands require additional data. For instance, the command that sets the debugging context to a specified module prompts for the module name.
3. An expression to be interpreted: A leading space puts the debugger into expression interpreter mode. The debugger interprets the characters following the space as a Mesa language expression and evaluates it in the current context. Once the space has been typed, a backspace will not put the debugger back in command mode. You must either enter an expression or press the DELETE key.

Copilot uses a command completion scheme that allows you to enter the shortest unique prefix of each command word. For example, to enter the List Configurations command you need only type LC - L for List and C for Configurations. CoPilot supplies the remainder of the command words ("ist" and "onfigurations"). In the remainder of this document, debugger commands are presented as they would be expanded by Copilot. The characters you type are given in uppercase and are underlined.

7.2.3 Setting context

The debugger needs to know the context in which to interpret commands and expressions. When it is invoked, the debugger sets its context to a value that may or may not be the context that you are interested in debugging. The following table summarizes the debugger context, given the swap reason.

Swap Reason	Resulting Context	
	Correct Context	Other Context
Interrupt		Context not set anywhere in particular
Uncaught Signal	Context set to process that raised the **SIGNAL/ERROR**	
Call Debugger	Context set to process that called the debugger	
Breakpoint	Context set to process that hit the breakpoint.	
Fault	Context set to process that caused the fault	

Table 7-1: **Context vs. swap reason**

Table 1 shows that in most cases the debugger sets the context as desired. However, you must sometimes change context to set breakpoints or examine variables in other parts of the system. Copilot provides several commands for displaying and setting the context:

- CUrrent context: displays the current context.
- List Configurations: lists all loaded configurations (a configuration is a linked program).
- List Processes: lists all processes and their current state.
- ReSet context: sets the context to what it was when the debugger was entered.
- SEt Configuration [config]: sets the current configuration to the named configuration. This command operates within the scope established by Set Root Configuration.
- SEt Module context [module]: sets the current module to the named module. This command operates within the scope of the current configuration.
- SEt Process context [process]: sets the current process to the named process.
- SEt Root configuration [config]: sets the current configuration to the named configuration. This command establishes a scope for both the SEt Configuration and SEt Module Context commands.

7.2.4 Breakpoints

You can cause program execution to be suspended at a certain point by setting a breakpoint. A breakpoint can be set at the beginning of any statement, on entry to a procedure, and on exit from a procedure. To set a breakpoint on a statement, perform the following steps:

1. Set the context to appropriate module (it may already be set).

2. Load the source program into a file window.

3. Make a selection anywhere in the statement of interest.

4. Select the **Break** item in the file window's **Debug Ops** menu.

To set a breakpoint on the entry to, or exit from, a procedure, set the context as above and then use the Break Entry or Break Xit command. These commands prompt you for the name of a procedure. In all cases, the debugger indicates that the breakpoint has been set and gives the breakpoint number. This number is used to identify a breakpoint in other commands.

The debugger provides a variety of breakpoint manipulation commands. For a full list, see section 24.3.2.1 of the *XDE User Guide*. The most commonly used commands are listed below.

- List Breaks: Lists all currently set breakpoints.

- Display Break [number]: Gives information about the specified breakpoint.

- CLear All Breaks: Clears all breakpoints.

- CLear Break [number]: Clears the specified breakpoint.

- ATtach Keystrokes [number, keystrokes]: Attaches the keystrokes to the specified breakpoint. When the breakpoint is hit, the keystrokes are passed to the debugger as if you had typed them.

- ATtach Condition [number, condition]: Makes the specified breakpoint conditional. That is, the breakpoint is taken only if the attached boolean expression is satisfied. As with all expressions, the conditional is given in Mesa syntax.

7.2.5 Displaying the call stack

Copilot allows you to examine the call stack of any process by using the Display Stack command. Typing this command displays the topmost procedure on the call stack of the current context, and enables a variety of subcommands. These subcommands can only be used in Display Stack mode, and the normal debugger commands cannot be used until you exit this mode. The expression interpreter can still be invoked by typing a space. The following list summarizes the subcommands:

- N: move to the next entry on the call stack (that is, the procedure that called the current procedure).
- B: move to the previous entry on the call stack (that is, the procedure called by the current procedure).
- G: display the global variables of the module containing the current procedure.
- P: display the parameters of the current procedure.
- R: display the return value of the procedure. Because return parameters are unnamed in C, Copilot will give *anon* as the name of the return variable.
- J [*n*]: Jumps *n* entries down the stack. If *n* is too large, the debugger responds with the upper bound on *n*.
- S: Prints the source line where the current procedure is stopped (this will usually be a procedure call). Also loads the source file into a file window and positions the text to the corresponding source line.
- L: Same as S, but just displays the source line in the debugger log.
- Q or **DELETE** key: Terminates Display Stack mode and returns to the main command processor.

As you move through the call stack, you change the context in which expressions are evaluated. A procedure's local variables are only visible if that procedure is equal to or below the current procedure on the call stack. Similarly, the global variables of a module are only visible if a procedure in that module is equal to or below the current procedure on the call stack.

7.3 Types and values

The debugger uses Mesa syntax to print types and values. This section describes the format that the debugger uses to display each C type and value. It also describes some naming anomalies and how to work around them.

7.3.1 Display format

The following subsections describe the format that the debugger uses to display values of each of the C data types. Copilot provides some flexibility in how values are displayed via the Copilot options window. To get this window, select the item labeled **Options** in the **Copilot** pop-up menu. Using the options window, you can change the display representation of various types. The following list summarizes the fields of the options window and describes how these names relate to C datatypes:

CARDINAL:	This item controls the display format for unsigned quantities (**unsigned int, unsigned short**, and **unsigned long**). They can be displayed in octal, decimal, or hexadecimal.
INTEGER:	This item controls the display format for signed quantities (**int**, **short**, and **long**). Once again, you can choose either octal, decimal, or hexadecimal.
POINTER:	Unused for C
LONG POINTER:	Controls the display format for all C pointer types. You may choose octal or decimal.
RELATIVE:	Unused for C
UNSPECIFIED:	Unused for C
Array elements:	When displaying an array, the debugger will display up to this many elements.
String length:	Unused for C
Apply:	Apply the selected options and remove the options window
Abort:	Reset any changes made to the options and remove the window

7.3.1.1 Simple scalar types

int: You may display these in either octal, decimal, or hexadecimal format. As mentioned above, unsigned values can be displayed differently than signed values. All sizes of **int** (**short**, **int** and **long**) are treated uniformly. The decimal number 25 is displayed as:

31B (octal) or **25** (decimal) or **19X** (hexadecimal)

char: The debugger displays characters with a single quote followed by a character. If the debugger encounters a non-printing character, it will display a mnemonic for it. For instance, Copilot displays a carriage return character as CR and a line feed as LF. Uninterpreted control characters are displayed with a leading ↑. For instance, control-H is displayed as ↑H. If the character code is greater than 127, then the debugger displays the character code in octal.

float: CoPilot displays floats as a string of digits followed by a decimal point, followed by a string of digits. Based on the magnitude of the float, CoPilot may decide to display the value using exponent notation. For example, 1,000,000 = 1e6 = 1 * 10 ↑ 6.

pointer: The debugger treats all C pointers uniformly. It displays them as either octal or decimal values followed by a ↑ to indicate that the value is a pointer. Note, however, that a pointer to a character has a different internal representation than a pointer to another type. A character pointer cannot be used directly with debugger commands that prompt for an address. The low order bit of a character pointer indicates which byte is of interest and the high order bit is turned on to show that it is a byte pointer. So, to find the word of memory refered to by a character pointer, you must remove the high-order bit and divide by two. For example, if a pointer has the value 20004567891 octal, you would divide 4567891 by two to find the word of interest. See §7.6 for a description of character pointer support.

7.3.1.2 Aggregate Types

array: The debugger displays an array in the following format:

(n)[el_0, el_1, el_2, el_3, ..., el_{n-1}]

where **n** is the size of the array and **el_0** through **el_{n-1}** are the elements of the array. The debugger displays an element of an array in the same way it would display any other variable of that type.

struct: Copilot displays a structure in the following format:

[fieldName:value, nextFieldName:value, ..., lastFieldName:value]

The debugger displays each field of the structure in the same way it would display any other variable of that type.

union: The debugger has to be given some information to do anything useful with a union. With no information, the debugger displays the value of any union as:

[OVERLAID[...]]

The debugger prints **OVERLAID** to denote the fact that several different representations of the value are being overlaid. To see the value using a particular representation, you must specify the variant of interest by typing the field name. The type of the field determines the representation to use. The debugger displays the resulting value in the same way it would display any other value of that type. For example, say we have a union declared as:

```
union myUnion {
   int   variant1;
   char variant2;
   float variant3;
} myUnionVariable;
```

The debugger displays:

```
myUnionVariable          as [OVERLAID[...]]
myUnionVariable.variant1  as an int
myUnionVariable.variant2  as a char
myUnionVariable.variant3  as a float.
```

7.3.1.3 Other Types

enum: Copilot displays a value of type **enum** with the name used in the source program. For example, consider the following code fragment:

```
enum colors {red, blue, green};
enum colors carColor;
carColor = red;
```

If you were to display carColor using the debugger, it would print the identifier *red* as the value.

function: Copilot displays the value of a function variable by giving the function's name and the name of the module that contains the function. For instance, consider the following code fragment taken from the file **MyProgram**:

```
char (*funcVar)(); /* Pointer to function returning char */

char charProc(c)
char c;
{
}

otherProc()
{
   char (*p)(); /* Pointer to function returning char */
   funcVar = charProc;
}
```

If you were to ask for the value of **funcVar** after it was assigned to in **otherProc()**, the debugger would display

```
funcVar = PROCEDURE charProc (
   in module MyProgram, G: nnnnnnB)
```

The number following the **G:** further identifies the module. It is an indication of where the module's global variables are stored.

7.3.2 Type values

As well as displaying the value of variables, Copilot can also display the value of types. That is, if a program gives a name to some type (using a **typedef**, **struct**, **union**, or **enum** construct), Copilot can display that type at debug time without the source file. For instance, the following code fragment declares an enum, a struct, a typedef, and a union:

```
enum colors {red, blue, green};
typedef struct Circ *Circ;
struct Circ {int cf1; Circ cf2;};
union simpleUnion {
   int variant1;
   char variant2;
   long variant3;
};
```

Note that in this case, the **typedef** name and the **struct** name are the same. This situation can also arise for **enums** and **unions**. To avoid naming conflicts, the compiler prepends STR to **struct** names, UNN to **union** names, and ENM to **enum** names. The compiler does not change **typedef** names.

The debugger displays type values using Mesa syntax. A programmer declares new types or aliases old ones by using the **struct, union, enum,** and **typedef** constructs. The following list describes how the debugger displays types declared using each of these constructs.

7.3.2.1 Struct's

For **struct**'s, the debugger displays the following:

```
STRname: PUBLIC TYPE = MACHINE DEPENDENT RECORD [
   fieldName: fieldType,
   nextFieldName: nextFieldType,
   ...,
   lastFieldName: lastFieldType]
```

Where:

STRname	is the name of the **struct** type prepended with STR
PUBLIC	indicates that the type is not PRIVATE in the Mesa sense (this can be ignored)
TYPE	indicates that this represents a type declaration
MACHINE DEPENDENT	indicates that field positions will not be changed by the compiler (this is always the case for C **structs**, so it can be ignored).
RECORD	is the Mesa substitute for the **struct** keyword.
fieldName	represents the name of a member.
fieldType	represents the type of the field.

For example, the debugger displays the following for **struct Circ** above:

```
STRCirc: PUBLIC TYPE = MACHINE DEPENDENT RECORD [cf1: int, cf2: LONG POINTER TO STRCirc]
```

7.3.2.2 Typedef's

For **typedefs**, the debugger simply displays the type that was equated to the **typedef** name. From the fragment above, **Circ** would be displayed as:

```
Circ: PUBLIC TYPE = LONG POINTER TO STRCirc
```

Where:

Circ	is the name of the type.
PUBLIC	indicates that the type is not PRIVATE in the Mesa sense (this item can be ignored).
TYPE	indicates that this represents a type declaration.

LONG POINTER TO STRCirc	is the definition of the new type.

7.3.2.2 Enum's

Copilot displays enumerated types in a straightforward way:

ENMname: PUBLIC TYPE = {item_1, item_2, ..., item_n}

Where:

ENMname	is the name of the **enum** type prepended with ENM.
PUBLIC	indicates that the type is not PRIVATE in the Mesa sense (this item can be ignored).
TYPE	indicates that this represents a type declaration.
item_i	is the textual name of one of the enumerated items.

From the fragment above, colors would be displayed as:

ENMcolors: PUBLIC TYPE = **{red, blue, green}**

7.3.2.3 Union's

The display format of a **union** type is somewhat complex. It takes the form:

```
UNNname: PUBLIC TYPE = MACHINE DEPENDENT RECORD [
  XX: SELECT OVERLAID PRIVATE * FROM
    YYY = > [variant: varType],
    ZZZ = > [nextVariant: nextVarType],
    ...,
    WWW = > [lastVariant: lastVarType],
    ENDCASE]
```

Where:

UNNname	is the name of the union type prepended with **UNN**.
PUBLIC	indicates that the type is not PRIVATE in the Mesa sense (this item can be ignored).
TYPE	indicates that this represents a type declaration.
MACHINE DEPENDENT	indicates that fields' position will not be changed by the compiler (this is always the case for C unions, so it can be ignored).
RECORD	is the Mesa substitue for the union keyword. Mesa implements unions as variant records, so the keyword for struct's is the same as for union's.
XX	is the name of a variant tag. Because C unions do not have tags, this name is generated by the compiler and can be ignored.
SELECT OVERLAID PRIVATE * FROM	This phrase is used to introduce the variants of the union

YYY =>	introduces a particular variant. The string in place of YYY can be ignored.
variant	is the member name of a particular variant.
varType	is the type of that variant.
ENDCASE	indicates the end of the variant list.

So, for **simpleUnion** in our fragment above, Copilot displays:

```
UNNsimpleUnion: PUBLIC TYPE = MACHINE DEPENDENT
RECORD [
   LDLR38: SELECT OVERLAID PRIVATE * FROM
      DLRTDLR40 = > [variant1: INTEGER],
      DLRTDLR41 = > [variant2: CHARACTER],
      DLRTDLR42 = > [variant3: LONG INTEGER],
      ENDCASE]
```

7.3.3 Anomalies

The XDE C compiler and debugger handle some C data types in an unconventional way. This section lists all such anomalies and describes how to deal with them when debugging.

7.3.3.1 Global arrays

To avoid space problems, the XDE C compiler allocates global arrays in a different area than other global variables. Because of this, global arrays are really pointers to arrays. This is completely transparent to the C programmer until debugging, then a global array will appear as a pointer to an array of the same type, which must be dealt with accordingly. That is, to see the contents of a global array rather than a pointer to the contents, you must dereference the global array variable.

7.3.3.2 Multiply defined names

The C language allows the same name to be used several times within a single scope. For example, you might declare a **struct** named myStruct and then create a **typedef** of that **struct** using the same name, such as:

```
struct myStruct {
   int field1;
   char field2;
} instance;
typedef struct myStruct myStruct;
```

To avoid naming conflicts, the XDE C compiler prepends an identifier to **struct**, **union**, and **enum** names (STR, UNN, and ENM respectively) in the debugger symbol table. So to refer to the struct above using the debugger, you type **STRmyStruct**. To refer to the **typedef**'d name, you would simply type **myStruct**.

It is important to understand that the compiler only modifies **struct, union,** and **enum** names, not the names of variables of those types. For example, in the fragment above we have declared a **struct** variable called instance. You refer to that variable by its declared name, instance, not by **STRinstance.** Furthermore, as mentioned above, the compiler only changes names in the debugger's symbol table. Within a program **struct, union,** and **enum** names are used normally.

7.3.3.3 Statics

Local **static** variables introduce another potential naming problem because the XDE C compiler makes them global data. That is, a programmer could declare a **static** called myStatic in two functions and would end up with two globals named myStatic. For this reason, the compiler prepends a function name to all locally declared **static** variables. The following example illustrates this point:

```
static myStatic;     /* myStatic => myStatic. Global statics
                        are untouched */
void func1()
{
   static myStatic; /* myStatic => func1__myStatic. */
}
void func2()
{
   static myStatic; /* myStatic => func2__myStatic. */
}
```

As for **struct, union,** and **enum** names, the compiler only changes the name of a local static in the debugger's symbol table. Within a program, local static vartiables are referred to normally.

7.3.3.4 Doubles

The debugger does not currently support doubles. CoPilot displays a double as if it were defined with the following C declaration:

```
struct double {
   long high;    /* 32 bits of data */
   long low:/* 32 bits of data */
};
```

In general, it is not reasonable to interpret a double displayed in this format as a floating-point number. See §7.6 for information about additional support for displaying doubles.

7.3.3.5 Strings

Copilot displays strings declared as an array of characters (**char** string[size]) using the array format described in §1.3.1.2 above. Although CoPilot has no facility for displaying a pointer to character (**char** *string) as a null terminated string, it can display a block of memory as Ascii characters. The ASCii Read command takes a memory location and a size (in bytes) and displays the block of memory in Ascii. For example, assume we have a **char** pointer named foo that points to the string "This is a test of the emergency broadcasting system". To display the first twelve characters pointed to by *foo*, convert the character pointer *foo* into a word pointer (see §7.3.1.1) and then give that value to the ASciiRead command:

```
ASciiRead: 4456792, n(10): 12
```

The debugger responds with:

```
This is a tes
```

See §7.6 for information about additional support for displaying C strings.

7.4 Evaluating expressions

This section describes how to evaluate expressions using CoPilot. As mentioned above, the debugger expects Mesa expressions, not C expressions. The following subsections describe how to convert expressions involving each of the basic C types into Mesa expressions. Remember that you must put CoPilot into expression interpreter mode by typing a leading space.

7.4.1 Single variable expressions

To display the value of a variable, enter an expression consisting of only that variable. For instance, to see the value of the float variable pi, type

```
pi↵
```

The debugger responds with the value of the expression, which in this case is simply the value of pi.

7.4.2 Dereferencing

Given a pointer variable, you can display its referent by using the dereferencing operator, an upward-pointing arrow (↑). For instance, say a program contains a pointer named arrayPtr to an array of 5 **ints**. Displaying the variable arrayPtr results in the value of the pointer not the value of the array. To see the array, type

```
arrayPtr↑↵
```

Copilot displays the pointer's referent in the format appropriate to its data type.

7.4.3 Subscripting

To display an element of an array (regardless of the element type), give the array name followed by an open square bracket, followed by the array subscript, followed by a closing square bracket. For example, to display the third element of an array called ra, one would type

ra[2]¢

It is important to note that Mesa does not share the C language view that pointers and arrays are almost equivalent. This means that the debugger will not accept a subscripted pointer variable.

7.4.4 Field Access

To access a field of a **struct**, use the familiar dot notation. So, to access a field named empName in the **struct** variable empRec, type

empRec.empName¢

CoPilot displays the specified field using the format appropriate to the field's type. Similarly, to access a some variant of a union variable, give the name of the union variable followed by a dot and the name of the variant.

To access a field through a **struct** or **union** pointer, you can dereference the pointer and then use the dot notation. This means that a C expression such as

unionPtr->variantName *becomes* unionPtr↑.variantName

Actually, you can make such references even more simply than that. You can type:

unionPtr.variantName¢

The debugger will notice that unionPtr is a pointer and dereference it automatically. Consequently, you can use the dot notation for both **struct/union** variables and **struct/union** pointers.

7.4.5 Assignment

Assignement in Mesa uses the "get" operator, a left-pointing arrow (←). So to assign an **int** variable x the value **755**, type

x ← 755¢

Similarly, to assign the value of x to the field empNo, which is pointed to by the **struct** ptr empRec, type

empRec↑.empNo ← x *or simply* empRec.empNo ← x

7.4.6 Address of operator

As in the C language, you can obtain the address of a variable using the "address of" operator. In C this is the & operator; in Mesa it is the @ operator. Mesa has pointer types, **POINTER** and **LONG POINTER**. All C pointers are **LONG POINTERS**. Unfortunately, the debugger generates a **POINTER** as the result of the @ operator. To get a C pointer (**LONG POINTER**), you must explicitly lengthen the result of an @ expression using the **LONG[]** construct. For example, to put the address of an int variable **myInt**, into an int * variable **myIntPtr**, type

myIntPtr ← **LONG[@myInt]**

At present there is no automatic way to generate a character pointer. You must generate a normal pointer and reverse the process described in §7.3.1.1.

7.4.7 Arithmetic operations

Table 7-2 summarizes which arithmetic operations can be applied to each of the basic arithmetic C types.

	short	int	long	char	pointer	float	double
+	yes	yes	yes	yes	yes[1]	no	no
-	yes	yes	yes	yes	yes[1]	no	no
*	yes	yes	yes	yes	yes[1]	no	no
/	yes	yes	yes	yes	yes[1]	no	no
MOD	yes	yes	yes	yes	yes[1]	no	no

Table 7-2: Applicability of arithmetic operators

[1] Although these operations are available for pointers, their meaning is questionable. (That is, what does it mean to multiply two pointers?)

7.4.8 Expressions involving function applications

You can evaluate expressions that involve function applications in a very straightforward way. Mesa uses square brackets rather than parentheses to delimit function parameters, so a C expression like

<u>4 * f(i)</u> *becomes* <u>4 * f[i]</u>

The debugger displays the expression value using the format corresponding to the expression's result type. Invoking a function in this way causes the same side effects that a normal function application causes.

7.4.9 Type coercion

Occasionally you must interpret a value of one type as being a value of another type. Using C, you can accomplish this by using a cast. In Mesa you use the LOOPHOLE construct. LOOPHOLE has two parameters, a value and a type: LOOPHOLE[**value, type**]. The result of a LOOPHOLE is the input value tagged with the input type. Because the debugger accepts Mesa expressions, you must perform type coercions by using LOOPHOLE.

There is a very important difference between a cast and a LOOPHOLE. A cast converts a value from one type to another; a LOOPHOLE does not. LOOPHOLE simply tells the type checker to accept the value as if it had the named type. A LOOPHOLE never converts its input to a different representation. The following comparison demonstrates the importance of the distinction between a cast and a LOOPHOLE:

Case 1: A cast from **long** to **float** in a C program

```
long k = 1;
float x;
x = (float)k;
```

Result: x has the value 1.0

Case 2: A LOOPHOLE from **long** to **float** using CoPilot

```
x ← LOOPHOLE[k, REAL]
```

Result: **x** has the same bit pattern as **k**, which is not 1.0 in the floating-point representation.

7.5 Limitations

This section reviews both the general and the C-specific limitations of the current debugger. We have touched on some debugger limitiations already, but we will repeat them here for completeness. The first subsection deals with general debugger limitations; the second addresses C-specific limitations.

7.5.1 Generic debugger limitations

Copilot has two primary limitations :

- Tracing: With some debuggers it is possible to suspend execution when a variable takes on a specific value, or when a variable changes value. This feature can be useful when a variable is being "smashed" by some unknown agent. Copilot does not provide integral support for this functionality.

- Single stepping: Copilot has no direct support for single-stepping statements at the source code level. That is, you cannot simply tell the debugger to execute one statement of a program. To simulate this, you must set a breakpoint on each statement that can follow the current statement in execution order.

7.5.2 C-specific limitations

There are five primary limtiations for C program debugging using Copilot:

- Syntax translation: Copilot expects expressions to be given using Mesa syntax. Also, the debugger displays values using Mesa format and notation. Consequently, Copilot users must on occasion transpose from C to Mesa and vice versa.

- Character pointer manipulation: You cannot directly dereference a character pointer, nor can you supply it as an address to a Copilot command. You must first convert it to a word pointer as described in §7.3.1.1. The debugging aid described in §7.6, CPrint, alleviates many of the problems associated with this limitation.

- External variables: At present, you cannot directly access **extern** variables. To reference an **extern**, you prepend the variable name with the name of the module that contains the storage for the variable. For example, assume that a file called **user.c** contains the declaration:

  ```
  extern int h;
  ```

  and the file **definer.h** contains the declaration:

  ```
  int h;
  ```

  To reference the variable h while debugging *user.c*, type

  definer$h␛

  The dollar sign ($) tells the debugger that the preceding name is the module containing h. External variables can be accessed directly if the context is set to the module that contains the storage for the variable.

- Cpp confusion: Several debugging problems arise from the use of the C preprocessor. First, all of the symbolic names defined using **#define** are lost. The compiler never sees these names because the preprocessor removes them. Since the compiler does not see the names, it cannot keep them around for the debugger.

  Second, the debugger cannot correlate included code with source positions. That is, if a C program **#includes** a file that contains a function, the debugger will not be able to find the source text for that function. This means that you cannot set source breaks in such a function, nor can the debugger show the source position of a suspended process if the position is in an included function.

- Naming anomalies: As mentioned above, you may have to take some special action to display certain types and values. Naming anomalies arise when the compiler must change a varaible or type name to avoid ambiguity. Of course, names are only changed from the debugger's point of view. The programmer uses all names normally. The following list summarizes the name transformations made the compiler:

  - The names of **struct**'s have STR prepended to them.

    struct foo {} = > STRfoo

  - The names of **union**'s have UNN prepended to them.

    union bar {} = > UNNbar

  - The names of **enum**'s have ENM prepended to them.

    enum mumble {**a, b, c**} = > ENMmumble

  - The names of **static**'s have a function name prepended to them.

    void proc1() {**static int testcase;**} = > proc1_testcase

7.6 A guide to CPrint, a C debugging aid

CPrint is a program that works in conjunction with Copilot to handle the C data types that are not handled well by Copilot alone. In particular, CPrint allows youto interpret a character pointer as a pointer to a null-terminated string of characters, and it allows youto display doubles in an intelligible format.

The first section below describes the basic functionality of CPrint. The second section describes the commands and provides other information necessary for operation.

7.6.1 Functionality

CPrint operates by intercepting debugger requests and handling certain requests on its own. Specifically, CPrint intercedes whenever Copilot is asked to display a pointer to character or a double. When a pointer to character passes by, CPrint reads the referenced characters and displays them. It displays characters until it finds a null or until it reaches a user-determined limit on string length. CPrint displays the characters as well as the pointer value and the string length. For example, assume cptr is a character pointer that points to the string "my mother the car." To display the string, you simply type the name of the pointer to Copilot, and CPrint will do the rest:

```
cptr⏎
```

The debugger responds with:

```
cptr = (20004567891, 17) "my mother the car"
```

CPrint only intercepts references to character pointers and a type called **DoubleReal.Double**. So, to display the value of a double, you must LOOPHOLE the variable to the type **DoubleReal.Double**. For instance, say you have a double variable called pi, to display the value of pi you enter the following LOOPHOLE expression:

```
LOOPHOLE [pi, DoubleReal.Double]⏎
```

The debugger responds with:

```
pi = 3.1415926
```

7.6.2 Operation

CPrint operation is very simple. All CPrint commands are given to the Executive. To make CPrint active, simply type

```
CPrint on⏎
```

to the executive. You may disable the functionality of CPrint at any point by typing the command

```
CPrint off⏎
```

to the executive. The remaining CPrint commands are summarized below:

- CPrint SetSize/n: Set the maximum length of displayed strings to n.
- CPrint SetPrec/n: Set the precision with which doubles are displayed to n.

- CPrint Notation/{Sci or Normal}: Set the display format of doubles to either scientific notation or normal decimal notation.

- CPrint Show: Show the settings of all CPrint options.

- Help CPrint: Display the help text for the CPrint Program.

In order to run CPrint, you must have the file **CPrint.bcd**. To interpret doubles, you must have the file **DoubleReal.bcd** on your search path, and you must be running either the C Environment or **FloatingPt.bcd** (the C Environment contains **FloatingPt.bcd**) in the debugger instance of the XDE.

(This page intentionally blank.)

A. PORTING C PROGRAMS TO THE XDE

This appendix describes some points you should consider when porting C programs to ViewPoint or XDE.

A.1 Differences in C environments

There are two types of differences between the C environment in ViewPoint and XDE and the C environment on other machines. The first is caused by the differences in architectures between Mesa PrincOps and register-based machines. The second is the result of differences between the ViewPoint and XDE environments and other programming environments.

A.1.1 Machine architecture

The Mesa PrincOps architecture supports 16-bit integers and 32-bit pointers. Many C programs coming from other machines assume that pointers and integers are the same size. This can create problems in porting programs. The program "Lint" can help in detecting such potential problems.

Because a Mesa processor is a word-addressable machine, character handling requires special attention. Anything that is of type "pointer to char" is a byte pointer. The representation of these pointers is different than that of a normal pointer because the Mesa architecture allows addressing only of words, not bytes. Byte pointers are 32 bits long, with the following interpretation:

1 TAG BIT	30 BITS FOR THE WORD POINTER	1 BIT FOR WHICH BYTE

Figure A-1: **A 32-bit byte pointer**

The high-order tag bit is always one. Because Pilot only implements 24 bits of virtual memory, using a byte pointer as a normal pointer results in an address fault. Similarly, using a normal pointer when a byte pointer is expected results in a byte pointer trap. The C compiler handles all conversions of the two pointer representations as needed except for one case: passing pointers as parameters to functions. No type checking is done between the caller and the callee, so the compiler doesn't have the information needed to know if a conversion is needed. Any necessary conversions must be done by the programmer with casts. If you get a byte pointer trap and think you have a normal pointer that points to text, try the debugger's Ascii read command.

If you want to convert a byte pointer to a normal pointer manually, get the octal representation of the pointer value. Then strip off the leading "2" and any zeroes that follow it. Divide the result by two and you'll get the word to which the byte pointer was referring. For example, "20000124520B" would be "124620B/2" = "52310B".

On Mesa processors, the parameters are passed in a register stack instead of a stack in memory. This leads to the requirement that the number and size of actual and formal parameters must match exactly on a function call.

A.1.2 The ViewPoint and XDE operating environments

When a C program consists of more than one C source module, those modules must be linked in the sequence in which they depend within a program. This is necessary because the global variables within these modules are initialized in the same sequence as they are linked. For example, if module A contains a global pointer that is initialized to point to an array in module B, then module B must come before module A when these modules are linked. Otherwise, the pointer in module A is initialized before storage is allocated for the array in module B, and the pointer will point to an illegal address.

In UNIX environments all the global variables are allocated to a "common" segment by the linker. If two or more C source modules contain a declaration of the same global variable, the linked program has only one instance of that variable. But in ViewPoint and XDE, the linked program will have as many instances of global variables as there are global declarations. When porting C programs the include files must be checked for variable declarations that include storage allocation (as opposed to extern declarations). In some C programs, one include file containing the declarations of global variables may be included in more than one C source module. To port such programs, it is necessary to declare these variables as global in one module (normally the module with the routine main) and then define them to be **extern** in the include file.

A.2 Porting steps

This section outlines the suggested steps in porting programs from another environment to ViewPoint or XDE. A short motivation for each step is provided along with a description of ways to deal with some of the obstacles to porting.

A.2.1 De-lint the source program

In general, it is a good idea to run Lint on the programs to be ported to ViewPoint or XDE. The use of the Lint switches "-hp" aid in locating portability sensitive source code. A note about the philosophy of the program Lint should be understood during these efforts; users are encouraged to review S.C. Johnson's *Lint--A C Program Checker*. The sections "A Word About Philosophy" and "Portability" are particularly helpful.

A common complaint with Lint is that it produces many error messages for legal C constructs. In Lint's defense it should be said that too much output is preferable to too little, and that the vast majority of porting obstacles that have been encountered so far could have been avoided by careful scrutiny of the Lint output. Use Lint!

Often, you may be willing to lose some special system-dependent function in exchange for a quicker port. If this is the case, commenting out code is one option, but another is to make use of the C preprocessor's conditional compilation facility. Instead of commenting out the code, use the phrases **#ifndef princops** and **#endif** to bracket the undesired code. It will be included the code in compilations on the host machine but will be deleted when the program is compiled on the PrincOps machine. (The C preprocessor on the PrincOps machine automatically defines **princops** to be true.)

A.2.2 Check for system-dependent function calls

Programs that were written on other systems may make extensive use of operating system calls that are not part of the C language. Basically there are two strategies for dealing with these calls: implement the call in software or rewrite the code so that the system call is not required.

As an example, consider porting a program from a UNIX environment where the program includes the system calls **open()** and **close()**. These calls use the Unix operating system's notion of file descriptors, for which there is no equivalent in ViewPoint or XDE. One approach would be to implement the calls **open()** and **close()**, perhaps by using a table to bind FILE pointers to integer file descriptors and then using the stdio routines **fopen()** and **fclose()** to do the actual file manipulation. Another option, in this case preferable, is to convert the **open()** and **close()** calls to **fopen()** and **fclose()** calls and to replace all occurrences of file descriptors with pointers to FILEs (that is, replace **read()** with **fread()** and **write()** with **fwrite()**). This technique has the desirable side effect of rendering the code less system dependent and hence more portable to subsequent environments.

But how do you determine which routines are system calls? The easiest way is to let the compiler help you. If you compile the example program discussed above with cc, an error message is printed for each undefined procedure (in this case, the list would include **open**, **close**, **read**, and **write**). Another technique is to use the XDE program Lister (see the *XDE User's*

Guide) to examine the bcd links (use **Lister bcdlinks[example.bcd]** then examine the resulting file **example.bl**).

A.2.3 Implement system-dependent function calls

In some cases it will not be possible to side-step the system dependencies of a program, and in fact those system dependencies may be crucial to the functionality of the program. In this case there is little to do but implement the call. Yet here again there are options: you can implement the function in C using lower-level system-independent calls, or you can implement the function by using the existing facilities of ViewPoint or XDE. For example, suppose that a program that manipulated directories was being ported to XDE, and at some point in the code you need to know the answer to the question: "Is the file pointed to by this FILE pointer actually a directory?" It would be possible (but painful!) to write the C routines necessary to dereference the pointer, examine the FILE contents, and decide whether it was actually a directory.

In this case it would be much easier to make use of the Mesa **MFile** interface procedure **MFile.GetProperties** to obtain an **MFile.Type (unknown, text, binary, directory or null)** for the file in question. Figures A-2, A-3, and A-4 show an example use of a Mesa implementation of a system-dependent function for a C program. This program simply checks to see if the file **foo** is a directory. The program is constructed by first compiling the Mesa programs:

```
>Compiler exampleMesaDefs exampleMesaImpl
```

and then compiling the C program and linking it with the Mesa implementation module:

```
cc -o example exampleCImpl.c exampleMesaImpl.bcd
```

This example is stored in the release directory in the folder "Examples" for programmer reference. In particular, see the file **Example.df**.

```
/* Example program to demonstrate Mesa interfaces */

#include <stdio.h>

#ifdef princops
mesa int exampleMesaDefs__ftype();
#define ftype(s) exampleMesaDefs__ftype(s)
#endif

main()
{
    if (ftype("FOO") = = 1) printf("FOO is a directory\n");
}
```

Figure A-2: Example**CImpl.c**

Several points should be noted in this example. In **Example.c**, the syntax

mesa int exampleMesaDefs__ftype();

is introduced. This extension to the C language is documented more fully in chapter 3.

```
-- exampleMesaDefs.mesa
-- ftype returns 1 if the file with the name pointed to by s is a directory, 2 if a file else returns 0.

DIRECTORY
    CString USING [CString];

exampleMesaDefs: DEFINITIONS =
{
    ftype: PROCEDURE [s: CString.CString] RETURNS [INTEGER];
}.
```

Figure A-3: **ExampleMesaDefs.mesa**

The Mesa definitions module (here example**MesaDefs.mesa**) must be produced and the corresponding **.bcd** file (here **exampleMesaDefs.bcd)** must be on the search path because the assembler uses this module to resolve function references.

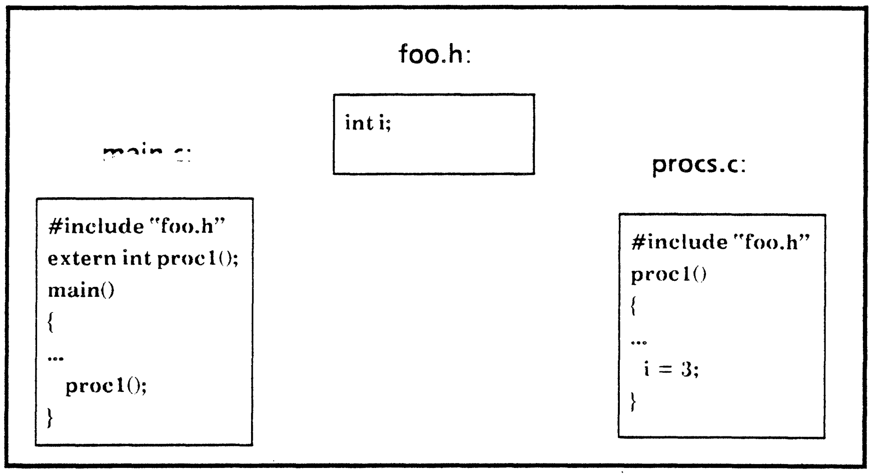
```
--exampleMesaImpl.mesa

DIRECTORY
    exampleMesaDefs USING [],
    CBasics USING [RegisterFrame],
    CString USING [CString, CStringToLongString],
    Heap USING [systemZone],
    MFile USING [Acquire, ByteCount, Error, GetProperties, Handle, Release, Type],
    Time USING [Packed];

exampleMesaImpl:PROGRAM
    IMPORTS CBasics, CString, Heap,MFile
    EXPORTS exampleMesaDefs =
{
    ftype: PUBLIC PROCEDURE [s: CString.CString] RETURNS [retval: INTEGER] =
    BEGIN
    fileName: LONG STRING←CString.CStringToLongString[s,Heap.systemZone];
    BEGIN ENABLE UNWIND = > IF fileName # NIL THEN Heap.systemZone.FREE[@fileName];
    create, write, read: Time.Packed;
    length: MFile.ByteCount;
    type: MFile.Type;
    deleteProtected, writeProtected, readProtected: BOOLEAN;
    file: MFile.Handle ← MFile.Acquire[name: fileName, access: anchor, release: [NIL, NIL]
                                   !MFile.Error = > {retval ← 0; GO TO return};];

    [create, write, read, length, type, deleteProtected, writeProtected, readProtected] ← MFile.GetProperties[file
                                   !MFile.Error = > {MFile.Release[file]; retval ← 0; GO TO return};];
    MFile.Release[file !MFile.Error = > {retval ← 0; GO TO return};];
    SELECT type FROM
        directory = > retval ← 1;
        text, binary = > retval ← 2;
        ENDCASE = > retval ← 0;
    IF fileName # NIL THEN Heap.systemZone.FREE[@fileName];
    RETURN[retval];
    EXITS return = > { IF fileName # NIL THEN Heap.systemZone.FREE[@fileName]; RETURN [0];};
    END;
    END;

[] ←CBasics.RegisterFrame[0];
}.
```

Figure A-4: **ExampleMesaImpl.mesa**

In any Mesa implementation module that will be used with C, care must be taken to catch all possible signals that might be raised by calls to other routines because most C programs have no notion of signals in the Mesa sense.

A.2.4 Check include files for storage allocation

As discussed in section A.1.2, the declaration of global variables in a file that is included in more than one module will not have the desired effect of sharing the single common global variable but instead will declare multiple global variables with the same name. The fix is to declare these variables in a single location (normally the module that contains the routine **main()**) and then to modify the include file to declare the same variables as **extern**. Figures A-5 and A-6 describe an example of this conversion.

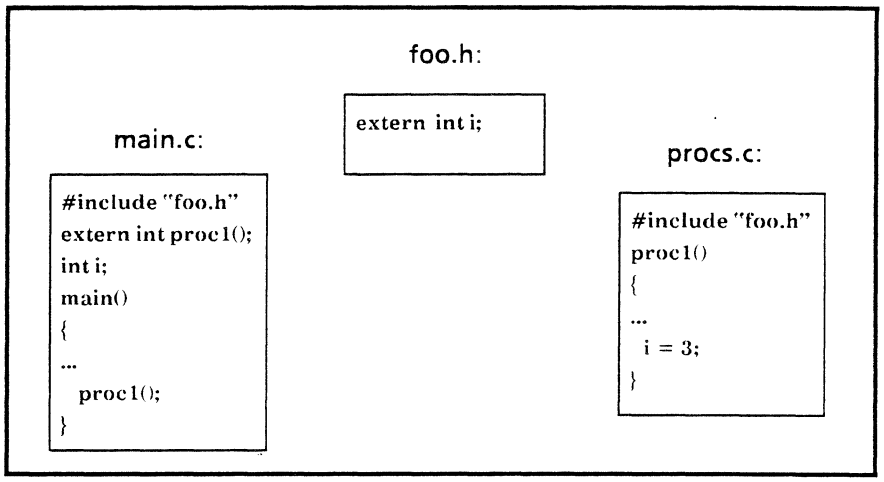


```
int i;
```

```
#include "foo.h"
extern int proc1();
main()
{
...
  proc1();
}
```

```
#include "foo.h"
proc1()
{
...
  i = 3;
}
```

Figure A-5: Configuration before modification



```
extern int i;
```

```
#include "foo.h"
extern int proc1();
int i;
main()
{
...
  proc1();
}
```

```
#include "foo.h"
proc1()
{
...
  i = 3;
}
```

Figure A-6: Configuration after modification

A.2.5 Other hints

Longs (and pointers) are not the same size as **ints**. See the table in chapter 3.

All C programs must be linked, even if there is only one source module. CC takes care of linking for you, but if you use the -c option with cc, you must at some point link the resulting module before you can run it. See chapter 4 for more details.

The start order of a multi-file configuration is significant. The order in which modules are started is the same as the order in which modules are specified on the command line to CC or to the linker.

The type of most numeric constants is **int** or **long**. This can have odd effects when combined with unsigned variables, especially in comparisons, division and mod. A comparison is unsigned if either operand is unsigned.

As a general rule, it is best to keep the source in a state so that it will compile and run on the original host. The process of porting tends to be iterative, so that at frequent intervals it is desirable to rerun the source on the original host to detect any machine-independent errors that may have been introduced while porting. In particular, the **#ifndef princops ... #else ...#endif** construct is very helpful in maintaining this compatibility.

A.3 Debugging

Debugging a C program in XDE is very similar to debugging a Mesa program, and in fact the syntax for using the CoPilot debugger is unchanged. See chapter 7 and the *XDE User's Guide* for the details of debugger use.

Some C-specific debugging hints are provided below.

A.3.1 Address faults and stack errors

A common problem encountered when porting C programs is the occurrence of address faults, particularly at the entry to function calls. These address faults are often the result of a mismatch in actual and formal parameters of a function call and frequently manifest themselves as address faults in **printf()**. The C language allows the number of actual parameters to be different from the number of formal parameters. This presents problems to a stack machine because the stack is expected to contain exactly as many actual parameters as formal parameters in the function that was called. To detect this problem the program can be compiled with the 'd' (stack-checking) switch. It checks that the stack is empty at all statement boundaries and that functions return the number of words the caller expects. Stack checking will slow a program down significantly because a stack dump and restore is required for each procedure call. Again, Lint is of particular aid in detecting mismatches of this kind.

A.4 References

[1] *XDE User Guide*. Version 3.0 [November 1984].

[2] *The C Programming Language*. Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, Inc., Englewood Cliffs, N.J. 1978.

[3] *Lint -- A C Program Checker*. S.C. Johnson.

UNIX is a trademark of Bell Laboratories.

(This page intentionally blank.)

B. COMPATIBILITY WITH MESA

This appendix describes points that must be considered when calling Mesa procedures from C programs. It describes problems that may be encountered in calling Mesa routines and explains how to construct data types in C that are equivalent to Mesa data types.

B.1 Accessing Mesa procedures

Mesa procedures must be declared in a Mesa interface for C programs to access them. chapter 3 (cc), section 3.4 (C language extensions) explains the syntax for calling Mesa procedures in C programs.

Although the C library functions are accessed from Mesa interfaces, C programs need not explicitly declare them as Mesa procedures. There is a set of Mesa interfaces for standard library functions, and the linker, when invoked through cc, assumes that library functions are in these interfaces. Both cc and the linker accept switches that expand this set of library interfaces (see chapters 3 and 4).

Programs that are ported to ViewPoint or XDE from other environments and use only CTool for a user interface may still need to explicitly call Mesa procedures to access features of the system that are inaccessable from any of the library functions. C programs that use the standard ViewPoint and XDE user interfaces (windows, menus, etc.) need to call Mesa procedures extensively.

B.2 Signals

Many procedures in Mesa interfaces raise signals to note exceptional conditions. Signals are described in the *Mesa Language Manual*, chapter 8 (Signaling and signal data types). C programs cannot catch signals. If a C program needs to call a Mesa procedure that might raise a signal, it must either:

1. Call an intermediary Mesa procedure that calls the procedure that can raise signals. The intermediary procedure catches the signals and can return a value indicating exceptional conditions if they arise. The C program calls the intermediary procedure rather than the procedure that can raise a signal. Many of the C library functions serve as such intermediary procedures to catch signals.

2. Ensure that the exceptional conditions for which signals are raised are not present before calling the Mesa procedure. Because it is not always possible to determine whether the exceptional conditions are present, this technique cannot always be used.

B.3 INLINE and MACHINE CODE procedures

INLINE procedures are Mesa procedures for which the code of the procedure is simply placed in the caller's code. They are described in the *Mesa Language Manual*, chapter 5 (Procedures, section 5.6).

C programs cannot call INLINE procedures. MACHINE CODE procedures in interfaces are implicitly INLINE procedures, and thus can not be called by C programs either.

B.4 Mesa data types

When calling Mesa procedures, it is important that the types used for the parameters and the return value are equivalent to those used by the Mesa procedure. The types must be of the same size and must be interpreted the same way in both Mesa and C. This section explains how to construct types in C that are equivalent to Mesa types.

B.4.1 Numeric types

B.4.1.1 Fixed point types

For all the fixed-point Mesa types there are simple equivalents in C. Table B-1 shows the C types equivalent to each fixed-point Mesa type.

Mesa type	C equivalent	Words
INTEGER	int or short	1
LONG INTEGER	long	2
CARDINAL	unsigned or unsigned short	1
LONG CARDINAL	unsigned long	2
NATURAL	unsigned or unsigned short	1

Table B-1: Mesa and C equivalent types

B.4.1.1 Floating point types

The Mesa type REAL is equivalent to the C type **float**. There is no Mesa type equivalent to double. Because values of type **float** are expanded to **doubles** when they are passed as parameters, there is no simple way to pass floating-point parameters to Mesa procedures. The expansion of **floats** to **doubles** can be prevented, however, by using a pointer to a **float** and casting it into a pointer to a **long**. The actual parameter is then the referent of the pointer to a **long**. Figure B-1 shows how to call a Mesa procedure that expects a REAL parameter:

```
{
    float f *fp;
    long *lp;
    .
    .
    fp = &f;
    lp = (long *)fp;
    mesaProc(...,*lp,...);
    .
    .
}
```

Figure B-1: **Calling a Mesa procedure**

B.4.2 Pointer types

B.4.2.1 Short pointers, long pointers, and relative pointers

There are three kinds of pointer types in Mesa: pointers, long pointers, and relative pointers. Mesa pointers are described in the *Mesa Language Manual*, chapter 3 (Common constructed data types), section 3.4 (The types POINTER and LONG POINTER), and chapter 6 (Other data types and storage management), section 6.3 (Base and relative pointers). A pointer in C (except for a pointer to a **char**) is equivalent to a long pointer. To call a Mesa procedure that requires a short pointer parameter, the C program should use a pointer to a variable in a frame and cast it into an **int**. The referent of such a short pointer must be a frame variable.

Relative pointers are offsets from another pointer, called a *base pointer*. The C type **unsigned** is equivalent to a relative pointer.

B.4.2.2 Byte pointers

In C, a pointer to a **char** is a special kind of pointer called a *byte pointer*. It can point to either the high byte or the low byte of a word (see section 6.2.4). A pointer to a **char** is not equivalent to any standard type in Mesa. In particular, a pointer to a **char** is not equivalent to the Mesa type LONG POINTER TO CHARACTER, which is an ordinary long pointer. If a pointer to a **char** is to be passed to any Mesa procedure that expects any kind of long pointer, it should be cast into a pointer to an **int** before being passed.

The CString Mesa interface (see section 6.2.4) defines a type that is equivalent to a C pointer to a **char**. It also contains procedures for converting between byte pointers and ordinary pointers.

B.4.3 Strings

Mesa and C represent strings differently. A string in C is a pointer to a **char** and is terminated by a null character. In Mesa it is a pointer to a record that holds the maximum length of the string, the current length of the string, and the characters. There is no special termination character. They are described fully in the *Mesa Language Manual*, chapter 6 (Other data types and storage management), section 6.1 (Strings).

The CString Mesa interface (see section 6.2.4) contains procedures for converting between C strings and Mesa strings.

B.4.3.1 Strings in ViewPoint

The interfaces for ViewPoint generally do not use Mesa strings for string parameters To support multilingual strings, ViewPoint uses its own representation of strings, defined in ViewPoint's XString interface (see the *ViewPoint Programmer's Manual*). The XString interface contains procedures for converting Mesa strings to XStrings. C programs can call procedures that have string parameters by first converting C strings to Mesa strings with the CString interface and then converting Mesa strings to ViewPoint's representation with the XString interface.

B.4.4 Records and arrays

In Mesa, records are automatically packed whenever possible. Because characters, booleans, and some subrange and enumerated types require less than one word of storage, records containing fields of these types may be packed. C structures containing the same fields as Mesa records and in the same order, therefore, are not necessarily equivalent. It may be necssary to use bit fields in the C structure declaration to construct a type that is equivalent to a Mesa record. The debugger's **type&bits** command can be used to determine the bit placement of each field in a Mesa record.

If a record in Mesa is declared to be a MONITORED RECORED, then it contains an implicit field of type MONITORLOCK (see section B.4.9). The debugger's **type&bits** command shows the placement of this hidden field.

Arrays in Mesa are not packed unless they are explicitly declared as packed arrays. There is no way of constructing types in C that are equivalent to packed arrays. If the size of the array can be determined, it is possible to declare a type that is of the same type, but the C program cannot index the array in the same way the Mesa procedure can.

B.4.5 Booleans

A Mesa parameter or variable of type **BOOLEAN** is the same size as a C **int** if it appears as a separate parameter or variable. If it is a component of a record or a packed array, it may occupy as little as one bit.

The boolean value **TRUE** is represented as 1, and the value **FALSE** is represented as 0.

B.4.6 Array descriptors

Array descriptors describe the length and location of an array. They consist of a pointer to the base of the array, and a one-word field describing the length. If a descriptor is declared to be a long descriptor, then the base pointer is a long pointer. Array descriptors are described fully in the *Mesa Language Manual*, chapter 6 (Other data types and storage management), section 6.2 (Array descriptors).

A C type equivalent to a long descriptor can be constructed with a structure containing a base field and a length field. The base field must come first. For example, for the Mesa type:

```
t: TYPE = LONG DESCRIPTOR FOR ARRAY OF INTEGER;
```

an equivalent C type is:

```
typedef struct {
        int *base;
        unsigned length};
```

B.4.7 Procedure types

A pointer to a function in C is equivalent in type to a procedure in Mesa.

B.4.8 Opaque types

An opaque type is a type that is declared in an interface, but all the information about the type is exported from an implementation module. The interface declaration may contain the size of the type; however, in that case, values of the type can be passed as parameters. Opaque types are described in the *Mesa Language Manual*, chapter 7 (Modules, programs, and configurations), section 7.6 (Exported (Opaque) types).

To hold a value of an opaque type in a variable in C, the type of the variable can simply be an array of int whose length is the size of the opaque type.

B.4.9 Special abstract types

Mesa includes several primitive types, whose values are not interpreted by Mesa programs, but which can be arguments to procedures or Mesa language constructs.

Variables in C that need to hold values of these types need only be the same size as the Mesa type. Table B-2 shows C types that can substitute for these Mesa types.

Mesa type	Possible C type	Words
PROCESS	int	1
CONDITION	long	2
SIGNAL	long	2
MONITORLOCK	int	1
UNCOUNTED ZONE	int *	2

Table B-2 **C substitutes for Mesa types**

C. ViewPoint Veneer Guide

This document describes the functionality of the ViewPoint veneer. It assumes the reader is familiar with both C and Mesa. The last section of this document contains a sample application that uses the veneer.

C.1 Purpose

The purpose of the veneer is to simplify writing ViewPoint applications in C. It does not relieve the C programmer of understanding ViewPoint programming, but it does make ViewPoint programming simpler and far less error-prone. It eliminates some of the awkwardness that results from using Mesa interfaces with features of the Mesa language that are not included in the C langauge. It also shields C programmers from needing to know all the details about the types and constants defined in ViewPoint interfaces.

By using the veneer, C programmers can also avoid repeating declarations and code in every C module. It can, therefore, be considered an extension of the C library.

C.2 Overview of veneer

The veneer consists of C header files and Mesa interfaces. Each header file and interface contain declarations and procedures relating to one ViewPoint interface. For some ViewPoint interfaces, a header file but no auxilliary Mesa interface is needed.

A header file called **Mesa.h** contains definitions of TRUE, FALSE, and NIL, and a **typedef** for array descriptors. This header file does not correspond to any ViewPoint interface.

Because ViewPoint applications also need to access Pilot and Services interfaces, the veneer covers some of these interfaces as well.

C.2.1 Contents of header files

The header files contain:

-- Typedefs for types defined in the corresponding ViewPoint interface.

-- Mesa declarations for procedures and variables in the corresponding ViewPoint interface and the auxiliary veneer interface.

-- Definitions of single- and double-word constants defined in the corresponding ViewPoint interface.

-- Macros for inline procedures defined in the corresponding ViewPoint interface.

Because interfaces depend on other interfaces, the header files include other header files in the veneer. Programmers need not be concerned with including things more than once, however, because each header file defines its name and is compiled only if its name is not already defined. For example:

```
/* File: StarWindowShell.h */
#ifndef StarWindowShell
#define StarWindowShell
#include Window.h
.
.
#endif
```

C.2.2 Contents of Mesa interfaces

The Mesa interfaces in the veneer include:

-- Procedures similar to those in the corresponding ViewPoint interface, but with the following changes:

1. They supply all or some of the default parameters to the corresponding procedure in the ViewPoint interface.

2. They replace **XString** parameters and return values with C strings.

3. They catch signals raised by calls to the corresponding ViewPoint procedure.

-- Procedures that return multi-word constants defined in the corresponding ViewPoint interface.

-- Procedures that compare multi-word values with constants defined in the corresponding ViewPoint interface.

These procedures can often reduce the amount of storage a C program needs for variables. They eliminate the need to declare multi-word records or arrays that are only used to hold constants to be passed as parameters to Mesa procedures or hold the return value of a Mesa procedure so that it can be compared with a constant.

C.3 Naming conventions

C.3.1 Files

Header files always have the same root name as the ViewPoint interface to which they correspond but have ".h" as their extension. Veneer interfaces have the same name as the ViewPoint interface to which they correspond but have a C prepended to their name. For example, the header file and interface that correspond to the ViewPoint interface **Display.mesa** (**Display.bcd**) are **Display.h** and **CDisplay.mesa** (**CDisplay.bcd**).

C.3.2 Items in header files

Items in the header files have the same name as the corresponding item in the ViewPoint interface, except that the interface name is separated by an underscore rather than by a dot. For example, the Mesa type **XString.Reader** appears as **XString_Reader** in **XString.h**.

C.3.2.1 Values of enumerated types

Because C does not permit the use of the same identifier as values in different enumerated types, the values for enumerated types have the name of the type and an underscore prepended to them. For example, the the Mesa type :

```
FormWindow.BooleanItemLabelType: TYPE = {
   string, bitmap};
```

appears in FormWindow.h as:

```
typedef enum {
   FormWindow_BooleanItemLabelType_string,
   FormWindow_BooleanItemLabelType_bitmap}
   FormWindow_BooleanItemLableType;
```

C.3.2.2 Variant record types

For a variant record in a Mesa interfaces, the equivalent C type in a header file is encoded as follows. The common fields of the variant record type are encoded the same way as for fields of non-variant record types. For each variant arm of the record, a separate type is declared whose name is the type of the entire record with an underscore and the variant adjective appended. The entire variant part of the record is a union of these separately declared types. If there is a tag field, it is encoded as a field of the common part of the record. For example, the Mesa type:

```
FormWindow.BooleantItemLabel: TYPE = RECORD[
   SELECT type: BooleanItemLabelType FROM
    string => [string: XString.ReaderBody],
    bitmap => [bitmap: Bitmap]
    ENDCASE];
```

appears in FormWindow.h as the three declarations:

```
typedef struct {
   XString__ReaderBody string}
      FormWindow__BooleanItemLabel__string;

typedef struct {
   FormWindow__Bitmap bitmap}
      FormWindow__BooleanItemLabel__bitmap;

typedef struct {
   FormWindow__BooleanItemLabelType type;
   union {
   FormWindow__BooleanItemLabel__string string;
   FormWindow__BooleanItemLabel__bitmap
   bitmap} var;
   } FormWindow__BooleanItemLabel;
```

C.3.3 Items in veneer interfaces

Procedures in the veneer interfaces that are similar to procedures in ViewPoint interfaces but with the changes noted above have the same name as the procedure in the ViewPoint procedure.

C.4 Interfaces covered by the veneer

This section documents the contents of each header file and veneer interface. Each subsection describes the header file and veneer interface that correspond to a particular ViewPoint (or Pilot or Services) interface.

C.4.1 Atom

C.4.1.1 Atom.h

Type: Atom

Constant: null

Procedures: Make, MakeAtom, GetPName, procedure in CAtom

C.4.1.2 CAtom.mesa

Make: PROCEDURE[*pName*: *CString.CString*] RETURNS[*atom*: *Atom.ATOM*];

The procedure *Make* is equivalent to the procedures *Make* and *MakeAtom* in the Atom interface except that it takes a C string as a parameter.

C.4.2 Containee

For the Containee interface, the veneer includes a header file but no Mesa interface.

C.4.2.1 Containee.h

Types: Data, DataHandle, Implementation, Ticket, PictureState

Constant: ignoreType,

Procedures: SetImplementation, GetImplementation, SetDefaultImplementation, GetDefaultImplementation, GetCachedType, SetCachedName, SetCachedType, InvalidateCache, InvalidateWholeCache, Ticket, ReturnTicket, GetCachedName.

C.4.3 Display

C.4.3.1 Display.h

Types: Handle, DstFunc, SrcFunc (from BitBlt interface), BitAddress, BitBltFlags, LineStyleObject, LineStyle, Brick

Constants: paintGrayFlags, bitFlags, replaceGrayFlags, boxFlags, xorGrayFlags, xorBoxFlags, replaceFlags, textFlags, paintFlags, xorFlags, eraseFlags, DashCnt

Procedures: Arc, Black, Gray White, Invert, Bitmap, Circle, Conic, Ellipse, Line, Point, procedures in CDisplay

C.4.3.2 CDisplay.mesa

Black: PROCEDURE[*window*: *Display.Handle*, *box*: *Window.Box*];

White: PROCEDURE[*window*: *Display.Handle*, *box*: *Window.Box*];

Gray: PROCEDURE[*window*: *Display.Handle*, *box*: *Window.Box*, *gray*: *Display.Brick*];

FiftyPercentGray: PROCEDURE[*window*: *Display.Handle*, *box*: *Window.Box*];

Bitmap:PROCEDURE[
  window: *Display.Handle*, *box*: *Window.Box*, *address*: *Display.BitAddress*,
  bitmapBitWidth: CARDINAL, *flags*: *Display.BitBltFlags*];

Invert: PROCEDURE[*window*: *Display.Handle*, *box*: *Window.Box*];

Circle: PROCEDURE[
  window: *Display.Handle*, *place*: *Window.Place*, *radius*: INTEGER];

Line: PROCEDURE[
  window: *Display.Handle*, *start*: *Window.Place*, *stop*: *Window.Place*];

The procedures *Black*, *White*, *Gray*, *Bitmap*, *Invert*, *Circle*, and *Line* call the procedures of the same name in the Display interface and supply the default bounds parameter (NIL). The procedure *Gray* also supplies the default destination function. *FiftyPercentGray* calls *Display.Gray* with*Display.fiftyPercent* as the gray *Brick*.

C.4.4 FormWindow

C.4.4.1 FormWindow.h

Types: ItemKey, ChoiceItems, ItemType, Visibility, TabType, ChangeReason, TextHintAction, BooleanItemLabelType, ChoiceItemType, ChoiceIndex, Bitmap, BooleanItemLabel, ChoiceItem

Procedures: Create, Destroy, Repaint, MakeBooleanItem, MakeChoiceItem, MakeMultipleChoiceItem, MakeCommandItem, MakeTextItem, MakeDecimalItem, MakeIntegerItem, DestroyItem, DoneLookingAtTextItemValue, SetWindowItemSize, SetBooleanItemValue, SetChoiceItemValue, SetDecimalItemValue, SetIntegerItemValue, SetTextItemValue, SetMultipleChoiceItemValue, SetTextItemValue, MakeWindowItem, GetWindowItemValue, GetBooleanItemValue, GetIntegerItemValue, GetMultipleChoiceItemValue, GetTextItemValue, LookAtTextItemValue, GetVisibility, HasAnyBeenChanged, HasBeenChanged, procedures in CFormWindow

C.4.4.2 FormWindow.mesa

Create: **PROCEDURE**[
  window: *Window.Handle*, *makeItemsProc*: *FormWindow.MakeItemsProc*];

MakeBooleanItem: **PROCEDURE**[
  window: *Window.Handle*, *myKey*: *FormWindow.ItemKey*,
  tag, *suffix*: *CString.CString*, *readOnly*:**BOOLEAN**,
  label: *FormWindow.BooleanItemLabel*, *initBoolean*:**BOOLEAN**];

MakeChoiceItem: **PROCEDURE**[
  window: *Window.Handle*, *myKey*: *FormWindow.ItemKey*,
  tag, *suffix*: *CString.CString*, *readOnly*: **BOOLEAN**,
  values: *FormWindow.ChoiceItems*, *initChoice*: *FormWindow.ChoiceIndex*];

MakeMultipleChoiceItem: **PROCEDURE**[
  window: *Window.Handle*, *myKey*: *FormWindow.ItemKey*,
  tag, *suffix*: *CString.CString*, *readOnly*: **BOOLEAN**,
  values: *FormWindow.ChoiceItems*,
  initChoice: **LONG DESCRIPTOR FOR ARRAY CARDINAL OF** *FormWindow.ChoiceIndex*];

MakeCommandItem: **PROCEDURE**[
  window: *Window.Handle*, *myKey*: *FormWindow.ItemKey*,
  tag, *suffix*: *CString.CString*, *readOnly*: **BOOLEAN**,
  commandProc: *FormWindow.CommandProc*, *commandName*: *CString.CString*];

MakeTextItem: **PROCEDURE**[
  window: *Window.Handle*, *myKey*: *FormWindow.ItemKey*,
  tag, *suffix*: *CString.CString*, *readOnly*: **BOOLEAN**,
  width: **CARDINAL**, *initString*: *CString.CString*];

MakeDecimalItem: **PROCEDURE**[
  window: *Window.Handle*, *myKey*: *FormWindow.ItemKey*,
  tag, *suffix*: *CString.CString*, *readOnly*, *signed*: **BOOLEAN**,
  width: **CARDINAL**, *initDecimal*:*XLReal.Number*];

MakeIntegerItem: **PROCEDURE**[
  window: *Window.Handle*, *myKey*: *FormWindow.ItemKey*,
  tag, *suffix*: *CString.CString*, *readOnly*, *signed*: **BOOLEAN**,
  width: **CARDINAL**, *initInteger*: **LONG INTEGER**];

MakeWindowItem: **PROCEDURE**[
  window: *Window.Handle*, *myKey*: *FormWindow.ItemKey*,
  tag: *CString.CString*,*size*: *Window.Dims*]
  RETURNS [*clientWindow*: *Window.Handle*];

SetChoiceItemValue: PROCEDURE[
  window: *Window.Handle*, *item*: *FormWindow.ItemKey*,
  newValue: *FormWindow.ChoiceIndex*];

SetMultipleChoiceItemValue: PROCEDURE[
  window: *Window.Handle*, *item*: *FormWindow.ItemKey*,
  newValues: LONG DESCRIPTOR FOR ARRAY CARDINAL OF *FormWindow.ChoiceIndex*];

GetTextItemValue:PROCEDURE[
  window: *Window.Handle*, *item*: *FormWindow.ItemKey*, *zone*: UNCOUNTED ZONE]
  RETURNS[*CString.CString*];

These procedures all call procedures of the same name in the *FormWindow* interface. They provide some of the default parameters, and their string parameters and return values are C strings rather than XStrings.

The procedures MakeWindowItem SetChoiceItemValue, and SetMultipleChoiceItemValue catch the error *FormWindow.Error*, thereby making it possible to attempt to set choice items to invalid values. The other procedures in this interface do not catch *FormWindow.Error*.

C.4.5 Heap

For the Heap interface, the veneer includes a header file but no Mesa interface.

C.4.5.1 Heap.h

Type: NWords

Constant: unlimitedSize,

Variables: systemZone, systemMDSZone, minimumNodeSize

Procedures: Create, CreateUniform, Delete, Flush, FreeNode, MakeNode

C.4.6 MenuData

For the MenuData interface, the veneer includes a header file but no Mesa interface.

C.4.6.1 MenuData.h

Types: ItemHandle, MenuObject, MenuHandle

Procedures: CreateItem, DestroyItem, DestroyMenu, ItemName, ItemData, CreateMenu

C.4.7 NSFile

C.4.7.1 NSFile.h

This header file contains some definitions from the NSName interfaces as well as from the NSFile interface.

Types: Handle, Type, SystemElement, ServiceRecord, Service, ID, Reference, Selections

Types from NSName: Organization, String Local, NameRecord

Procedures: Close, procedures from CNSFile

C.4.7.2 CNSFile.mesa

```
Close: PROCEDURE[file: NSFile.Handle];

IsNullReference: PROCEDURE[
   reference:NSFile.Reference] RETURNS [BOOLEAN];

GetNoInterpretedSelections: PROCEDURE
   RETURNS[NSFile.InterpretedSelections];

GetNoExtendedSelections: PROCEDURE
   RETURNS[NSFile.ExtendedSelections];

GetSelectionsDefaultValue: PROCEDURE
   RETURNS[defultVal:NSFile.Selections];
```

The procedure *Close* calls *NSFile.Close* and catches *NSFile.Error*.

IsNullReference compares the *NSfile.Reference* passed with the constant *NSFile.NullReference*.

The procedures *GetNoInterpretedSelections* and *GetNoExtendedSelections* return the constants *NSFile.noInterpretedSelections* and *NSFile.noExtendedSelections*, respectively.

The default value for the type *NSFile.Selections* can be acquired with the procedure *GetSelectionsDefaultValue*.

C.4.8 PropertySheet

C.4.8.1 PropertySheet.h

Types: MenuItemType, MenuItems

Constants: nullPlace, propertySheetDefaultMenu, optionSheetDefaultMenu

Procedures: Create, CreateLinked, GetFormWindows, InstallFormWindow, SwapExistingFormWindows, SwapFormWindows, procedures in CPropertySheet

C.4.8.2 CPropertySheet.mesa

Create: PROCEDURE[
  formWindowItems: *FormWindow.MakeItemsProc*, *menuItemProc*: *PropertySheet.MenuItemProc*, *size*: *Window.Dims*, *title*: *CString.CString*]
  RETURNS[*shell*: *StarWindowShell.Handle*];

CreateLinked: PROCEDURE[
  formWindowItems: *FormWindow.MakeItemsProc*, *menuItemProc*: *PropertySheet.MenuItemProc*, *size*: *Window.Dims*, *title*: *CString.CString*, *linkWindowItems*: *FormWindow MakeItemsProc*]
  RETURNS[*shell*: *StarWindowShell.Handle*];

InstallFormWindow: PROCEDURE[
  shell: *StarWindowShell.Handle*, *menuItemProc*: *PropertySheet.MenuItemProc*, *title*: *CString.CString*, *formWindow*: *Window.Handle*];

SwapExistingFormWindows: PROCEDURE[
  shell: *StarWindowShell.Handle*, *new*: *Window.Handle*, *newTitle*: *CString.CString*] RETURNS [*old*: *Window.Handle*];

SwapFormWindows: PROCEDURE[
  shell: *StarWindowShell.Handle*, *newFormWindowItems*: *FormWindow.MakeItemsProc*, *newTitle*: *CString.CString*]
  RETURNS[*old*: *Window.Handle*];

The procedures in this interface call procedures of the same name in the PropertySheet interface. They supply some of the default parameters to the procedures in the *PropertySheet* interface, and their string parameters are C strings rather than XStrings.

These procedures do not catch the error *PropertySheet.Error*

C.4.9 Selection

For the Selection interface, the veneer includes a header file but no Mesa interface.

C.4.9.1 Selection.h

Types: ManagerData, RequestorData, ValueProcs, Value, ValueHandle, Target, Difficulty, Action, CopyOrMove

Constants: maxStringLength

Procedures: Convert, ConvertNumber, Free, CopyMove, Set, ActOn, Clear, ClearOnMatch, Copy, Move, HowHard, CanYouConvert, Enumerate, Match, UniqueTarget, UniqueAction

C.4.10 SimpleTextDisplay

C.4.10.1 SimpleTextDisplay.h

Procedure: StringIntoWindow, procedure in CSimpleTextDisplay

C.4.10.2 CSimpleTextDisplay.mesa

StringIntoWindow: PROCEDURE[
  string: *CString.CString*, *window: Window.Handle*, *place*: *Window.Place*,
  maxNumberOfLines: CARDINAL]
  RETURNS [*lines*: CARDINAL, *lastLineWidth*: CARDINAL];

The procedure *StringIntoWindow* calls *SimpleTextDisplay.StringIntoWindow*, supplying some of the default parameters. The string parameter is a C string instead of an XString.

C.4.11 StarWindowShell

C.4.11.1 StarWindowShell.h

Types: Handle, ShellType, State, ArrowFlavor, ArrowScrollAction, MoreFlavor, When, ThumbFlavor, ScrollData, PopOrSwap

Constant: nullHandle

Procedures: Create, GetHost, Pop, SetHost, ShellFromChild, SleepOrDestroy, StandardClose, StandardCloseAll, StandardCloseEverything, AddPopupMenu, DestallBody, Destroy, DestroyBody, EnumerateDisplayed, EnumerateDisplayedOfType, EnumerateMyDisplayedParasites, EnumeratePopupMenus, InstallBody. Push. SetBodyWindowJustFits, SetBottomPusheeCommands, SetContainee, SetIsCloseLegalProc, SetMiddlePusheeCommands, SetName, SetNamePicture, SetPreferredDims, SetPreferredPlace, SetReadOnly, SetRegularCommands, SetTopPusheeCommands, SubtractPopupMenu, Swap, CreateBody, EnumerateBodiesInDecreasingY, EnumerateBodiesInIncreasingY, GetBody, GetAdjustProc, GetAvailableBodyWindowDims, GetBodyWindowJustFits, GetReadonly, GetSleeps, GetZone, HaveDisplayedParasite, IsBodyWindowOutOfInterior, IsCloseLegal, IsCloseLegalProcReturnsFalse, GetContainee, GetIsCloseLegalProc, GetLimitProc, GetPusheeCommands, GetRegularCommands, GetScrollData, SetScrollData, GetState, SetState, GetTransitionProc, GetType, SetAdjustProc, SetLimitProc, StandardLimitProc, VanillaArrowScroll, VanillaThumbScroll, procedures in CStarWindowShell.

C.4.11.2 CStarWindowShell.mesa

Create: PROCEDURE[
  transitionProc: *StarWindowShell.TransitionProc*, *name*: *CString.CString*]
  RETURNS [*StarWindowShell.Handle*];

CreateBody: PROCEDURE[
  sws: *StarWindowShell.Handle*, *repaintProc*: PROCEDURE[*Window.Handle*]]
  RETURNS[*Window.Handle*];

Push: PROCEDURE[*newShell*:*StarWindowShell.Handle*];

The procedures in this interface supply many of the default parameters to the procedures of the same names in the StarWindowShell interface. The *Create* procedure takes a C string for the *name* parameter, rather than an XString.

C.4.12 Window

For the Window interface, the veneer includes a header file but no Mesa interface.

C.4.12.1 Window.h

Types: Place, Dims, Box, BoxHandle, Clarity, Gravity, Object, Handle

Variable: rootWindow

Procedures: EnumerateInvalidBoxes, InvalidateBox, Validate, ValidateTree, InsertIntoTree, RemoveFromTree, EnumerateTree, GetParent, GetSibling, GetChild, SetParent, SetSibling, SetChild, EntireBox, GetBox, GetDims, IsPlaceInBox, IsDescendantOfRoot

C.4.13 XString

C.4.13.1 XString.h

Types: Context, ReaderBody, Reader, WriterBody, Writer, Character

Procedures: First, NthCharacter, Equal, FromSTRING, ReaderFromWriter, WriterBodyFromSTRING, AppendChar, AppendReader, AppendSTRING, CopyReader, CharacterLength, Empty, procedures in CXString

C.4.13.2 CXString.mesa

```
CSTRING: TYPE = CString.CString;

FromSTRING: PROCEDURE [cs: CSTRING, zone: UNCOUNTED ZONE] RETURNS [XString.ReaderBody];

AppendReader: PROCEDURE[
   to: XString.Writer, from: XString.Reader, extra: CARDINAL];

AppendSTRING: PROCEDURE[
   to: XString.Writer, from: CSTRING, extra: CARDINAL];

CStringFromReader: PROCEDURE[
   reader: XString.Reader, zone: UNCOUNTED ZONE]
   RETURNS[CString.CString];
```

The procedures in this interface do not catch errors raised by procedures in the *XString* interface.

FromSTRING produces an *XString.ReaderBody* from a C String. Storage for the characters is allocated from *zone*. Clients should call *XString.FreeReaderBytes* with the same zone when finished with the *ReaderBody*.

AppendReader calls *XString.AppendReader*, supplying a default parameter.

The procedure *AppendSTRING* converts the *from* parameter to a Mesa string and calls *XString.AppendString*, supplying a default parmaeter.

CStringFromReader produces a C string from a *XString.Reader*. The *XString.Reader* must describe a string consisting only of Ascii characters.

C.5 Sample application

The example below is a simple ViewPoint application that is written in C and uses the ViewPoint veneer. It uses standard ViewPoint user interfaces such as icons, Star window shells, and property sheets (see the chapters StarWindowShell, PropertySheet, FormWindow, and Containee in the *ViewPoint Programmer's Manual*).

This application creates its own icon, which can be copied, moved, selected, and opened like those of other applications. When the icon is opened, a Star window shell is created that has a single body window displaying some simple text. The user can choose the text to be displayed with a property sheet. A property sheet is created when you press the PROP'S key while an icon for the application is selected. The form window in the propery sheet contains a single text item, the text that is displayed in the opened windows. Initially the text displayed is "Hello world!". One global string is used for all copies of the icon, so changing this string changes the text displayed when any copy of the icon for this application is opened.

C.5.1 Sample.c

```
/* Sample.c */
/* A sample Viewpoint application in C, using the Viewpoint veneer. */

/* veneer headers */
#include "XString.h"
#include "Containee.h"
#include "Display.h"
#include "StarWindowShell.h"
#include "NSFile.h"
#include "SimpleTextDisplay.h"
#include "Window.h"
#include "Atom.h"
#include "MenuData.h"
#include "Selection.h"
```

```
#include "PropertySheet.h"
#include "FormWindow.h"
#include "Heap.h"
#include "Mesa.h"

/* Standard C library header */
#include "strings.h"

#define HELLOFILETYPE 30305L /* An NSFile.Type for icon of this application. */

#define maxMsgLength 50 /* Maximum length of text displayed in window. */

#define myKey 0 /* Used for form window item. */

typedef int word;

/* Atoms for the Containee__GenericProc */
Atom__ATOM open, props, canYouTakeSelection, takeSelection, takeSelectionCopy;

Containee__Implementation old, new;

/* Following 2 ints needed as referents for return */
/*   values of Containee__GenericProc*/
int true = TRUE;
int false = FALSE;

word ch;

char message[maxMsgLength]; /* Message to be printed in window */

mesa NSFile__Reference Prototype__Find();
mesa word SimpleTextFont__AddClientDefinedCharacter();
mesa NSFile__Handle Prototype__Create();

/* bitmap for icon */
word icon[256] = {
  0177777, 0177777, 0177777, 0177777, 0177777, 0177777, 0177777, 0177777,
  0152525, 052525, 052525, 052527, 0165252, 0125252, 0125252, 0125253,
  0152525, 052525, 052525, 052527, 0165252, 0125252, 0125252, 0125253,
  0152525, 052525, 052525, 052527, 0165252, 0125252, 0125252, 0125253,
  0152525, 052525, 052525, 052527, 0165252, 0125252, 0125252, 0125253,
  0152525, 052525, 052525, 052527, 0165252, 0125252, 0125252, 0125253,
  0152525, 052525, 052525, 052527, 0165252, 0125252, 0125252, 0125253,
  0177777, 0177777, 0177777, 0177777, 0177777, 0177777, 0177777, 0177777,
  0140000, 0, 0, 03, 0140000, 0, 0, 03,
  0140000, 0, 0, 03, 0140000, 0, 0, 03,
  0140040, 01, 0200, 03, 0140060, 015, 0103300, 03,
  0140044, 01005, 0101300, 040003, 0140032, 016401, 0100303, 0120003,
  0140073, 0601, 0100303, 0130003, 0140073, 017601, 0100303, 030003,
  0140063, 016601, 0100303, 030003, 0140063, 021601, 0100304, 070003,
  0140167, 017603, 0100703, 0170003, 0140063, 07401, 0100301, 0160003,
  0140000, 0, 0, 03, 0140000, 0, 0, 03,
  0140000, 0, 0, 03, 0140000, 040, 0, 03,
  0140000, 0160, 0, 03, 0140000, 060, 0110000, 03,
  0140000, 0660, 064000, 03, 0140000, 0260, 0166000, 03,
  0140000, 060, 0166000, 03, 0140000, 060, 0146000, 03,
  0140000, 060, 0146000, 03, 0140000, 0161, 0156000, 03,
  0140000, 060, 0146000, 03, 0140000, 0, 0, 03,
  0140000, 0, 0, 03, 0140000, 0, 0, 03,
  0140000, 0, 0, 03, 0140000, 0, 0, 03,
  0140000, 01, 0, 03, 0140000, 016, 0100000, 03,
  0140000, 016, 0140000, 03, 0140000, 015, 0140000, 03,
  0140000, 014, 0140000, 03, 0140000, 014, 0100000, 03,
  0140000, 014, 0140000, 03, 0140000, 021, 0140000, 03,
  0140000, 017, 0140000, 03, 0140000, 07, 0100000, 03,
  0140000, 0, 0, 03, 0140000, 0, 0, 03,
  0140000, 0, 0, 03, 0140000, 0, 0, 03,
  0177777, 0177777, 0177777, 0177777, 0177777, 0177777, 0177777, 0177777};

main()
```

```
{
 strncpy(message, "Hello world!", maxMsgLength); /* Set initial message. */
 MakeGenericProcAtoms();
 FindOrCreateIconFile();
 SetImplementation();
}

MakeGenericProcAtoms()
{
 open = CAtom__Make("Open");
 props = CAtom__Make("Props");
 canYouTakeSelection = CAtom__Make("CanYouTakeSelection");
 takeSelection = CAtom__Make("TakeSelection");
 takeSelectionCopy = CAtom__Make("TakeSelectionCopy");
}

FindOrCreateIconFile()
{
 XString__ReaderBody name;
 extern struct ReaderBody ReaderBodyFromCString();

 if(CNSFile__IsNullReference(Prototype__Find(HELLOFILETYPE,0,0,NIL)))
  {
  name = CXString__FromSTRING("Hello",Heap__systemZone);
  CNSFile__Close(
   Prototype__Create(
    &name, /* name */
    HELLOFILETYPE, /* type */
    0, /* version */
    0, /* subtype */
    0L, /* size */
    FALSE, /* isDirectory */
    NIL /* session */ ));
  }
}

SetImplementation()
{
 extern word SmallPictureProc();
 extern long GenericProc();
 extern void PictureProc();
 Containee__Implementation dummy; /* compiler needs this */
 extern void MakeSmallPictureChar();

 old = new = Containee__GetImplementation(HELLOFILETYPE);
 new.genericProc = GenericProc;
 new.pictureProc = PictureProc;
 MakeSmallPictureChar();
 new.smallPictureProc = SmallPictureProc;
 dummy = Containee__SetImplementation(HELLOFILETYPE,new);
}

/* GenericProc is called when the user performs an action on an instance of */
/* the icon (such as selecting it or opening it. */
long GenericProc(atom, data, changeProc, changeProcData)
Containee__DataHandle data;
Atom__ATOM atom;
int *changeProcData;
void (*changeProc)();
{
 long val;
 extern StarWindowShell__Handle MakeWindowShell(), MakePropSheet();

 if(atom = = open)
    val = (long)MakeWindowShell();
 else if(atom = = props)
    val = (long)MakePropSheet();
 else if(atom = = canYouTakeSelection)
   {
   if(Selection__CanYouConvert(Selection__Target__file))
```

```
      return ((long)&true);
    else
      return ((long)&false);
    }
  else if ((atom == takeSelection) || (atom == takeSelectionCopy))
     return ((long)&true);
  else
     return ( old.genericProc(atom, data, changeProc, changeProcData));
  if (changeProc)
   changeProc(changeProcData,data,CNSFile__GetSelectionsDefaultValue(),TRUE);
  return (val);
}

/* PictureProc paints the icon, the selected icon, and the ghost icon that */
/* appears while the icon is open. */
void PictureProc(data, window, box, old, new)
Containee__DataHandle data;
Window__Handle window;
Window__Box box;
Containee__PictureState old, new;
{
  union {
   Display__BitBltFlags asFlags;
   int asInt} flags;

  if (new == Containee__PictureState__garbage) return;
  box.dims.w = box.dims.h = 64;
  box.place.x += 4;
  box.place.y += 4;
  if (new == Containee__PictureState__ghost)
   {
   CDisplay__White(window,box);
   CSimpleTextDisplay__StringIntoWindow(
    "Hello",window,box.place.x + 10, box.place.y + 4,1);
   }
  else
   {
   flags.asInt = Display__replaceFlags;
   if (new == Containee__PictureState__highlighted)
    /* Create video-inverted icon if it is selected. */
    flags.asFlags.srcFunc = Display__SrcFunc__complement;
   CDisplay__Bitmap(window, box, icon, 0, 64, flags.asInt);
   }
}

/* SmallPictureProc is called when the small icon picture needs to */
/* be painted (such as when copying the icon). */
SmallPictureProc(data, type, normalOrReference)
Containee__DataHandle data;
NSFile__Type type;
Containee__PictureState normalOrReference;
{
  return (ch);
}

void MakeSmallPictureChar()
{
  static word bits[16] = { /* bitmap for small icon picture.*/
   000000, 000000, 014030, 014030, 014030, 014030, 014030, 017770,
   017770, 014030, 014030, 014030, 014030, 014030, 000000, 000000};
  ch = SimpleTextFont__AddClientDefinedCharacter(16,16,16,bits,0);
}

StarWindowShell__Handle MakeWindowShell()
{
  StarWindowShell__Handle shell;
  extern void WriteMsg();

  shell = CStarWindowShell__Create(NIL,"C Application");
  CStarWindowShell__CreateBody(shell, WriteMsg);
```

```
 return (shell);
}

StarWindowShell__Handle MakePropSheet()
{
 StarWindowShell__Handle pSheet;
 extern void FormProc();
 extern int MenuProc();

 pSheet = CPropertySheet__Create(
  FormProc, MenuProc, 300, 300, "Sample props");
 return(pSheet);
}

/* FormProc is called when the form window in the */
/* property sheet is being created. */
void FormProc(window, clientData)
Window__Handle window;
int *clientData;
{
 CFormWindow__MakeTextItem(
  window, myKey, "Message", NIL, FALSE, 200, message);
}

/* MenuProc is called when the user clicks Done or Cancel in the property sheet.*/
int MenuProc(shell, formWindow, menuItem, clientData)
StarWindowShell__Handle shell;
Window__Handle formWindow;
PropertySheet__MenuItemType menuItem;
int *clientData;
{
 char *msg;

 if(menuItem = = PropertySheet__MenuItemType__done)
  {
  /* Copy the value of the text item to message. */
  msg = CFormWindow__GetTextItemValue(formWindow, myKey, Heap__systemZone);
  if(msg != NIL)
   strncpy(message, msg, maxMsgLength);
  else
   message[0] = '\0';
  };
  return(TRUE); /* Destroy the property sheet */
}

/* WriteMsg is called whenever an instance of the Star window shell */
/* needs repainting. It displays the current message in the body window. */
void WriteMsg(window)
Window__Handle window;
{
 CSimpleTextDisplay__StringIntoWindow(message, window, 10, 10, 1);
}
```

D. Sample XDE C Application

The example below--**ExecEcho.c**--is an XDE application written in C. This program is intended to be run in the Executive (see the *XDE User Guide)*, not in CTool (see chapter 2). It follows the XDE program paradigm of registering with the environment call-back procedures that are invoked in response to user actions. This program registers a command with the executive.

The main procedure of this program just registers the command **Echo.~** with the executive. The call- back procedure invoked when you enter this command to the executive is Echo. This procedure simply echoes the parameters and switches you enter.

D.1 ExecEcho.c

```
/* ExecEcho.c */
/* A simple XDE application written in C. */

typedef int *handle; /* any pointer type (except for ptr. to char). */

struct TokenAndSwitches {
    handle token,switches};

#define nil (handle)0
#define normal 0

mesa void Exec__AddCommand(), Exec__PutChar(), Heap__FreeNode();
mesa struct TokenAndSwitches Exec__GetToken();
mesa handle Exec__FreeTokenString(), Heap__systemZone,    CString__CStringToLongString();
mesa Exec__DefaultUnloadProc();
mesa (*(Exec__OutputProc())) ();

/* main registers the command "Echo.~" with the Executive. */
main()
{
 handle commandName;
 extern echo();

 /* Create a Mesa string for the command name */
 commandName = CString__CStringToLongString("Echo.~", Heap__systemZone);
 Exec__AddCommand(commandName, echo, nil, Exec__DefaultUnloadProc ,nil);
 Heap__FreeNode(Heap__systemZone, commandName);
}

/* echo is an Exec.ExecProc. */
echo(h, clientData)
handle h, clientData;
{
 struct TokenAndSwitches ts;
 int (*outputProc)();

 outputProc = Exec__OutputProc(h);
 for (;;) {
  ts = Exec_GetToken(h);
  if (!ts.token && !ts.switches) break;
  if (ts.token) {
   ((void (*)())(*outputProc))(ts.token,nil);
   ts.token = Exec_FreeTokenString(ts.token);}
  if (ts.switches) {
   Exec_PutChar(h,'/');
   ((void (*)())(*outputProc))(ts.switches,nil);
   ts.switches = Exec_FreeTokenString(ts.switches);}
  Exec_PutChar(h,' ');
  }
 return(normal);
}
```