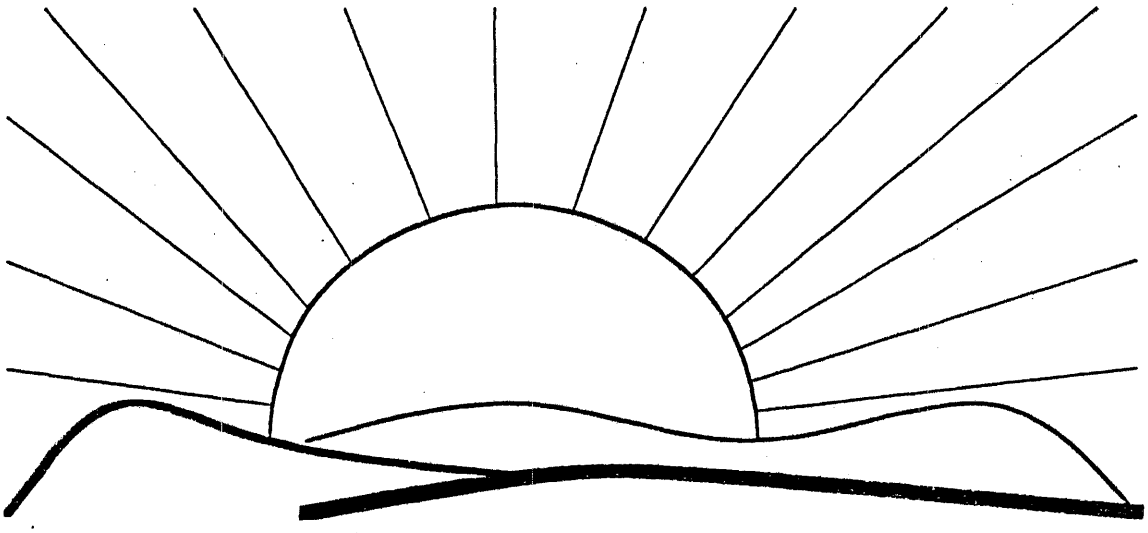
V-System 5.0 Reference Manual



Eric J. Berglund, Per Bothner, Kenneth P. Brooks, David R. Cheriton,
Stephen E. Deering, J. Craig Dunwoody, Ross S. Finlayson, David R. Kaelbling,
Keith A. Lantz, Timothy P. Mann, Robert J. Nagler, William I. Nowicki,
Paul J. Roy, Marvin M. Theimer, Willy E. Zwaenepoel

Computer Systems Laboratory
Departments of Computer Science and Electrical Engineering
Stanford University

18 October 1984

This research was supported by the Defense Advanced Research Projects Agency under contracts MDA903-80-C-0102 and N00039-83-K-0431.

Table of Contents

List of Tables

— 1 — Introduction

The V-System is a message-based distributed operating system designed primarily for high-performance workstations connected by local networks. It permits the workstation to be treated as multi-function *component* of the distributed system, rather than solely as a intelligent terminal or personal computer. Ultimately, it is intended to provide a general-purpose program execution environment similar to some degree to UNIX. The programs are intended to interact with each other, and with programs running on traditional timesharing systems, to produce an integrated distributed system.

1.1. The User Model

One of the most important functions for the workstation is to provide state-of-the-art user interface support. In particular, the workstation should function as a *front end* to all available resources, whether local to the workstation or remote. To do so, the V-System adheres to three fundamental principles:

1. The interface to application programs is (reasonably) independent of particular physical devices or intervening networks.
2. The user is allowed to perform multiple tasks simultaneously.
3. Response to user interaction is fast.

In addition, facilities are being developed to permit a consistent interaction discipline across applications.

When the user boots his workstation he may communicate with one of two entities: an *executive* or the *view manager*. The user executes commands (application programs) from within an executive, which is the equivalent of a UNIX shell or TOPS-20 EXEC. The applications may run local to the workstation or remote. They may be written with the particular workstation in mind, or run in "terminal emulation" mode. They may require I/O modalities other than traditional text, namely, graphics.

Each application may be associated with one or more separate, device-independent *virtual graphics terminals* (VGT). A VGT may be created by the user (via the view manager) or by the activity itself. Each VGT may be used to emulate either a page-mode VT-100 terminal or a 2-dimensional raster graphics terminal.

When the user wishes to initiate a new application concurrent with existing applications, he must first create a new VGT, with an associated executive. To do so, the user communicates with the view manager. The executive serves as a command interpreter from which the desired activity may be initiated. The user can create a new executive, with VGT, at any time, asynchronous to any existing activities. When a particular activity requires additional virtual terminals, it is free to create them. These VGTs will be deallocated when the activity terminates, whereas VGTs created by the user may only be deallocated by the user.

Virtual terminals are mapped to the screen when and where the user desires. Each such mapping is termed a *view*. When an activity creates a new VGT, it prompts the user to specify the default view. Thereafter, the user may create as many additional views as he wishes. To some extent, he may manipulate views of the same VGT independent of all other views of that VGT, for example, pan or zoom. All VGT management is performed via the view manager.

1.2. The System Model

The V-System adheres to the *server model*: The world consists of a collection of *resources* accessible by *clients*[1] and managed by *servers*. A server defines the abstract representation of its resource(s) and the operations on this representation. A resource may only be accessed or manipulated through its server. Because servers are constructed with well-defined interfaces, the implementation details of a resource are of concern only to its server. Note that a server frequently acts as a client when it accesses resources managed by other servers. Thus, *client* and *server* are merely roles played by a process or module.

Clients and servers may be distributed throughout the (inter)network. By default, access to resources is *network transparent*; a client may access a remote resource with the same semantics as it accesses a local resource. The result is an environment in which clients may communicate with servers without regard for the topology of the distributed system as a whole. However, we do not intend that a client cannot determine or influence the location of a particular resource, rather that a transparent mechanism is available. Moreover, we allow for clients and servers that were not written with network-transparent access in mind.

Logically, then, the V-System consists of a distributed kernel and a distributed set of server processes. A standard program environment is defined, the principal instance of which is the C program library. The C library includes runtime support for standard C and UNIX-like library functions to facilitate the porting of existing C programs.

1.2.1. The Distributed Kernel

The distributed kernel provides network-transparent interprocess communication based on *synchronous* message-passing. It consists of the collection of kernels resident on the participating machines. The host kernels may be implemented at a *base level* (as on the SUN workstation) or a *guest level* (as under VAX/UNIX). The host kernels are integrated via a low-overhead *inter-kernel* protocol that supports transparent interprocess communication between machines.

1.2.2. Servers

Servers include:

virtual graphics terminal server
: Provides all terminal management functions. One per workstation.

Internet server
: Provides ARPA Internet IP/TCP support.

pipe server
: Provides asynchronous, buffered communication facilities similar to UNIX pipes.

team server
: Provides team creation, destruction, and management. One per workstation.

exception server
: Fields process exceptions and dispatches them to registered handlers, such as debuggers. One per workstation.

storage server
: Provides disk storage.

device server(s)
: Interfaces to a specific physical device, such as the console, mouse, serial line, or disk.

local name server
: Provides locally defined character string names for (possibly) remote resources. One per workstation.

---

[1] A *client* is a program requesting access to a resource, typically on behalf of a human *user*.

1.3. The Application Model

Using the kernel well requires understanding the model of processes and messages that the kernel provides, and how they are intended to be used. Processes represent logical *activities* within the application. They are intended to be sufficiently inexpensive to allow the use of multiple processes to achieve the desired level of concurrency. In particular, multiple processes may share the same address space or *team*, to facilitate fine-grain sharing of code and data. A team must be entirely contained on a single machine.

Processes can be dynamically created and destroyed. When a process is created, it is assigned a unique *process identifier* that is used subsequently to specify that process.

Synchronous message-passing facilitates communication between processes that looks to the sender like a procedure call. That is, the sender blocks until a reply to his request is received. Greater flexibility is provided to the receiver to allow scheduling of requests. Messages are addressed to the process identifier of the recipient; there is no concept of a *mailbox* or *port* distinct from a process.

Messages are short and fixed-length. To facilitate transfer of large amounts of data, a separate data transfer facility is provided. Specifically, a process can pass, in a message, access to an area in its team space. This facility follows the procedure paradigm in being used primarily to access what are logically "call-by-reference" parameters. Synchronization between the two processes involved in the data transfer is guaranteed by virtue of the fact that the recipient will not reply to the sender (and hence awaken him) until the transfer is complete.

The kernel implements a low-level naming service that provides efficient access to server processes. A process can register its process identifier as corresponding to a particular *logical process identifier*. Clients can subsequently query the kernel as to the process identifier corresponding to a specific logical process identifier.

Process scheduling is strictly priority-based. The effective priority of a process is the sum of its *process priority*, which is defined and fixed when the process is created, and its *team priority*. Team priorities can be dynamically varied by a server process to provide time-sliced scheduling.

1.4. Outline

The remainder of this manual consists of five parts:

Part 1	**Commands:** describes the user interface and available application programs.
Part 2	**Program Environment:** defines the V-System program environment in terms of the existing C program library.
Part 3	**Servers:** defines the standard I/O protocol and presents the server interfaces.
Part 4	**Kernel:** describes the distributed kernel.
Part 5	**Appendices**

Any part of the V-System may change without notice. Therefore, this documentation should be regarded as advisory.

Part I:
Commands

— 2 —
Using the V Executive

2.1. Introduction

The V executive is the part of the V system that accepts user commands from the keyboard and causes them to be executed. It corresponds to the Unix shell or Tops-20 Exec. There are currently several versions of the executive, including two called *exec* and *vgtsexec*.[2] The two versions differ only in their handling of terminal I/O. The exec program runs a single executive, which uses the kernel console device, while the vgtsexec uses the Sun Virtual Graphics Terminal Service to provide any number of simultaneous execs. Although initially the vgtsexec provides one executive, the user can create and delete executives using the Exec Control command of the view manager, described in the next chapter.

The basic operation of the executive is to read command lines and execute commands. The first field on a command line is the command name; the rest are arguments to be passed to the command. Fields are separated by spaces. A command name can be a built-in exec command, the name of a file containing a program compiled to run under the V system, or the name of a program to be run on a server, such as Unix. The executive provides a simple search path mechanism for commands. It looks first for a V program in the current context (i.e., directory), next in the current [bin] context, and finally in the [public] context. If it still cannot find it, it will try to execute the command remotely, on the server that is providing your current context.

The executive waits for each command to exit, unless the last field on the command line is the single character **&**. In this case, the command runs in the background, while the executive continues to accept commands from the keyboard. In the vgtsexec, there is a view manager option to terminate a program running in the foreground, but the plain exec currently provides no way to do this. A program running in the background may be terminated using the *destroy* command (see chapter 4).

Other exec features are described in section 2.7.

2.2. Running the V Executive

When you come up to an idle Sun workstation, it may be in one of several states. If the screen is blank, it is probably running V, but idle. The VGTS blanks the screen on idle workstations after a few minutes of inactivity. Move the mouse slightly or press any key on the keyboard to restore the display. A previous user may have left one or more of his sessions (see below) active. The command

```
logout
```

will terminate them all and get you off to a fresh start.

If the workstation is running some other program, dead, powered down, or the like, it will be necessary to reboot it, as described in the following paragraphs.

There are several brands of Sun workstation in existence, and booting procedures vary depending on the brand. The two major kinds are those made by Cadline, which are black, and those made by Sun Microsystems (SMI), which are white. Many other computers based on the same 68000 CPU board may also run the V system, but details may be different.

---

[2] See section SERVEREXEC for a description of the *serverexec*, which is used only on dedicated server machines.

A Cadlinc workstation in a random state can be reset to the PROM monitor by typing <CTRL><SHIFT><BREAK>, pressing the reset button, or (in desperation) power-cycling the workstation. It is best to try pressing the comma key on a Cadlinc's numeric keypad before resetting it. If the V kernel is active at that point, this key instructs it to turn off the mouse, necessary for proper operation of the PROM monitor. Otherwise, you may have to power cycle the workstation or keyboard to regain control.

On the SMI workstation, hold down ERASE EOF (White Keyboard) or SET-UP (Black keyboard) and hit the "A" key. There is no reset button on SMI workstations, and the SMI mouse does not need to be shut off.

Suns that have an ordinary terminal as their console can usually be brought into the PROM monitor by hitting the terminal's BREAK key. Sometimes there is a reset button or switch attached.

It is always necessary to reset the workstation by pressing the reset button or using the Sun monitor's **k1** command before running the V kernel. On SMI Suns, the **k1** command destroys the type font used by the PROM monitor to draw characters on the display, but this is restored by the next **b** command. You can also use **k2** on SMI Suns, which repeats the power-on diagnostics and thus takes much longer than **k1**, but does not destroy the font.

To run the V executive on a Sun workstation, reset the workstation and type the command

```
n V
```

to the Sun monitor. (Use **b** instead of **n** on SMI Suns.) If your Sun has a frame buffer, this command loads the vgtsexec, or a small version of the vgtsexec if you have 256 Kbytes of memory or less. If you have no frame buffer, the **n V** command loads the plain exec. You can force the plain exec to be loaded by typing

```
n VV
```

to the monitor.

Both V and VV are special versions of the Vload program, "hardwired" to load a particular command. See chapter 12.

2.3. Contexts and the Local Name Server

A *context* in the V system is a generalization of the *directories* provided by other systems such as Unix. Each process (and thus each executive) has its own current context. A filename is normally interpreted in the current context, unless it begins with a square bracket ('['). Any filename that begins with a square bracket is sent to the local name server, which interprets the part of the name in brackets, then forwards the request off to the specified context. For example,

```
[diablo]/usr/files
```

means the name **/usr/files** is to be interpreted in the context named **[diablo]**.

The local name server predefines several context names, and others are defined by the *login* program and the executive. Users can define their own local names using the *define* and *undefine* commands. The command

```
listdir [context]
```

lists the local context names currently defined.

2.3.1. Changing the Current Context

The *cd* (change directory) command can be used to change the current context for an exec. The command format is

```
cd pathname
```

The pathname is interpreted in the (previous) current context. If the pathname is omitted, [home] is assumed (see section 2.4.1). When an exec is created, its current context is set to the current value of [home].

2.3.2. Getting Context Names

The *context* or *pwd* command will print a name for a context. It tries to find the most informative of the possible ways of naming the context. The command format is

`context` *pathname*

If the pathname is omitted, the command prints a name for the current context. This is the most common use.

2.3.3. Defining and Undefining Names

The command

`define` *name1 name2 ... nameN oldname*

or

`define` *name1 name2 ... nameN* `-lp` *logicalpid*

defines local names [name1] through [nameN] to refer to the same context as the current value of oldname or, if the "-lp" option is used, to refer to the context corresponding to the supplied logical pid. (System logical pids are defined in <Venviron.h>.) Brackets are optional on name1 through nameN, while oldname is interpreted in the current context unless surrounded by brackets. Any previous meanings for name1 through nameN are lost.

The command

`undefine` *name1 name2 ... nameN*

removes any existing local definitions of [name1] through [nameN]. Brackets are optional on these names.

2.4. Sessions

The V system uses the concept of a *session* to provide a relatively secure form of file access over a local network. To gain access to files on a host machine, it is necessary to create a session on that machine, by providing a valid user name and password to a server process running on the host. The session created has that user's file access permissions, so the existence of a V server on a machine does not add any additional complications to security or create any new security holes. Both the server process and the session it creates appear as ordinary V processes which can send and receive messages using the distributed V kernel interprocess communication protocol.

2.4.1. Login

The *login* command is used to create sessions. The command format is

`login` *hostname sessionname*

where both the host name and session name are optional. If the host name is omitted, the login program will prompt for it. If the session name is omitted, it defaults to be the same as the host name.

The login program always prompts for a user name and password. The password is not echoed when typed. An error message will be printed if the session cannot be created, or the user had an incorrect name or password.

The login program registers the user's home directory on the newly created session with the local name

server. Thus after a

```
login diablo
```

command, the name

```
[diablo]papers/naming.mss
```

refers to the file **papers/naming.mss** under the user's home directory on the host named diablo.

The login program defines the local name [home] as an alias for the user's home directory on the last session created, and the local name [bin] as an alias for the V public directory on the host providing that session, if it maintains one.

After the login command is run, the exec automatically changes its current context to the new value of [home]. Remember that this does *not* change the current context for any other process, including any of the other execs that may be running on the workstation.

2.4.2. Logout

The *logout* command is used to terminate sessions. The command format is

```
logout sessionname ...
```

where the session names are optional. If one or more names are given, each of the named sessions is terminated. If no names are given, all sessions known to the local name server are terminated. After it finishes, the logout command prints a count of the number of sessions logged out. If a session name was given and no such session was found, an error message is printed.

Logging out a session can cause the current contexts of one or more processes on the workstation, the name [home], and/or the name [bin] to become invalid. Executives try to recover from this situation, but other programs may not be able to. Do not log out a session if some program on your workstation is still using it.

2.4.3. Accessing Files Without a Session

For convenience, the V servers provide a way of accessing a certain limited set of files without first creating a session. Any of the programs kept in the standard V public directory may be run without creating a session. The name [public] is predefined by the local name server to refer to this service.

On a workstation with no sessions in existence, the names [home] and [bin] are normally both defined to equal [public]. The current directory of the first exec created when V is booted is also set to [public].

The name [public] has the special property that it is mapped to a logical process id (and well-known context id) instead of a specific server process. Each time the name is used, it is automatically mapped to a currently existing server, the one which responds first to the name server's GetPid request. Other names which are defined to equal [public], as mentioned above, get its current value when they are defined; they do not cause a GetPid on each use.

2.5. Remote Program Execution on a Session Server

If the executive fails to find an appropriate load file for a command, it will attempt to execute the command on the server providing its current context by invoking the *fexecute* program. Thus, for example, when a V server on Unix is providing the current context, all the standard Unix commands like *finger*, *make*, or *ls* are available. The output of the Unix command is printed on the standard output file.

You can also supply input to remote commands. The character echoing and line editing on this input are done on the workstation, not by the session server machine. You can type

```
control-t c [return]
```

to send an end of file to the remote command, or

```
control-↑ e [return]
```

to cause the remote command to exit. Type **control-↑** twice to send a single **control-↑** character to the remote command.

Since both the input and output are done through pipes, and input is a line at a time, many Unix programs which expect to be run on tty devices (such as *emacs*, *telnet*, *more*, etc.) do not work in this mode. Such programs can only be run by logging in to the Unix machine, perhaps using one of the V telnet programs to connect to it (see chapter 4).

The V servers do not provide execution of Unix commands without a session. If the executive tries to execute a Unix command in the [public] context, the V server returns an "Illegal request" error.

2.6. Remote Execution on a Designated V Host

A command can also be executed remotely by explicitly designating another host. This is done by specifying the process id of the team server for the host on which the command is to be run. (Syntax details are described in 2.7.6.) Remote execution of this type is transparent to the user in that I/O is still directed to the local host.

2.7. Executive Facilities for Command Specification and Modification

The executive provides various facilities for specifying and editing command lines and for redefining various aspects of command execution. The syntax and semantics of each is described below.

2.7.1. Line Editing Facilities

Command lines can be edited with Emacs-style line-editing keys. More specifically, the following editing commands are available. CTRL-x means striking the Control key and the x key simultaneously; ESC-x means striking the Escape key and then the x key.

CTRL-a	Move cursor to beginning of the command line.
CTRL-b	Move cursor back one character.
CTRL-c	Kills the Break Process, usually the command running in the current executive.
CTRL-d	Delete character under the cursor.
CTRL-e	Move cursor to the end of the command line.
CTRL-f	Move cursor forward one character.
CTRL-g	Abort the command. The line editor will pass the command line, followed by a CTRL-g, to the client program, which is responsible for detecting the CTRL-g and reacting to it.
CTRL-h	Delete the character before the cursor. Equivalent to the DEL key.
CTRL-k	Delete the command line from the cursor to the end of the line.
CTRL-t	Transpose the two characters preceding the cursor.
CTRL-u	Delete the command line up to the cursor.
CTRL-w	Delete from the cursor to the beginning of the current word.

CTRL-z	Causes an End of File indication to be sent to the application reading the line. This will terminate the Executive if no application is running.
ESC-b	Move cursor to the beginning of the current word.
ESC-d	Delete from the cursor to the end of the current word.
ESC-f	Move cursor past the end of the current word.
ESC-h	Delete from the cursor to the beginning of the current word. Same as CTRL-w.

Printing characters are normally inserted at the cursor. Commands are submitted to the executive for execution by hitting carriage return. This can be done regardless of where in the command line the cursor is.

2.7.2. Command History References

The executive also maintains a history of the last 20 command lines that the user has typed in. These command lines may be referenced by typing the character ! immediately followed by a prefix of the desired command line. Thus if the command line

```
cp /ng/ng/V/cmds/exec/exec.c /tmp/exec.c
```

was typed in, then it can be referenced by typing (for example)

```
!cp
```

If a non-unique prefix is specified then the most recent command with that prefix is taken. Another special form of reference is !!, which references the previous command line.

When a command line is referenced it is redisplayed for further line editing and verification. Thus in the previous example typing

```
!cp
```

will cause the executive to display

```
cp /ng/ng/V/cmds/exec/exec.c /usr/sun/Vboot/exec.c
```

with the cursor sitting at the end of the line. The user can then hit carriage return to reexecute the line or can edit it first to derive a new command.

The command *history* will cause the executive to list the command lines it has stored in its history record. The most recently executed command will be at the bottom of the list.

2.7.3. Command Aliases

Command names can be aliased by means of the *alias* command. Thus, for example, typing

```
alias e ved
```

will cause the command name "e" to be replaced by "ved" in subsequent command lines. Note that aliasing is done *only* for command names and not for command arguments. (Remember that the command name is the first word of a command line.)

Aliases specify a string for replacement of the alias word. Thus one can create aliases such as

```
alias test /ng/mmt/test/testcopy -d
```

Then typing something like

```
test file1 file2
```

will cause the command

```
/ng/mmt/test/testcopy -d file1 file2
```

to be submitted to the executive for execution.

A list of all defined aliases can be obtained by typing *alias* without any arguments. The command *unalias* is used to remove an alias definition. Specifying a new alias definition for a command name simply replaces the old one.

2.7.4. I/O Redirection and Pipes

I/O redirection and specification of pipes between two (or more) commands is done using the same syntax as is used by the Unix shells. Thus input can be redirected to come from a file by specifying

```
cmd < file
```

and output can be redirected to a file by specifying

```
cmd > file
```

or

```
cmd >> file
```

The latter form specifies that the output should be appended to the file whereas the first form will overwrite any data already existent in the file. Error output can be redirected by specifying `>?` or `>>?`. The forms `>&` and `>>&` redirect both standard output and standard error to the same file.

A special form of redirection is available for bidirectional files, such as the serial lines available on Suns. Specifying

```
cmd <> file
```

causes the command's input and output to be redirected to the same file. To be precise, the file is opened in FCREATE mode, and standard output is redirected to the instance thus created. Standard input is redirected to come from an instance whose id is equal to the output instance id plus 1. This matches a convention used by several V-System I/O servers. The form `<>&` also redirects standard error to the same instance as standard output.

Pipes can be set up between several commands by separating them with a | on the command line. Thus, for example, the command line

```
cmd1 | cmd2 | cmd3 > log
```

will create two pipes and redirect I/O so that the output of cmd1 will be used as the input to cmd2, the output of cmd2 will be used as the input to cmd2, and the output of cmd2 will be redirected into the file `log`.

All the special characters described above must be surrounded by spaces for the executive to recognize them. Redirection clauses must appear after all arguments to be passed to the command.

2.7.5. Concurrent Commands

Commands can be specified as being *concurrent* by including an & at the end of the command line. This causes the executive to return immediately to the user for another command rather than waiting until the current command completes. Also, while nonconcurrent (foreground) commands are terminated if their executive is deleted, concurrent (background) commands will continue even if the executive that initiated them goes away.

The & must be preceded by a space for the executive to recognize it.

2.7.6. Execution of Commands on Another Host

Commands can be designated to execute on another host by including

```
@ TeamServerPid
```

at the end of the command line. (Note: an & can be specified in addition to this.) Here TeamServerPid is the hexadecimal process id of the team server residing on the remote host where the command is to be executed.[3]

Remote execution is transparent to the user in that the I/O of the command is still directed to the local host and will be displayed in the same manner as if the command were executing locally.

The @ sign must be surrounded by spaces for the executive to recognize it. The remote execution clause, if present, must follow all arguments to the command (but may be intermixed freely with redirection clauses).

[3]Using the hex pid is a temporary measure until some form of host name service is available.

— 3 —
The View Manager

The view manager provides an interface between the user and the VGTS. The programmer's interface to the VGTS is described in the *V-System Programming Environment Manual*, and the internal structure of the VGTS is described in the *V-System Servers Manual*. The program creates SDFs and objects within them, and associates these objects with Virtual Graphics Terminals (VGT's). Through the view manager, the user maps these VGT's onto physical screens, and manipulates the resulting views. The VGTS multiplexes both the output devices (the screen) and the input devices (keyboard and mouse) among all the programs that use them. The VGTS is no longer physically integrated with the executive, although the view manager does provide an interface to the exec server. The line-editing functions described in section 2.7.1 are provided by the VGTS, like any terminal agent.

3.1. VGTS Conventions

Virtual terminals appear as white overlapping rectangles on the screen, with a black border and a label near the top edge. There is at most one virtual terminal (usually a pad, or text-only virtual terminal) that is getting input from the keyboard, along with possibly other virtual terminals getting input from the mouse. This is indicated by a flashing black box for a cursor in the text virtual terminal, and a black label on all the views that are accepting mouse input. Note that all virtual terminals are always *active* in the sense that any application may run or change the display in any virtual terminal at any time independent of this selection; it only applies to input.

Clicking the left or middle button of the mouse in a non-selected virtual terminal will cause it to be selected for input. Views of selected pads will be brought to the top. The input pad can be changed by using control up-arrow (octal 036) followed by a single command character. The only command characters interpreted by the VGTS are 1-9 to select the given pad for input.

There are a few conventions for using the mouse with the VGTS. A "Click" consists of pressing any number of buttons down and releasing them at a certain point on the screen. While the buttons are down there may be some kind of feedback, like an object which follows the cursor. The click is usually only acted upon when all the buttons are released, so if you decide you have made a mistake after pressing the buttons you can slide the mouse to some harmless position before releasing the buttons. Holding all three buttons down is also interpreted as a universal abort by most programs and the view manager. The click event is sent to the program associated with the view in which the event occurred (through its VGT).

When a V-System program requests the creation of a pad, the cursor will change to the word "Pad". At this point, hold down any button, and an outline of the view which will be created will be tracked on the screen. Position the view where desired, and let go of the button.

3.2. View Manager Menus

The view manager menus can always be invoked by moving the cursor to the grey background area or any virtual terminal not getting input (except in the banner area) and pressing the Right button. The following commands are available from the view manager menus:

Create View — Creates another view of an existing VGT. Move the cursor to the desired position of any one of the four corners for the new viewport. Hold any button down, and move the cursor

to the diagonally opposite corner. An outline of the new view will follow the cursor as it moves with the button down. Let the button up, and then point at the VGT that you would like to see with the left or middle buttons, or hit the right button and select the VGT from the menu. Normally only used with graphics VGTs.

Delete View — Click one view which is removed from the screen. Warning: if you delete the last view of a VGT, it does not destroy the VGT or the process associated with it. You can still create views of the VGT by using the right button menu in the Create View command.

Move Viewport — Press any button to select a viewport to move. While the button is being held down, the outline of the viewport will move, following the cursor. Lift up the button at the desired position. None of the other view parameters are changed. A shortcut to this function is obtained by pressing the middle button while pointing to the banner of the desired viewport. The viewport outline will follow the cursor until the middle button is released.

Make Top — Brings the view to the top, potentially obscuring other views. A shortcut to this function is obtained by pressing the left button while pointing to the banner of the desired viewport.

Make Bottom — Brings Brings the view to the bottom, potentially making visible other views. A shortcut to this function is obtained by pressing the right button while pointing to the banner of the desired viewport.

Exec Control — Selects a submenu to create another executive, destroy an executive (and the teams running in it), kill a program, or control paged output mode. When you are creating an executive, the outline of the new pad will follow the cursor as you hold the button down. Lift the button up at the desired position, or press all three buttons to abort. A shortcut to the Exec Control menu is obtained by pressing both the middle and right buttons while the cursor points to the gray background or the display area of a viewport not requesting mouse information.

Graphics Commands — Selects another menu of commands that are usually only applied to graphics views. These are described below:

Center Window — Click the position that you want to become the center of the viewport. Does not change the position of the viewport on the screen, just the object within the view. Doing this to pads is almost always a mistake.

Move Edges — Push any button down next to an edge or corner, move that edge or corner to the new position, and let the button up. The edge outline should follow the cursor as long as you hold the button down. Does not move the object being viewed relative to the screen.

Move Edges + Object — Similar to the previous command, but this one drags the underlying object around with the moved edge or corner, while the previous command keeps it stationary with respect to the screen.

Zoom — Invokes a Zoom mode, indicated by a change in the cursor to the word "Zoom". You can get out of this mode in two different ways: First, clicking the left or middle buttons when the cursor is inside a view of a pad returns from the view manager and selects that pad for input. As a side effect that view is also brought to the top. Secondly, you can click the right mouse button. The cursor should change back to the normal arrow.

The left and middle buttons in Zoom mode zoom out and in respectively. That is, the left button makes the object look smaller, and the middle button makes it look larger. You can remember this because the "outer" (left) button zooms out, and the "inner" (middle) button zooms in. A shortcut to this mode is available by clicking the middle and left

buttons at the same time while the cursor points to the gray background or the display area of a viewport not selected for input.

Expansion Depth — Click to determine the view, then select the new expansion depth from the menu. Symbols will not be expanded more than this many levels into the hierarchy. Instead they will be drawn as outlines with text for their names if there is room. The default expansion depth is infinity, so all levels will be normally expanded.

Redraw — Redraws all the views on the screen; necessary only during debugging.

Toggle Grid — Click once to turn the grid on if it is off, or off it is on in the view you select. The grid dots are every 16 screen pixels, and always line up with the origin.

Debug — Enables lots of extra printouts, for maintenance use only. This command asks for confirmation, to discourage its accidental invocation. It will not turn on debugging unless the response begins with the letter **y**.

A shortcut to the Graphics Commands menu is obtained by pressing both the left and right buttons while the cursor points to the gray background or the display area of a viewport which is not requesting mouse information.

3.3. Paged Output Mode

When paged output mode is on, the terminal agent stops writing to a pad when the pad fills up with output. The terminal agent then displays the message "Type <space> for next page" and waits for the user to issue a command which unblocks the pad. This section describes the available commands.

Most commands are optionally preceded by an integer argument ***k***. Defaults are in brackets. Star(*) indicates that the argument becomes the new default.

<space> — Display the next ***k*** lines [current page size]

z, Z — Display the next ***k*** lines [current page size]*

<CR>, <LF> — Display the next ***k*** lines [1]

q, Q — Throw away all output until the next time input is sent to the application program.

s — Skip forward ***k*** lines [1]

S — Skip forward to the last line

f — Skip forward ***k*** pages [1]

F — Skip forward to the last page

<backspace>, DEL — Erase the last character of the numeric argument

. — Repeat the previous command

If the user types a character which is not a valid command, the character is treated as a normal input character. If line editing mode is on, the CTRL-c and CTRL-z commands (see section 2.7.1) have their usual effect here.

3.4. Mouse Escape Sequences

Inside a pad, when connected to some host through a telnet program, the buttons have the following effect:

Left Button — Sends the sequence escape M *<xXy>* which positions the Emacs cursor at the position of the click.

Middle Button — Selects the clicked pad for input, and brings the view selected to the top.

Right Button — View manager menu, described in the previous section.

Left+Middle Buttons
Sends the sequence escape M *<xXy>* null which sets the Emacs mark to the clicked position.

Left+Right Buttons
Sends the sequence escape M *<xXy>* ↑W which deletes in Emacs from the mark to the clicked position.

Middle+Right Buttons
Sends the sequence escape M *<xXy>* ↑Y which inserts the kill buffer at the clicked position in Emacs.

The above escape sequences are enabled by turning on the **ReportEscSeq** bit in the cooking mode of the virtual terminal. See the VGTS chapter of the library manual for more details.

3.5. Mouse Emulation via the Keyboard

For the benefit of hardware configurations without a working mouse, the VGTS can interpret certain keyboard escape sequences as mouse input. The VGTS will only intercept these escape sequences if they are sent as a rapid burst of characters, as is the case when they are sent by pressing a function key. If the escape sequences are typed manually, the VGTS will detect the space between characters and pass them through in the normal fashion.

The following is a list of the escape sequences used and the function keys with which they are normally associated on an ANSI (VT100-style) keyboard.

ESC [A (ANSI Down Arrow)
Move the mouse cursor down.

ESC [B (ANSI Up Arrow)
Move the mouse cursor up.

ESC [C (ANSI Right Arrow)
Move the mouse cursor to the right.

ESC [D (ANSI Left Arrow)
Move the mouse cursor to the left.

ESC O P (ANSI PF1)
Toggle the value of the left mouse button. The new value of the left mouse button is displayed in the view manager window.

ESC O Q (ANSI PF2)
Toggle the value of the middle mouse button. The new value of the middle mouse button is displayed in the view manager window.

ESC O R (ANSI PF3)

Toggle the value of the right mouse button. The new value of the right mouse button is displayed in the view manager window.

ESC O S (ANSI PF4)

Toggle mouse emulation mode. When mouse emulation mode is OFF, all escape sequences except for ESC O S (ANSI PF4) will be passed through as normal, allowing the associated function keys to perform application-defined functions. The new state of mouse emulation mode is displayed in the view manager window.

When the VGTS receives input from a "real" mouse, mouse emulation is permanently disabled. If your mouse fails, you must use the "newterm" command to create a new VGTS in order to use mouse emulation.

— 4 —
Command Summary

4.1. Workstation Commands

The following briefly summarizes the currently available commands for V.

amaze — A multi-person distributed game. Does not (yet) run under the vgts. See chapter 10.

biopsy — Prints information about all the processes on the workstation, sorted by team. Several options are recognized. The -l option also includes the filename from which each team was loaded. (This generally makes the output longer than one screenful.) The -t option followed by a pid or the *suffix* of a team's filename will cause information to be printed only about the team associated with the pid or filename. More than one pid or filename can be specified - information for each will be printed. To obtain detailed information about one or more processes, invoke biopsy with just the pid(s) of the relevant process(es).

bits — A program for manipulating bitmaps and fonts. See chapter 9, and the online `help` file.

boise — Prints files on the Boise laser printer.

Several switches are allowed, preceding the filenames:

-r	Print rotated, that is, in landscape (horizontal) mode.
-n *name*	Use *name* to label the output. If this option is not given, the "For user:" field is left blank.
-b *banner*	Use *banner* in the "File:" field instead of the filename.
-h *hostname*	Host name to use instead of "V-System".
-m *mode*	Print mode. Possible modes are

0	Line printer (the default). For printing ordinary text files.
1	DVI. For printing TeX output.
2	Press. Not implemented.
3	HP2680a. For files in HP2680a "spool file" format.

-w	File is in the Sail ("WAITS") character set instead of standard ASCII. (Line printer mode only).

If no filenames are given, *boise* reads its standard input.

cd — Change directory: change the current context.

checkers — Lets you play a game of checkers against the workstation. This is also a good demonstration of the vgts' graphics capabilities. See chapter 10.

clear — Clears the pad.

context	Prints an expanded name for the current context, or if a context name is given, for that context. Also known as *pwd* (print working directory), by analogy with Unix.
cp	Copy the first file to the second file.
copydir	Invoked as: **copydir** *fromdir todir* copies the *fromdir* directory subtree to *todir*. *todir* must previously exist. New files are created in a default mode, while the mode of existing files (being updated) is left alone.
dale	Distributed version of yet Another Layout Editor is a VLSI layout editor that provides graphics editing of SILT chip descriptions. Documented in a Stanford CSL Technical Report.
date	Prints the date as maintained by the local workstation kernel, and as maintained by the session host. The kernel-maintained time on a workstation is set from a time server when the exec is started. The command **date -s** sets the local time to the network time.
define	Defines one or more local names for a context. The first argument(s) are the new names to be defined. The last argument is a context name, specifying the value to be given to the new names.
debug	The V debugger. See chapter 6.
destroy	Takes the name of a team (or any suffix) as an argument, and sends a message to the team server asking it to destroy that team. If the argument begins with the characters **0x**, it is taken as a process id, and that process is destroyed. This is useful for killing processes run in the background.
dopar	A program similar to **doseq**, except that it allows the executions of its command argument take place in parallel on different hosts. The program prompts for the names of hosts on which to execute the command (for each context). If "*" is entered, then the service server will select an 'arbitrary' available host.
doseq	This program takes two string arguments: a list of context names, and a command to execute. The command is executed in each context in turn. **doseq** is often useful in makefiles. **domake** is a synonym for **doseq**.
draw	An interactive drawing program that runs under the VGTS. See chapter 8.
echo	Echos its arguments.
fexecute	Force a command to be executed on the server providing the current context, as described in section 2.5.
help	A program which prints out a little bit of information about the V system. **help ?** prints a list of topics on which help is available.
internetserver	A version of the Internet Server, as described in the V-System Servers manual.
iphost	If given a single host name as an argument, *iphost* lists all IP addresses corresponding to that host. If no argument is given, the IP address of the local workstation is printed.
iptelnet	A multi-window IP/TCP telnet program using the VGTS. This program has a copy of the VGTS linked into it, so it is only useful under the bare kernel or the STS. Use *iptn* under the VGTS.
iptn	IP/TCP-based telnet implementation. It can run under the STS, or in a VGTS window. A

destination host name or address may be given as a command argument; if none is given, *iptn* prompts for one. A host name is a string of non-white-space characters starting with a non-numeric character. A host address is a string of the form a.b.c.d, where a,b,c and d are decimal integers. Both names and addresses may be followed by a dot and a decimal port number (with no intervening spaces).

While connected to a remote host, *iptn* recognizes a set of commands prefixed by ctrl-↑. Ctrl-↑ ? prints a list of all such commands.

After disconnecting from a remote host, *iptn* prompts for another host. To exit *iptn*, enter ctrl-c or ctrl-z in response to the prompt.

If there is no internet server on your workstation when *iptn* is loaded, it runs one in the background. The -l flag inhibits loading a local server, instead looking for a public internet server running on another V host.

The -d flag enables debug mode. In this mode, all transmitted and received telnet protocol commands are printed, and all received non-printable characters are printed in an escaped notation. Debug mode can be toggled on and off by typing ctrl-↑ d while connected to a remote host.

listdir — Lists the names defined in a context, and prints some information about each. If no argument is given, the current context is assumed.

login — Command to start a session on a computer running a V server.

logout — Command that terminates sessions.

newterm — Change terminal agents. Takes one argument, the filename of a new terminal agent to take the place of the existing one. All executives running under the old terminal agent are destroyed; the new one will presumably provide means of creating a new one. For example, **newterm sts** replaces the vgts with the Simple Terminal Server, which does no graphics but simply presents the workstation as an ascii terminal. If no argument is given, it defaults to "vgts". ***Warning:*** If the named program is not in fact a terminal agent, you will probably lose control of your workstation.

pagemode — Enable or disable paged output mode in the current executive. Takes one argument, which may have one of two values: "on" or "off". When paged output mode is on, the terminal agent stops writing to a pad when the pad fills up with output. The terminal agent then displays the message "Type <space> for next page" and waits for the user to issue a command which unblocks the pad. The user interface for paged output mode is described in section 3.3.

query — Prints out the result of performing various 'query' operations. In particular, **`query kernel`** prints the result of the **`QueryKernel`** operation, **`query config`** prints the contents of the workstation's configuration file, and **`query ethernet`** prints the result of querying the "ethernet" device. **`query ?`** lists the possible options.

rm — Takes one or more filenames as arguments, and removes each file.

serial — This program provides a full-duplex conversation between its standard input and output, and a device connected to one of the serial ports of the workstation. The argument is a device name, specifying the line to be opened. It defaults to *[device]serial0* if omitted. Names of the form *[device]serialn* (with *n* a single digit) can be abbreviated by giving only the digit. If the serial line is connected to a modem or a terminal port on another computer, this program allows the Sun to act as a terminal. The flag **`-b bitrate`** can be used to specify the bit rate (baud rate) of the connection; it defaults to 9600 bps.

show — Displays a **.dvi** file or a **.press** file. It is driven from a menu in the invoking pad: by selecting the appropriate field, you can move around from page to page, with either relative movement, absolute page number or the TeX generated **\count0** numbers. You can invoke it with **show** *filename*, or you can set the filename in the menu. You can "scroll" a page by pressing a mouse button inside the view, moving the mouse and releasing the button. It handles the TeX generated **dvi** files pretty well, though magnification is ignored and some fonts are missing. Biggest problems: it only handles a small subset of **press** format, there are no good **scribe** fonts for it, and it is a bit slow.

telnet — A multi-window PUP-based telnet program using the VGTS. This program has a copy of the VGTS linked into it, so it is only useful under the bare kernel or the STS. Use *tn* under the VGTS.

testexcept — Simple interactive program for testing the exception server.

timekernel — Program to measure the time for Send/Receive/Reply kernel primitives.

tn — PUP user telnet program. It can be run under the STS or in a VGTS window. *tn* takes an optional argument specifying the host to connect to. While running, the following keyboard commands are available:

ctrl-↑ c	Close the connection.
ctrl-↑ d	Close the connection and delete the VGT, if created by tn. (Only available when running with the VGTS.)
ctrl-↑ e	Close all connections and exit from tn.
ctrl-↑ o	Create a new VGT and open another connection in it. (Only available when running with the VGTS.)
ctrl-↑ b	Create a big (48-line) VGT and open another connection in it. (Only available when running with the VGTS.)
ctrl-↑ ctrl-↑	Transmit a ctrl-↑ character.

tn is capable either of using the raw Ethernet device on the workstation, or going through a local internet server. If there is a local internet server, tn must use it, since the kernel Ethernet device is single-user. Even if there is no local internet server when tn is loaded, to be compatible with *iptn*, tn will load a local internet server and work through it if there is no public internet server elsewhere on the network that could be used by iptn. To force tn to use the raw Ethernet device if it can, invoke it with the command line **tn raw** *hostname*.

Only one copy of tn may be run on a workstation at one time.

type — Type out one or more files on the terminal. Types a page-full and then stops and waits for input. Pressing [SPACE] brings up another page, while [RETURN] brings up another line. Hit **q** or **↑C** to quit.

undefine — Removes the definitions of one or more local context names.

ved — A text editor, similar to Emacs, which runs under the vgts. Described in chapter 7.

4.2. Commands on Session Hosts

There are also several useful commands that can be invoked on session hosts (usually a Vax/Unix system). Use these commands once you have logged into a machine through a telnet connection. Most of these commands also have versions that run locally on the workstation under the vgts, and the Unix versions can also be run remotely under the vgts, using the exec's remote execution feature (section 2.5).

dale	A version of the Yale layout editor that runs under the vgts.
photo	Reads and displays a ".sun" format raster file.
siledit	A program which edits .SIL format files. SIL, a Simple Interactive Layout program, is a graphics editor for logic designs and illustrations.
silpress	A program which takes a .sil format file and produces a .press format file that can be printed on the Dover.

— 5 —
Executive Control Commands

The following commands give the user access to the execserver functions.

checkexecs	Kill off any exec whose standard input server or standard output server has died.
delexec	Delete an executive, specified by its exec id. The first exec created when the workstation is booted will always have an id of 0.
do	Create an exec with a named file as its input. This file should contain a list of V commands, exactly as you would type them, one to a line.
killprog	Kill the program, if any, running in the specified executive.
queryexec	Find out the status of the specified executive. Useful mainly for system testing. See **QueryExec** in the Program Environment manual.
startexec	Create an exec in a new pad. The new exec will have the same context as the exec from which startexec was invoked, NOT the [home] context. For most purposes the view manager's **Create Executive** commands are to be preferred over this one, as the view manager will not work on an executive created by **startexec**. **startexec** prints out the exec id and process id of the new exec.

— 6 —
The V Debugger

6.1. Synopsis

```
debug [-d] progName progArg1 progArg2 ...
```

6.2. Description

6.2.1. Invoking the Debugger With a Program

Debug is an assembler-level symbolic debugger for .r files created by the 68000 linker (ld68). It can be called as a command to the V exec and takes the following arguments:

-d	If the VGTS is available, then this argument causes an IO pad to be created for the debugger which is separate from the one used by the program to be debugged. This option is a necessity for programs which read keyboard input via separate reader processes since these may interfere with the debugger's keyboard input requests.
progName	The name of the program to be debugged.
progArg*n*	The nth argument of the program to be debugged.

Thus, to debug a program which is normally invoked by:

```
progName arg1 arg2
```

one types

```
debug progName arg1 arg2
```

If a separate IO pad is desired (*for Vgts resident environments only*) then one would type

```
debug -d progName arg1 arg2
```

6.2.2. Postmortem Debugger

The debugger can also be used as a "postmortem" debugger. The V execs (both the Vgts-based one and the non-Vgts-based one) have been structured so that if an exception occurs in the program currently being run, the debugger is automatically loaded and given control.

6.2.3. Common Usage

A program invoked with the debugger will start out at the debugger's command level. Breakpoints may be set and the program code and global variables may be examined. The program can then be started using the commands described below.

A frequent "postmortem" use of the debugger is to obtain a stack trace to find out where a program incurred an exception and then quit. This is done by typing **s** after having been transferred into the postmortem debugger to get a stack trace, and **q** to quit:

```
! prog arg1 arg2

Bus error on read from address     f in process     2ed0024
Instruction        Program Counter      Status Register
    1010               10172                10
B0> 10174      4880                      main+2C          extw d0
.s
    stack trace
.q
!
```

6.3. Commands

The debugger begins by displaying the line of code at which execution has paused, and then gives a period ('.') as a prompt. The user can then enter commands using the keyboard. Most commands are terminated with a carriage return; exceptions will be noted in the command descriptions. The only characters that may be used to erase previously typed input are backspace (\b) and delete (DEL). The entire line may be erased by typing CTRL-u. When eliminating optional arguments in commands which take more than one argument, be sure to include the correct number of commas for the command. In this way the debugger can determine which argument is to be assumed.

6.3.1. Definitions

Within the command descriptions below, an *expression* is some combination of numeric constants, register symbols, globally visible symbols from the program being debugged, and the operators +, -, and |, representing 2's complement addition, subtraction, and bitwise inclusive or, respectively. Blanks are not significant except in strings. All operations are carried out using 32-bit arithmetic and evaluated strictly left to right.

Register symbols are symbols which represent the various processor registers. The following symbols are recognized:

%d0 - %d7	Data registers 0 - 7.
%a0 - %a7	Address registers 0 - 7.
%fp	Frame pointer (synonym for %a6).
%sp	Stack pointer (synonym for %a7).
%pc	Program counter.
%sr	Status register.

In all commands except the replace-register (**rr**) command a register symbol represents the contents of the specified register. In the replace-register command it represents the address of the register specified.

Globally visible program symbols are names of program routines or global program variables.

The single character '.' (dot) is treated as a symbol representing the last memory location examined. Its value upon entrance to the command level of the debugger is set to the current value of the program counter.

6.3.2. Execution Control Commands

expression, *number*, **b**

Set breakpoint *number* (in the range 2-15 decimal) at *expression*. *expression* must be a legal

instruction address. If *number* is omitted the first unused breakpoint number is used. If *expression* is 0 the named breakpoint is cleared, or if *number* is omitted then all breakpoints are cleared. If *expression* is omitted all breakpoints are printed. Note: if *expression* is omitted then *number* must also be omitted or must be preceded by a comma in order distinguish it from being interpreted as the *expression* argument.

expression, **g** Go. Start or resume execution at *expression*. If *expression* is omitted, then start execution at the current pc value.

expression, **gb** Go past breakpoint. Like *go* with no argument, except that if we are presently stopped at a breakpoint, then *expression* counts the number of times to pass this breakpoint before breaking. If *expression* is omitted, then 1 is assumed.

expression, **x** Execute the next *expression* instructions, starting from the current pc and printing out all executed instructions. If *expression* is omitted, then 1 is assumed. Note: traps are executed as single instructions; i.e. the instructions executed in a trap routine are not displayed or counted.

expression, **y** Same as x except that subroutine calls are executed as single instructions; i.e. do not descend into the called subroutine.

xx xx is a synonym for y

; A synonym for x, except that each instruction executed is displayed on the same line as the command, providing a more compact display. No carriage return is needed to terminate this command; the semi-colon triggers execution. The typeout mode referred to in the command descriptions is described under the t command.

sp Toggle the flag that determines whether the whole team stops at an exception or just the process that incurred the exception. The debugger's default behavior is to stop the whole team when an exception occurs, not allowing any of its processes to proceed until one of the above Execution Commands restarts the team. (Of course, at that point ANY of the processes could resume execution -- i.e., single-stepping one process could allow another to execute indefinitely.) If this command is typed, an exception in any one process will not halt any of the other processes on the team. Typed again, the debugger goes back to its original behavior.

q Quit. Exits the debugger and kills both the debugger and the program being debugged.

6.3.3. Display Commands

The following commands are executed immediately without waiting for a carriage-return (CR) to be typed, and their output overwrites the current line. (This provides a more compact display format.)

expression/

expression\ Display the contents of *expression*. The typeout mode used is determined from the program symbol table and the current typeout mode. The value of dot is set to *expression*.

/

**** Display the contents of dot after having respectively incremented (/) or decremented (\) it. The typeout mode used is determined from the program symbol table and the current typeout mode.

@

expression@ Display the contents of the memory locations *pointed* to by the value of dot or *expression*,

respectively. The typeout mode used is determined from the program symbol table and the current typeout mode. The value of dot is set to the address of the memory location just displayed. Note that %pc will yield the contents of the memory location pointed to by the pc register (i.e. the current instruction) and that %pc@ will attempt to place an additional indirection on that memory location. %pc@ is almost always an invalid reference.

=

expression= Display the value of dot or *expression*, respectively.

The following display commands are executed when a carriage-return is typed.

d Display the contents of all the registers.

s Print out a stack trace describing the chain of subroutine calls and their parameters. Warning: the debugger's stack trace examines the values of parameters as they currently exist on the stack, not as they were when the routine was called. Routines which change the values of their parameters will similarly affect the stack trace output.

expression, *numlines*, **n**
Display the next *numlines* memory locations, starting at *expression*. If *expression* is omitted, then display starts at dot. If *numlines* is omitted, then 24 lines are displayed.

expression, *numlines*, **p**
Display the previous *numlines* memory locations, starting at *expression*. If *expression* is omitted, then display starts at dot. If *numlines* is omitted, then 24 lines are displayed.

type, **t** Temporarily set typeout mode to *type* where *type* is one of:

'c'	type out bytes as ascii characters.
'h'	type out bytes in current output radix.
'w'	type out words (2-bytes) in current output radix.
'l'	type out longs (4-bytes) in current output radix.
's', *strl.engih*	type out strings. Set the maximum length of strings to be *strl.engih*. The maximum string length determines how far the debugger is willing to look for the end of a string, which is assumed to be a '\0' byte. For programming languages such as Pascal which don't terminate their strings with a '\0' byte this limit is important to prevent endless string searches. The string maximum length is sticky (i.e. it need be set to the desired value only once). The default value is 80.
'i'	type out as symbolic assembler instructions.

Note that the type characters must be surrounded by single quotes. If no argument is supplied then the default typeout mode is used. This mode tries to set the typeout mode based on the type of symbol(s) being displayed and uses 'l' format when the mode is not obvious. The new typeout mode stays in effect until execution is resumed with one of the Execution Control Commands.

type, **tt** Permanently set typeout mode to *type*. The typeout mode is set to the default typeout mode if *type* is omitted.

base, **ir** Set the input radix to *base*. If *base* is illegal (less than 2 or greater than 25, decimal) or omitted, then hexadecimal is assumed. (This is the default radix.)

base, **or** Set the output radix to *base*. If *base* is illegal (less than 2 or greater than 25, decimal) or omitted, then hexadecimal is assumed. (This is the default radix.)

offset, **of** Set the maximum offset from a symbol to *offset*. If *offset* is illegal (less than 1) or omitted, then hexadecimal 1000 is assumed. (This is the default offset.) This command is useful when examining areas of the team, such as the stack, which are more accurately labeled by hex addresses than by symbol+offset notation.

charcount, **sl** Set the maximum number of characters in a symbol which will be displayed to *charcount*. If *charcount* is illegal (less than 1 or greater than 128) or omitted, then 16 is assumed.

6.3.4. Replacement and Search Commands

expression1, *expression2*, *type*, **r**
Replace the contents of the memory location specified by *expression1* with *expression2*. *expression2* is interpreted to have type *type*. Note: It is not currently possible to replace strings with this command, and instructions should be specified in 16-bit quantities and replaced with type 'i'. If *expression2* is omitted, then the value 0 is used.

register, *expression*, **rr**
Replace the contents of the specified register with *expression*. If *expression* is omitted, then the value 0 is used. *expression* is interpreted to be a 32-bit quantity.

expression, *lowlimit*, *highlimit*, *type*, **f**
Search for (find) *pattern* in the range *lowlimit* (inclusive) to *highlimit* (exclusive). *expression* is interpreted as an object of type *type*. Objects are assumed to be aligned on word (2-byte) boundaries except for 1-byte types and strings, which are aligned on byte boundaries. A mask (set with the mask command) determines how much of the *expression* is significant in the search, unless *expression* is a string constant. The first three arguments to the search command are sticky; thus if any of them are omitted then their previously specified value is used. f is the only debugger command which allows the specification of a string constant as *expression*. A string constant is delimited by the character " on either side; to use " in the string itself, precede it with a \. An example of a string is: "This is a string with \" in it". The typeout limit of strings determines how much of the string is significant in the search, not the search mask.

expression, **m** Set the search mask to *expression*. If *expression* is omitted then 0 is used. -1,m forces a complete match, f,m (that's hex f) checks only the low order 4 bits, 0,m will make the search pattern match anything.

6.3.5. Help Commands

h Print a brief description of each of the debugger's commands.

w Print a set of internal debugger statistics. This was implemented for the convenience of the designers and may change frequently in content and format. It replaces the obsolete qq which, due to the debugger's unsophisticated command parsing will behave exactly as does **q**.

6.4. Bugs

The debugger as it is currently implemented has some "features" one must be aware of.

Currently, each instance of the debugger can debug only one team at a time. Programs that create and load new teams will cause problems because the debugger assumes that it is always dealing with the same program

image.

If a breakpoint is encountered anywhere between the receipt of a message and a later attempt to call RereadMessage() on that message then the breakpoint exception will destroy the message value, yielding garbage in the subsequent call to RereadMessage().

The debugger assumes that any trace trap exceptions have been caused by its own single-stepping mechanism. Though it will recognize the first one, and print an error message, subsequent trap exceptions can cause intolerable behavior.

The stackdump routines depend upon knowing the string names of the kernel routines to produce correct stack traces which include those routines. Right now, this list is being kept up to date by hand.

Putting breakpoints in code which is shared by two or more processes can be hazardous to your mental health.

— 7 —
Ved: A Text Editor

Ved is the V system text editor. It runs entirely on a Sun workstation, using a session host only for file service. Its basic keyboard commands are a subset of Emacs. However, the mouse adds a whole new style of interaction with the editor. The multiple window capability of the VGTS is put to good use, as well. And the user will quickly notice that it responds much faster than Emacs on a normally loaded system.

Ved manages one or more editing windows. Each window is thought of as a viewport onto a *buffer* of text, a continuously accurate display of some portion of that text. A change to the buffer is followed immediately by a corresponding change to the display. In each buffer there is a cursor, which is guaranteed always to be in the portion of the text displayed. Each buffer normally has a filename associated with it, the file from which it was read or the file to which it was most recently written.

7.1. Starting up

Ved runs under the V system executive, which is invoked as described in the previous chapter. Once inside the executive, type

```
ved
```

or

```
ved filename
```

The filename can be in any of the forms recognized by the V exec -- a relative pathname, an absolute pathname, or [session host] followed by a pathname of either type. Ved proceeds to read in the named file given, then requests a pad, its first editing window. This is indicated by the mouse pointer, which changes to the word "Pad". Move the mouse to the desired upper left corner of the pad and click any button. The pad will appear, and in it the first screenful of text will be displayed. The pad in which ved was invoked is reserved for displaying error messages and typing special text, such as filenames or search strings, which is not to be inserted into any buffer. In normal use it is convenient to shrink this window down to a few lines at the bottom.

At the top of the editing window, there is a banner. When the banner is inverted, then this window is selected for input either by the mouse or the keyboard. The banner specifies the ved window number which is used by the window selection command (described in section 7.10) and the Vgt number (see section 3.2). The rightmost area is reserved for the file name associated with this window. If the file name has an asterisk (*) prefix, then ved thinks that this buffer has been modified since the last write or save of the specified file.

As an added feature, there is a inverted line of text at the bottom of every ved window. This is the fixed menu area of the window. It can be used to enter some frequently used commands using the mouse instead of the keyboard (a full description of the fixed menu is in section 7.6.2).

7.2. Motion

The following commands are available to move the cursor within a file:

The four arrows Move the cursor in the direction indicated.

↑F, ↑B, backspace

Horizontal cursor motion.

esc f, esc b, esc backspace
: Word-oriented cursor motion. esc-f goes forward to the end of a word; esc-b and esc-backspace go back to the beginning of a word.

↑P, ↑N : Vertical cursor motion -- scrolling if necessary.

↑A, ↑E : Cursor to beginning, end of line.

esc comma, esc period
: Cursor to top, cursor to bottom of visible region.

esc <, esc > : Cursor to beginning, end of text.

↑G : Get out of special states. Whether you have just typed Escape or ↑X and didn't want to, or are busy typing a search string, or whatever, ↑G will get you back to the normal state.

↑X↑Z : Quit the editor. If there are any modified buffers, you will be asked if you want to save them. Here and in similar cases, if you are warned and then decide you don't want to do the command at all, type ↑G to escape back to normal editing.

↑C : Also quits, but first asks for confirmation, which should be answered with "y" or Return if you mean to quit. ↑C is kept for Unix compatibility, protected with a message because a multi-window edit can take some time to set up, and should not be at the risk of a single keystroke. In the future, ↑C is intended to serve a "quit local edit" function, when ved or something like it is a service available to programs like mail and send.

7.3. Paging and Scrolling

↑V, esc v : Page down, page up.

↑L : Redraw the display.

PF1 : Scroll -- move the viewport down one line relative to the text

PF2 : Scroll backward -- move the viewport up one line relative to the text

↑Z, esc z : Synonyms for PF1, PF2 respectively.

esc PF1 : Moves the viewport 1/2 page down the text -- half a ↑V.

esc PF2 : Moves the viewport 1/2 page up the text.

esc downarrow, esc uparrow
: Synonyms for esc PF1, esc PF2.

7.4. Simple Editing

Typing any printing character, or TAB, inserts the character typed. Other special characters are handled as follows:

↑D : Delete forward from the cursor -- the character under the cursor.

DEL : Delete backward from the cursor.

esc d : Delete word forward.

esc DEL, esc h	Delete word backward.
Return	Insert a Linefeed, not a CR character -- gets the desired effect.
↑O	Insert a Linefeed, leaving the cursor before it.
↑K	As in Emacs. Delete the contents of one (logical) line, or the carriage return on an empty line, into the killbuffer. A sequence of ↑K commands **uninterrupted** by any other command causes the whole section thus deleted to go into the killbuffer. ↑K after any other command restarts the killbuffer from scratch.
↑Y	Yank -- insert contents of the killbuffer at the cursor. The killbuffer is unchanged. The cursor ends up at the beginning of the insertion, and the Mark (see below) ends up at the end.
esc y	Yank, but without disturbing the Mark. The cursor ends up at the end of the insertion.
Linefeed	Insert a newline (Linefeed) and then indent the new line to the indentation of the previous line, using tabs where possible. If the previous line is empty, it will look up until it finds a nonempty line and use that as the standard of indentation.
esc Tab	Add indentation to this line equal to the indentation of the previous line. Intended use: if you type Return and wish you had typed Linefeed, this will make up the difference.
↑Q	Quote the following character. Allows you to insert non-printing characters (such as the useful ↑L, formfeed, which forces a page break on most printers) into the text.
↑\	Quote the following character and insert it with the high bit set. ↑Q and ↑\ are the only exceptions to the ↑G command: they will quote a following ↑G, but that simply means the insertion of a character, which can easily be deleted.

7.5. File Access

Whenever ved writes a file, it preserves the previous version of that file (if there was one) by renaming it to its former name followed by ".BAK". Thus myfile.c becomes myfile.c.BAK .

↑X↑V	Visit a file, whose name will be requested. The new file replaces the current one, so if the current buffer is modified you will be asked before proceeding.
↑X↑S	Write the buffer back to the file from which it was read.
↑X↑W	Write the buffer to a file whose name will be requested.
↑X↑I	Insert file at the cursor. You will be asked for the file name. Cursor and Mark are set just as in ↑Y above.
esc ~	Forget that the buffer has been modified.
↑X b	Toggle the .BAK safety feature. Creation of .BAK files makes file writing take about 4 times as long as it otherwise would, so if you really want that speedup, this will turn off the making of .BAK files. ↑X b again will turn it back on.
↑X c	Change current context. This command allows a user to change the way character string names are interpreted. A context is similar to a directory in that it defines which object is associated with a name. The file name represented in the banner of the pad should always be context independent. (See section 2.3.)

7.6. The Mouse

The mouse offers an alternative way of doing several common editing functions, such as placing the cursor and deleting or moving text. The mouse has two functions: fixed menu selection and editing.

7.6.1. Editing With the Mouse

Left button — Click and release it at any character in the text: sets the cursor at that character. Click it at one character, move the mouse to another point in the window, and release: selects the text between the point of clicking and the point of release. While you are moving the mouse with the left button held down, the region which would be selected if you released it at this moment is displayed in inverse video. When you release, your selection is defined and remains displayed in inverse video. Carriage returns are invisible, so the selection of a carriage return is shown by black space from the end of the text on that line to the end of the window. Note that a selection and a normal cursor are mutually exclusive. The importance of this will become apparent below. If you have a selection and click the left button, with or without moving, the former selection is deselected and a new cursor position or selection is chosen.

Middle button — When you have a selection, clicking the middle button deletes it into the killbuffer. If you have no selection, nothing happens. The position of the mouse is irrelevant.

Right button — Brings back the contents of the killbuffer and makes it selected. If there is nothing in the killbuffer, nothing happens. If there was a selection already, its contents are swapped with the contents of the killbuffer. If there was no selection, the contents of the killbuffer replace the cursor.

7.6.2. Fixed Menu

The fixed menu that appears at the bottom of every ved window provides the user with mouse oriented file perusal capabilities. Clicking the middle or right mouse buttons in the fixed menu area will execute the command that is nearest the mouse cursor. All the commands in the menu could be entered from the keyboard, therefore they are not described here. Refer to the sections on searching, scrolling, and regions for descriptions.

In the fixed menu area, the semantics of the each of the buttons differ. The middle button (in general) means *forward* whereas the right button means *backward*. For instance, clicking the middle button at the **Full-Page** command will cause the window to be scrolled forward one full page and the right button will cause a reverse scroll. The commands **Half-Page**, **Scroll-Line**, and **Search** behave in this same manner. The **Tag** command has exactly the same semantics for both buttons. **Mark/Point** is the only "different" command; in it, the middle button causes a jump to the Mark and the right button sets the mark at the point. Note that the left button has no effect on menu selection, to maintain continuity during dynamic selection. The **Search** and **Tag** commands will either use the selected string as the pattern or prompt the user for one in the case of no selection.

7.7. Searching and Replacing

↑S — Search for string. Prompts for a string, and finds the first instance of that string after the cursor. Prints "Not found" if there is no such instance. If you type Return without typing any search string, the previous search string is used -- ↑S Return is equivalent to PF3 as described below. Here and elsewhere, a newline can be inserted into the search string using the Linefeed key. It is echoed as an inverse-video backslash. Non-printing characters can be searched for, and are echoed as like "↑A". If the search succeeds, the string found is selected, and several special commands (described in The Right Hand and

the Left, below) are available. In particular, typing s will repeat the search.

↑R — Reverse search. Just like ↑S but searches backward.

PF3 or esc s — Repeat search. Forward search for the string most recently used in a ↑S or ↑R command. Works regardless of whether there is currently a selection or not.

PF4 or esc r — Repeat search backward. Like esc s but searches backward.

esc q — Query Replace. Prompts for a search string, then a replacement string. Then searches till it finds the search string, and selects that text. Type y (yes) to replace, n (no) to leave it alone and go on. Other options are described below. These special commands are available whenever there is a selection, so Query Replace is easily re-enterable. Just use PF3 to find and select the next instance of the search string, and away you go.

esc g — Go to line. Prompts for a line number, and moves the cursor to the head of that line in the file. The first line is numbered 1. If the number is too large, it will go to the end of text and notify you of the true line number there.

7.8. The Right Hand and the Left

When there is a selection, the cursor is not in a single spot, so it would not make much sense to insert characters at the cursor. So various printing characters are used as special selection-mode commands. The most basic of these commands are all assigned to left-hand keys. Thus one possible mode of operation is for the user to have his right hand on the mouse, selecting things, and his left hand at the usual place on the keyboard, giving commands which are not available on the mouse buttons. Others of these commands are designed for use with the search and replacement facility.

Non-printing characters other than those described below deselect, then perform their usual function as if the cursor had been at the beginning of the selection.

space bar — Deselect. The cursor lands at the beginning of the selection. All printing characters not mentioned here also have this effect, but the space bar is recommended.

Tab — Deselect, but the cursor lands following the end of the selection.

d — Delete. Exactly identical to the middle mouse button.

e — Exchange. Exactly identical to the right mouse button.

c — Copy in place. A copy of the current selection is inserted right after it, and becomes the new selection.

g — Grab. The current selection is copied into the killbuffer without deleting it.

s — Search for the next instance of the selected string. This becomes the search string, as used in future Repeat Search or search-and-replace commands.

r — Reverse version of s.

PF3, PF4 — Repeat search -- they perform their usual function, using the stored search string rather than the current selection.

PF1, PF2 — Scroll -- as usual, but unlike other commands they do not deselect unless the selection is being scrolled off the screen.

↑L — Redisplay, with the selection near the top of the screen. Good for long selections which run off the bottom of the screen.

y	Yes replace. Replace the selection with the stored replacement string.
n	No don't replace. Search for the next instance of the stored search string.
backspace	Undo replacement. Search backward for the first instance of the replacement string and replace it with the search string. The resulting string is selected.
Y	Yes replace but don't move on. The selection is replaced and the result remains selected.
u	Undo in place. The current selection (which hopefully is the replacement string) is replaced with the search string.
S	Search for next instance of the replacement string.
R	Reverse version of S.
q	Start query replace. Takes the current selection as the search string, and prompts for a replacement string. Replaces the current selection, and goes on to the next instance of it, just as "y" would do.
Q	Set replacement string. The current selection is copied into the replacement string. This makes it possible to alter a Query Replace in mid-flight.

7.9. Mark and Region

Ved maintains an invisible point in the buffer called Mark. Until otherwise set, it is at the beginning. It can be set by ↑X↑M or Control-@ (Control-spacebar is the same as Control-@ on some keyboards). "Region" refers to all the text between Mark and the cursor. The following commands use these concepts:

↑X ↑M, ↑@	Set the Mark at the current cursor position.
↑X ↑X	Exchange Mark and cursor (changing the display if necessary to keep the cursor on the screen).
↑X ↑K	Kill Region. Region vanishes and becomes the killbuffer -- so this command can be undone with ↑Y. Note that in Unix Emacs this function is normally bound to ↑W.
↑X ↑R	Write Region. Prompts for a file name, and writes the region into that file. The buffer is unchanged.

7.10. Windows and Buffers

Ved is normally started with one editing window, but it can support several. Each editing window is associated with a separate editing buffer, which includes the text, cursor position, selection if any, associated filename, and whether this buffer has been modified. Multiple windows on the same buffer are not supported. Since the correspondence is one to one, hereafter we refer to "window" meaning "window and its associated buffer". At any time one window is selected for editing, and is foremost on the screen. Window selection can be changed by clicking a mouse button in an unselected window, or by pressing the appropriate number key on the keypad. Windows are numbered, starting at 1, in the order of their creation.

The search and replacement strings and the killbuffer are universal across windows. Thus it is possible to kill some text in one window and yank it into another. It is likewise possible to search for a string in one window, then select another window and repeat-search on the same string.

The window from which ved was invoked is special. It cannot receive input except during certain commands, at which time it is selected automatically. It is never receptive to mouse input.

↑X g	Get file. Prompts for a file name, and reads it into a new window. If no file name is given, creates an empty window. Here and in all other cases, when a window is to be created the mouse cursor will change to "Pad" and let you indicate where the window is to go. If you abort the pad creation by pressing all three buttons, the command is aborted.
↑X d	Delete the current window. Will warn you if it is modified. The next lower numbered window becomes selected. If the last window is deleted, ved quits, because it cannot live without a selected window.
↑X y	Yank to window. The killbuffer is copied into a new window.
↑X a	Pull Apart. Kills the Region in the current window and transfers it to a new window.
↑X m	Merge windows. Asks the user to indicate a secondary window, and transfers its contents into the current window at the cursor position. The secondary window is then deleted. The secondary window is indicated by clicking the mouse in it.
↑X1 - ↑X9	Select the corresponding window.

Mouse click in any unselected window
Select it.

7.11. Crash Recovery

In an ideal world, this program would never crash. But in fact it sometimes does -- but it is so designed that it has to crash in two stages to lose your text. Normally a crash only breaks the first stage, in which case you will get the message

```
Editor crash!  Shall I try to save this buffer?
```

If you have any changes, and you value them, and the crash did not come during a save, it is probably a good idea to answer "y". A .BAK file will be made, so the danger of total loss is small. If this succeeds you will be asked

```
Try to continue?
```

If you answer "y", the inner editor will be recreated with the buffers just as they were. For some display-related errors, a ↑L at this point will set everything right. However, you are on shaky ground, and the best thing to do first is save any modified buffers in other windows.

If you are dumped into the debugger by an editor crash, the debugger command **Suicide,g** will destroy the process that got the exception. This will usually activate ved's crash recovery facility, as described above.

Ved tries to detect the cases in which it runs out of memory. In some activities, such as reading in a file, it will simply refuse. In others, such as a kill or an insertion, you will get the message

```
Out of memory!  Please do one of the following:
  Pick a window to delete
  c - continue (after you free something)
  q - save and quit
  ↑C - quit without saving
```

Ved cannot proceed without more memory, and cannot exit gracefully from this activity, so you have to help it out. To pick a window, select it with one mouse click and signal it with a second click. It will be saved if modified, then deleted to reclaim its storage. If you have anything else going on on your Sun, you can delete a view or terminate a program or delete an exec to free some storage. After doing so, type c to continue. If this won't work, type q to try to save everything and quit gracefully. It will save the current buffer last, trying to avoid the dangers of saving a half-modified text. ↑C is a last resort, a quick and dirty quit.

7.12. Hints on Usage

Ved has no repeat factor like the ↑U of Emacs. Use the hold-repeat feature of the arrow keys to move the cursor around -- they happen fast enough that this is rather workable. Take advantage of the scrolling features -- you will quickly become addicted to the convenience of getting your material centered on the screen exactly as you want it. When making scattered changes, you will find the mouse very helpful. Arrow-repeat will get you there fast, but a mouse click will get you there now. Likewise select-and-delete is the fastest way to delete a small piece of text so you can type something to replace it.

Ved is almost too large now to run on a 256K workstation. Use it only with one or two page files on such workstations. Attempts have been made to catch the event that ved runs out of memory, and give you a chance to save, but this is not reliable. .

If you get into a weird state, try ↑L, it often restores sanity. If that fails, a save may work anyway -- it uses only the textual data structures, and it is the display structures that usually foul up.

Esc followed by a number key invokes one of the debugging routines. Avoid them.

— 8 —
Draw: A Drawing Editor

The Draw program is meant to fill a specific void in the V-System software. Specifically, the lack of an analog for the Alto Draw program is addressed. The V Draw program is not identical to its Alto counterpart, although as much symmetry as possible was included. If you have any questions about the behavior of the program, try using the Help command. It will (hopefully) provide some meaningful information.

This program runs under the vgtsexec only. Since it uses splines, it will not run under the small version of the VGTS configured for 256k Sun workstations.

8.1. Conceptual Model

The conceptual model behind this program is one similar to a person drawing regular objects on pieces of paper. The drawing pen has a number of different nibs (tips) which can be selected. Similarly, the "ink" and fill pattern used to shade areas also comes in several flavors. The ink can be either TRANSPARENT or OPAQUE. If transparent ink is used to fill an object, anything under the object will show through. If opaque ink is used, underlying objects will be obscured. The **Bring to Front** (raise) and **Push to Back** (lower) commands are useful for shuffling which objects lie on top of each other. Each object lies entirely in its own plane, so it is impossible to create works similar to those popularized by M. Escher.

Curves are generated using B-Splines of various orders. By default, all curves are of order 3, and thus use quadratic interpolation. The Alter command can be used to change the order of the interpolating splines. Automatic filling (shading) of closed objects (specifically, CLOSED CURVES, CLOSED POLYGONS, and certain TEMPLATES) is possible.

A GROUP is a collection of existings objects lumped together and treated as a single unit. Groups are useful for replicating completed symbols and figures in several places.

A variety of standard shapes are provided, and are referred to as TEMPLATES. Templates for Arrowheads (open and closed, wide and narrow) exist, as well as templates for rectangles, circles, and ovals. Each object on the screen has a type, so while it is quite possible to create a rectangular closed polygon which appears identical to a rectangular template, they are of distinct type. This is important to bear in mind because whenever you are asked to select an object on the screen, the program will only examine objects of a certain type, and so some confusion might arise when the program doesn't find the thing you are right on top of.

8.2. Screen Layout

When the program is first invoked, it will create two new windows on the screen. The large empty one is the main drawing area (known as "drawing area" to the Vgtsexec), and the smaller one is the commands window (known as "draw menu" to the Vgtsexec). The drawing area is zoomable (for instructions on how to use the Vgtsexec, see the V Commands Manual), and the grid spacing available at normal magnification is the same as that used by the program. Since the program has no way of knowing what magnification you are using, it aligns to the unzoomed grid values. The VGTS will place grid points at a constant separation, regardless of magnification. You may create additional views, move existing views, etc., to your satisfaction. The default drawing area is in the proportion of 8.5 by 11, and centered. A frame is put around the actual size of a drawing page to provide some reference points if you zoom the view or change its centering. The frame is normally not visible, as it lies entirely outside the default view. It will not appear in any output.

The Menu window is divided into three separate menus. One consists of action commands (**Rotate**, **Scale**, **Move**, **Copy**, **Draw**, **Alter**, **Erase**, **Push to Back**, and **Bring to Front**), which place the program into a state where it is waiting for you to specify certain actions. Typically, you will need to specify an object type (ALL OBJECTS, TEXT, OPEN CURVE, CLOSED CURVE, CURRENT OBJECT, OPEN POLYGON, CLOSED POLYGON, GROUP, or TEMPLATE) and then some data points. A second series of menu options runs along the bottom of the menu window. These are the commands which control various defaults within the program. For example, if you wish to change the default font which new text is displayed in, select the Text default option in the lower left corner of the menu (not the object type TEXT under the "Objects" column), and make the desired selection in the popup menus which will appear. The third section of the Menu is the list of permanent menu selections (**Exit**, **Help**, **Misc**, **Undo**, **Abort**, and **Done**). These commands are valid most of the time. In particular, you can always hit **Help**.

The original window which you used to run the draw program will serve as a combination history log and prompt file. The program will print many prompts in this window, telling you what it expects you to do next, and what it didn't understand of your last action. When you ask for Help, it will appear in this window.

8.3. Command Input

The program accepts all command input through the mouse. Clicking the mouse near a command in the Menu is sufficient to indicate to the program that you wish to specify that command. Clicking the mouse in the drawing area will either specify a data point or a command, depending on which mouse buttons are used. More on that later. Sometimes input is required from the keyboard. Due to limitations of the VGTS, when the program is requesting input from the keyboard, clicking the mouse will have no immediate effect. Once the program gets around to asking for mouse clicks again, all of the saved clicks will be processed.

Occasionally the VGTS will have difficulties synchronizing communications. This almost invariably occurs when you hit a character on the keyboard while the program is expecting a mouse click. When this happens, a message similar to

Sync error - Expected 037, got 040

will appear. After this happens, things usually get a little strange. The program will starting complaining about

Internal Error: Bad mouse buttons 170 at (5, 89)

(maybe with other numbers) or more commonly

Missed! Please select a menu command.

and refuse to recognize anything you do as intelligible. Do Not Despair!! The remedy for this is to deliberately force *more* sync errors (by alternately typing a character on the keyboard and attempting innocuous commands with the mouse, like Help) until a full cycle has been completed. This typically requires you to force three more synchronization errors, and then everything will be completely functional again.

8.4. Control Points and Sticky Points

When you create a curve, you will be asked to specify the Control Points of the spline. These points are the places which you wish the curve to pass near. The more control points you put in one place, the nearer the curve will come to that place. Also, placing multiple control points at a single point will make the curve much "sharper" at that point. Except for the end points of open curves, and multiple control points, the curve will not pass through any of the control points.

Sticky points (similar to Knots) are points which actually lie on the curve. They are calculated by the program to help you with the alignment of objects. There will be the same number of control points and sticky points on curves. Polygons are a special case, in that since the control points of a polygon actually lie on the "curve", the program considers them to be sticky points too. This means that the sticky points on

polygons lie at the corners and in the middle of each edge. Sticky points for bounding boxes (e.g., for TEXT objects) are the same as those for polygons.

8.5. Mouse Buttons

When the mouse is clicked inside the Menu, it is unimportant which mouse buttons you use. Within a popup menu (a list of choice which "pops up" after you do something), you can abort by either clicking outside the menu or by pressing all three mouse buttons down and releasing them. In general, you don't have to release (or press) the buttons all at once, but the mouse position is based upon where the cursor is when you release the last button.

Clicking the mouse inside the drawing area can cause one of several different commands (and mouse locations) to be used by the program. The use of mouse keys within the drawing area is as follows:

Buttons	Effect
X - -	Specifies a data point right where you are pointing.
- X -	Requests the program to find a sticky point.
- - X	Requests the program to use the nearest grid point.
X X -	The Almost Done command. (see below)
X - X	Requests that a Checkpoint be made.
- X X	Equivalent to the "UNDO" command.
X X X	Equivalent to the "ABORT" command.

Sticky points are points located on or near existing objects on the screen. They are useful for connecting lines to objects, specifying points actually on the object, etc. GROUPS themselves do not have sticky points, although the objects within a group may. Curves have one sticky point per control point. These points are located midway between each pair of control points. When you request that the program select a sticky point, it will choose the nearest such point which is within a given radius (about 1 inch). If you are further trying to specify a point on a specific type of object, the search for a suitable point is begun again from the previous result. Naturally, if the original mouse click relocated to a sticky point on an object of the proper type, that will be the closest point for any further searches.

Grid points are spaced every 16 pixels (at normal magnification). If you wish to see these grid points, use the **Toggle Grid** command within the Vgtsexec. For printed output, pixels are assumed to be distributed at 100 per inch.

The **Almost Done** command is quite similar to the **Done** command described below, in that it tells the program you are satisfied with the selections you have made, and that you wish the program to accept them. Unlike the **Done** command, it does not tell the program that you are completely finished with whatever you were doing. Instead, it permits you to, for example, erase several objects of the same type without having to go to the Menu each time and specify the **Erase** command and the object type.

It differs from a "repeat" command in that it does not force the program to create a checkpoint before beginning the next command. As a special case, when in conjunction with the **Draw** command, **Almost Done** and **Abort** behave slightly differently. **Abort** will cause the last item you drew to be removed, and **Undo** will subsequently remove all of the others. Normally, **Abort** would cause all changes since the command began to be removed.

This command is also quite useful when drawing a series of objects of similar type. You can specify that you wish to draw a closed curve, place the control points for the curve, and then confirm with the right two mouse keys (**Almost Done**). The program will complete the curve you have outlined, and wait for you to specify another closed curve, just as if you had confirmed with **Done**, and then selected **Draw Closed Curve** again.

8.6. Selecting Objects

The standard method of selecting an object is to first choose the object type and then to point at the desired object on the screen, using some combination of mouse buttons which specify a data point. If you select the wrong object type, simply point at and choose a different object type before you select the object itself. As a short cut, the program maintains a notion of what it considers to be the "Current Object". This will be the last object you selected. If you choose the object type CURRENT OBJECT, and it is unambiguous as to what the current object is, this will suffice. After you select an object, that object will be "highlighted" on the screen. This normally consists of frame or bounding box appearing around the object. If the program misinterprets your pointing, use the **Undo** command (also available by pressing the right two buttons on the mouse) and then point to a different location on the screen.

The most notable exception to this process is the method for selecting groups. Since individual objects can appear in several groups, a popup menu appears when you select the GROUP object type, listing all of the existing groups. Either choose one of these groups from the menu, or click outside the menu to abort.

8.7. Action Commands

There are nine action commands. Each is useful for manipulating one or more objects. Typically, each action command will require the selection of an object type.

Rotate This command will permit you to select an object, specify a fixed point about which the rotation is to take place, and two points which will define the angle of rotation.

Text is rotated about its positioning point. Only the position of the text is changed -- the orientation of individual letters is constant.

Scale This command will permit you to select an object, specify a fixed point for the scaling, and two points which define the scaling vector. This command is useful for expanding and contracting objects.

Scaling text will not change its size or font. It will change the location of the string based upon its positioning point.

Move This command will permit you to select an object, and then specify a pair of points which define a displacement vector. This vector tells the program how far and in which direction to move the object. By using this command, you can move existing object about on the screen.

Copy This command is similar to **Move**, except that it leaves behind an image of the object. If you copy anything other than a group, the two resulting items are completely independent. Changing one will not affect the other. Groups, on the other hand, work differently. When you copy a group, it simply creates a new image of the same group, appending "**, copy** *n*" to the old group name, where *n* is a small number.

Draw This command lets you create new objects. Since it is anticipated that most of the time spent in this program will be drawing new objects, this is (so to speak) the default command. If you ever select an object type without giving a command first, the program will assume that you implicitly meant to draw an object of that type.

Alter This command is useful for changing the characteristics of an existing object. It will permit you to move the control points on splines, change the filling and nib selections used to draw objects, etc. ***Currently Unimplemented.***

Erase This command allows you to delete (erase) objects from the screen. If you decide you just don't want to see an object any more, erase it. If you decide that the last command you did was a mistake, you probably want to use the **Undo** command.

Push to Back Also known as Lower, this command will place the selected object behind all of the other objects. This is useful when you use opaque ink to fill something, and it winds up obscuring an object you want to see.

Bring to Front Also known as Raise, this command functions much like the one above, except that it will place the selected object on top of all of the other objects. Note that you can still point to objects you can't see -- the program will find sticky points on completely obscured objects with no difficulty.

8.8. Object Types

There are nine selections within the Object type menu. Some of the selections are obvious, some are not. All are meaningful for selecting existing objects, but the "Draw All Objects" command is not.

All Objects This will permit you to select all objects at once. If you decide, for example, to move everything on the page a bit to the left, then this is the object type you want. The **Clear Screen** command (available under **Misc**) simply does an "Erase **All Objects**", and then sets the dirty flag to false.

Text As is obvious, this will select any text string. It doesn't matter what font the text is in -- it is all of object type text.

Open Curve An open curve is a spline with open end conditions. When you create one, the first and last control point you specify will actually be on the curve, and the curve will be tangent to its convex hull at these points. (It gets straight at the ends.)

Closed Curve Closed curves are splines with closed end conditions. To create one, you specify all of the control points you desire (the program will build a frame for the spline to help you visualize the resultant curve. You do not need to try to get the first and last control points in the same place -- the program will close objects automatically. Closed curves can be filled.

Current Object This will select the current object, if one exists. If you attempt to "Create **Current Object**", the program will interpret this as a request to create an object of the same type as the current object.

Open Polygon A bunch of connected straight lines. (Internally, these are just splines of order 2, thus using linear interpolation.)

Closed Polygon Also straightforward. Closed polygons can be filled. The program will automatically add the last edge, closing the polygon.

Group Groups are a bit tricky. A group consists of several existing objects, lumped together and treated as a singled named object. To create a group, you select as many existing items as you like, and they are all placed into the group. Once items are inside a group you *cannot* select them by normal means. They are in effect hidden inside the group.

Groups are useful if you have a complicated, complete figure which you wish to deal with as a whole. Due to internal limitation, you can't have more than some fixed number of groups active at one time.

Template Templates are standard shapes. The broad classes of standard shapes are arrowheads, circles, ovals, and rectangles. Arrowheads can either be open or closed. Closed arrowheads are filled with black ink. The nib used to draw an arrowhead will be the same size as the the current nib, but will be circular.

Rectangles and Ovals are both created by specifying two points on a long diagonal of the

bounding box. You can give either the upper left and the lower right or the lower left and the upper right. The program doesn't know the difference.

Circles are created by specifying the center of the circle and a point on the circle. Since circles are actually high order splines (5^{th} order, using quartic interpolation), they are not truly circular. If you draw a really large circle, it is possible that the point you specify for being on the circle will be off by as much as a quarter of an inch or so. For normal sized circles, there will be no difficulties.

8.9. Default-setting Commands

There are a number of Menu selections which control various defaults within the program. They can be selected at any time. They are:

Text When you select this command, a popup menu will appear. You can use this menu to specify either the method of positioning text, or the general class of font you wish to use.

There are three different ways of positioning text -- you can specify (with a data point entered via the mouse) either the bottom left-hand corner of the bounding box for the text, the center of the bottom edge of the bounding box for the text, or the bottom right-hand corner of the bounding box. Initially, the program positions text based upon the center of the bottom edge of the text. You can only specify a point on the lower edge of the text -- the program will automatically compute where the bounding box for the particular piece of text lies. If you are confused about where text should appear, try positioning a few strings, using the exact positioning (leftmost) mouse button.

If you wish to change the font in which new text is displayed, choose one of the font menu selections. A second popup menu will appear, listing all of the available fonts. The "Standard Fonts" category contains the fonts which can be considered to be mundane text fonts -- variations on Helvetica, Times Roman, and a few randoms like Clarity 12 and Cream 12. The "Unusual Fonts" selection contains everything else -- Old English 18, Hebrew, Cyrillic, Greek, APL, Math, CMR, etc. There are some fonts, like Gates 32 and Template 64, which are very difficult to use unless you are quite familiar with them. They are included primarily for completeness sake.

Fill on/off This toggle controls whether or not closed objects (Closed Arrowheads, Rectangles, Circles, Ovals, Closed Curves, and Closed Polygons) are filled by default. If you wish to use created filled objects, set this switch to **Fill on**. Toggle the switch by pointing at it and clicking the mouse.

Nib If you wish to change the nib (paint brush?) used to draw lines, select this command. There are four nibs, each of which comes in four sizes. The program initially uses a Circular nib of size 1. The four nib shapes are Square, Circular, a horizontal Dash, and a vertical Bar. The four sizes are (unsurprisingly) 0, 1, 2, and 3. Size 0 is a single pixel, so all of the nibs look the same at that size. An example of the current nib, at the current size, is displayed in this area.

Fill This command allows you to change the fill pattern used to shade closed objects. A small square section of the current fill pattern appears in this area. When you select **Fill**, a popup menu will appear, permitting you to either toggle the use of opaque vs. translucent ink or select a general class of fill pattern.

The "Striping Patterns" consist mostly of lines drawn at various angles. The program initially uses the one pattern least like straight lines but still considered a striping pattern: Chain Link.

The "Gray Tones/Area Patterns" selection consists more of patterns which are either various shades of gray, or are regular and (to some) uninteresting. These patterns are useful for highlighting objects.

The "Textured Patterns" are supposed to be more representative of actual textures. The names for these patterns are supposed to be suggestive of their appearance, but many of the names are nonetheless obscure.

8.10. Permanent Menu Commands

There are a few commands which are generally useful, and are thus considered permanent. Not all are meaningful at all times, but are still useful enough to be given an entry on the Menu.

Exit This command will exit the program. It can be used at any time. If you have performed any command since either clearing the screen or writing a file, the program will ask you to confirm that you actually wish to exit even though there are unsaved changes. Typing "**y**" will confirm. Typing anything else will abort the command. Note that since the program is reading input from the keyboard when asking you to confirm, any mouse clicks you make will simply be queued, and when you do type at the keyboard they will all be dealt with. In particular, be careful about using the **Abort** command here.

Help This command will provide a brief description of any other command you like. To get help on a specific command, just select that command after you select help. To get help with the mouse buttons, push any one button in the drawing area. To exit help, select **Help** again. You can ask for help at any time.

Misc This command is actually a front for a collection of less frequently used commands. The various commands are for clearing the screen (only valid if no other command is in progress), reading and writing files, generating press files, and toggling debug printouts. The debug printouts are long and plentiful -- they are also meaningless to the uninitiated.

Reading a SUN draw file does not clear the screen -- it adds the new objects to the current display. ***Press files currently unimplemented.***

Undo There are actually two forms of the **Undo** command. If you select it while at the top level (when there are no other commands in progress), the effect will be to revert to the previous checkpoint, in effect undoing the last command. The program maintains a list of 10 checkpoints, so you can undo up to 10 commands.

Checkpoints are copies of the complete state of the drawing area. The program will make one before it starts any action command. In this way, the **Abort** command doesn't need to keep track of incremental changes during the processing of a command.

If you give the undo command while in the process of specifying another command, it will attempt to undo your last choice. If you are specifying data points, **Undo** will delete the last point you entered. If you are selecting an object, **Undo** will unselect the object and permit you to choose again.

Abort This command will abort any command in progress. In general, **Abort** will also revert to the last checkpoint, since action commands all make a checkpoint before they begin processing.

Done This command is used to tell the program that you are happy with all of your choices, and are done specifying parameters. After you hit **Done**, the program will attempt to perform whatever command you are executing, and will display the results. Note that by pressing down the left two mouse buttons while in the drawing area, you can give the **Almost**

Done command, which will confirm all of you selections, but not stop the command. In this way you can, for example, create several objects at once. Note, however, that if you use the **Almost Done** command, you are not guaranteed that a checkpoint will be made. If you create several objects with **Almost Done**, and then hit **Abort**, some (perhaps even all) of the new objects may be aborted also.

—9—
bits: a bitmap and font editor

bits is a special-purpose editor for working with bitmaps and fonts. It makes intensive use of the VGTS. The VGT of the executive under which **bits** is started up, is used to display various status information, as well as being the menu of commands to execute. When started, **bits** will ask for you to create a new view in which the actual editing is performed. If you request to view sample text, you will be asked to create a third VGT (see below). These last two VGTs can be zoomed.

9.1. Command Input

In this chapter, when you are asked to do the command **[xxx]**, it means that you should select and click the mouse at the field **[xxx]** of the status/command VGT. You get the same feedback as with pop-up menus, with the field in inverse video. Some of these fields, when activated, expect you to type in some number or string. In those cases, you have the full power of the line editor, until you type a <return>. (To abort input, type CTRL-g.)

9.2. Rasters

The important thing to remember is that **bits** handles *pointers* to bitmaps. These we call *rasters*. A raster also contains size and offset data, so it can point to *part* of a bitmap. You can name a raster using the **[Store with new name]** command, and later retrieve it from the **Table of saved rasters**. You can thus save multiple pointers to the same bitmaps under different names. If you change bits in one of the bitmaps, the bits will also change in the other rasters, since they refer to the same bitmaps. Use the **[Save a fresh copy]** command to make a virgin copy of a bitmap, which is guaranteed to have no other rasters pointing at it.

9.3. Changing Raster Size

To change the *size* of a raster, point at the boundary, click the *middle* button and "move" the boundary to where you want it. You can also change the size of a raster with the **[Width]** and **[Height]** commands. To do this, select one of these fields, and type in a number. The absolute value typed in becomes the new size. If the value is positive, the old and new rasters coincide at the top left corner; if the value is negative, they coincide at the bottom right corner.

Note that when you change a raster's size, all other rasters pointing at the same bitmap will be adjusted to point at whatever bits they used to point at. This is true even when you *increase* the size. (When the size is increased, and the underlying bitmap is larger than the part pointed to by the current raster, the hidden part of the bitmap will appear. If this isn't enough, a new bitmap will be allocated, and all the pointers adjusted.)

9.4. Bitmap I/O

You can read and write bitmaps in **.sun** format (as used by the photo program), using the **[Read raster]** and **[Write raster]** commands. To write a raw raster in hex suitable for putting in a C program, use the **[Write hex]** command.

9.5. Painting

To *set* (blacken) a pixel, point at it with the mouse, and click the *left* button. To *clear* (whiten) a pixel in a bitmap, use the *middle* button.

9.6. Inverting a Raster

Selecting **[Invert black and white]** inverts the interpretation of black and white pixels. This interpretation is actually stored as part of the raster object, so no pixels are actually changed (except on the display).

9.7. Raster Operations (BitBlt)

You can do a general 2-operand BitBlt with the **[Raster operation]** command. The current (displayed) raster is used as one of the operands (the "destination"), so this should be selected first. Then give the **[Raster operation]** command, after which you will be asked to select an operation. Available are plain copy, 'and', 'or' (paint) and 'xor'. In addition, the **[Invert Source]** modifier first inverts the source. **[Invert Destination]** does the same for the destination, which means inverting the destination operand *and* the output result. Finally, you must select the other operand (the "source") from the name table.

You can also select **[Get the empty raster]** as a source. This gives you an infinite plane of white pixels. This, together with the **[Invert Source]** option, allows you to conveniently clear or set any rectangle.

9.8. Reflection and Rotation

Selecting **[Reflect/Rotate]** will do one of these transformations. (A popup menu asks for the particular transformation.) Note that the result is a "fresh" raster: There are no other rasters *or* tables pointing at its bitmap.

9.9. [Replace in table]

This command asks you to select an element in the raster table or the current font. The element is replaced by the current raster. If a **[Table of saved raster]** element is replaced by the Empty Raster, its space is freed.

9.10. Making a Copy of the Screen

You can make copy of the frame buffer, with a little bother. Select **[Get framebuffer]**, which gets a pointer to the frame-buffer. You should now use **[Height]** and **[Width]** to reduce the time and space required to deal with it. (The framebuffer is *big*.) You should **[Save a fresh copy]** to see what's going on, and then use the middle button to select the part that interests you. This will be slow, since such a big raster is involved, and you will also have to use the Vgts window manager commands.

9.11. Fonts

A *font* is a collection of *characters*. From **bits**' perspective, a character is a bitmap with some extra information. **bits** currently knows about fonts in the following formats:

- sf format ("Sun format"), which is specially optimized for the Sun graphics hardware. (The name

should probably be changed, since it conflicts with Xerox' Spline Format.)

- The same format, but the font is stored in an archive (library) of relocatable binary files. Thus fonts can be linked in with programs, *or* read in at run time. The standard fonts are stored in **/usr/sun/lib/libsfsonts.a**.
- Pxl format, which can be generated by MetaFont, and is used by a lot of the TeX people.

To read / write a font, select the desired field in the **Read font | Write font** table. Note that you cannot write a font to an archive.

9.11.1. Displaying Fonts

When a character in a font is displayed, there are funny lines sticking out of the bitmap picture. The two horizontal lines show where the baseline of the character is. The lower vertical line shows the starting position ("origin") of the character. The top line indicates the width of the character, and shows where the next characters should start. You can select any of these lines (with the middle button), and adjust them with the mouse.

9.11.2. Font parameters

This is a section of the pad with magic numbers about the current font. They can all be changed, but you should know what you are doing.

Design size is the size in points at which the font is designed for. **Magnification** is one thousand times the number of times the image is magified, relative to a default Pxl resolution of 200 pixels/inch. To be compatible with the Altos, we have decided that the resolution of the Sun display should be defined to be 80 pixels/inch. This means that the 1.0 magnification will have the magnification parameter of 400, which is somewhat small. Both these are TeX/Pxl parameters.

[Raster alignment] is the bit boundary character bitmaps should be aligned on in **sf** font files. It must be 1, 8, or 16.

9.12. Sample Texts

To study how a text string would look at no magnification, select **[Sample text]**. You should then type in the text you want displayed. This text will be placed in a new VGT. To change the text, just reselect **[Sample text]**, the old text will be placed in the line editor buffer, to simplify small changes. If you edit the font, select **[Redraw]** to update the sample.

Note that in the sample, the character **'\'** is special. It is used to indicate special non-ascii characters, as in C. Specifically, **'\'** followed by a 3-digit octal number is the character with that ordinal value. **** displays ****, and **\b**, **\t**, **\e**, **\r** and **\n** are BackSpace, Horizontal Tab, Escape, Carridge Returen and Line Feed, respectively. **\@**, **\A**, ... **_** are control characters: **↑@**, **↑A**, ... **↑_**.

9.13. Printing a Raster

There is a Unix program to convert a **.sun** file to a **.press** file. To run it (on some Stanford VAXen), do:

```
/usr/sun/src/graphics/pix/sunpress -p X.press X.sun
```

This, together with the **[Get framebuffer]** command, allows you to print a hardcopy of the screen on a Dover printer.

9.14. Bugs and Problems

.sun files use 1 to mean 'white' while **bits** uses 0. This means that you should **[Invert black and white]** after reading and before writing, if you want to use the bitmaps for programs like **sunpress** and **photo**.

The are some limitations on how bitmaps are displayed by the VGTS. A bitmap can only be should magnified 1, 2, 4, 8, or 16 times, so other zoom factors will be wrong. Also, it is over-conservative when clipping rasters, which means that a whole row of bits could be missing.

Raster operations do not take into account that rasters may be overlapping.

bits is not very robust against things like running out of memory. Caution would imply that you save your work often.

— 10 —
Amaze

Amaze is a game for two to five players which runs under the plain (non-VGTS) exec (VV). If you see the letters VGTS in a small window on your screen you are not running the plain exec. See section 2.2 for instructions on how to start up the plain exec.

To run *amaze*, type the command

amaze

If no one else is playing, it will type "New game starting" and then draw the maze. Otherwise it responds with "Joining game as player number x" and then draws the maze. Your player token, called a monster, will be sitting in the center of the screen just above a checkered flashing door. From this point, you control your monster through the keyboard. The commands are:

```
i       Move the monster up.
,       Move the monster down.
j       Move the monster left.
l       Move the monster right.
k       Hold the monster at its current position.
a       Let the monster's moves be selected randomly.

e       Fire the monster's missile up.
c       Fire the monster's missile down.
s       Fire the monster's missile left.
f       Fire the monster's missile right.
         (Note: the missile can be fired only once every six
           seconds.)

h       Hide the monster from other players -- no shooting allowed
        while hidden.
v       Let the monster be seen again -- can shoot again, too.
         (Note: monsters stay hidden for ten seconds, but once
           they become visible, they remain visible for 16 seconds.)

0       Set monster velocity to 0.
1       Set monster velocity to 1.
2       Set monster velocity to 2.
3       Set monster velocity to 3 -- the starting velocity.
4       Set monster velocity to 4.
5       Set monster velocity to 5.

q       Quit the game, but continue to watch other players.
↑       Rejoin the game just above the door.
r       Rejoin the game at a random corner in the maze.
Ctrl-C  Terminate your involvement with the game.
```

Note that to leave the game entirely you hold down the CTRL key and type 'c'.

To rejoin the game after being shot by another monster, use either the ↑ or the r command. The game currently does not keep score of the number of hits you inflict or suffer.

Problems and questions should be directed to Eric Berglund -- berglund@Diablo.

checkers allows you to play a game of checkers against the computer. This version of the program executes entirely on the player's workstation.

On starting the program, the view manager will prompt you for the position of the VGT representing the checkerboard.

The player moves the 'red' (white) pieces; the program's pieces are black. You are expected to make the first move. You can, however, force the program to move first by "passing". (See the paragraph describing the menu, to follow.) To make a move, move the mouse to the square containing the piece that you wish to move, and click either the left or the middle mouse button. If this piece can be legally moved, it will then be highlighted. Complete the move by moving the mouse to the destination square and once again clicking the left or the middle button.

If the move that you have selected is legal, your piece will be moved, and the program will then make its move. Note that having selected a piece to move, you can abort this selection by clicking an illegal destination square (the source square itself, for example). If a capture of an opposing (ie. black) piece is possible, your next move must be a capture. A message indicating such "forced captures" will be displayed just below the board. In such a case, the program will not allow you to make a move that is not a capture. Multiple captures are handled correctly - if you move a piece by making a capture, your move will not be completed until all possible captures with this piece have been made.

The standard rules of checkers apply. If a piece reaches the eighth rank of the board, it is promoted to a king; kings may move in any direction. A side wins either by capturing all of the opposing pieces, or if the opposing side can make no legal move.

When it is your turn to move, you may also use the right mouse button to select from a menu of options, which are described below:

Redraw — This causes the VGTS to redraw the entire board. This command should rarely be necessary.

Pass (skip turn) — This command can be used if you want the program to make the first move. You can also use this to avoid any capturing obligations.

Change search depth — By default, the program searches 4 half-moves ahead when choosing its next move. That is, it considers its own move, your response to this move, its next move, and your response to that. The "Change search depth" command allows you to change the depth of lookahead to any value from 1 to 8. Don't select any of the higher depths unless you have a lot of patience, however. The program takes about 20s to respond to a typical opening move when the depth is 6, about 50s when the depth is 7, and about 3 minutes when the depth is 8. (These times were taken on a 10 MHz SM1 workstation - Cadlines will be slightly slower.) Note that you may find out the current search depth by selecting "Change search depth", and then clicking outside the 'depth' menu.

Edit board — This command puts you into *Edit mode*, which allows you to cheat by adding pieces to, or removing pieces from, the board. Edit mode is described below.

Back up one move — This allows you to retract (eg. to correct) your last move.

Resign — The quick and cowardly way to end the game.

The program chooses it's move by performing a 'brute-force' search, using alpha-beta pruning. It evaluates

the board positions at the 'leaves' of the search tree using a simple heuristic based on the number and position of pieces on each side. A 'value indicator' to the right of the board indicates the value of the current position, as seen by the program. (If the indicator is above the halfway mark, for example, then the program 'thinks' that you are winning.) There are also counters immediately above and below the value indicator, giving the number of pieces on each side. The value indicator and the piece counters are updated whenever the program completes its move.

You can make changes to the board (between moves) in *Edit mode*. In this mode, a special menu is displayed to the right of the board. To add a piece to the board (or change an existing piece), click the square in the menu that contains that piece. You may place a copy of this piece on any (shaded) square of the board, by clicking that square. You may do this repeatedly; it is not necessary to select from the menu each time. Note that you use the 'empty square' to delete one or more pieces from the board. You may remove all pieces from the board by clicking "Clear". When you have finished making changes to the board, click "Done" to leave Edit mode. It will still be your turn to move next.

Mail comments and/or gripes to Ross Finlayson - rsf@diablo.

— 11 —
Fscheck: File System Checking Program

This program is a file system disk checker as well as simple file system editor that can be used to inspect and modify file system disk data structures. In addition, it gives one the capability to create and initialize new file systems. Fsheck must only be used when there is no other file system activity. It also should only be used by persons responsible for maintaining the file system.

11.1. Invocation

One can invoke fscheck from within the V system executive by typing

```
fscheck
```

or

```
fscheck devicename
```

If no device name is specified, fscheck attempts to open two devices, [device]disk0 and [device]disk1. Non-existence of a second device does not affect correct operation of the program. Note that the devices must be attached to the workstation from which the command is invoked and the kernel running on the workstation must include the proper disk driver (see the Kernel Section for details on which kernel should be booted).

11.2. Commands

Commands are provided to check the global data structure consistency of each file system, inspect and modify individual node descriptors (ND), and initialize new file systems.

a [+r] [+s] — check the consistency of the file system block allocation. If +r is specified, the bitmap is reconstructed. If +s is specified, error messages about blocks marked in the bitmap but not allocated to a file are suppressed.

b block — print the nd number of all node descriptors that point to the given block number. Normally, there is at most one. If the allocation is inconsistent, a block may be allocated more than once.

c — update the checksum in the current ND, print it, and set the current field to the checksum field.

f — print the pathname of the current ND relative to the file system being checked.

g field — set the field corresponding to the given name as the current field and print the current field.

i — initialize file system information. Prompts the user for the name, drive number, start block, and length of each file system in the disk subsystem and writes the information into the file "fstab" on the root file system. Note that the start block of the first (root) file system should correspond to the START_FD_FILE definition (usually 40) in "/V/servers/storage/storagedefs.h". Warning: this command should only be executed when new file systems are being created.

l — print all links from and to the current ND.

n <path> | <nd> — set the ND corresponding to the given pathname or number to be the current ND. Pathnames must be specified as absolute pathnames (i.e. starting with "/"). If a pathname cannot be followed, the current ND is set to the last node visited while looking up the pathname. This occurs if the node does not exist or the path from the root cannot be followed (e.g. a node in the path has a bad checksum).

p — print all the fields of the current ND.

q — quit.

s number — set the current file system to be the one indicated by number.

t — check the consistency of the file system tree structure.

w — write the current ND back to disk. The ND number is taken from the current value of the number field. If the current ND describes an allocated node (i.e. its name field is not the null string), it is written only if its checksum is correct. If it describes an unallocated node, it is written unconditionally. Checks that the number field is correct before writing the current ND out.

<RETURN> — advance to the next field and print its name and value. Hitting <RETURN> after an "n" command prints the first field.

. — print the current field.

↑ — set the current field to the previous field and print it.

= <number> | <str> — store the given number or string in the current field and print it. The number may be decimal, octal('O' prefix), or hexadecimal('$' prefix). A string is a sequence of characters and must be enclosed by double quotes. A null string is represented by "". Strings are accepted only when the name field is being modified. Note that modifications are not effective until a "w" command is issued.

11.3. Initializing a new disk subsystem

Once the disk drive(s) have been formatted (using *diskdiag*), the characteristics of each of the multiple possible file systems should be specified. This can be accomplished by creating the root file system (as described below) and subsequently running the "i" command. Then, using the "s" command to successively switch to each new file system, the rest of the file systems should be created.

11.3.1. Creating a new file system

To build a new file system, one should allocate blocks to the ND file (ND 1) and to the bitmap file (ND 2). (If the file system being created is the root file system then a single block should be allocated to "fstab" (ND 3).) This is done by modifying these node descriptors so that each refers to non-overlapping extents of disk blocks. Also, the link fields in each node descriptor should be updated so that a proper tree structure exists, i.e. ND 2 is the son of ND 1 and ND 3 is the brother of ND 2.[4] After this is done, a "t" command should be used to check the consistency of the new tree structure, and an "a +r" command must be issued so that the bitmap reflects these newly allocated blocks.

[4] In the near future, the task of creating a new file system will be automated.

11.4. Checking file system integrity

Once the "s" command has been used to set the current file system to the one you want to check, test the consistency of block allocation using the "a" command. Used with the +r option, it rebuilds the bitmap file in the case of missing blocks (i.e. blocks marked as allocated in the bitmap but not actually allocated to any file) or blocks allocated to a file, but marked as free in the bitmap. Blocks allocated more than once have to be handled manually. In this case, use the "b" command to determine to which ND's they are allocated. Use the "n" and the "f" commands to determine the pathnames of those ND's. Make copies of the conflicting files and remove the old ones. Note that the information in the files may be damaged.

Second, check the tree structure using the "t" command. If there are missing links, find out what they should be using "n" and "l". If there are nodes completely disconnected from the file system, remove them or else determine from their father pointers where they should be in the tree structure. The easiest way to remove a disconnected node is to mark the corresponding ND as unallocated (setting the name field to the null string) and then using "a +r" to recover the blocks that were allocated to that file.

— 12 —
Standalone Commands

This chapter discusses standalone programs, i.e., programs that do not run under the V kernel, that are useful with the V-System.

12.1. Vload

Vload is the V-System bootstrap loader. The Vload program loads the V kernel and initial team into memory and starts up the kernel.

There are several versions of Vload. Currently, all versions use the V I/O protocol and V IKC protocol to load programs over the Ethernet.[5] On the Sun-1, the Sun 3 Mbit Ethernet board and Excelan 10 Mbit Ethernet boards are supported as boot devices. On the Sun-1.5 and Sun-2, the 3Com 10 Mbit Ethernet board is supported.

Vload determines the files to load and other actions to take at run time, depending on what was typed on the command line and what information is stored in the configuration database for the workstation being booted (see section WORKSTATIONCONFIG). For each of its parameters, Vload gives first priority to command-line information, if any, second priority to the defaults for this workstation recorded in the configuration database, if any, and third priority to a default value determined at compile time.

Team and kernel filenames are interpreted in the V-System "[public]" context, unless they begin with a square bracket. In the latter case, the name inside brackets is taken as a machine name, in the same name space used by the *login* command. If "#" is given as the kernel file name, no kernel is loaded. Instead, the file specified as first team is loaded into the kernel's memory area and executed as a standalone program.

Besides file names, two other parameters are also required: "world" and "options." The world may be either V (production) or xV (experimental). The only option currently recognized is 'b', which causes a break to the PROM monitor before the kernel is started.

The following sections describe the defaults and special characteristics of the three versions of Vload in use at this writing.

12.1.1. 3 Mbit Ethernet

This version of Vload is intended for booting Cadline, SMI Sun-1, and other Sun-1 workstation configurations with 3 Mbit Sun Ethernet boards. These workstations ordinarily use a version of the Stanford PROM monitor that incorporates PUP bootstrap code. The first step in booting these workstations is to load Vload using the bootstrap PROMs. This can be done by typing a keyboard command (`b filename` for SMI workstations, `n filename` for others), or automatically on powerup or reset (see below).

For these workstations, the kernel resides from 0x1000 to 0x10000, and teams are loaded at 0x10000.

The compiled-in default values for Vload's parameters in this version are as follows:

world V team

[5] In the near future, there will be a version of Vload that can boot a fileserver machine directly from its local disk.

team1-vgts kernel
Vkernel/sun1+en options
null

The only command line information visible to Vload is the name it was invoked under. Therefore, Vload is installed under several different names, and its action depends on its name. The names and actions are listed below.

V	When called under this name, Vload will load the team *team1-vgts* and the default kernel for this workstation, using the default options. The team and kernel are loaded from a V storage server (production versions) rather than an xV storage server (experimental versions), that is, the *world* parameter is set to *V*.
VV	The team is *team1-sts*, and the world is *V*.
xV	The team is *team1-vgts*, and the world is *xV*.
xVV	The team is *team1-sts*, and the world is *xV*.
Vload	The user is prompted for *team*, *kernel*, and *options*. The default value is used for any field where the user enters a blank line. The world is *V*.
xVload	Same as Vload, except that the world is set to *xV*.
null	If the name is null, Vload assumes it was autobooted. Default values are used for all parameters.
others	If a copy of Vload is installed under any other name, it will use its name as the team name to be loaded, set the options to null, and use defaults for the kernel and world.

No special setup is required to get an SMI processor to autoboot—it will do so automatically 30 seconds after powerup or a k2 command. The PUP boot PROM requests boot file number 1 by number, which causes a file called 1.Boot to be loaded from the first responding PUP EFTP server. We have arranged for this file to be a copy of Vload, so the boot action is as described under the *null* name above.

A non-SMI processor can be made to autoboot by installing the proper jumpers in its configuration register. (See the *Sun User's Guide* for a full description of the configuration register.) Bits 7-4 of the configuration register are an index into a table of bootfile names stored in the PROM. An in-place jumper or closed DIP switch corresponds to a 0 bit; no jumper or an open switch corresponds to a 1. These bits should be set to the number corresponding to the name "Vload." The "W ff" command typed to the PROM monitor causes it to list the bootfile names and corresponding numbers that it knows about. Vload is usually number 5, corresponding to jumpers on bits 5 and 7. Vload's action will be as described under the *null* name above.

12.1.2. Excelan Ethernet

This version of Vload is intended for booting Cadline, SMI Sun-1, and other Sun-1 workstation configurations with Excelan 10 Mbit Ethernet boards. Ordinarily, this version of Vload is used only with workstations using a special version of the PROM monitor that incorporates TFTP bootstrap code. The first step in booting these workstations is to load Vload using the bootstrap PROMs. This can be done by typing a keyboard command, not described here.

The compiled-in default values for Vload's parameters in this version are as follows:

world V team
team1-vgts kernel
Vkernel/sun1+ex options
null

The only command line information visible to Vload is the name it was invoked under. Therefore, Vload is installed under several different names, and its action depends on its name. The names and actions are listed below.

xlnV	When called under this name, Vload will load the team *team1-vgts* and the default kernel for this workstation, using the default options. The team and kernel are loaded from a V storage server (production versions) rather than an xV storage server (experimental versions), that is, the *world* parameter is set to *V*.
xlnVV	The team is *team1-sts*, and the world is *V*.
xlnxV	The team is *team1-vgts*, and the world is *xV*.
xlnxVV	The team is *team1-sts*, and the world is *xV*.
xlnVload	The user is prompted for *team*, *kernel*, and *options*. The default value is used for any field where the user enters a blank line. The world is *V*.
xlnxVload	Same as Vload, except that the world is set to *xV*.
others	If a copy of Vload is installed under any other name, it will use its name as the team name to be loaded, set the options to null, and use defaults for the kernel and world.

There is currently no way to autoboot a workstation with TFTP boot PROMs. This limitation will be removed in the future.

12.1.3. 3Com Ethernet

This version of Vload is intended for booting Sun-1.5s and Sun-2s with 3Com 10 Mbit Ethernet boards. These workstations boot using either a local disk or tape, or the SMI network disk protocol. The network disk protocol does not allow specifying a file name, so the V-System ND boot server is only capable of loading one file—Vload. Vload, however, can read the entire command line typed by the user.

The compiled-in default values for Vload's parameters in this version are as follows:

world	V team team1-vgts kernel Vkernel/sun1.5+cc options *null*

Zero or more arguments may be passed on the command line to Vload. If the first argument to Vload is one of the special values described below, it is stripped off and the special action listed is taken. After this check, the first three remaining arguments are respectively used to override the defaults for team name, kernel name, and options. Values set by these arguments have priority over values that may have been set by the first argument.

V	Sets the world to *V*, and the team to *team1-vgts*. (This team name will be overridden by the next argument if present.)
VV	The team is set to *team1-sts*, and the world is *V*.
xV	The team is set to *team1-vgts*, and the world is *xV*.
xVV	The team is set to *team1-sts*, and the world is *xV*.
null	If no arguments are present, the default values are used for all parameters.
vmunix	The SMI boot PROMs have this name hardwired in for autobooting, so it is treated the same as a null first argument.

others If the first argument is not one of these values, the default world is used, and the arguments present specify team name, kernel name, and options, as described above.

12.2. Postmortem

Postmortem is intended to provide some help in diagnosing system deadlocks, kernel aborts, and other disastrous errors. Any time the system seems to be hanging, you can break to the PROM monitor, and type the command

```
n postmortem
```

Substitute **b** for **n** on SMI workstations.

Postmortem examines the kernel data structures left behind after a crash, and prints out the state of each process, if any exist, the pid of the currently active process, and the ready queue.

It is important *not* to use the monitor **k1** or **k2** command or press the workstation reset button before running postmortem. These actions cause memory to be cleared. The PROM monitor on a Cadlinc workstation will not operate properly if the mouse is active, but fortunately, it is possible to turn off the mouse without power cycling the workstation by unplugging the keyboard and plugging it back in. This should not be necessary if you were able to press the comma key on the numeric keyboard while the kernel was still running.

12.3. Ipwatch

The *ipwatch* family of programs provide a way of monitoring the Ethernet to debug protocol implementations or search for the cause of strange behavior. Ipwatch knows about most common types of packets seen on the Stanford network, including most PUP protocols, Internet protocols such as IP, TCP, and ICMP, XNS protocols, and the V interkernel protocol. It can print packet traces on the screen, or save them in a file. *Enwatch* is a version for the Sun 3 Mbit Ethernet board, while *ecwatch* works on the 3Com 10 Mbit board. *Exwatch* works with the Excelan 10 Mbit board.

To run enwatch, reset the workstation completely, and type the command

```
n enwatch
```

for Cadlinc workstations, or

```
b enwatch
```

for SMI workstations. The program is menu driven, and most options are self-explanatory.

Currently, all versions of *ipwatch* use the PUP Leaf protocol on the 3 Mbit Ethernet to write packet traces to files, and they run only on the Sun-1 with Stanford PROMs.

12.4. Diskdiag

The *diskdiag* program is a diagnostic program that allows one to manually access specific sectors on the disk. It is useful for verifying the correct interaction between the disk controller and disk drives, as well as for initializing a new disk. Diskdiag is configured to run on a system with a Xylogics 450 or Interphase 2181 disk controller and Fujitsu M2351 and M2284 disk drives.

To run diskdiag, type the command

```
b ec() diskdiag #[6]
```

[6]Some SMI workstations with older PROM revisions require that nd() be used in place of ec().

for SMI workstations, or

```
n diskdiag
```

for Cadline workstations. There are commands available to **format(f)**, **read(r)**, **seek(s)**, and **write(w)**. The user is prompted, as necessary, for more information on each of these commands.

In addition, it is possible to **label(l)** a drive with the configuration parameters needed by the disk driver in the kernel. Executing the format command automatically labels the disk after the format is complete. The **verify(v)** command reads the label off of disk and prints it on the console.

The **partition(p)** command prompts the user for the start block and length of each partition on the disk and creates a disk partition table. Existence of a disk partition table is optional as it is not needed by any system software. The **examine(x)** command allows one to examine the contents of the disk partition table.

Reinitializing the diskdiag program is accomplished using the **again(a)** command.

12.5. Offload and Offload38

The *offload* program uses PUP EFTP to load standalone programs into a Sun-1 equipped with a Sun 3 Mbit Ethernet interface, at a user-specified memory location. This program is useful on Sun-1 workstations equipped with standard Stanford PUP boot PROMs, because they are only capable of loading programs that reside at the default address of 0x1000.

To use offload, first reset the workstation, then give the command

```
n offload
```

to the Sun PROM monitor. (Substitute 'b' for 'n' on SMI workstations.) The program will prompt for

1. The name of the program to be loaded. The default directory is the miscserver's standard default directory, as described under Vload.
2. The load origin of the program, in hex. This should be the same value specified to cc68 or ld68 with the -T option when the program was linked. Offload will refuse to load a program that would overlap part of the memory it uses; use offload38 if this is a problem (see below).
3. Where to put a copy of the program's b.out header. This is usually not needed; enter '0' to omit it.
4. Whether to load the program's symbol table into memory. This is generally not needed. See the *Sun User's Guide* for a description of how program symbol tables appear in memory.
5. Whether to jump to the program's entry point or return to the PROM monitor after the program is loaded. After returning to the monitor, the command

   ```
   g 1000
   ```

   will restart offload to load another file.

Offload itself resides at 0x1000 so that it can be loaded by the PROM monitor. If it is necessary to load a program that would overlap offload's default location, use offload to load *offload38* at 0x38000. This program is identical to offload except for its starting address. The command

```
g 38000
```

will restart offload38 after a return to the monitor.

The following dialog can be used to load a nonstandard kernel that is too large for Vload. User input is underlined.

```
>n offload
Sun Offset Loader - Version 2.2 - 2 Feb 1983
Loader resides from 1000 to 60e8
Program to load: offload38
Origin (hex): 38000
Place b.out header at (hex; 0 if not needed): 0
Load symbols? (y/n): n
Execute? (y/n): y

Sun Offset Loader - Version 2.2 - 2 Feb 1983
Loader resides from 38000 to 3e0e8
Program to load: /usr/sun/Vboot/team1-sts
Origin (hex): 10000
Place b.out header at (hex; 0 if not needed): ffe0
Load symbols? (y/n): n
Execute? (y/n): n
>g 38000

Sun Offset Loader - Version 2.2 - 2 Feb 1983
Loader resides from 38000 to 3e0e8
Program to load: your nonstandard kernel
Origin (hex): 1000
Place b.out header at (hex; 0 if not needed): ffc0
Load symbols? (y/n): n
Execute? (y/n): y
```

Using "/usr/sun/Vboot/team1-sts" as above loads the standard version of the plain exec. You can substitute team1-vgts or your own special first team.

Part II:
Program Environment

— 13 —
Program Environment Overview

This manual, the *V-System Program Environment Manual* describes the execution environment provided for C programs written to run in the V system (and in particular the V kernel), primarily for programs in the C language. This program environment is designed to minimize the difficulty of porting C programs (and C programmers) from other C program environments, such as that provided by UNIX[7], and to provide access to the distributed process and message facilities provided by the V kernel and V servers.

The program environment consists of three major components:

- The base C language implemented by the compiler.
- Routines that are part of the C program library in most C implementations.
- Functions that access V facilities.

The basic C language is not described here. The reader is referred to *The C Programming Language* by B. W. Kernighan and D. M. Ritchie, Prentice-Hall 1978 for a tutorial on the language and standard C library routines.

Standard C library routines are only described here to the degree they differ in the V program environment from other implementations, particularly the Unix C library. The reader is referred to the above-cited book or *The Unix Programmer's Manual* for details on these standard functions.

The V-specific functions are described in detail in the following chapters.

While there has been a strong attempt to provide a superset of the standard C program environment, there is no real definition of "the standard C program environment." While C as a programming language does not define I/O facilities, memory management, etc., an ill-defined de facto standard has arisen from the extensive use of C with the Unix operating system. Attempts to port C programs have resulted in a slightly more portable standard program environment than originally used with Unix. However, there is not, to the authors' knowledge, a definition of what a portable C program can reasonably expect of its program environment. The functions included in the V program environment for C, excluding V and SUN workstation specific routines, constitute our proposal for such a standard portable C program environment.

The differences between the V C program environment and the Unix C program environment fall into four major categories

- Functions that are Unix system calls which may be provided as V library routines, e.g., stime().
- Functions that are slightly changed in their implementation, but provide (essentially) the same functionality, e.g., malloc().
- Functions that are SUN workstation-specific, because they are not necessary in standard Unix on, say, a VAX[8]. For example, the long division routines are in this category, as are the emulator traps,
- Functions that are particular to the V-System, like Create() and Ready().

---

[7] UNIX is a trademark of Bell Laboratories.

[8] VAX is a trademark of Digital Equipment Corporation.

13.1. Groups of Functions

The description of functions is structured by subdividing them according to functional groups as follows.

emt	C language interface to the on-board PROM monitor emulator traps. See the *Sun User's Guide* for more information.
exec	V-System program execution functions.
fields	Functions that enable a pad to be used as a menu, similar to a data entry terminal.
io	Input/output related routines.
math	Mathematical functions.
mem	Memory management and allocation routines.
naming	V-System name management functions.
numeric	Arithmetic and numeric functions.
process	V-System process service functions and V kernel traps.
strings	Character string manipulation routines.
time	Clock and time conversion services.
vgts	Virtual Graphics Terminal Service interface routines.
others	Miscellaneous other functions.

This functional subdivision is also reflected in the structure of the program source for the V C library, where every subdivision corresponds to a subdirectory of the C library directory.

13.2. Header Files

The following header files define manifest constants, type definitions and structs used as part of the V C program environment. They are included as usual by a "#include <headername>" directive in C programs.

Venviron.h	Standard header file for V kernel types and request/reply codes.
Vethernet.h	Ethernet-specific header information. This is very low-level information; most users will want to use the network server instead.
Vexceptions.h	Exception types and exception request format.
Vgts.h	Virtual graphics terminal server interface. This should be included in any programs that do graphics.
Vio.h	I/O Protocol header file. Types and mode constants for file manipulation functions described in chapter of this manual.
Vmouse.h	Mouse device-specific header information. Most programs will use the Vgts to handle graphics input.
Vnet.h	Network server definitions. This is included in any programs that use the network.
Vprocess.h	Processor state structure and other process-specific header information.
Vserial.h	Manifests for the serial lines. Again, very low level for most users; use the higher level library interface instead to be more portable.

Vsession.h	Manifests and message structs for session services. These are remote servers, often called Unix or V servers, that provide transparent file access over a network.
Vteams.h	Team header file. Structures used to communicate with the team server and to pass information to teams when they are created.
Vtime.h	Structures used in time services, primarily for getting time from a session server.

— 14 —
Program Construction and Execution

A V-System C program is constructed and executed similar to a C program on Unix.

14.1. Writing the C Program

An application program on the V-System starts to execute as a single process[9], with priority 4. It is allocated an initial stack area of about 4000 bytes, just above its uninitialized data segment. If this is not large enough, one of the first actions of the team's root process should be to use the Create() library function to create processes with larger stacks.

Note that large dynamically allocated areas of memory should be allocated using malloc, calloc, or a similar memory allocator, and not be allocated on the process stack. **Warning:** There is no run-time checking for overflowing the process stack allocation. The program behavior from stack overflow can be sufficiently bizarre as to cause good programmers to seek refuge in monasteries. If the stack overflow caused the process in question to get an exception, the standard exception handling routine will usually detect the overflow and print a message; however, not all stack overflows cause an exception in the process that generated them, and sometimes the stack is back in bounds by the time the exception occurs.

The file Venviron.h is a header file defining the types and constants that arise as part of the interface to the kernel. It is included by the line

```
#include <Venviron.h>
```

Other V header files, listed in the previous section, are included similarly.

14.2. Compiling and Linking

When the application program is compiled and linked, references to kernel operations and other standard routines must be resolved by searching the library file libV.a (kept in /usr/sun/lib on Stanford Unix systems). The application must be relocated so that its text segment starts at 0x10000. These defaults are automatically selected with the -V option of the cc68 command. The compile command:

```
cc68 -V -r programfile
```

produces a .r file for running with the kernel. The program environment provided by the libV.a library is given in the later sections of this manual.

14.3. Program Execution

There are two models for executing V C programs, namely: using the V executive and bare kernel mode.

---

[9] For a complete discussion of processes, message passing, and other services provided by the V kernel, see the kernel manual.

14.3.1. Execution With the Executive

Use of the V executive is described in the V-System commands manual. Basically, one types the name of the file containing the program to the command interpreter followed by zero or more command arguments. The program is then loaded and executed.

When the V executive is used, the program execution begins at a procedure called **main()**, passed a count of the number of arguments to the program and an array of pointers to the program string arguments, as given on the command line. Each new team is passed standard input, output, and error files through the TeamRoot message.

The following example shows how a program can read its command line arguments. The variable **argc** contains the number of arguments including the command name. The arguments are kept in **argv[0]** through **argv[argc-1]**; the command name is **argv[0]**, **argv[1]** is the first argument, **argv[argc-1]** is the last argument, and **argv[argc]** is NULL. This matches the Unix convention.

```
main( argc, argv )
    int argc;
    char *argv[];
    /* Echo arguments */
  {
    int i;

    for( i = 0; i < argc; ++i )
        printf( "%s ", argv[i] );
    putchar("\n");
  }
```

The executive sets the new team's team priority to 30.

14.3.2. Bare Kernel Mode

In bare kernel mode, execution also begins at main(), but no arguments are available.

None of the standard servers ordinarily included in the V executive are available, unless the program includes one or more of them itself (as described in the V servers manual).

A program to be executed in bare kernel mode is loaded by a special loader program called *Vload*:

```
n Vload
```

typed to the SUN monitor causes it to load and execute the loader. (Use **b** in place of **n** on SMI workstations.) *Vload* then prompts for the name of a file containing the program. The use of this loader is described more fully in the *Standalone* chapter of the V commands manual.

14.4. The Team Root Message

Each team is passed a team root message at the time it is started. This is the message passed to the team by the **Reply()** call that sets it running. The team root message is a structure of type RootMessage, as defined in the standard header file <Vteams.h>. A function called **TeamRoot()** (automatically included in every program by the -V option of cc68) receives the team root message, stores a copy of the team root message in an area pointed to by the global variable **RootMsg**, initializes the team's standard i/o, and calls **main()**. If **main()** returns, **TeamRoot()** calls **exit()**. The team root message can be accessed from within a team (not usually necessary) by declaring it as

```
extern RootMessage *RootMsg;
```

The team root message contains the following fields:

stdinserver	Process id of the server providing this team's standard input file.
stdoutserver	Process id of the server providing this team's standard output file.
stderrserver	Process id of the server providing this team's standard error file.
stdinfile	Instance id of this team's standard input file.
stdoutfile	Instance id of this team's standard output file.
stderrfile	Instance id of this team's standard error file.
rootflags	A set of flags indicating whether the team is to overwrite or append to its standard output and standard error, whether standard input, output, or error have been redirected, and whether it is to release its standard input, output, or error instances upon exit.
nameserver	Process id of the server providing the team's initial *current context* (i.e., current working directory).
contextid	The context id of the team's initial current context.
kernelpid	Process id to which the team is to send to obtain secondary kernel services (see the V kernel manual). Normally the same as the team creator's kernel pid, provided the new team is running on the same workstation as its creator.

14.5. The Per-Process Area

Each process has a region of team memory reserved for its own use, called its *stack space*. On the Sun, a process's stack grows downward from the highest address in this region. A portion of the stack space, called the *per-process area*, is used to store a few process-global variables. On the Sun, this area begins at the lowest address of the stack region. A team-global variable called **PerProcess** points to this area. It is reset by the kernel to point to the correct area on every process switch.

The standard per-process area is described by the PerProcessArea structure in the header file <Vio.h>. It contains the following values:

stdio	An array of three File pointers describing the process's standard input, output, and error files. <Vio.h> defines the macros **stdin**, **stdout**, and **stderr** to be **PerProcess->stdio[0]**, **PerProcess->stdio[1]**, and **PerProcess->stdio[2]** respectively. Note that only pointers, not the File structures themselves, are kept in the per-process areas.
nameserver	Process id of the server providing the process's current context.
contextid	Context id of the process's current context.
stackSize	The size of the process's stack space, in bytes.

The **TeamRoot()** function initializes the team root process's per-process area from the values passed in the team root message. The **Create()** library function, used to create new processes, initializes each new process's per-process area to be a copy of that of its creator (except for the stackSize field). This causes each child process to inherit its creator's standard I/O and current context.

— 15 —
The V-System Configuration Database

15.1. Introduction

When a diskless workstation boots up, it has a limited amount of information about its own configuration and identity. A boot program can probe to see what devices are attached, and some workstations may have configuration registers, additional switches, or a small amount of nonvolatile memory. If a workstation has an Ethernet board, there will be a PROM or DIP switch on the board containing its Ethernet address. There may also be some machine-specific information in PROMs on the processor board. If the workstation is booted by typing a command, rather than automatically on power-up, the user may be asked to type in some information.

From this information, the workstation software needs to deduce several things, including at least:

1. What version of the kernel to load (68000, 68010).
2. Which Ethernet board to use for interkernel communication, if there is more than one.
3. What to run as the initial team.
4. Whether to run the VGTS, the STS, or some other program as the terminal server.
5. What commands to execute before turning control over to the user, if any. (For example, we may wish to run a print server on this workstation, or automatically bring up an internet server in the background.)
6. What Internet address to use for this workstation.
7. What the name of this workstation is (e.g., SUN-MJ402).
8. What type of terminal is connected to the workstation, if the STS is to be used.

In general, there is no reliable algorithm for determining most of these things. In fact, many are the result of essentially arbitrary human decisions — for example, the workstation name.

15.2. Configuration Database

As a solution to this problem, the V-System maintains a configuration database, containing information about each workstation. The information is organized as sets of keyword/value pairs, one per workstation.

There is one standard library function provided for extracting information from the configuration database:

```
SystemCode QueryWorkstationConfig(keyword, value, maxlength)
    char *keyword, *value;
    int maxlength;
```

Given a character string representing the keyword, this routine returns the corresponding value as another character string. The variable **keyword** points to the keyword, **value** points to the place to put the value, and **maxlength** is the size of the buffer, which should include space for a terminating null byte. The routine returns a system error code if there is no configuration information recorded for the querying workstation

(NOT_FOUND), there is some configuration information, but no value corresponding to the given keyword (BAD_ARGS), or the buffer was too short to hold the value (BAD_BUFFER), else returning OK. In the buffer-too-short case, it will return as much as there is room for. In unusual situations, other error codes may be generated; these can be treated as failures or considered equivalent to NOT_FOUND.

15.3. Implementation

Ordinarily, programs should not be aware of the implementation of the configuration database; this implementation may change in the future. The QueryWorkstationConfig() function should be the only interface used. Since there is no standard library function provided to modify the configuration database, however, system maintainers need to be aware of its implementation. The current implementation allows the configuration database to be modified with an ordinary text editor, and the changes installed with the same tools that are used for installing new binary program images on storage servers.

The V configuration database is currently implemented as a set of *configuration files*, one for each workstation. Each configuration file must be available on every publically-available V storage server.[10] requests from nonlocal clients.)

The name of each workstation's configuration file is derived from its hardware Ethernet address—a convenient unique identifier.[11] The files are kept in a subcontext named "config", under the server's public context. (See section 30.) For a workstation with Ethernet address 0260.8c01.9954 (a typical 3Com-assigned address), the configuration file could then be read by a workstation as a file named "[public]config/C.02608c019954"; this is in fact how QueryWorkstationConfig() is implemented.

A configuration file is an ASCII text files, consisting of a set of keyword/value pairs, arranged in no particular order. Each keyword appears at the beginning of a new line, and is separated from its corresponding value by a colon (':'). A line beginning with a colon serves as a continuation of the value on the previous line. This format has been designed to be easy to read and easy to parse. (Note that spaces both before and after the colon may be considered significant by programs, so take care when creating or editing config files.)

At Stanford, the master copies of configuration files are kept in the directory /xV/config on Pescadero, and only those copies should be edited. The command "make install" (run as user *ds*) is used to install changes.

Currently Defined Keywords

The following keywords are in use at this writing. A list of keyword names and their meanings is presently kept in the same directory as the config files themselves, in a file called "keywords."

name — The name of this workstation. Should match the name used in local IP name tables for this workstation's IP address. There is no default.

ip-address — The workstation's Internet Protocol address, given in the conventional [a.b.c.d] notation, where a, b, c, and d are decimal integers. On the 3 Mbit Ethernet, the default value of d is the 8-bit Ethernet host address, while default values of a, b, and c are determined by the Internet server. For 10 Mbit Suns, this keyword should always be present.

ip-gateways — Name of a file containing a list of Internet gateways to be used by this workstation. The file name is given relative to the standard [public] context. If this keyword is omitted, the Internet server will not forward datagrams through any gateways, i.e., only local traffic will

[10] Publically-available storage servers are defined as those that respond to GetPid(STORAGE – SERVER, ANY – PID

[11] Currently, on Sun-2 workstations with 3Com Ethernet interfaces, the address assigned to the Ethernet board is used, not the address assigned to the processor.

be supported.

kernel — Filename of the program to be loaded as the kernel, for use by Vload. The name is given relative to the standard [public] context. If this keyword is omitted, Vload uses a compiled-in default, currently Vkernel/sun1+en for the 3 Mbit Ethernet version, Vkernel/sun2+ec for the 10 Mbit.

team — Filename of the first team, as above. If it is omitted, Vload uses a compiled-in default, currently team1-vgts.

world — Either V or xV. Used by Vload. If omitted, Vload uses a compiled-in default, currently V.

boot-options — Boot options for use by Vload. Currently the only option is **b**, meaning "break before starting kernel." The default is a null string.

startup-script — Filename of the startup script. Currently used only by team1-server, for workstations that autoboot as servers. No default. In the future, the definition of this keyword will be changed to allow the startup script to be placed directly in the config file, and all (or most) versions of the first team will use it.

alt-ether-addr — Alternate ethernet addresses for this workstation, one per line. These are addresses the workstation may use, other than the one the config file is named for. 10 Mbit addresses should be given in hexadecimal, in the form xxxx.yyyy.zzzz. 3 Mbit addresses may be given in octal. The default is null. This keyword must be present for use by the Vax Unix ND server for workstations that boot using the ND protocol under a different Ethernet address than the one the config file is named for. This is true of SMI Sun-2's with PROM revision N or later.

ndboot — The Vax Unix ND boot server looks for a configuration file when deciding whether it should answer boot requests, and will refuse to respond if there is none or it contains the line "ndboot:no". (This procedure allows our ND server to coexist with SMI Network Disk servers on the same net.) Thus, the default value for this option is "yes" if a config exists for this workstation, otherwise "no."

terminal-type — Type of terminal used as a console. Used by the STS. The default is to assume the Stanford PROM terminal emulator for Cadlines, or something ANSI-compatible (like the SMI PROM terminal emulator) otherwise. The only other recognized value for this option is "h19".

Usage

In general, we have implemented programs that use this service in such a way that if a configuration file or specific keyword/value pair is missing, some reasonable default is used where this is possible. Also, where it is easy to reliably determine something by examining the hardware present, it is best to do that instead of putting the information in the configuration file. Following these principles means that fewer updates to the configuration files are needed to keep workstations running correctly when something changes.

In some cases, the value of a keyword may be the name of a file, perhaps because it is more convenient for the client to read the information from a file, or because the information associated with the keyword is quite bulky. In the present implementation, such files are kept in the "[public]config/" directory along with the configuration files themselves. Files whose names begin with "S." are startup command scripts for workstations that boot automatically. Files whose names begin with "G." are gateway information files used by the internet server.

— 16 —
Input and Output

The input and output routines can be divided into three categories:

1. Basic I/O routines like `getchar()` that are supported but differ in their implementation from the standard Unix versions.
2. I/O support routines like `printf()` that are identical with the standard Unix version.
3. V-specific I/O routines like `Read()` and `Write()` that are used in several cases to implement the standard C routines in the V message-based world.

16.1. Standard C I/O Routines

The following standard C I/O routines are available:

```
chdir()     clearerr()  fclose()   feof()
ferror()    fflush()    fgetc()    fgets()
fopen()     fprintf()   fputc()    fputs()
fread()     freopen()   fscanf()   fseek()
ftell()     fwrite()    getc()     getchar()
gets()      getw()      printf()   putc()
putchar()   puts()      putw()     rewind()
scanf()     sprintf()   setbuf()   sscanf()
ungetc()
```

However, `fopen()` returns a pointer value of type *File, where File is defined in <Vio.h> and is a totally different record structure from that used by, for instance, the Unix standard I/O. Also, `setbuf()` is a no-op under V.

16.2. V I/O Conventions

Program input and output are provided on files, which may include disk files, pipes, mail-boxes, terminals, program memory, printers, and other devices.

To operate on a file, it is first "opened" using `Open()` if the file is specified by a pathname, otherwise by `OpenFile()` if the file is specified by a server and instance identifier. The mode is one of the following:

FREAD	No write operations are allowed. File remains unchanged.
FCREATE	Any data previously associated with the described file is to be ignored and a new file is to be created. Both read and write operations may be allowed, depending on the file type described below.
FAPPEND	Data previously associated with the described file is to remain unchanged. Write operations are required only to append data to the existing data.
FMODIFY	Existing data is to be modified and possibly appended to. Both read and write operations are allowed.

Both open functions return a pointer to an open file descriptor that is used to specify the file for subsequent operations. `Close()` removes access to the file. `Seek()` provides random access to the byte positions in the file. Note: the value returned from a byte position that has not been written is not defined.

Each program is executed with standard input, output and error output files, referred to as **`stdin`**, **`stdout`**, and **`stderr`** respectively.

The file type indicates the operations that may be performed on the open file as well as the semantics of these operations. The file type is specified as some combination of the following attributes.

READABLE The file can be read.

WRITEABLE The file can be written.

APPEND_ONLY
Only bytes after the last byte of the data previously associated with the file can be written.

STREAM All reading or writing is strictly sequential. No seeking is allowed. A file instance without the STREAM attribute must store its associated data for non-sequential access.

FIXED_LENGTH
The file instance is fixed in length. Otherwise the file instance grows to accommodate the data written, or the length of the file instance is not known as in the case of terminal input.

VARIABLE_BLOCK
Blocks shorter than the full block size may be returned in response to read operations other than due to end-of-file or other exception conditions. For example, input frames from a communication line may differ in length under normal conditions.

With a file instance that is VARIABLE_BLOCK, WRITEABLE, and not STREAM, blocks that are written with less than a full block size number of bytes return exactly the amount written when read subsequently.

MULTI_BLOCK Read and write operations are allowed that specify a number of bytes larger than the block size.

INTERACTIVE The open file is a text-oriented stream. It also has the connotation of supplying interactively (human) generated input.

Not all of the possible combinations of attributes yield a useful file type.

Files may also be used in a block-oriented mode by specifying FBLOCK_MODE as part of the mode when opening the file. No byte-oriented operations are allowed on a file opened in block mode.

See the *V-System Servers Manual* for more details on the semantics of the various possible file types and modes.

16.3. V I/O Routines

16.3.1. Opening Files

```
File *Open(pathname, mode, error)
    char *pathname;  unsigned short mode;  SystemCode *error;
```

Open the file specified by **`pathname`** with the specified mode and return a file pointer for use with subsequent file operations.

mode must be one of FREAD, FCREATE, FAPPEND, or FMODIFY, with FBLOCK_MODE if block mode is required. If **Open()** fails to open the file, it returns NULL and the location pointed to by **error** contains a standard system reply code indicating the reason. If an error occurs and **error** is NULL, **Open()** calls **abort()**.

```
File *OpenFile(server, instanceidentifier, mode, error)
    ProcessId server;  InstanceId instanceidentifier;
    unsigned short mode;  SystemCode *error;
```

Open the file instance specified by the **server** and **instanceidentifier** arguments and return a file pointer to be used with subsequent file operations.

mode must be one of FREAD, FCREATE, FAPPEND, or FMODIFY, with FBLOCK_MODE if block mode is required. If the instance is to be released when **Close()** is called on this file pointer, FRELEASE_ON_CLOSE must also be specified as part of the mode. If **OpenFile()** fails to open the file, it returns NULL and the location pointed to by **error** contains a standard system reply code indicating the reason. If an error occurs and **error** is NULL, **OpenFile()** calls **abort()**.

```
File *_Open(req, mode, server, error)
    CreateInstanceRequest *req;  unsigned short mode;
    ProcessId server;  SystemCode *error;
```

Open a file by sending the specified I/O protocol request message **req** to the server specified by **server** and return a file pointer to be used with subsequent file operations. This function is only used when additional server-dependent information must be passed in the request message, or the file is to be opened on a server that cannot be specified by a character string pathname as in **Open()**.

The request **req** may be either a CreateInstanceRequest or a QueryInstanceRequest. **mode** must be one of FREAD, FCREATE, FAPPEND, or FMODIFY, with FBLOCK_MODE if block mode is required. If **_Open()** fails to open the file, it returns NULL and the location pointed to by **error** contains a standard system reply code indicating the reason. If an error occurs and **error** is NULL, **_Open()** calls **abort()**.

```
SystemCode CreateInstance(pathname, mode, req)
    char *pathname;  unsigned short mode;  CreateInstanceRequest *req;
```

Open the file specified by **pathname** in the given mode using the specified CreateInstanceRequest, but do not set up a File structure for it. A CreateInstanceReply is returned at the location pointed to by **req**. The function returns a standard system reply code, which will be OK if the operation was successful.

16.3.2. Closing Files

```
Close(file)
    File *file;
```

Remove access to the specified file, and free the storage allocated for the File structure and associated buffers. If the file is WRITEABLE and not in FBLOCK_MODE, the output buffer is flushed.

```
SpecialClose(file, releasemode)
    File *file;  unsigned releasemode;
```

Close the specified file, as in **Close()**. If **SpecialClose()** releases the file instance associated with the

specified File structure, the release mode will be set to **releasemode**. **Close()** sets the release mode to zero. See the I/O protocol section of the V servers manual for a explanation of release modes.

```
ReleaseInstance(fileserver, fileid, releasemode)
    ProcessId fileserver;  InstanceId fileid;  unsigned releasemode;
```

Close the file instance specified by **fileserver** and **fileid**, using the specified release mode. This function is used only when there is no File structure for the given file.

16.3.3. Byte Mode Operations

The standard Unix functions mentioned above may be used on files opened in byte mode (i.e., not opened in FBLOCK_MODE). Several other functions are also available on such files, as described below.

```
int Seek(file, offset, origin)
    File *file;  int offset, origin;
```

Set the current byte position of the specified open file to that specified by **offset** and **origin** and return TRUE (nonzero) if successful.

If **origin** is ABS_BLK or ABS_BYTE, the byte position is set to the **offset**-th block or byte in the file starting from 0. If **origin** is REL_BYTE, **offset** specifies a signed offset relative to the current byte position. If origin is FILE_END, **offset** is the signed byte offset from the end of file.

If the file is FIXED_LENGTH, an attempt to seek beyond the end of file causes **Seek** to return FALSE and the byte position to remain unchanged. The end of file position is one beyond the last byte written. The value of bytes in the file previous to the end of file that have not been explicitly written is undefined.

Seek() may not be used on files opened in block mode. **SeekBlock()** should be used on such files. **Seek()** is identical to **fseek()**.

```
unsigned BytePosition(file)
    File *file;
```

Return the current byte position in the specified file. The value returned is correct only if the current byte position is less than MAX_UNSIGNED. This function is identical to **ftell()**.

```
Flush(file)
    File *file;
```

Flush any buffered data associated with the file, providing it is WRITEABLE. Flushing a file causes local buffered changes to the file data to be communicated to the real file. If the file is in block mode or not WRITEABLE, no action is performed. This function is identical to **fflush()**.

```
Resynch(file)
    File *file;
```

Resynchronize the next block to read and write in the file with the server. Any buffered bytes are lost. This operation is only valid for streams, and is only needed when there is more than one File structure associated with a single file instance. This will happen, for example, if two teams are sharing the same standard output. Normally it should not be needed for files used in a single team.

```
SystemCode Eof(file)
    File *file;
```

Any of the byte mode read or write operations may return EOF (Exception On File) as a special value indicating an inability to read or write further in the file. Eof returns a standard system reply code indicating the nature of the exception. This may be a true end-of-file, i.e., the current byte position exceeds the last byte position of the file, or some type of error.

```
ClearEof(file)
    File *file;
```

Clear the exception on the specified file. This only clears the local record of the exception; it does not affect the circumstances that caused the exception to occur. See **Eof()**.

```
int BufferEmpty(file)
    File *file;
```

Test whether or not a file's local buffer is empty. If this function returns TRUE (nonzero), the next **getc()** will cause an actual read. If it returns FALSE (zero), the next **getc()** will return immediately with a byte from the buffer.

16.3.4. Block Mode Operations

The following functions are most useful on files opened in block mode. Unless otherwise noted, they may also be used on files opened in byte mode.

```
unsigned Read(file, buffer, bytes)
    File *file;  char *buffer;  unsigned bytes;
```

Read the specified number of bytes from the file starting at the beginning of the current block location of the file and store contiguously into the byte array starting at **buffer**, returning the actual number of bytes read.

If the number of bytes read is less than the number of bytes requested, the reason is indicated by the standard reply code returned by **FileException()**. The number of bytes requested may not be more than the block size of the file (returned by **BlockSize()**) unless the file has the type attribute MULTI_BLOCK. **Read()** is intended for use on files opened in block mode only. Note: **Read()** does *not* increment the current block number stored in the File structure for the given file.

```
unsigned Write(file, buffer, bytes)
    File *file;  char *buffer;  unsigned bytes;
```

Write the specified number of contiguous bytes from the buffer to the file starting at the beginning of the current block location of the file, and return the actual number of bytes written.

The number of bytes to be written must be less than or equal to the block size (as returned by **BlockSize()**) unless the file has the type attribute MULTI_BLOCK. If the number of bytes written is less than the number of bytes requested, the reason is indicated by the standard reply code returned by **FileException()**.

Write() should be used only on files opened in block mode. Note: **Write()** does *not* increment the current block number stored in the File structure for the given file.

```
unsigned BlksInFile(file)
    File *file;
```

Return the number of blocks in the specified file. Meaningful if the file is FIXED_LENGTH or is a WRITEABLE, non-VARIABLE_BLOCK, STREAM file.

```
unsigned BlockPosition(file)
    File *file;
```

Return the current block position in the specified file.

```
SeekBlock(file, offset, origin)
    File *file;  int offset;  int origin;
```

Set the current block position of the specified open file to that specified by **origin** and **offset**. The new block position is the block offset from the specified block origin. **origin** is one of FILE_BEGINNING, FILE_END or FILE_CURRENT_POS.

```
unsigned BlockSize(file)
    File *file;
```

Return the block size in bytes of the specified file.

```
unsigned FileException(file)
    File *file;
```

Return the standard reply code indicating the last exception incurred on the specified file. This is used primarily on files opened in FBLOCK_MODE. **Eof()** is used on byte-oriented files.

16.3.5. Server-Specific Operations

This section describes routines in the I/O library which are specific to particular servers.

```
SystemCode CreatePipeInstance(readOwner, writeOwner, buffers, reply)
    ProcessId readOwner, writeOwner;  int buffers;
    CreateInstanceReply *reply;
```

Interact with the pipe server to create a pipe, with the specified owners for the reading and writing ends of the pipe, and the specified number of buffers. **buffers** should be between 2 and 10 inclusive. The reply to the create instance request is returned at the location pointed to by **reply**; it contains the file instance id of the writeable end of the pipe. The id of the readable end is equal to this value plus 1. **OpenFile()** may be used to set up File structures for either or both ends of the pipe. **CreatePipeInstance()** returns a standard system reply code, which will be OK if the operation was successful.

```
File *OpenTcp(localPort, foreignPort, foreignHost, active,
        precedence, security, error)
    unsigned short localPort, foreignPort;  unsigned long foreignHost;
    int active, precedence, security;  SystemCode *error;
```

Interact with the Internet server to create a TCP network instance, and return a pointer to a File structure opened in byte mode that can be used to send data on the corresponding TCP connection.

To obtain a second File structure that can be used to read from the connection, use the call

```
f2 = OpenFile(FileServer(f1), FileId(f1) + 1,
        FREAD + FRELEASE_ON_CLOSE, &error)
```

where **f1** is the value returned by **OpenTcp()**. Note that it is necessary to release both the readable and writeable instances to cause the connection to be deallocated. Releasing the writeable instance closes the caller's end of the connection. Data can still be read from the readable instance until it is released, or other end closes (resulting in an END_OF_FILE indication).

The parameters **localPort**, **foreignPort**, and **foreignHost** specify the sockets on which the TCP connection is to be opened. **active** specifies whether the connection should be active (i.e., send a connection "syn" packet), or passive (i.e., listen for an incoming "syn" packet). **precedence** and **security** specify the precedence and security values to be used for the connection. Specifying zero for these parameters will cause appropriate default values to be used.

If the open is unsuccessful, **OpenTcp()** returns NULL, and a standard system reply code indicating the reason for failure is returned in the location pointed to by **error**; else OK is returned in this location.

```
File *OpenIp(protocol, error)
    char protocol;  SystemCode *error;
```

Interact with the Internet server to create an IP network instance, and return a pointer to a File structure opened in block mode that can be used to write IP packets to the network.

To obtain a second File structure that can be used to read IP packets, use the call

```
f2 = OpenFile(FileServer(f1), FileId(f1) + 1,
        FREAD + FBLOCK_MODE + FRELEASE_ON_CLOSE, &error)
```

where **f1** is the value returned by **OpenIp()**. Note that it is necessary to release both the readable and writeable instances even if only one of them is used.

The **protocol** specifies which value of the *protocol* field in the IP packet headers is of interest. The readable instance will only return packets with the requested protocol value, and the client program should only write packets with the specified protocol field to the writeable instance, though this is not currently checked by the server. If **protocol** is zero, it specifies "promiscuous" mode, in which all IP packets are returned which are not of protocol types that have been requested by another client, and packets of any protocol type may be written.

If the open is unsuccessful, **OpenIp()** returns NULL, and a standard system reply code indicating the reason for failure is returned in the location pointed to by **error**; else OK is returned in this location.

```
File *OpenPup(socket, error)
    unsigned long socket;  SystemCode *error;
```

Interact with the Internet server to create a PUP network instance, and return a pointer to a File structure opened in block mode that can be used to write PUPs to the network.

To obtain a second File structure that can be used to read PUPs, use the call

```
f2 = OpenFile(FileServer(f1), FileId(f1) + 1,
        FREAD + FBLOCK_MODE + FRELEASE_ON_CLOSE, &error)
```

where **f1** is the value returned by **OpenPup()**. Note that it is necessary to release both the readable and writeable instances even if only one of them is used.

The **socket** parameter specifies which value of the *socket* field in the PUP headers is of interest. The

readable instance will only return packets sent to the requested socket, and the client program should only write packets with the specified source socket to the writeable instance, though this is not currently checked by the server. If **socket** is zero, it specifies "promiscuous" mode, in which all PUPs are returned which are not to sockets that have been requested by another client, and packets with any source socket number may be written.

If the open is unsuccessful, **OpenPup()** returns NULL, and a standard system reply code indicating the reason for failure is returned in the location pointed to by **error**; else OK is returned in this location.

16.3.6. Miscellaneous I/O Functions

```
InstanceId FileId(file)
    File *file;
```

Return the file instance identifier associated with the open file. This was either generated as part of **Open()** or specified as an argument to the **OpenFile()** operation that opened the file.

```
ProcessId FileServer(file)
    File *file;
```

Return the file server identifier associated with the open file. This was either generated as part of **Open()** or specified as an argument to the **OpenFile()** operation that opened the file.

```
unsigned FileType(file)
    File *file;
```

Return the file type, which indicates the operations that may be performed on the open file as well as the semantics of these operations.

```
unsigned Interactive(file)
    File *file;
```

Return TRUE (nonzero) if the file has the type attribute INTERACTIVE, else FALSE (zero).

```
File *OpenStr(str, size, error)
    unsigned char *str;  unsigned size;  SystemCode *error;
```

Make the specified string look like a file. The file is FIXED_LENGTH, with one block of size **size**, and the end of file set to the end of this block. **str** must point to an area at least **size** bytes in length. A file opened by **OpenStr()** is identified as such by its file server (as returned by **FileServer()**) being equal to 0.

```
SystemCode RemoveFile(pathname)
    char *pathname;
```

Remove (delete) the file specified by **pathname**.

```
SystemCode SetBreakProcess(file, breakprocess)
    File *file;  ProcessId breakprocess;
```

Sets the break process associated with the specified file (which must be INTERACTIVE) to **breakprocess**.

If a break occurs on the file after a break process has been set, the IO_BREAK reply will be returned to any outstanding read requests, and the specified break process will be destroyed.

```
SystemCode SetInstanceOwner(fileserver, fileid, owner)
    ProcessId fileserver, owner;  InstanceId fileid;
```

Set the owner of the specified file instance to be **owner**.

```
PrintFile(name, file)
    char *name;  File *file;
```

Print the value of each field in the given File structure on the standard output, identifying the file by the name **name**. Useful in debugging servers and I/O routines.

```
SystemCode ChangeDirectory(name)
    char *name;
```

Change the current context for the calling process to be the context specified by **name**, and return a standard system reply code indicating OK if successful, else the reason for failure. **name** is interpreted in the (previous) current context. This function is identical to **chdir()**, except that the latter returns 0 to indicate success or -1 to indicate failure.

— 17 —
Numeric and Mathematical Functions

17.1. Numeric Functions

Most of the functions in the numeric section of the library are not called directly in user programs; they are accessed by the C compiler as needed. The following functions are useful in user programs:

```
unsigned abs(value)
    int value
```

Integer absolute value.

```
int rand()
```

Random number generator. Generates pseudo-random numbers in the range from 0 to 2^{31}-1. This is a very poor generator, identical to the one provided in Berkeley Unix 4.1.

```
srand(seed)
    unsigned seed;
```

Reseed the `rand()` random number generator.

17.2. Mathematical Functions

The math-related functions in the V library are listed below. They are similar to the "section 3M" functions of the Unix library. See the Unix manual for documentation.

```
sin()       cos()       tan()       asin()
acos()      atan()      atan2()     sinh()
cosh()      tanh()      j0()        j1()
jn()        y0()        y1()        yn()
hypot()     cabs()      gamma()     fabs()
floor()     ceil()      exp()       log()
log10()     pow()       sqrt()
```

— 18 —
Memory Management

Blocks within a managed pool of memory can be dynamically allocated and freed within the address space of a team using the functions described below. Note that there is one pool of free storage for all processes in the team. Programmers must be careful to synchronize the processes allocating and freeing this storage. These routines provide essentially the same functionality as the standard C library. The memory allocation routines are provided on a per-team basis.

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

Returns a pointer to a memory block that is **size** bytes long. NULL is returned if there is not enough memory available.

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

The memory pointed to is returned to the free storage pool. **ptr** must point to a block allocated by one of the routines listed here.

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

Changes the size of the block pointed to by **ptr** to be **size** bytes. Returns a possibly moved pointer.

```
char *calloc(elements, size)
    unsigned elements, size;
```

Equivalent to **malloc(elements*size)**, except the area is cleared to zero. Provided for allocating arrays.

```
cfree(ptr, elements, size)
    char *ptr;  unsigned elements, size;
```

Frees storage allocated by **calloc()**. Actually, this function is identical to **free(ptr)**, which may be used instead. **elements** and **size** are ignored.

```
unsigned Copy(destination, source, count)
    char *destination, *source;  unsigned count;
```

A block transfer function. Transfers **count** bytes from **source** to **destination**. Returns **count**.

```
unsigned blt(destination, source, count)
    char *destination, *source;  unsigned count;
```

Identical to **Copy()**.

```
char *Zero(ptr, n)
    char *ptr;  unsigned n;
```

Zero memory. Writes **n** bytes of zeros starting at **ptr**, and returns **ptr**.

```
clear(ptr, n)
    char *ptr;  unsigned n;
```

Clear memory. Writes **n** bytes of zeros starting at **ptr**.

```
swab(pfrom, pto, n)
    char *pfrom, *pto;  unsigned n;
```

Swap the bytes in **n** 16-bit words starting at the location **pfrom** into a block starting at the location **pto**.

The following functions are of interest only to those managing memory (using the kernel primitives) in addition to that provided by the above routines. The current implementation of **malloc()** prevents these routines from adding space below the current top of the pool.

```
GiveToMalloc(start,length)
    char *start;  int length;
```

Add the **length** bytes of memory at **start** to the pool used by the allocators described above, returning the number of bytes actually installed after alignment and error-checking is done.

```
char * GetMoreMallocSpace(min,actual)
    int min, *actual;
```

Malloc() calls this function to acquire more space for its pool; a default version is supplied, which is replaced if the programmer supplies a routine of this name. **GetMoreMallocSpace()** should return a pointer to at least **min** bytes of space and set ***actual** to the number of bytes made available; **NULL** may be returned if no more space is to be added to the pool.

In the default version, free memory is determined and extended based on the memory map and memory usage of the team (using the V kernel operations **GetTeamSize()** and **SetTeamSize()**).

— 19 —
Processes and Interprocess Communication

The process-related functions in the V C library provide services and/or interfaces between processes and the V kernel. They have no direct analog in the standard Unix C library.

19.1. Kernel Operations

These functions provide a convenient interface to kernel-provided services. Some of the functions execute kernel trap instructions, while others send messages to a pseudo-process inside the kernel..

A kernel operation executes as a single indivisible function call as far as the C programmer is concerned. Each kernel operation takes zero or more arguments and returns a single value.

In the descriptions below, the *active process* or *invoking process* always refers to the process that executed the kernel operation.

Some operations such as SetTeamPriority and SetTime are intended to be used only by "operating system" or management processes and should not be used by application programs.

```
int AwaitingReply(frompid, awaitingpid)
    ProcessId frompid, waitingpid;
```

Return true (nonzero) if **awaitingpid** is awaiting reply from **frompid**; otherwise return false. Note: if **awaitingpid** is send blocked on **frompid**, but **frompid** has not yet received the message, this function will return false.

```
ProcessId Creator(pid)
    ProcessId pid;
```

Return the process id of the process that created **pid**. If **pid** is zero, return the creator of the invoking process. If **pid** does not exist or is the root process of the initial team, return 0.

```
ProcessId CreateProcess(priority, initialpc, initialsp)
    short priority;  char *initialpc, *initialsp;
```

Create a new process with the specified priority, initial program counter and initial stack pointer and return its unique process identifier.

The priority must be between 0 and 127 inclusive, with 0 the highest priority. **initialpc** is the address of the first instruction of the process to be executed outside of the kernel. Generally, **initialsp** specifies the initialization of the stack and general registers and is processor-specific. In the case of the Motorola 68000, **initialsp** is a simple long word value that is assigned to the user stack pointer.

The process is created awaiting reply from the invoking process and in the same team space. The segment access is set up to provide read and write access to the entire team space of the newly created process. The creator must reply to the newly created process before it can execute. If there are no resources to create the process or the priority is illegal, a pid of 0 is returned.

Usually programmers will prefer the **Create()** call described later in this chapter.

```
ProcessId CreateTeam(priority, initialpc, initialsp)
    short priority;  char *initialpc, *initialsp;
```

Create a new team with initial or root process having the specified priority, initial program counter, and initial stack pointer.

CreateTeam() is similar to **CreateProcess()** except the new process is created on a new team. The new team initially has a null team space. It is intended that the creator of the team will initialize the team address space and root process state using **SetTeamSize()**, **MoveTo()**, and **WriteProcessState()**.

CreateTeam returns 0 if there are no resources to create the team or the root process, or the priority is illegal.

```
Delay(seconds, clicks)
    unsigned seconds, clicks;
```

Suspend the execution of the invoking process for the specified number of seconds and clicks (where a click is a machine-specific unit, usually one clock interrupt).

Delay() returns 0 after the time period has passed, or the number of clicks remaining in the delay time if the process has been unblocked by **Wakeup()**. A clock interrupt on the SUN workstation is 10 milliseconds.

```
SystemCode DestroyProcess(pid)
    ProcessId pid;
```

Destroy the specified process and all processes that it created. When a process is destroyed, it stops executing, its pid becomes invalid, and all processes blocked on it become unblocked (eventually).[12] **DestroyProcess()** returns OK if **pid** if successful, else a reply code indicating the reason for failure. **DestroyProcess(0)** is suicide.

Usually programmers will prefer the **Destroy()** call described later in this chapter.

```
ProcessId Forward(msg, frompid, topid)
    Message msg;  ProcessId frompid, topid;
```

Forward the message pointed to by **msg** to the process specified by **topid** as though it had been sent by the process **frompid**.

The process specified by **frompid** must be awaiting reply from the invoking process. The effect of this operation is the same as if **frompid** had sent directly to **topid**, except that the invoking process is noted as the forwarder of the message. Note that **Forward()** does not block.

Forward() returns **topid** if it was successful, 0 if unsuccessful. If **topid** is invalid, **frompid** is unblocked with an indication that its **Send()** failed.

```
ProcessId Forwarder(pid)
    ProcessId pid;
```

[12] Processes blocked on a nonexistent processes are detected and unblocked by the clock interrupt routine checking periodically.

Return the process id that forwarded the last message received from **pid**, providing **pid** is still awaiting reply from the invoking process. If the message was not forwarded, **pid** is returned. If **pid** does not exist or is not awaiting reply from the invoking process, 0 is returned.

```
ProcessId GetPid(logicalid, scope)
    int logicalid, scope;
```

Return the pid of the process registered using **SetPid()** with the specified **logicalid** and **scope**, or 0 if not set.

The scope is one of:

LOCAL_PID Return a locally registered process only.

REMOTE_PID Return a remotely registered process only.

ANY_PID Return a local or remote process pid.

If **logicalid** is ACTIVE_PROCESS, the pid of the invoking process is returned. If a scope of *remote* is specified, the kernel broadcasts a request for a process identifier registered as this logical id to other workstations running the V kernel on the network. If the scope is *any*, the kernel first looks for a locally registered process; if one is not found, it then looks for a remote process. In this way, a kernel can discover the process identifiers of the standard server processes from other kernels, or at least from the kernel that is running the server process of interest.

```
ProcessId GetTeamRoot(pid)
    ProcessId pid;
```

Return the process identifier of the root process of the team containing **pid**, or zero if **pid** is not a valid process identifier. A **pid** of zero specifies the invoking process.

```
char *GetTeamSize(pid)
    ProcessId pid;
```

Return the first unused location in the team space associated with **pid**, as set by **SetTeamSize()**. If **pid** is zero, the size of the invoking process's team is returned. If **pid** does not exist, 0 is returned.

```
unsigned GetTime(clicksptr)
    unsigned *clicksptr;
```

Return the current time in seconds. The standard is to represent time as seconds since January 1, 1970 GMT. If **clicksptr** is not NULL, the number of clock interrupts since the last second is stored at that location.

```
SystemCode MoveFrom(srcpid, dest, src, count)
    ProcessId srcpid;  char *dest, *src;  unsigned count;
```

Copy **count** bytes from the memory segment starting at **src** in the team space of **srcpid** to the segment starting at **dest** in the invoking process's space, and return the standard system reply code OK.

The **srcpid** process must be awaiting reply from the invoking process and must have provided read access to the segment of memory in its space using the message format conventions described for **Send()**. **MoveFrom()** returns a standard system reply code indicating the reason for failure if any of these conditions are violated.

```
SystemCode MoveTo(destpid, dest, src, count)
    ProcessId destpid;  char *dest, *src;  unsigned count;
```

Copy **count** bytes from the segment starting at **src** in the invoking process's team space to the segment starting at **dest** in the team space of the **destpid** process, and return the standard system reply code OK.

The **destpid** process must be awaiting reply from the invoking process and must have provided write access to the segment of memory in its space using the message format conventions described under **Send()**. **MoveTo()** returns a standard system reply code indicating the reason for failure if any of these conditions are violated.

```
QueryKernel(pid, groupSelect, reply)
    ProcessId pid;  int groupSelect;  Message reply;
```

Query the kernel on the host where process **pid** is resident. A **pid** of zero specifies the invoking process's kernel.

The **groupSelect** field specifies what information is to be returned in the **reply** message. The available group selection codes are MACHINE_CONFIG, to return information about the processor configuration, PERIPHERAL_CONFIG, to return a list of peripherals available on the machine, KERNEL_CONFIG, to return the kernel's configuration parameters, MEMORY_STATS, to return memory usage statistics, and KERNEL_STATS, to return other kernel statistics. These codes, and the corresponding structures that may be returned, are defined in the standard header file <V/querykernel.h>.

```
ProcessId QueryProcessState(pid, pb)
    ProcessId pid;  ProcessBlock *pb;
```

Copy the state of the process into the structure pointed to by **pb**. The various fields in the structure are defined in <V/process.h>. Their meanings should be self-explanatory.

The message buffer is only available if **pid** is the invoking process or is awaiting reply from the invoking process. If not, the appropriate fields in the structure are zeroed.

If **pid** is zero, the process state of the invoking process is returned. If **pid** does not exist, 0 is returned; otherwise, **pid** is returned.

```
ProcessId ReadProcessState(pid, state)
    ProcessId pid;  Processor_state *state;
```

Copy the machine-specific processor state into the structure pointed to by **state**. The information returned is a subset of that returned by **QueryProcessState()**.

If **pid** is zero, the processor state of the invoking process is returned. If **pid** does not exist, 0 is returned; otherwise, **pid** is returned.

```
ProcessId Receive(msg)
    Message msg;
```

Suspend the invoking process until a message is available from a sending process, returning the pid of this process, and placing the message in the array pointed to by **msg**.

```
ProcessId ReceiveWithSegment(msg, segbuf, segsize)
    Message msg;  char *segbuf;  unsigned *segsize;
```

Suspend the invoking process until a message is available from a sending process, returning the pid of this process, and placing the message in the array pointed to by **msg** and at most the first ***segsize** bytes of the segment included with the message in the buffer starting at **segbuf**. The actual number of bytes in the portion of the segment received is returned in ***segsize**.

```
ProcessId ReceiveSpecific(msg, pid)
    Message msg;  ProcessId pid;
```

Suspend the invoking process until a message is available from the specified process, returning the pid of this process, and placing the message in the array pointed to by **msg**.

If **pid** is not a valid process identifier, ReceiveSpecific returns 0.

```
ProcessId Reply(msg, pid)
    Message msg;  ProcessId pid;
```

Send the specified reply message to the process specified by **pid** and return **pid**.

The specified process must be awaiting reply from the invoking process. Zero is returned if the process does not exist or is not awaiting reply.

```
ReplyWithSegment(msg, pid, src, dest, bytes)
    Message msg;  ProcessId pid;  char *src, *dest;  unsigned bytes;
```

Send the specified reply message and segment to the process specified by **pid** and return **pid**.

The specified process must be awaiting reply from the invoking process. Zero is returned if the process does not exist or is not awaiting reply. The segment size is currently limited to 1024 bytes. A **ReplyWithSegment()** with a nonzero segment size may only be used to reply to an idempotent request (see **Send()**).

```
RereadMsg(msg, pid)
    Message msg;  ProcessId pid;
```

Reread into **msg** the message received from the process specified by **pid**, providing it is still awaiting reply from the invoking process.

RereadMsg() copies the contents of the message buffer last received from **pid** into the array **msg**, providing the process specified by **pid** still exists and has not been replied to. If **pid** is zero, it is taken to mean the invoking process and rereads the last reply message. This operation also allows a newly created process to read the initial reply message from its creator.

```
int SameTeam(pid1, pid2)
    ProcessId pid1, pid2;
```

Return true (nonzero) if the processes specified both exist and are on the same team; otherwise return false. If either pid is zero, the invoking process is assumed.

```
ProcessId Send(msg, pid)
    Message msg;  ProcessId pid;
```

Send the message in **msg** to the specified process, blocking the invoking process until the message is both

received and replied to. The array specified by **msg** is assumed to be 8 long words. The reply message overwrites the original message in the array.

If **Send()** completes successfully, it returns the pid of the process that replied to the message. The pid returned will differ from that specified in the call if the message is forwarded by the receiver to another process that in turn replies to it. If the send fails (for instance, because the intended receiver does not exist), **Send()** returns the pid of the process the message was last forwarded to (the pid it was sent to, if it was never forwarded). The kernel indicates the reason for the failure by overwriting the first 16 bits of the message with a standard system reply code. (This places it in the *replycode* field for reply messages that follow the standard system format.)

All messages must follow the kernel message format conventions as follows. The first 16 bits of the message are considered to be a request code or reply code. The highest order 6 bits are assigned special meanings.

Bit 0 is 0 if a request message is being sent, or 1 if a reply message.

Bit 1 is 1 if the request code or reply code is considered a standard system code. Applications can use special request codes and reply codes internal to their programs but use standard ones for interfacing to other programs and the system. The remaining 4 bits are interpreted with the following special meanings only if the message is a request.

Bit 2 is 1 if the request is considered to be idempotent. This is just a hint to discriminate between requests that do not need duplicate suppression and those that do.

Bit 3 is 1 if the request specifies a segment. If 1, the kernel interprets the last 2 words of the message as specifying a pointer to the start of the segment and the size in bytes of the segment, respectively. The kernel then makes the segment available to the receiving process using MoveTo and MoveFrom. Access to the segment is controlled by the following two bits, which only have meaning if bit 3 is 1.

Bit 4 is 1 if read access is provided to the segment.

Bit 5 is 1 if write access is provided to the segment.

It is intended and assumed that most requests can be assigned a request code that is stored in the first 16 bits of the request message, so that the bits are set correctly for the request by the value of the request code.

```
SetPid(logicalid, pid, scope)
    int logicalid, scope;  ProcessId pid;
```

Associate **pid** with the specified logical id within the specifed scope. Subsequent calls to **GetPid()** with this **logicalid** and **scope** return this pid. This provides an efficient, low-level naming service.

The scope is one of:

LOCAL_PID Register the process in the local scope only.

REMOTE_PID Register the process in the remote scope only.

ANY_PID Register the process in both the local and remote scopes.

The *local* scope is intended for servers serving only the local workstation. The *remote* scope is for network-accessed server processes serving several workstations (but not the local workstation). The *any* scope permits both local and remote access.

```
SetTeamPriority(pid, priority)
    ProcessId pid;  short priority;
```

Set the team priority of the team associated with **pid** to the specified priority and return the previous team priority.

Each process effectively runs with the absolute scheduling priority of its team's priority plus the priority specified when the process was created. **SetTeamPriority()** changes the absolute scheduling priority of each process on the team by modifying the team priority. This operation is intended for implementing macro-level scheduling and may eventually be restricted in use to the first team.

If **pid** is zero, the invoking process's team priority is set.

```
char *SetTeamSize(pid, addr)
    ProcessId pid;  char *addr;
```

Sets the first unused address for the team containing **pid** to **addr**. The new team size may be either greater or smaller than the previous size. The new team size is returned; this will normally be equal to **addr**. If there was not enough memory available to grant the request, the return value will be less than **addr**; if **addr** was below the starting address for team spaces on the host machine, the team space will be set to null and its starting address will be returned. Thus **SetTeamSize(pid, 0)** is a machine-independent way of setting a team space to null.

A pid of 0 specifies the invoking process. Only the creator of the team or members of the team may change the team size and (consequently) the specified process must be local.

```
SetTime(seconds, clicks)
    unsigned seconds, clicks;
```

Set the kernel-maintained time to that specified by **seconds** and **clicks**.

The standard time representation used is the number of seconds since January 1, 1970 GMT, plus the number of clock interrupts since the last second.

```
ProcessId Wakeup(pid)
    ProcessId pid;
```

Unblock the specified process if it is delaying using **Delay()** and return **pid**. If the process does not exist or is not delaying, return 0.

```
int ValidPid(pid)
    ProcessId pid;
```

Return true (nonzero) if **pid** is a valid process identifier; otherwise return false.

```
ProcessId WriteProcessState(pid, state)
    ProcessId pid;  Processor_state *state;
```

Copy the specified process state record into the kernel state of the process specified by **pid** and return **pid**.

The specified process must be the invoking process, or awaiting reply from the invoking process. **WriteProcessState()** returns 0 if the process does not exist, is not awaiting reply or there is a problem with the state record. The kernel checks that the new state cannot compromise the integrity or security of the kernel.

A **pid** of 0 specifies the invoking process. A process that writes its own processor state affects only the

machine-independent per-process area information kept as part of the state record (see section 14.5).

19.2. Other Functions

```
ProcessId Create(priority, function, stacksize)
    short priority;  char *function;  unsigned stacksize
```

Create a new process executing the specified function with the specified priority and stack size. The new process is blocked, waiting for a reply from the creator. The function **Ready()** should be used to start the process running. The new process is on the same team as its creator, and inherits the creator's standard input, output, and error files, and the creator's current context (current working directory).

Create returns the pid of the new process, or zero if a process could not be created. This function is usually preferable to calling the kernel operation **CreateProcess()** directly.

```
ProcessId Ready(pid, nargs, a1, ..., an)
    ProcessId pid;  unsigned nargs;  Unspec a1, ..., an;
```

Set up the stack of the specified process and reply to it, thus placing it on the ready queue. The values **a1, ..., an** appear as arguments to the root function of the new process, while **nargs** is the number of arguments passed. Zero is returned if there is a problem, else **pid** is returned.

```
Destroy(pid)
    ProcessId pid;
```

Destroy the specified process. If the destroyed process was on the same team as the invoking process, the memory allocated to its stack by **Create()** is freed. Warning: Do not invoke **Destroy()** on a process that was not created by **Create()**; use **DestroyProcess()** in that case.

```
Suicide()
```

Destroy the invoking process and free its stack. **Suicide()** is identical to **Destroy(0)**, and the same warning applies.

```
exit()
```

Terminate the execution of the team (i.e., program), after closing all open files. Using the V executive, control is returned to the command interpreter. In bare kernel mode, control is returned to the PROM monitor.

```
abort()
```

Abort execution of the team by causing an exception in the calling process.

— 20 —
Naming

The *naming* section of the library includes a number of functions that provide a convenient interface to V-System naming protocol messages. Functions for creating and terminating storage server sessions are also included.

```
SystemCode AddContextName(name, serverpid, contextid)
    char *name;  ProcessId serverpid;  ContextId contextid;
```

Add **name** as a local name for the context specified by **(serverpid, contextid)**, and return OK, or a standard system reply code if an error occurred. This function creates and sends an ADD_CONTEXT_NAME request message to the context prefix server.

```
SystemCode AddLogicalName(name, logicalpid)
    char *name;  ProcessId logicalpid;
```

Add **name** as a local name for the default context specified by **logicalpid**, and return OK, or a standard system reply code if an error occurred. This function creates and sends an ADD_CONTEXT_NAME request message to the context prefix server.

```
SystemCode AliasContextName(newname, oldname)
    char *newname, *oldname;
```

Define **newname** as a local name for the context specified by **oldname**. **oldname** is interpreted in the current context. Returns OK if the name was defined successfully, or a standard system code indicating the reason for failure.

```
SystemCode CreateSession(host, user, password, sessionname, owner)
    char *host, *user, *password, *sessionname;
    ProcessId owner;
```

Create a session on the storage server (usually a Unix server) specified by **host**, using the given user name and password, and define **sessionname** as a local name for the user's home directory on this session. If **owner** is nonzero, the session owner is set to be the specified process; otherwise, the invoking process becomes the session owner. A session is automatically terminated when its owner no longer exists.

The given session name is considered the primary name for the session (the SESSION bit is set in its descriptor), and its definition should not be removed until the session is terminated.

CreateSession() returns OK if successful, else a standard system code indicating the reason for failure.

```
SystemCode DeleteContextName(name)
    char *name;
```

Remove the definition of the context name **name**, but do not delete the context it refers to. Return OK if

successful, else a system reply code indicating the reason for failure.

The name is interpreted directly by the context prefix server, not in the current context, since the function is ordinarily used only to remove names from the context prefix server's directory.

```
ProcessId DirectToCurrentContext(request)
    NameRequest *request;
```

Direct a request to the current context, or to the context prefix server if the name begins with a square bracket ('['). The function returns the pid to which the request should be sent, and puts the proper context id into the NameRequest message. This routine is provided to avoid duplicating the code that implements the square bracket convention in a large number of functions. **request** may be of any request type that fits the standard NameRequest template given in <Vnaming.h>.

```
SystemCode GetContextId(name, serverpid, contextid)
    char *name;  ProcessId *serverpid;  ContextId *contextid;
```

Interpret the given name in the current context, and return a corresponding (serverpid, contextid) pair in the locations pointed to by **serverpid** and **contextid**. The function returns OK if successful, or a standard system error code if an error is detected, such as the given name specifying an object that is not a context.

```
SystemCode GetContextName(name, namelen, serverpid, contextid, nameserver)
    char name[];  unsigned *namelen;
    ProcessId *serverpid;  ContextId *contextid;
    ProcessId nameserver;
```

Perform an inverse mapping from the specified (serverpid, contextid) pair to a character string context name. The request is sent to the server specified by **nameserver**. The array **name** must be ***namelen** characters in length; ***namelen** is modified to contain the actual length of the name upon return. ***serverpid** and ***contextid** are modified upon return to indicate the context in which the name is valid. **GetContextName()** returns OK if the mapping was successful, or a standard system error code if a failure occurred.

```
SystemCode GetFileName(name, namelen, serverpid, contextid, instanceid)
    char name[];  unsigned *namelen;
    ProcessId *serverpid;  ContextId *contextid;
    InstanceId instanceid;
```

Perform an inverse mapping from the specified (serverpid, instanceid) pair to a character string file name. The array **name** must be ***namelen** characters in length; ***namelen** is modified to contain the actual length of the name upon return. ***serverpid** and ***contextid** are modified upon return to indicate the context in which the name is valid. **GetContextName()** returns OK if the mapping was successful, or a standard system error code if a failure occurred.

```
SystemCode TerminateSession(sessionname)
    char *sessionname;
```

Terminate the session specified by **sessionname** and invalidate the name. Return OK on success, else a standard system code indicating the reason for failure. The session name is interpreted by the local context prefix server. The function checks that the SESSION bit is set in the name's descriptor; if it is not, NONEXISTENT_SESSION is returned.

— 21 —
Program Execution Functions

This chapter describes a number of functions relating to program execution. Most of these functions are used internally in the V executive; some of them may also be useful in user-level programs that need to start up other programs as part of their operation. All the functions in this chapter are subject to change.

21.1. Program Execution

```
ProcessId LoadProg(argv, concurrent, teamServer, rtMsg, drtMsg, error)
    char *argv[];  int concurrent;  ProcessId teamServer;
    RootMessage *rtMsg, *drtMsg;  SystemCode *error;
```

LoadProg() interacts with the team server to create a new team and load a program image into the new team space. It includes path searching code, which currently always looks for the program along the default path of

1. The current context
2. The context "[bin]"
3. The context "[public]"

If all these fail, **LoadProg()** loads the *fexecute* program, which, when started, will attempt to execute the program on the storage server that is providing the current context.

The array **argv** contains pointers to the character string arguments to be passed to the new team. By convention, **argv[0]** should point to the name of the program. The last element of the array must be a null pointer. The **concurrent** argument specifies whether the team is to be "owned" by the process executing the **LoadProg()** call (if **concurrent** is zero) or by the team server itself (if it is nonzero). The team server destroys any team whose owner ceases to exist; thus, programs to be run "in the background" should be flagged as concurrent. The **teamServer** argument specifies which team server is to create the team. This is useful for running programs remotely. If **teamServer** is zero, the program is run locally.

The **rtMsg** argument holds the root message to be passed to the new team. This message specifies file instances to be used for standard input, output, and error, the initial current context, and some other information. The fields in the message are described in section 14.4. The **drtMsg** argument is the root message to be used to start up the postmortem debugger if a process team on the new team incurs an exception. The debugger root message should specify a real keyboard and display as standard input and output, even if the standard i/o for the program being loaded is redirected. These root messages are stored by the team server.

The function returns the process id of the new team's root process, or 0 in case of an error. A standard system code is returned in the location pointed to by **error**. The new team can be started running by replying to the pid returned, using the same root message as was passed to **LoadProg**.

```
ProcessId ExecProg(argv, concurrent, teamServer, rtMsg, drtMsg, error)
    char *argv[];  int concurrent;  ProcessId teamServer;
    RootMessage *rtMsg, *drtMsg;  SystemCode *error;
```

ExecProg() interacts with the team server to create a new team and load a program image into the new team space, as in **LoadProg()**. It then starts the new team running by replying to it. The arguments to **ExecProg()** are exactly the same as those to **LoadProg()**.

```
ProcessId RunProgram(argv, concurrent, teamServer, error)
    char *argv[];  int concurrent;  ProcessId teamServer;
    RootMessage *rtMsg, *drtMsg;  SystemCode *error;
```

RunProgram() performs the same function as **ExecProg()** except that it uses the standard I/O bindings to initialize the **rtMsg** and **drtMsg** parameters that are passed in to **ExecProg()**.

```
ProcessId LoadNewTeam(teamServer, name, concurrent, argv,
        rtMsg, drtMsg, error)
    ProcessId teamServer;  char *name;  int concurrent;  char *argv[];
    RootMessage *rtMsg, *drtMsg;  SystemCode *error;
```

LoadNewTeam() is an internal routine called by **LoadProg()**. It does no path searching; the name of the file to load the program image from is given by the **name** argument. The other six arguments are as described above, under **LoadProg()**, though they appear in a different order.

LoadNewTeam() calls **ValidProgram()** to check whether the specified file appears to contain a valid program image, interacts with the team server to create the new team, and sets up the arguments on the new team's stack.

```
ProcessId LoadTeam(filename, priority, stacksize, error)
    char *filename;  short priority;
    int stacksize;  SystemCode *error;
```

Create a new team with the specified root process priority, and load the program contained in the specified file into it. The number of bytes specified by **stacksize** is allocated at the end of the team space as a stack area for the team root process unless **stacksize** is -1, in which case the default 4000 bytes are allocated. If the operation is successful, the pid of the new team's root process is returned; otherwise 0 is returned. If **error** is not NULL, a standard system reply code is returned in the location to which it points.

This function does not request the team server to create and load the team; it creates the team and performs the team load itself. It is normally preferable to use one of the other functions described above, all of which make use of the team server.

```
SystemCode RemoteExecute(processFile, programname, argv, mode)
    File *processFile[2];  char *programname;
    char *argv[];  unsigned short mode;
```

Cause the specified program to be executed on the server machine providing the invoking process's current context by opening a file in FEXECUTE mode. This function is used by the *fexecute* program.

The **argv** parameter is an array of null-terminated strings which are to passed as arguments to the program. The array itself is terminated by a null pointer. **mode** should be FREAD or FCREATE. A File structure describing a stream from which the program's standard output can be read is returned in **processFile[0]**. If the mode is FCREATE, a File structure describing a writeable stream that is fed into the program's standard input is returned in **processFile[1]**. **RemoteExecute()** returns OK if successful, else a standard system code describing the error condition.

Closing the writeable file passes an end-of-file indication on to the remote program. Closing the readable

file terminates the program.

21.2. Other Functions

```
File *ValidProgram(filename, error)
    char *filename;  SystemCode *error;
```

This function opens the file specified by **filename** and checks whether it has a valid "magic number," marking it as an executable V program image. If it is a valid program, **ValidProgram()** returns a pointer to a File structure describing the open file; if not, it closes the file again and returns NULL. A standard system code is returned in the location pointed to by **error**. The error code END_OF_FILE indicates that the file was too short to be a valid program, while BAD_STATE indicates that the magic number was invalid.

```
SetUpArguments(pid, argv)
    ProcessId pid;  char *argv[];
```

SetUpArguments() is the function called by **LoadProg()** to set up the arguments on a newly created team's stack. Users will not normally need to call it directly. The array **argv** has the format described under **LoadProg()**, above. The process id **pid** specifies the root process of the team whose arguments are to be set up.

```
ParseLine(start, argv, maxArgs)
    char *start;  char *argv[];  int maxArgs;
```

ParseLine() parses a command line into separate words, null terminating each one, and filling in an array of pointers to each word. Spaces and tabs are recognized as word separators. This routine is used by the V executive to construct an **argv** array to pass to **LoadProg()**.

The **start** argument points to the command line, which should be a null-terminated character string. The string is modified by inserting null characters after each word. The array of pointers created is returned in **argv**, which should be defined in the calling program to be of size **maxArgs**. **ParseLine()** terminates the array with a null pointer. If there are too many words in the command line to fit in the array, only the leftmost **maxArgs - 1** words are returned.

— 22 —
Control of Executives

Instances of the V executive, or command interpreter, are normally created and controlled directly by the user interacting with the sytstem. However, this control is also available to programs through the following functions:

```
int CreateExec(execserver, inserver, infile, outserver, outfile,
                errserver, errfile, nameserver,context, flags, execpid,
                error)
    ProcessId execserver;
    ProcessId inserver, outserver, errserver;
    InstanceId infile, outfile, errfile;
    ProcessId nameserver;
    ContextId context;
    short flags;
    ProcessId  *execpid;
    SystemCode  *error;
```

Create an instance of the executive with the specified standard input, standard output, standard error output, and context. Each of the three standard i/o files is specified by two parameters, the server pid and the instance identifier within that server. This means that all these instances must be opened before Create Exec is called. Context is specified by two parameters, a name server pid and a context identifier within that nameserver. The GetContextId function will map a context name into such a pair. Execserver is the pid of the exec server to which the request is being made. The **Flags** parameter determines which if any of the standard i/o instances are to be owned by the newly created executive; it may be any combination of RELEASE–INPUT,RELEASE– OUTPUT, and RELEASE–ERR. If for example RELEASE–INPUT is specified, the executive will own its standard input instance and will release it on termination.

CreateExec returns an exec indentifier, a small integer which will be used to refer to this executive in other executive control requests. In the location pointed to by execpid it returns the process id of the new executive. In the location pointed to by error it returns a system error code; if this code is not OK, the exec identifier and execpid are meaningless.

WARNING: a server process cannot call CreateExec with a file instance pointing to that server itself, or the server and the execserver will become deadlocked waiting for each other. A server that needs to do this should create a subprocess to call CreateExec.

```
SystemCode DeleteExec(execserver, execid)
    ProcessId execserver;
    int execid;
```

Delete the executive specified by execid, along with the program running under it if any. It need not have been created by this process; there is no concept of ownership of execs. Note that this is not the only way executives vanish; they also terminate on end of file on the standard input. DeleteExec will return NOT–FOUND if execid is invalid.

System.....

Inquire about the state of the specified exec. If successful, it returns a code of OK, and the following information: inexecpid the process id of the exec; in program, the process id of the program running under it, if any; in status, the status of the exec. Status can be one of

EXEC–FREE Exec is waiting for a command.

EXEC–LOADING
exec is in the process of loading a program.

EXEC–RUNNING
A program is running under this exec. In this case and this case only, program returns relevant information.

EXEC–HOLD Exec has been created but not yet started. Hopefully this state should never be observed, as it is taken care of within CreateExec.

```
SystemCode KillProgram(execserver, execid)
    ProcessId execserver;
    int execid;
```

Kill the program, if any, running under the specified exec. Returns OK is successful, NOT–FOUND if execid was invalid, NONEXISTENT–PROCESS if there was no program running under that exec.

```
SystemCode CheckExecs(execserver)
    ProcessId execserver;
```

Causes the execserver to do a check on all executives. Any of them whose standard input server or standard output server (but NOT standard error server) has died is destroyed during the check. This should be called after an action that might have destroyed an i/o server which was providing standard i/o for one or more executives.

— 23 —
Service Registration and Selection Functions

This chapter describes a number of functions which deal with the globally visible service server; which provides registration and selection facilities for globally visible services. A description of the service server and the details of how to interact with it are provided in its servers manual chapter. This chapter assumes that the reader is familiar with the servers manual chapter and bases the form of its explanations on that assumption. All the functions in this chapter are subject to change.

23.1. Registration Facilities

```
InstanceId RegisterServer(nameType, nameIndex, typeIndex,
                                    ownerPid, desc, descLen, error)
    int nameType;
    int nameIndex;
    int typeIndex;
    ProcessId ownerPid;
    char *desc;
    int descLen;
    SystemCode *error
```

RegisterServer registers the server descriptor (actually any object descriptor) pointed to by **desc** with the service server. **descLen** indicates how long the descriptor is in bytes. The owner of the descriptor is specified by **ownerPid**. It is assumed that the descriptor contains both a valid server name and type field, whose starting indices within the descriptor are given by **nameIndex** and **typeIndex**. The type field is assumed to be a null-terminated string field. There are two types of name field allowed: a process id or a null-terminated string field. **nameType** specifies which type of name field is being used. The allowable values for **nameType** are defined in the **Vservice.h** include file. **error** is used to return a status value indicating whether the operation was successful or why it failed. If the operation is successful then **RegisterServer** returns an id number for the server's registration entry. This is used for unregistering the server (and possibly other things in the future).

```
SystemCode UnregisterServer(serverId)
    InstanceId serverId;
```

UnregisterServer removes a server's registration entry from the service server's database. It takes as argument the id number returned from the original **RegisterServer** operation.

23.2. Selection Facilities

```
InstanceId CreateSelectionInstance(serverType, pattern, patternFcn,
                                   howMany, desc, descLen, error)
    char *serverType;
    char *pattern;
```

```
int patternFcn;
int howMany;
char *desc;
int descLen;
SystemCode *error;
```

CreateSelectionInstance() specifies a set of registered objects to associate with a *selection instance* and returns the first entry of the instance in **desc**. **descLen** specifies the maximum size that the descriptor returned may be. If the selected descriptor is larger than that then only the first **descLen** bytes are returned. The server type under which selection is to take place is specified by the **serverType** field, which is a null-terminated string. The pattern-matching function to be used is specified by **patternFcn**. Values that this parameter may assume are defined in the **Vservice.h** include file. The pattern to match against registered descriptor entries is pointed to by **pattern**. The format of the pattern as far as this function is concerned (its interpretation within the service server will depend on which pattern-matching function is specified) is a null-terminated string. **howMany** specifies whether one or more selections is desired. If only one selection is desired then that selection is returned in **desc** *and no selection instance is created.* This provides a means of circumventing the overhead of establishing a full-blown connection for obtaining just one selection. **error** is used to return the status of the operation performed. If the operation is successful then **CreateSelectionInstance** returns an instance id for the selection instance established. (This value is meaningless if **howMany** equals 1.)

— 24 —
Graphics Functions

The Virtual Graphics Terminal Service (VGTS) allows the display of structured graphical objects on a workstation running the V system. This chapter describes the interface of a client (application) program to the VGTS to provide facilities for the creation, destruction, and editing of structured display files (SDFs). The user interface to the VGTS is described in the View Manager chapter (chapter 3) of the Commands Manual. For simple text applications, the VGTS implements the standard I/O protocol. The functions in this chapter are primarily for graphics applications.

24.1. Terminology

The central concept of the VGTS is that application programs should only have to deal with creating and maintaining abstract graphical objects. The details of viewing these objects are taken care of by the VGTS. This is in contrast to traditional graphics systems in which users perform the operations directly on the screen, or on an area of the screen referred to as a viewport or window. Thus the VGTS deals with declarative information rather than procedural; you describe what the objects are rather than how to draw them.

The following are the types of objects managed by the VGTS:

SDF	A structured display file is a name space in which symbols and items are defined. Each item can be given a unique identifier by the client.
Item	Items can be either graphical primitives such as rectangles, lines, or text, or symbols, which consist of other items.
Symbol	A list of items (primitives or calls to other symbols) used to represent the hierarchical structure of the display file.
VGT	A virtual graphics terminal, can be either an emulation of an ANSI standard text terminal, or a general graphics terminal which can display an instance of a symbol in some SDF.
Event	Graphical input is in terms of events in the coordinate space of some virtual graphics terminal. For example, a mouse click used to select a displayed object.
View	Both applications and users can create views of Virtual Graphics Terminals. A view consists of a viewport on some screen of some workstation, a window onto some VGT giving the world coordinates of the viewed area, and some other viewing parameters. The same VGT can appear in several different views, with independent control of all parameters.

Items within an SDF are named with 16 bit identifiers chosen by the application. It is assumed that the application will maintain some higher-level data structures, along with the appropriate mapping to these internal item names. Items that will never be referenced can be given item number zero. The item names are global to each SDF, but applications may also have several SDFs for different name spaces. The item identifiers are hashed into a symbol table, so there are no constraints on their values. Item numbers can refer to both definitions of symbols and their instances.

For example, a picture of a bicycle might define a symbol for a wheel. This definition of the wheel symbol is given item number 4. There may then be two instances of item number 4, that are given item numbers 5

and 6. The individual spokes of the wheel are components of symbol number 4, but are all given item number 0, since we will never want to refer to any of them. If it is desired to delete or move any individual spoke, then the items may be given numbers.

Each item has the following parameters:

Item — A 16 bit unique (within the SDF) identifier for this object, or zero. This identifier is referenced by the client when performing editing operations.

Type — One of the predefined types described below, either a primitive type or one to indicate structure. Currently eight bits are allocated to this.

TypeData — Eight bits of type-dependent information, like the stipple pattern number for a filled rectangle. Other attributes are stored here, such as the font index for general text.

Xmin — Minimum X coordinate of the bounding box. All coordinates are in "world" coordinates, stored as signed 16 bit signed integers.

Xmax — Maximum X coordinate of the bounding box.

Ymin — Minimum Y coordinate of the bounding box.

Ymax — Maximum Y coordinate of the bounding box.

Pointer — Depending on the type, this is either a pointer to some data like an ASCII text string, or for symbol calls, a pointer to the called symbol.

Sibling — All the component items in a symbol are linked together via this chain. Normally it should not be visible to the client, unless the client wants to step through a symbol definition in order.

24.2. SDF Primitive Types

Some of the meanings of the fields depend on the type of the item. The following are the types of items that occur in display records in a structured display file:

SDF_FILLED_RECTANGLE
: A filled rectangle. The TypeData field determines the stipple pattern, or color on the Iris system. Refer to the `Vgts.h` include file for the available colors.

SDF_HORIZONTAL_LINE
: Horizontal line from (Xmin,Ymin) to (Xmax,Ymin). Ymax is ignored.

SDF_VERTICAL_LINE
: Vertical line from (Xmin,Ymin) to (Xmin,Ymax). Xmax is ignored.

SDF_POINT
: A point, which usually appears as a 2 by 2 pixel square at (Xmin,Ymin).

SDF_SIMPLE_TEXT
: A simple text string, which appears at (Xmin,Ymin) as its lower left corner. Currently only a single fixed-width font is available. The values of Xmax and Ymax need not surround the text, but they are used as aids for redrawing, so should correspond roughly to the real bounding box.

SDF_GENERAL_LINE
: A generalized line, from (Xmin,Ymin) to (Xmax,Ymax). Note that Xmin etc. are slightly misleading names. The SDF manager actually sorts the endpoints and calculates the bounding box correctly.

SDF_OUTLINE Outline for a selected symbol. Xmin, Xmax, Ymin and Ymax give the box for the outline. The TypeData field specifies bits to select each of the edges: LeftEdge, RightEdge, TopEdge or BottomEdge.

SDF_HORIZONTAL_REF
A horizontal reference line at (Ymin + Ymax)/2. Reference lines consist of a thick line with two tick marks at the ends, and some associated text. They are intended for use in computer aided design applications like the Yale layout editor.

SDF_VERTICAL_REF
A vertical reference line at (Xmin + Xmax)/2.

SDF_SEL_HORIZ_REF
A thick (selected) horizontal reference line at (Ymin + Ymax)/2.

SDF_SEL_VERT_REF
A thick (selected) vertical reference line at (Xmin + Xmax)/2.

SDF_TEXT A string of general text, with a lower left corner at (Xmin,Ymin). The TypeData field determines the font number. Xmax is recalculated from the width information for the font. See section 24.6 for an example.

SDF_RASTER A general raster bit-map with a lower left corner at (Xmin,Ymin), and upper right corner at (Xmax,Ymax). The TypeData field determines if the raster is written with ones as black or white. The pointer field points to the actual bitmap, in 16 bit-wide swaths.

SDF_SPLINE A spline object, of which a special case is a polygon. The pointer field points to a SPLINE structure as defined in the include file <splines.h>.

There are a few other types that are not visible to the user. For example, symbol definitions and calls are represented as items with most of the same attributes.

Note: The following SDF item types are not yet implemented for the SMI model 120 framebuffer: SDF2us()TEXT, SDF2us()RASTER, SDF2us()GENERAL_LINE and SDF2us()SPLINE.

24.3. SDF Manipulation Procedures

The following are the currently defined procedures used to manipulate the SDF. When called from C, all return values except the actual C expression value are passed via pointer parameters. If any pointer is NULL, no value is returned for that parameter.

```
short CreateSDF()
```

Create a structured display file, and return it. Return -1 if the VGTS runs out of resources. This must be done before any symbols are defined.

```
short DeleteSDF(sdf)
    short sdf;
```

Return all the items defined in the given SDF to free storage. This includes all strings, polygon structures, and spline structures associated with items in the SDF. Returns **sdf**.

```
short DefineSymbol(sdf, item, text)
    short sdf, item;
```

```
    char *text;
```

Enter symbol into the symbol table, and open it for editing. The **sdf** is returned from a previous CreateSDF call. The **text** is an optional descriptive name for the symbol, used in the hit selection routines for disambiguating selections. Returns **item** if successful, or zero on some error.

```
short EndSymbol(sdf, item, vgt)
    short sdf, item, vgt;
```

Close the given symbol so no more insertions can be done, and cause the VGT to be redrawn to reflect the new SDF. Called at the end of a list of **AddItem()** and **AddCall()** calls defining a symbol, started with **DefineSymbol()** or **EditSymbol()**. Returns **item** if successful. Note that the VGT number is only a "hint," because an object can exist in several different VGTs. The client can always call **DisplayItem()** to force a VGT to be redrawn.

```
short AddItem(sdf, item, xmin, xmax, ymin, ymax,
        typedata, type, string)
    short sdf, item, xmin, xmax, ymin, ymax;
    unsigned char type, typedata;  char *string;
```

Add an item to the currently open symbol. Returns the item name if successful, or zero on errors. **string** is an optional pointer to a text string used only for text types and reference lines, or special object descriptors for rasters and splines. The **item** number can be zero to indicate that the item will never be referenced.

```
short AddCall(sdf, item, xoffset, yoffset, calledSymbol)
    short sdf, item, xoffset, yoffset, calledSymbol;
```

Add an instance of the called symbol to the currently open symbol. The called symbol instance is placed at (Xoffset,Yoffset). Returns **item** if successful, 0 otherwise.

```
short DeleteItem(sdf, item)
    short sdf, item;
```

Delete an item from the currently open symbol definition. The item name will be removed from the hash table. Symbol calls can be deleted just like any other item, but symbol definitions are deleted by the DeleteSymbol function. Again, returns zero on errors, the item name if successful.

```
short InquireItem(sdf, item, xmin, xmax,
        ymin, ymax, typedata, type, string)
    short sdf, item;  short *xmin, *xmax, *ymin, *ymax;
    unsigned char *type, *typedata;  char *string;
```

All parameters except **sdf** and **item** are pointers. For each non-null pointer, the value of the field for that item is returned. Zero is returned if the item could not be found; otherwise **item** is returned.

```
short InquireCall(sdf, item)
    short sdf, item;
```

Return the item name of the symbol called by the indicated item. Returns zero if the item is not a call, or could not be found.

```
short ChangeItem(sdf, item, xmin, xmax,
        ymin, ymax, typedata, type, string)
    short sdf, item, xmin, xmax, ymin, ymax;
    unsigned char type, typedata;  char *string;
```

Change the parameters of an already existing item. Return zero if the item did not exist, otherwise `item`. This is equivalent to deleting an item and then reinserting it, so the item must be part of the open symbol.

```
short EditSymbol(sdf, item)
    short sdf, item;
```

Open an already existing symbol definition for modification. This has the effect of calling `DefineSymbol()` and inserting all the already existing entries to the definitions list. The editing process is ended in the same way as the initial definition process -- a call to `EndSymbol()`. Returns `item` if successful, 0 otherwise.

```
short DeleteSymbol(sdf, item)
    short sdf, item;
```

Delete the definition of a symbol. `item` must be a symbol definition. Any dangling instances of this symbol will remain, but will contain nothing. Returns `item` if successful, else 0.

To continue the example of the previous section, to create the bicycle figure we would use code like the following:

```
short sdf;

sdf = CreateSDF();
DefineSymbol(sdf,4,"Wheel");
    .
AddItem(sdf,0,xmin,xmax,ymin,ymax,0,SDF_GENERAL_LINE,NULL);
    .
   (add the components of the wheel symbol)
    .
EndSymbol(sdf,4,0);

DefineSymbol(sdf,3,"Bicycle");
AddCall(sdf,5,x1,ymin,4);
AddCall(sdf,6,x2,ymin,4);
EndSymbol(sdf,4,0);
```

24.4. VGTs and Views

Once a client has defined some graphical objects, it also needs to provide information on which objects can be viewed. Every *VGT* (Virtual Graphics Terminal) is an item (usually a structured symbol) that is associated with one or more views, that actually appear on the screen. Each VGT can exist in zero or more views, but each view has exactly one VGT associated with it. The "SDF Numbers" can be thought of as separate object *definition* spaces, while the VGTs are object *instance* spaces. Symbol definitions are shared between VGTs, but instances of symbols are not.

The VGTS lets a user view objects in any VGTs anywhere on the screen in *views*. Each view has a zoom factor, a window on the world coordinates of some VGT, and screen coordinates which determine its viewport. Although the client can create default views, the VGTS user can change them with the window manager, and create and destroy more of them. Routines for the client's manipulation of VGTs and views:

```
int CreateVGT(sdf, type, topItem, string)
```

```
    short sdf;  int type;  short topItem;  char *string;
```

Create a VGT, return the VGT number, and put the indicated item as the top-level item in the VGT. The type can be some combination of TTY, GRAPHICS, and ZOOMABLE. The Pads created by making TTY VGTs can presently only be manipulated by the VGTS or through the I/O protocol interface (See the description of OpenPad in section 24.7.2). If the ZOOMABLE bit is set, the view zooming factor can be changed by the user. The topItem can be zero to indicate a blank VGT. Returns negative on errors.

```
int DeleteVGT(vgt)
    short vgt;
```

Destroy the given VGT. All the views of the VGT will also be destroyed.

```
DisplayItem(sdf, topItem, vgt)
    short sdf, topItem;  int vgt;
```

Change the top-level item in a VGT. The new item is displayed in every view that contains the VGT.

```
int DefaultView(vgt, width, height, wXmin, wYmin,
        zoom, showGrid, pWidth, pHeight)
    short vgt, width, height, wXmin, wYmin, zoom, showgrid;
    short *pWidth, *pHeight;
```

Create a view of the given VGT, with the user determining the position on the screen with the graphics input device. The **width** and **height** parameters give the initial size of the view if they are positive. Zero (or negative) values indicate that the user should determine the size with the mouse at run-time. See the View Manager section of the commands manual (chapter 3) for more information about the user interface.

If the **pWidth** and **pHeight** pointers are non-NULL, then the shorts which they point to receive the selected width and height. **wXmin** and **wYmin** are the world coordinates to map to the left and bottom edges of the viewport. The **zoom** factor is the power of two to multiply world coordinates to get screen coordinates. The zoom factor may be negative, to denote that a view is zoomed out. If **showGrid** is non-zero a grid of points every 16 pixels is displayed in the window. Returns negative on error.

24.5. Graphical and Character Input

The VGTS maintains an event queue for each instance, and the VGTs associated with the given file instance. The mode bits of the instance give the kind of events that will be queued. The following functions are available to handle the event queues:

```
LISTTYPE FindSelectedObject(sdf, x, y, vgt, searchType)
    short sdf, x, y, vgt;
    char searchType;
```

Return a list of items that are at or near the selected location within the VGT. Along with each item is a set of edges, to indicate that the hit was near one or more edges of the object. The **searchType** selects one of several modes of hit detection, as given in the <Vgts.h> include file. Usually the constant value **All** will be used. The return type LISTTYPE is also defined in this file.

```
typedef struct MinElement
  {
    short                    item;
    short                    edgeset;
    struct MinElement        *next;
  } MINREC, *MINPTR;

typedef struct ListInfo
  {
    MINPTR            Header;
    short             NumOfElements;
  } LISTTYPE;
```

```
short popup(menu)
    PopUpEntry menu[];
```

Provide a "Pop-Up" menu. The menu argument points to an array of PopUpEntry structures, each of which is a string and a code. The array is terminated by a NULL string. The code of the menu item selected by the user is returned. If the user clicks outside the menu a negative value is returned.

```
typedef struct
 {
   char *string;               /* String to display. */
   unsigned char menuNumber;   /* Number returned if entry selected. */
  } PopUpEntry;
```

24.6. Defining and Using Fonts

```
short DefineFont(name, fileName)
    char *name, *fileName;
```

Defines a font to be used in subsequent SDF_TEXT items. The **name** is a pointer to a string giving the name of the font, for example, "Helvetica10B". The font is read by the VGTS from the file with the pathname given as the second argument. The **fileName** argument can be null to indicate a read from the standard place. The fontID returned by this call is used as the TypeData field of the AddItem call for these characters. A negative return value indicates an error. For example,

```
short roman = DefineFont("TimesRoman12", NULL);

AddItem(sdf, 0, x, x, y, y, roman, SDF_TEXT, "Hello")
```

will display the string "Hello" in the Times Roman font at 12 point size, at the position (x,y) on the screen.

24.7. Using the VGTS

The constants for mouse search types, VGT usage types, etc. are found in the include file **Vgts.h**. The stub routines are available in the default V library, so just including the option **-V** on your **cc68** command line for linking should work. Do NOT include the **-lVgts** option on your command line.

Use **-lVgts** on your **cc** command line for transparently running programs on a Unix system. Use **-I/usr/sun/include** to get the file **Vgts.h**. This package uses an escape sequence which can be used through PUP Telnet, IP Telnet, or with the remote command execution facility of the executive. Please contact the author for the details of this protocol if you wish to implement it on some other operating systems. There already are efforts underway for using this protocol from TOPS-20 assembler programs (e.g. SUDS) and InterLisp.

24.7.1. Cooking Your Pads

The following mode bits are maintained for each pad to indicate the amount of rawness of the I/O:

CR_Input — Change the CR (return) character to LF (UNIX newline) on input. This is for the benefit of UNIX programs which expect '\n' as a line terminator.

LF_Output — Change LF to CR-LF on output. That is, every line-feed operation is preceded by a return.

Echo — Echo input characters. This bit should be off for programs which can also run on the kernel console device.

LineBuffer — Wait for a line of input before returning. The user interface to the line editing feature is described in section 2.7.1.

PageOutput — Block the writer each time a pad fills up with output, and wait for the user to issue a command which unblocks the pad. The user interface to the PageOutput feature is described in section 3.3. This bit is "on" by default.

PageOutputEnable — When turned on in a ModifyPad request, this bit causes the new value of the PageOutput bit to be assigned to a user-controlled, "sticky" enable/disable bit. The PageOutputEnable bit should only be used by "privileged" programs (such as executives) as a means to allow the user to "permanently" disable paged output mode. A QUERY_FILE request will return the actual value of the PageOutputEnable bit.

DiscardOutput — When set, this bit causes all output to a pad to be ignored. It is automatically set when the user types 'q' to a pad that is blocked at the end of a page in PageOutput mode. It is automatically cleared whenever the VGTS sends input to a program that is reading from the pad. The bit may also be cleared "manually" via ModifyPad. Application programs should call ModifyPad to clear this bit before sending a prompt to a pad, to insure that the prompt is not discarded along with any previous output that was discarded at the user's request.

ReportTransition — Report every change of buttons on the graphical input device as a significant event.

ReportClick — Report events only when all the buttons have been released on the graphical input device.

NoCursor — Do not display a cursor in the indicated pad.

The default when pads are created, or commands are initially run by the executive, is for all the keyboard bits to be on, and the mouse bits to be off.

24.7.2. Other Interface Routines

The following routines to communicate with the VGTS via the I/O protocol interface are in the V library:

```
File *OpenPad(name,lines,columns,error)
        char *name;
        short lines, columns;
        SystemCode *error;
```

Returnsa file descriptor for a new pad. **error** is a pointer to the reply code, normally OK. A NULL pointer is returned on an error. Note that the file descriptor returned is open for writing. If you want to read from it, you must use OpenFile to create another file descriptor with the same fileserver and fileid.

```
SelectPad(file)
        File *file;
```

Causes the indicated file to be selected for input, and brought to the top.

```
ModifyPad(file,mode)
        File *file;
        int mode;
```

Sets the Cooked mode of the file. mode is some combination of the bits described in the previous subsection.

```
int QueryPad(file)
        File *file;
```

Returns the Cooked mode of the file, some combination of the bits described in the previous subsection.

```
int QueryPadSize(file,plines,pcols)
        File *file;
        short *plines, *pcols;
```

Gets the number of rows and columns of the specified pad, storing them in the shorts pointed to by **plines** and **pcols**. The value returned is the same as for the preceding function.

```
GetTTY()
```

Puts the terminal in raw mode. The UNIX version of this routine does the appropriate UNIX operation if standard input is a tty device, otherwise it sends the proper code for the remote execution facility.

```
ResetTTY()
```

Restores the mode before the last **GetTTY()**. Runs under UNIX as well, checking standard input properly.

```
GetGraphicsEvent(file,px,py,pbuttons)
        File *file;
        short *px, *py, *pbuttons;
```

Waits for a graphical event in the indicated VGT, and returns the world X and Y coordinates in the shorts pointed to by **px** and **py**. The state of the buttons is returned in the short pointed to by **pbuttons**. Use the file pointer **stdin** to get events in VGTs that were created with **CreateVGT()**.

```
SystemCode GetGraphicsStatus(file,px,py,pbuttons)
        File *file;
        short *px, *py, *pbuttons;
```

Returns after any motion the world X and Y coordinates in the shorts pointed to by **px** and **py**. The state of the buttons is returned in the short pointed to by **pbuttons**. The value returned will be EOF if the graphics cursor is not within a view of the given VGT.

```
GetEvent(file,px,py,pbuttons,cbuf)
        File *file;
```

```
        short *px, *py, *pbuttons;
        char *cbuf;
```

Waits for any event in the indicated VGT, and returns the world X and Y coordinates in the shorts pointed to by **px** and **py**, and the buttons in the short pointed to by **pbuttons** if the event is graphical, or else returns the characters in the buffer pointed to by **cbuf**. The return value is zero for a graphical event and the byte count for keyboard events.

RedrawPad(file)

Waits until the indicated pad is redrawn.

```
PadFindPoint(vgt,nlines,x,y,pline,pcol)
        short vgt, nlines, x, y;
        short *pline, *pcol;
```

Converts a set of world coordinates in x and y into a line and column position within a pad. Currently the **vgt** parameter is unused, and the number of lines must be specified in **nlines**.

24.8. Example Program

The following program can be run either under Unix or under the V system executive. The **#ifdef VAX** directives allow the programmer to conditionally compile code for one environment or the other. It first creates an SDF and VGT, then displays 100 random objects of various kinds.

```
/*
 * test.c - a test of the remote VGTS implementation
 * Bill Nowicki September 1982
 */

# include <Vgts.h>
# include <Vio.h>


# define Objects 100    /* number of objects */

short sdf, vgt;

Quit()
  {
     DeleteVGT(vgt,1);
     DeleteSDF(sdf);
     ResetTTY();
     exit();
  }


main()
  {
    int i;
    short item;
    long start, end;
```

```
# ifdef VAX
    printf("Remote VGTS test program\n");
# else VAX
    printf("VGTS test program\n");
# endif VAX
    fflush(stdout);
    GetTTY();
    sdf = CreateSDF();
    DefineSymbol( sdf, 1, "test" );
    AddItem( sdf, 2, 4, 40, 4, 60, NM, SDF_FILLED_RECTANGLE, NULL );
    EndSymbol( sdf, 1, 0 );
    vgt = CreateVGT(sdf, GRAPHICS+ZOOMABLE, 1, "random objects" );
    DefaultView(vgt, 500, 320, 0, 0, 0, 0, 0, 0);

    time(&start);
    for (i=12; i<Objects; i++ )
      {
        short x = Random( -2, 155);
        short y = Random( -10, 169);
        short top = y + Random( 6, 100 );
        short right = x + Random( 4, 120 );
        short layer = Random( NM, NG );


        EditSymbol(sdf, 1);
        DeleteItem( sdf, i-10);
        switch (Random(1, 6) )
          {
            case 1:
              AddItem( sdf, i, x, right, y, top, layer,
                       SDF_FILLED_RECTANGLE, NULL );
              break;


            case 2:
              AddItem( sdf, i, x, x+1000, y, y+16, 0, SDF_SIMPLE_TEXT,
               "Here is some simple text" );
              break;


            case 3:
               AddItem( sdf, i, x, right, y, y+1, 0,
                        SDF_HORIZONTAL_LINE, NULL );
               break;


           case 4:
               AddItem( sdf, i, x, x+1, y, top, 0,
                        SDF_VERTICAL_LINE, NULL );
               break;


           case 5:
               AddItem( sdf, i, x, right, y, top, 0,
                        SDF_GENERAL_LINE, NULL );
               break;
```

```
            case 6:
                AddItem( sdf, i, x, right, top, y, 0,
                        SDF_GENERAL_LINE, NULL );
                break;
            }
          EndSymbol( sdf, 1, vgt );
        }

      time(&end);
      if (end==start) end = start+1;
      printf("%d objects in %d seconds, or %d objects/second\r\n",
         Objects, end-start, Objects/(end-start));
      printf("Done!\r\n");
      Quit();
  }


Random( first, last )
  {
        /*
         * generates a random number
         * between "first" and "last" inclusive.
         */
    int value = rand()/2;
    value %= (last - first + 1);
    value += first;
    return(value);
  }
```

— 25 —
Fields: Using a Pad as a Menu

These routines allow you to set up a table of *fields* in a pad. They can be selected with the mouse, so that you can have a menu. The advantages over the standard pop-up menu are that you can have more choices, you can display more information with each choice, and the menu is always there.

With each field, you can associate a value, which can be displayed and edited.

The menu is an array of **Fields**. These are defined in **<fields.h>**. Each **Field** consists of:

```
typedef struct
  {
    short row;          /* field's row number in pad */
    short col;          /* leftmost character of field */
    short width;        /* width of field */
    long *value;
    int (*proc)();
    char *format;       /* format in which to display *value */
  } Field;
```

row and **col** indicate where in the pad the field begins. (row = 1 and col = 1 is the top left corner of the pad.) **width** is the length of the field in characters. Only one-line fields are supported. **proc** is not used by the package itself. The intended useage is:

```
field = GetField(...);
if (field) (*field->proc))(field->value);
```

or perhaps:

```
if (field) (*field->proc))(field);
```

format is discussed below.

25.1. Formats

format is a format like those used by **printf** and **scanf**. Together with the **value**, it determines the string to be displayed in the field. This string must be a least **width** characters long. It is a zero-terminated C (asciz) string. Formats are of the form:

prefix [conversion] suffix

Here prefix and suffix is constant text which is displayed. If a **%** is to be displayed, it must be written as **%%**. The following utility routine will do a string copy analogous to **strncpy**, except that **%**s are automatically copied:

```
char * StrToFormat(f, s, n)
    char *f;   /* destination string buffer where '%'s are to be doubled */
    char *s;   /* source string */
    int n;     /* count - buffer size */
```

The optional *conversion* describes how **value** is to be displayed/read. Its form is:

%[[-][0][fieldwidth][.precision][^]c

Here the **%** indicates the beginning of the conversion specification. The conversion type letter *c* marks the end of the conversion specification. The format is exactly as used by **printf**, except that there may be a data length specification λ. If **value** is a **short *** rather than a **int ***, λ must be given as **h**. If the **value** is a **double *** rather than a **float ***, λ must be **l**, or the conversion type letter *c* must be capitalized.

When fields are displayed, **sprintf** is used to do the conversion. The length specification λ is only used to dereference **value** (except for fields where the conversion type letter is **s**); it is stripped from the format before being passed to **sprintf**.

On input to fields, only the length specification λ and the type code *c* are passed to **sscanf**. If the type code is **e** or **g**, it is changed to **f**.

25.2. The Field Table as a Menu: Selecting an Action

```
Field * GetField(menu, menuLength, buttons, pad)
    Field *menu;
    int menuLength;
    short buttons;
    File *pad;          /* output pad */
```

If **button** != **0**, it is assumed that the mouse is down on procedure entry. **GetField** returns when the button state changes; if it changes to non-zero, **GetField** fails by returning zero. If **button** == **0**, **GetField** will first wait for an event. (It will fail unless it is a mouse button being pressed down.)

As long as the user keeps the mouse button down, display the selected field (if any) in inverse video. When the user releases the button, return the last selected **Field**, or if none, return 0.

The menu is terminated by the first negative **row** field, or when the **menuLength** count is exhausted.

25.3. Displaying Fields

```
PutField(buffer, field)
    char *buffer;   /* destination string buffer */
    Field *field;   /* source format and value */
```

More or less like **sprintf(buffer, field->format, *field->value)**.

```
DisplayFields(menu, menuLength, pad)
    Field *menu;
    int menuLength;     /* see GetField function */
    File *pad;          /* output pad where fields are to be written */
```

Display in the pad all the **string** fields, at the positions given by the **row** and **col** components.

The **width** components are ignored. This allows convenient display of material which the user cannot select ("write-protected" fields) either by using fields with **width** <= **0** or by having a **string** longer than the **width**.

25.4. User Input to Fields

```
EditField(field, stuff, out, in)
    Field *field; /* field whose *value is to be edited */
    int stuff; /* 0: old text should be cleared: 1: stuff into editor */
    File *out, *in; /* input and output sides of pad to use */
```

Move the cursor to the conversion part of the **field**. If **stuff** is 0, the old value is cleared from the screen; if it is 1, the old value is placed in the line editing buffer. Enter line-edit mode, and wait for the user to type in a line. If the user types ↑G, abort, redisplay old value and return -1. Else parse the line using **field->format**. If this succeeds, update ***field->value**, returning 1, else 0. In any case, redisplay things correctly.

EditStdFld(field)

Equivalent to **EditField(field, 1, stdout, stdin)**

ReadStdFld(field)

Equivalent to **EditField(field, 0, stdout, stdin)**

25.5. An Example

```
/* This is a program which adds up integers, optionally scaled */
#include <stdio.h>
#include <fields.h>
double Scale = 1.0, Total = 0.0;
int Value = 0;

Quit() { ... cleanup actions ...; exit(-1);}

NewValue(f)
    Field *f;
  {
    if (ReadStdFld(f) == 1)
        Total += Value * Scale;
  }

Fields Menu[] =
  {
/* VAL (defined in fields.h) coerces pointers and values to (int *) */
    {1, 41, 10, VAL &Scale, EditStdFld, "Scale: %G      "},
    {1,  1, 15, VAL &Value, NewValue,   "New value: %-8d"},
    {2,  1,  0, VAL &Total, 0,          "Total: %G.        "},
    {5,  1,  8, 0,          Quit,       "--Quit--"},
    LASTFIELD /* defined in fields.h */
  };

main()
  { Field *field;
    while (1)
      {
        putc('L' & 31, stdout); /* write FormFeed to clear screen */
        DisplayFields(Menu, 999, stdout);
        field = GetField(Menu, 999, 0, stdout);
        if (field) (*(field->proc)) (field);
      }
  }
```

Since the screen is updated every time here, we do not have to worry about garbage being left behind when the field becomes shorter. However, I have shown two solutions which can be used when this is not desired: In the `Value` field, we make *sure* the field doesn't become shorter, by left justification if needed. This loses if we want to output punctuation after the value, as in the `Total` field. In this case, we can make sure that we output enough trailing spaces to erase the garbage.

25.6. Limitations

No facilities yet for arrays.

— 26 —
SUN PROM Monitor Emulator Traps

The emulator trap interface functions in the V C library are listed below. These are extremely dependent on the version of the SUN workstation PROM monitor being used. The use of these functions should be avoided if at all possible; none of them are present in the Unix C library. For more information see the *Sun User's Guide*. Note that not all the traps mentioned there are available under the V kernel, since processes always run in user state.

```
int emt_getconfig()
```
Returns the current value of the "configuration register."

```
int emt_getmemsize()
```
Returns the size of the on-board RAM in bytes.

```
char emt_getchar()
```
Busy-wait input from the console. Will not work unless the kernel console device is closed to prevent it from "stealing" the characters.

```
int emt_putchar(c)
    char c;
```
Busy-wait output to the console.

```
int emt_ticks()
```
Returns the number of milliseconds since the monitor was last booted. Incremented at each memory refresh.

```
int emt_version()
```
Returns the version number of the PROM monitor.

```
int fbmode(newmode)
    int newmode;
```
Queries/sets the frame buffer mode.

```
setecho(flag)
    int flag;
```
Controls whether characters read using **emt_getchar()** are echoed.

— 27 —
Miscellaneous Functions

27.1. Time Manipulation Functions

The time-related functions in the V C library are described below. A few of them are not present in the Unix C library.

`stime(), time(), ftime()`

These are Unix system calls and are implemented here with simple library functions which emulate the Unix functions by performing the appropriate V kernel operations **`SetTime()`** and **`GetTime()`**. They have the same interface and functionality as in Unix; however, **`ftime()`** has the timezone hardwired as Pacific Time, since the V-System provides no time zone information.

`ctime(), localtime(), gmtime(), asctime(), timezone()`

These are identical to the Unix library functions.

```
sleep(seconds)
    unsigned seconds;
```

The invoking process is suspended from execution for the specified number of seconds. The actual time may be considerably longer than that specified if the process is not the highest priority ready process when its sleep time expires. **`sleep()`** is not sensitive to **`Wakeup()`**'s. Use the V system call **`Delay()`** for a **`Wakeup()`**-able suspension.

`unsigned GetRemoteTime()`

Returns the time according to the TIME_SERVER in seconds since January 1, 1970, GMT. Returns zero if it fails, e.g., no time server responded. Currently the Unix servers act as time servers.

27.2. Strings

The string-related functions in the V-System C library are described below.

27.2.1. Unix String Functions

The following functions are identical to the functions of the same name provided by Unix. See the *Unix Programmer's Manual* for documentation.

```
atof()      atoi()      atol()      crypt()
ecvt()      gcvt()      index()     rindex()
strcat()    strncat()   strcmp()    strncmp()
strcpy()    strncpy()   strlen
```

27.2.2. Verex String Functions

There is also another set of string manipulation functions which were ported from Verex. These include the following:

```
int Any(c, string)
    char c;  char *string;
```

Determine whether there is any occurence of the byte **c** in the string **string**, and return true (nonzero) if so, else false (zero).

```
char *Concat(dest, s1, s2, s3)
    char *dest, *s1, *s2, *s3;
```

Concatenate the strings **s1**, **s2**, and **s3**, store the result in **dest**, and return **dest**. **dest** must have enough room to store the resulting string. If any of **s1**, **s2**, **s3** are null pointers, the remaining arguments are ignored.

```
int Convert_num(string, delim, base)
    char *string;  char **delim;  unsigned base;
```

Parse the given string to extract a number of base **base** and return its value. If **base** is zero, the initial character of the string determines the base, as follows

#	Base 2
0 (zero)	Base 8
$	Base 16
otherwise	Base 10

Upon return, ***delim** is modified to contain a pointer to the delimiter that terminated the number.

```
char *Copy_str(string)
    char *string;
```

Copy the given string into a newly allocated region of memory and return a pointer to the copy. The new region is allocated using **malloc()** and may thus be freed using **free()** when the copy is no longer needed.

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

Compare the strings **s1** and **s2**. Return true (nonzero) if the strings are equal, else false (zero). Strings are considered to be equal if and only if they are of equal length (up to the terminating null byte) and each corresponding byte is the same.

```
int Hex_value(c)
    char c;
```

Return the value of **c**, interpreted as a hex digit. Return -1 if **c** is not a hex digit.

```
char *Lower(string)
```

```
    char *string;
```

Convert all alphabetic characters in **string** to lower case and return **string**.

```
unsigned Null_str(string)
    char *string;
```

Return true (nonzero) if **string** is a null string (i.e., of length zero), else return false (zero).

```
char *Shift_left(string, chars)
    char *string;  unsigned chars;
```

Delete the leftmost **chars** characters of **string** by shifting the remaining characters to the left, and return **string**. **string** must be at least **chars** characters long, but this condition is not checked.

```
unsigned Size(string)
    char *string;
```

Return the number of characters in the given string, i.e., the index of the null byte that terminates the string.

```
char *Upper(string)
    char *string;
```

Convert all alphabetic characters in **string** to upper case and return **string**.

27.3. Other Functions

```
qsort(base, nel, width, compare)
    char *base;  int nel, width;  int (*compare)();
```

Implements the quicksort algorithm. **base** is a pointer to the base of the data; **nel** is the number of elements; **width** is the width of an element in bytes; and **compare** is a function to compare two elements. The function **compare** must return an integer less than, equal to, or greater than zero, if the first argument is less than, equal to, or greater than the second, respectively.

```
setjmp(env)
    jmp_buf env;

longjmp(env, value)
    jmp_buf env;  int value;
```

setjmp() saves the stack environment in **env**, so that a later call to **longjmp()** will act like a return was made from the function which contained the call to **setjmp()**, with return value **value**.

```
char *ErrorString(error)
    SystemCode error;
```

Returns a pointer to a string describing the system request or reply code **error**, in human readable terms. Use this in error messages instead of printing the numeric value of the code.

```
PrintError(error, msg)
    SystemCode error;  char *msg;
```

Prints the string **msg** and an explanation of the SystemCode **error** on the standard error file.

Part III:
Servers

— 28 —
Servers Overview

All system services other than those implemented by the kernel are provided by sending a message to one of the system server processes. This manual describes the protocol for requesting these services, including the format of the request message, the format of the reply message, the possible values for the message fields, and the process that handles the request. This information is generally not required by application programmers because the protocol is implemented in a library of standard functions that provide system services via simple function calls. The library is described in the V-System program environment manual. More sophisticated use of the system requires the more detailed information in this manual.

This chapter describes some general message format standards used in communicating with servers. The next two chapters give details of two standard protocols, the V-System I/O Protocol and V-System Naming Protocol. The remaining chapters give the details of the particular servers, describing which of these protocols they implement, additional server-specific request types they provide, and the server-specific semantics of the services and requests each implements.

28.1. Message Format Conventions

System server protocols obey several system-wide conventions. The first short word of every request message contains a request code indicating the service requested. The first short word of every reply message contains a code indicating the successful completion of the request execution or the reason that the request was not executed normally. A requesting process can assume that the request has been completely executed when the reply message is received with a successful reply code (although in cases such as disk write-behind this may not be strictly true).

28.2. Standard System Request Codes

Each system request is allocated a unique request code to be placed in the first word of the request message when requesting that service. The request codes obey the message format conventions imposed by the kernel, as described for **Send( )** in the V environment manual. The manifest constant definitions for these request codes are defined in the standard C include file <Venviron.h>.

28.3. Standard System Reply Codes

The reply code returned in a message from a server is normally one of the following standard system replies:

OK — Operation successful.

ABORTED — An operation was aborted. For example, a network connection that has been aborted returns this code.

BAD_ADDRESS — Request contains an invalid memory address.

BAD_ARGS — Request contains field(s) with illegal or inconsistent values.

BAD_BLOCK_NO

The block number specified in an I/O request does not specify an existing block. If the file instance has attribute STREAM, the block number does not specify the block which is sequentially next in reading or writing.

BAD_BUFFER A buffer specified in the request lies (perhaps partially) outside the client's address space.

BAD_BYTE_COUNT
The byte count is larger (or smaller) than that supported by the server. On a file instance without the MULTI_BLOCK attribute, this is returned if the number of bytes requested to read or write is greater than the block size.

BAD_PROCESS_PRIORITY
The request specified an illegal value for a process priority.

BAD_STATE Request invalid at this time.

BUSY The server cannot satisfy the request at this time, probably because the requested resources are allocated to another client.

CURRENT_CONTEXT_INVALID
Normally only returned by library routines, not servers. The routine has detected that the current context of the calling process is invalid, probably because its process-id component refers to a nonexistent process.

DEVICE_ERROR
File or device-dependent error has occurred.

DUPLICATE_NAME
The request attempted to assign the same name to two different objects.

END_OF_FILE Attempt to read beyond file boundaries.

ILLEGAL_REQUEST
Invalid request code. The request was probably sent to the wrong type of server, one which could not perform that function.

INTERNAL_ERROR
The server detected an inconsistency in its own state. This error code may indicate a bug in the server.

INVALID_CONTEXT
The request contained a context identifier (see chapter 30) that was invalid.

INVALID_FILE_ID
The request contained an invalid file instance identifier.

INVALID_MODE
The mode specified as part of a CREATE_INSTANCE request is not valid.

IO_BREAK Returned from interactive files.

KERNEL_TIMEOUT
A timeout occured in the kernel when trying to send to a remote process. This error differs from NONEXISTENT_PROCESS in that the sending kernel did not receive a negative acknowledgement from the remote kernel, but for most purposes it can be handled in the same way. This error code is only generated by the kernel, but may be passed on by other servers.

MODE_NOT_SUPPORTED

The mode specified as part of a CREATE_INSTANCE request is not supported by this server.

NO_MEMORY The server was not able to obtain enough memory to satisfy the request.

NO_PDS The server was not able to create a process or team needed to satisfy the request.

NO_PERMISSION
Some kind of restricted operation was attempted.

NO_SERVER_RESOURCES
The server has (temporarily) inadequate resources to satisfy the request.

NONEXISTENT_PROCESS
The request was sent or forwarded to a nonexistent process, or a nonexistent process was specified in the request. This error code is only generated by the kernel, but may be passed on by other servers.

NONEXISTENT_SESSION
The request referred to a session (see chapter 36) which does not exist, or to an object which is not a session.

NOT_AWAITINGREPLY
The process specified in a request was not awaiting reply from the client.

NOT_FOUND The object named in the request was not found.

NOT_READABLE
Specified file instance does not have the attribute READABLE which is required for the requested operation.

NOT_WRITEABLE
Specified file instance does not have the attribute WRITEABLE which is required for the requested operation.

POWER_FAILURE
Operation was unsuccessful due to a power failure.

REQUEST_NOT_SUPPORTED
The server recognizes the request, but does not support it.

RETRY Client should repeat request.

SERVER_NOT_RESPONDING
The server failed to receive a response from another server specified in the request.

TIMEOUT An attempt to satisfy the request failed because of a timeout. Usually applied to network connections.

The `ErrorString()` function described in the V Environment manual will return a character string version of many of the system reply and request codes. The string form is much more informative than printing the codes in numeric form.

— 29 —
The V-System I/O Protocol

A standard input/output protocol is defined in V to provide transfer of data between processes in a uniform fashion. Using this protocol, a *client* process views and accesses data managed by a *server* process as a *file*. A file is a "view" of the data associated with an object or activity managed by a server. An object viewed as a file is a sequence of variable-size records or *blocks*.

To operate on an object viewing it as a file, it is necessary to create an *instance* of that file. The protocol is *object-based* in the sense that it is defined in terms of operations on a object, the file instance. File instance operations include: creating a file instance, querying a file instance, setting the file instance owner, reading, writing, and releasing file instances. There are also operations for setting a prompt string and break process associated with a file instance which are restricted to interactive file instances. A server that supports this protocol is called an I/O server or file instance server. (The term "file server" might be more appropriate if it did not have a different established meaning in the research literature on distributed systems).

A file instance is created by a server in response to a client request, which specifies the file, i.e. the object or data and the particular view and usage required. Conceptually, a file instance is an object which is created at the time of the client's CREATE_INSTANCE request, and (possibly) initialized to contain the same data as an existing, permanent file. When the instance is released by the client, the data contained in the instance is atomically written back to the corresponding permanent file. For some servers (for example, the internetwork server), however, there is no permanent file corresponding to an instance, while for others (for example, the device server), there is effectively no distinction between the instance and the permanent file -- changes in the instance are immediately reflected in the underlying file or I/O device. The current implementation of some storage servers (e.g., the V Unix server) also causes changes in an instance to be immediately reflected in the underlying file.

A file instance is uniquely identified by the server process identifier and the *instance identifier* returned by the CREATE_INSTANCE request. The creating process is made the owner of the file instance. The lifetime of the file instance and the validity of the instance identifier does not exceed that of the owner of the file instance. The owner of a file instance can be changed by the SET_INSTANCE_OWNER request.

The reply message to a CREATE_INSTANCE or QUERY_INSTANCE request specifies the server, file instance identifier, block length in bytes, file type, last block (written) in the file instance, number of bytes in the last block, and the next block to read.

The file *type* indicates the operations that may be performed on the file instance as well as the semantics of these operations. These types are defined in the include file <Vio.h>; file types are specified as some combination of the following attributes.

READABLE READ_INSTANCE operations are allowed on the file instance.

WRITEABLE WRITE_INSTANCE operations are allowed on the file instance.

APPEND_ONLY WRITE_INSTANCE operations are only effective to bytes in the file instance beyond the last byte associated with the instance at the time it was created.

STREAM All reading and writing is strictly sequential. The first READ_INSTANCE operation must specify the block number returned as *nextblock* in the reply to the CREATE_INSTANCE request. This next block number to read is incremented after each READ_INSTANCE operation. Its current value is returned by a QUERY_INSTANCE. A server must store

the last block read and allow it to be read again, to provide duplicate suppression on requests.

Similarly, each WRITE_INSTANCE operation must specify the block number returned as *lastblock* by CREATE_INSTANCE or QUERY_INSTANCE. This block number is incremented after every write operation. A server must ignore requests to rewrite the last block written, returning a reply code of OK, to provide duplicate suppression on requests.

A file instance without the STREAM attribute stores its associated data for non-sequential ("random") access. That is, on a non-stream file, for any *n*, block *n* may be read or written at any time, and reading block *n* will return the same data as was last written to block *n*.

Since each file models a single sequence of data blocks, objects which provide bidirectional communication, such as serial lines or network connections, are most appropriately modeled as a pair of file instances, one a READABLE STREAM, the other a WRITEABLE STREAM. Some servers may allow both instances to be created by a single CREATE_INSTANCE request.[13]

FIXED_LENGTH
: The file instance is fixed in length. The length is specified by the last block and last byte returned from a create or query instance request. Otherwise the file instance grows to accommodate the data written or else the length of the file instance is not known (as in the case of terminal input).

VARIABLE_BLOCK
: Blocks shorter than the full block size may be returned in response to read operations other than due to end-of-file or other exception conditions. For example, input frames from a communication line may differ in length under normal conditions.

  With a file instance that is VARIABLE_BLOCK, WRITEABLE, and not STREAM, blocks that are written with less than a full block size number of bytes return exactly the amount written when read subsequently.

MULTI_BLOCK
: Read and write operations are allowed that specify a number of bytes larger than the block size.

INTERACTIVE
: The file instance is a text line-oriented input stream on which a prompt can be set using the SET_PROMPT request and a break process can be defined using the SET_BREAK_PROCESS request. It also has the connotation of supplying interactively (human) generated input.

Not all of the possible combinations of attributes yield a useful file type. The file instance types supported by each server are documented with each server.

A client must specify a mode of usage for the file instance when creating it. The mode is one of FREAD, FCREATE, FMODIFY and FAPPEND. The modes of usage have the following semantics.

FREAD
: No write operations are to be performed, only reads.

FCREATE
: Any data previously associated with the described file is to be ignored and a new file

[13] A few existing servers bend this rule by assigning the same instance id to the input and output streams, even though block number *n* of the input stream is unrelated to block number *n* of the output stream. Strictly speaking, this behavior is in violation of the protocol, and we plan to change these servers eventually. A single STREAM that is both READABLE and WRITEABLE would have to return the data written to block *n* if block *n* is later read back. This type of file might be used to model a Unix-like pipe, but in fact, the V-System pipe server (see chapter 33) takes a different approach, creating a separate instance for each end of the pipe, with the connection between them invisible to the protocol.

instance is to be created. Write operations are permitted; read operations are also permitted if the file instance has type attribute READABLE.

FAPPEND Data previously associated with the described file remain unchanged. Write operations are permitted only to append data to the existing data.

FMODIFY Existing data is to be modified and possibly appended to. Both read and write operations are required. This is only supported on file instances that are not STREAM.

A server creates a file instance of a suitable type for the specified usage mode if it can. For example, the storage server provides file instances with type attributes READABLE, FIXED_LENGTH and MULTI_BLOCK in response to a CREATE_INSTANCE request specifying FREAD usage mode.

One of three modifiers may be used on the mode field of a CREATE_INSTANCE request.

FDIRECTORY Indicates that the given name specifies a context directory. See section 30.7.

FEXECUTE Specifies that the given file is to be executed as a program on the storage server machine. The mode must be FREAD or FCREATE. Respectively, one or two file instances are returned, which allow reading from the program's standard output, and optionally (in FCREATE mode) writing into its standard input. When two instances are created, the fileid of the second (readable) file instance is obtained by adding 1 to the fileid of the writeable instance (which is returned in the reply message). This mode modifier need not be supported by all storage servers.

FSESSION Specifies that a session is to be created on the server machine, using the (null-separated) user name and password passed in the filename field of the CREATE_INSTANCE request. The file server pid returned is the process id of the session. Releasing the file instance id returned will terminate the session. The session will also be terminated after the death of the instance owner. This mode modifier is only supported by storage servers that use the concept of "session." See section 36.

The following subsections give the format of the request message and the format of the reply, plus a description of the semantics for each operation in the protocol. These message formats are defined in the C include file <Vioprotocol.h>.

29.1. CREATE INSTANCE

requestcode CREATE_INSTANCE

filenameindex The index of the first byte in the filename to use in the name mapping.

type Type of file to create an instance of. This is used, for example, to specify the device to the device server and protocol to the internet server.

filemode Desired usage mode indicating FREAD, FCREATE, FAPPEND or FMODIFY, plus optionally FDIRECTORY, FEXECUTE, or FSESSION.

unspecified Server-dependent information specifying the file to be created.

contextid Specifies the context within the server in which the filename is to be interpreted. (See section 30.2.)

filename Pointer to a byte array containing the symbolic name of the server or file.

filenamelen Number of bytes in filename, not including the terminating null byte.

replycode	Standard system reply. If the reply code is not OK, the file instance was not created and the remainder of the reply is not defined.
fileid	File instance identifier. This is the number used in subsequent operations on the file.
fileserver	Process identifier of the server managing this file. This is not necessarily the same as the id to which the request was sent.
blocksize	Maximum size in bytes of a block.
filetype	Type attributes of the file instance as described at the beginning of this section.
filelastblock	Index of the last block in the file or of the last block written to the file instance if it is a STREAM file. Indexing is 0-origin.
filelastbytes	Number of bytes in the last block. For file instances which are not WRITEABLE and not FIXED_LENGTH, this field and the *filelastblock* field should return the maximum unsigned integer.
filenextblock	Number of the next block that can be read if this file is a READABLE STREAM.

The CREATE_INSTANCE request is issued either directly to the server or sent via a name server process. In the former case, the use of the fields of the request is server-dependent and is documented for each server. In the latter case, the *unspecified* field is not filled in by the client. The name server maps the symbolic name to a server and a server-dependent description of the file and then forwards the request to the appropriate server. An I/O server may not use the *filename*, *filenamelen*, and *filenameindex* fields if it does not support symbolic naming.

The *fileid* and *fileserver* uniquely identify the file instance created. The file instance exists until released or until the requesting process ceases to exist.

29.2. QUERY INSTANCE

requestcode	QUERY_INSTANCE
fileid	File instance identifier.

replycode	A standard system reply. If the reply code is not OK, the file instance was not queried and the remainder of the reply is not defined.
fileid	File instance identifier, same as the request for compatibility with the reply to the CREATE_INSTANCE request.
fileserver	Server process identifier.
blocksize	The maximum size in bytes of a block.
filetype	Type attributes of the file instance as described at the beginning of the section.
filelastblock	Index of the last block in the file or the last block written to the file instance if it is a STREAM file. Indexing is 0-origin.
filelastbytes	The number of bytes in the last block.

filenextblock Number of the next block that can be read if the file is a READABLE STREAM.

In response to a QUERY_INSTANCE request message, the server queries the file instance specified by fileid for the parameters supplied in the reply message. The reply message has the same format and semantics as the reply to a CREATE_INSTANCE request except for the reply code. For example, a reply code of NOT_FOUND to a CREATE_INSTANCE request indicates that the file specified does not exist, while a reply code of INVALID_FILE_ID to a QUERY_INSTANCE request indicates the file instance does not exist.

29.3. RELEASE INSTANCE

---

requestcode RELEASE_INSTANCE

fileid File instance identifier

releasemode Server-dependent action to perform when releasing the instance. This field is set to zero on a normal close.

---

replycode A standard system reply code.

In response to a RELEASE_INSTANCE request, the server invalidates the instance identifier, reclaims server resources dedicated to the instance and possibly performs some server-dependent function with the file instance data. A *releasemode* of 0 indicates normal completion of the use of the file instance. For example, in the case of the printer server, the file instance data is printed. In the case of the storage server, the data atomically replaces the previous version of the stored file data. A non-zero release mode causes the data to be discarded.

A server may release a file instance with a non-zero release mode if it detects that the process that created the instance no longer exists. A server should maximize the time before reusing a file instance identifier.

29.4. READ INSTANCE

---

requestcode READ_INSTANCE

fileid File instance identifier

blocknumber Index of the block in the file from which the read is to begin.

bufferptr Address of the data buffer in which the data is to be moved if more than IO_MSG_BUFFER bytes are read. That is, IO_MSG_BUFFER is the maximum number of data bytes that fit in the message.

bytecount Number of bytes to be read.

---

replycode Standard system reply code.

fileid Same as in request.

shortbuffer IO_MSG_BUFFER bytes containing the data bytes read if less than or equal to

IO_MSG_BUFFER bytes.

bytecount Number of bytes read.

In response to a READ_INSTANCE request, the server transfers up to *bytecount* bytes from the file instance starting at the block numbered *blocknumber*. If the number of bytes read is less than the number requested, the reply code indicates the reason. If the file instance has the type attribute VARIABLE_BLOCK and the block being read was not the full block size specified for the file instance, this case is not an error, and the reply may be OK, or END_OF_FILE if the last block was read. Note that a client may ignore the reply code if the returned byte count is equal to the requested byte count, so servers should set the byte count to zero on error conditions.

If the number of bytes read is less than or equal to IO_MSG_BUFFER, the data read is contained in the reply message starting at *shortbuffer*. If it is greater than IO_MSG_BUFFER, the data read is transferred into the space of the requesting process starting at the address *bufferptr*.

If the file instance has the type attribute STREAM, the block number specified must be the next block to read for this instance, which is incremented after the read. Reads always start at the beginning of the specified block. The values of bytes read that were not explicitly written are undefined. The number of bytes requested must be less than or equal to the block size unless the file instance has the type attribute MULTI_BLOCK.

29.5. WRITE INSTANCE

---

requestcode WRITE_INSTANCE, or WRITESHORT_INSTANCE if *bytecount* is less than or equal to IO_MSG_BUFFER.

fileid File instance identifier.

blocknumber Index of the block in the file instance at which the write is to begin.

shortbuffer Data bytes to be written if less than or equal to IO_MSG_BUFFER.

bufferptr Address of the data buffer if no more than IO_MSG_BUFFER bytes are being written. Otherwise, this field may be overwritten by the data bytes.

bytecount Number of bytes to be written.

---

replycode Standard system reply code.

bytecount Number of bytes written.

In response to a WRITE_INSTANCE or WRITESHORT_INSTANCE request, the server transfers up to *bytecount* bytes to the file instance starting at the block numbered *blocknumber*. If the number of bytes written is less than the number requested, the reply code indicates the reason. As with READ_INSTANCE, a client may ignore the reply code if the returned byte count is equal to the requested byte count, so servers should set the byte count to zero on error conditions.

If the number of bytes to write is less than or equal to IO_MSG_BUFFER, the data is assumed to be contained in the request message starting at *shortbuffer*. If it is greater than IO_MSG_BUFFER, the data is transferred from the space of the requesting process starting at the address *bufferptr*. Writes always start at the beginning of the specified block. Note that the separate request code WRITESHORT_INSTANCE is used

when the data is contained in the message only to be consistent with the kernel message format conventions. There is no READSHORT_INSTANCE needed because the data is passed back in the reply. That is, WRITE_INSTANCE specifies that segment access is being passed while WRITESHORT_INSTANCE specifies no segment access.

If the file instance has type attribute STREAM, the block number specified must be one greater than the last block in this file instance, which is incremented after the write. The number of bytes to write must be less than or equal to the block size unless the file instance has the type attribute MULTI_BLOCK.

29.6. SET INSTANCE OWNER

requestcode	SET_INSTANCE_OWNER
fileid	File instance identifier
instanceowner	Process identifier of new file instance owner.

replycode	Standard system reply code.

In response to a SET_INSTANCE_OWNER request, the server sets the file instance owner process to that specified by *instanceowner*. The requesting process must be the current owner of the file instance. The initial owner of a file instance is the process that created the instance.

29.7. SET BREAK PROCESS

requestcode	SET_BREAK_PROCESS
fileid	File instance identifier
breakprocess	Process to be "broken" when next break generated on this file instance.

replycode	Standard system reply code.

In response to a SET_BREAK_PROCESS request, the server sets the break process associated with the file instance to the process specified by *breakprocess*. When a break is generated on this file (the IO_BREAK reply returned to any outstanding read operations), the server issues a DestroyProcess kernel operation on the specified process.

This request is only supported on file instances with type attribute INTERACTIVE.

29.8. SET PROMPT

requestcode	SET_PROMPT
fileid	File instance identifier

promptstring Prompt string, which must be less than IO_MSG_BUFFER bytes long.

---

replycode Standard system reply code.

In response to a SET_PROMPT request, the server sets the prompt string output previous to every read operation to that specified. This request is only supported on file instances with type attribute INTERACTIVE.

29.9. QUERY FILE and NQUERY FILE

---

requestcode QUERY_FILE

fileid File instance identifier

---

requestcode NQUERY_FILE

nameindex The index of the first byte in the file name to use in the name mapping.

unspecified Up to the last three 32-bit words in the message.

namecontextid Context in which the name is to be interpreted.

nameptr Pointer to a memory segment containing the file name.

namelength Length of the segment in bytes.

---

replycode Standard system reply code.

unspecified Server dependent information.

In response to a QUERY_FILE or NQUERY_FILE request, the server returns server specific information about the file or file instance. For example, the VGTS returns the "cooking" bits, and the internet server returns connection information. A QUERY_FILE request specifies the file using an instance identifier, while a NQUERY_FILE request uses a character-string name. Both types of request return the same information.

29.10. MODIFY FILE and NMODIFY FILE

---

requestcode MODIFY_FILE

fileid File instance identifier

unspecified Server-dependent information.

---

requestcode NMODIFY_FILE

nameindex The index of the first byte in the file name to use in the name mapping.

unspecified	Server-dependent information. Up to the last three 32-bit words in the message.
namecontextid	Context in which the name is to be interpreted.
nameptr	Pointer to a memory segment containing the file name.
namelength	Length of the segment in bytes.

---

replycode	Standard system reply code.

The MODIFY_FILE and NMODIFY_FILE requests are supported by some servers to modify some attributes of the file or file instance. For example, the VGTS uses MODIFY_FILE to turn echoing on and off.

A MODIFY_FILE request specifies which file is to be modified by passing an instance identifier, while an NMODIFY_FILE request passes a character string name.

— 30 —
The V-System Naming Protocol

A number of V-System services use character string names to specify the objects to be operated on, and many standard message types include space for such a name. Examples include the CREATE_INSTANCE request and several other requests described above as part of the I/O Protocol.

Name mapping in the V-System is performed by a collection of cooperating server processes rather than a single, monolithic "name server." The *V-System Naming Protocol* consists of a uniform format for request messages that contain symbolic names, and a small set of request types which must be handled specially by any server that implements the protocol. The protocol also specifies conventions for forwarding partially-interpreted requests from one server to another.

30.1. Character String Names

Syntactically, a *character string name (CSname)* is a sequence of zero or more bytes, of a specified length or else terminated by a null byte. Operationally, a character string name is a byte string as above that is used to specify an object relative to a server that can interpret the name. There is no universal limit on the length of character string names. Two CSnames are equal if and only if they are byte-wise identical and equal in length (where a null in the name takes precedence over the length specification).

Although CSnames may contain arbitrary bytes, they are generally specified or chosen by the client (as opposed to the server) and are usually human-readable ASCII strings.

The term *character string name handling server (CSNH server)* refers to any server that performs character string name mapping, regardless of what else it does. The term *CSname request* describes any request containing a character string name that must be mapped in order to perform the requested operation.

30.2. Contexts and Context Ids

In general, the interpretation of a string name depends on the context in which the name is used. Formally, a context is a set of (name, object)-tuples. A context can have an arbitrary set of members in theory. In the V-System, the context of a name includes (1) the server to which the name is to be sent, and (2) the place within that server's naming hierarchy where interpretation is to begin, or more generally, the context within the server. A server is specified by its process id, while a context within a server is specified by a *context identifier*. A context identifier is a 32-bit identifier assigned by the server. Thus in general, a context is specified by a (server-pid, context-id) pair.

This definition does not specify detailed semantics for contexts, leaving it to individual servers. This is similar to the I/O protocol where, for example, the semantics of writing to a file instance is not specified but is server-dependent. Thus, each name server must specify the semantics of its contexts. For example, while a file server may implement a purely hierarchical name space and only implement contexts that modify the semantics of so-called relative pathnames, a internetwork server may implement contexts that correspond to different networks, or sets of hosts talking particular protocols, etc.

A context-id has the same lifetime as the server. Thus, after a context-id is acquired by a client, there is no need (and no way) to release it when the client is finished using it. A context-id identifies the context itself, not an "instance" of the context. Therefore, we have made context-ids relatively long (32 bits).

Basically, character string name mapping is structured as three levels: server, context and CSname. However, a CSname may be structured hierarchically, as in the case of a filesystem pathname. The naming protocol is independent of this structure, though usually each component in a hierarchical name will be the character string name of a context in which the rest of the name is interpreted.

It is expected that, given a character string name, a server and a context id, the interpretation of that character string name is fully specified independent of the operation requested.

30.3. Well-Known Context Ids

We require that context-id 0 (called DEFAULT_CONTEXT) represent a valid context on every CSNH server. In general, DEFAULT_CONTEXT should be a reasonable default for clients that are not sure which context within a server a name should be mapped in, but do know the server. For example, a server that provides access to a Unix file system should map DEFAULT_CONTEXT to the root directory (known as "/"). A server that provides only one context should number it 0.

Other small context identifiers (less than 16, say) are reserved for use as "well-known" contexts. There is a need for some servers to publish certain context ids, similar to DEFAULT_CONTEXT, and some servers may provide certain contexts which have special properties. Currently defined well-known contexts include

DEFAULT_CONTEXT
: As described above.

PUBLIC_CONTEXT
: Holds publically-available V programs on storage servers.

LOGIN_CONTEXT
: The home directory of the owner of a session, on storage servers that implement the concept of a session.

ANY_CONTEXT
: A special value used with the GET_FILE_NAME and GET_CONTEXT_NAME operations. When returned by one of these operations, it indicates the name is an "absolute" name, valid in any context on the given server. When passed in the contextid field of a GET_CONTEXT_NAME request, it acts as a wild card, i.e., the server receiving the request may return the name of any context on the server specified in the request.

30.4. Name Request Format

All V-System request messages that contain CSnames are built on a common skeleton, defined as the NameRequest structure in the standard header file <Vnaming.h>.

requestcode	Any valid request code that grants read access to a segment.
nameindex	The byte offset of the name, within the segment specified by the last two long words of the message.
unspecified	Request-specific information, up to the last three long words in the message.
namecontextid	A 32-bit identifier for the context in which this name is to be interpreted.
nameptr	Pointer to the segment containing the symbolic name.
namelength	Length of the segment containing the name.

The reply is not specified by this protocol because it is generally dependent on the operation requested.

The name need not be first in the segment but is considered to start at the byte offset specified by *nameindex*. If the name is not last in the segment, it must be terminated by a null. A CSNH server may reject a request if the total segment size is too long for it to handle.

30.5. Name Parsing and Forwarding

A CSNH server follows the following algorithm in handling a request containing a CSname.

If the server does not provide pointers to contexts in other servers as part of its name space, it may interpret the name in any way it chooses.

Otherwise, the server begins by looking at the name itself, not the request code. Since this request may have been directed to another server (to which it will eventually be forwarded by this algorithm), the request code is irrelevant at this point.

Names are ordinarily interpreted left-to-right, if the server implements hierarchical naming. The server initializes the variable *CurrentContext* to the context id specified in the request. As each component of the name is parsed, it is looked up in the current context. If the name specifies a context, *CurrentContext* is updated. If the new context is implemented by some other server, the *nameindex* field in the request message is updated to point to the first character of the name not yet parsed, the *namecontextid* field is set to *CurrentContext*, and the request is forwarded to the server that implements the context.

A server with a flat name space may ignore the contextid field of requests, but it must set this field when forwarding requests to other servers.

30.6. Standard CSNH Server Requests

There are several standard CSNH requests, which should be implemented by all CSNH servers, and others which need only be implemented by context prefix servers (see chapter 42), but may be implemented by others as well. All of the request and reply formats described below are subsets of the ContextRequest structure defined in the standard system header file <Vnaming.h>.

30.6.1. GET CONTEXT ID

requestcode	GET_CONTEXT_ID
nameindex	The byte offset of the name, within the segment specified by the last two long words of the message.
namecontextid	Context in which to interpret the given name.
nameptr	Pointer to the segment containing the symbolic name.
namelength	Length of the segment containing the name.

replycode	Standard system reply code.
serverpid	The serverpid component of the named context.
contextid	The contextid component of the named context.

entrytype	Optional, server-specific type information.
instanceid	File instance id associated with the context, if any. Server-specific.
otherinfo	Optional, server-specific information.

Given a CSname that names a context, this request returns a (serverpid, contextid) pair which identifies the same context.

30.6.2. GET CONTEXT NAME

requestcode	GET_CONTEXT_NAME
serverpid	The serverpid component of the context for which a name is to be found.
contextid	The contextid component of the context.
nameptr	Pointer to a buffer in which the name is to be returned.
namelength	Size of the buffer.

replycode	Standard system reply code.
serverpid	The serverpid component of the context in which the returned name is valid.
contextid	The contextid component of the context in which the returned name is valid.
nameptr	The value provided is returned unchanged.
namelength	Length of the returned name.

Returns a CSname corresponding to the specified (serverpid, contextid) pair, if one is known to the server receiving the request, plus the server and context-id required to fully qualify the CSname. The context-id returned will be ANY_CONTEXT, if possible, and the server will ordinarily be the one to which the request was sent.

Since the inverse mapping from (serverpid, contextid) to CSname is not well-defined in general, a server may sometimes fail to satisfy this request despite its best efforts. Also, there may be many possible choices for the name that is to be returned. Servers should attempt to return a name that is as informative to a human user as possible.

30.6.3. GET FILE NAME

requestcode	GET_FILE_NAME
instanceid	A file instance id for the file whose name is desired.
nameptr	Pointer to a buffer in which the name may be returned.
namelength	Size of the buffer.

replycode	Standard system reply code.
serverpid	The serverpid component of the context in which the returned name is valid.
contextid	The contextid component of the context in which the returned name is valid.
nameptr	The value provided is returned unchanged.
namelength	Length of the returned name.

Returns a CSname for the file associated with the specified file instance, plus the server and context-id required to fully qualify the file name. The context-id returned will be ANY_CONTEXT, if possible, and the server will ordinarily be the one to which the request was sent.

30.6.4. ADD CONTEXT NAME

requestcode	ADD_CONTEXT_NAME
nameindex	The byte offset of the name, within the segment specified by the last two long words of the message.
serverpid	Server pid to assign to name.
contextid	Context id to assign to name.
entrytype	Server-specific type information.
instanceid	Instance id associated with context, if any.
otherinfo	Server-specific.
namecontextid	Context in which to interpret (or define) the given name.
nameptr	Pointer to the segment containing the symbolic name.
namelength	Length of the segment containing the name.

replycode	Standard system reply code.

The ADD_CONTEXT_NAME operation defines a new CSname to refer to an existing context. The existing context is specified in the (serverpid, contextid) fields of the request. The specified CSname is interpreted according to the naming protocol, in the context specified by *namecontextid*, until the mapping algorithm reaches a context in which the remainder of the name is not defined, at which point it is added to that context.

This operation need only be implemented by context prefix servers, but of course all CSNH servers must be able to forward it in accordance with the naming protocol.

30.6.5. DELETE CONTEXT NAME

requestcode	DELETE_CONTEXT_NAME
nameindex	The byte offset of the name, within the segment specified by the last two long words of the

	message.
namecontextid	Context in which to interpret the given name.
nameptr	Pointer to the segment containing the symbolic name.
namelength	Length of the segment containing the name.

replycode	Standard system reply code
serverpid	The serverpid component of the name's former value.
contextid	The contextid component of the name's former value.
entrytype	Server-specific type information formerly associated with the name.
instanceid	File instance identifier formerly associated with the name, if any.
otherinfo	Server-specific information formerly associated with the name.

Delete the specified context name, making it no longer meaningful. The context associated with this name is not deleted. This operation need only be implemented by context prefix servers, but of course all CSNH servers must be able to forward it in accordance with the naming protocol.

30.7. Context Directories and Object Descriptors

Each context consists of a set of (name, object)-tuples and is implemented by a server process. The discussion so far has concentrated on performing operations on specific objects and the protocol for specifying a particular object. However, an important aspect of system operation is supporting query operations about objects or sets of objects. A simple example is that of listing the names of all objects in a given context. In general, one may wish to list a variety of information about objects in a context, perhaps ignoring some of the objects based on their properties.

Each CSNH server implements one or more *context directories* of objects that it manages. A context directory appears as a file of records, with each record specifying an object in the associated context. A directory file is accessed using the I/O protocol with the CREATE_INSTANCE request specifying the name of the context to be used. The FDIRECTORY bit is set in the mode field of such a request. A client can then use the standard I/O routines to read the contents of the directory and derive the information required. The selection of the information required is done by the client, not the server. The client may also be able to modify some or all of the fields of a directory record by writing it, using the standard I/O protocol. A server is not obligated to make all fields presented in a directory modifiable. If a client attempts to change a non-modifiable field, that field should be left unaltered, but any other changes indicated in the request should be carried out.

The FDIRECTORY bit is primarily for the benefit of Verex-like file systems, which permit each node in the naming hierarchy to be (in Unix terms) both a file and a directory. It discriminates between access to the data content of such a node, and the context directory associated with it. Also, servers that do not implement character sting naming at all can use this bit to distinguish between requests to access one of the objects they manage and requests to read their context directory.

Each record in a directory starts with a *descriptor-type* field that specifies the format of the record describing the object. For space economy, this field is an identifier that specifies a description of the record format stored elsewhere in a system database of such formats. (The standard formats and descriptor type identifiers are defined in the header file <Vdirectory.h>.) Applications can read a directory and extract the required information by referring to the descriptor-type field and these format descriptions, even when a directory

contains heterogeneous records.

A similar query activity involves accessing the descriptor of a single object. For efficiency and consistency, this is supported by a separate READ_DESCRIPTOR function on the object (as opposed to being subsumed by the context directory facility), which returns the same record as found in the context directory. A corresponding WRITE_DESCRIPTOR operation is available for modifying an object's descriptor.

There is no implication that a server need store information about objects as it is presented in a context directory. For instance, the Unix file system stores the names of files separate from their descriptors with the association provided by so-called "i-node numbers." A context directory entry in this case is fabricated dynamically by replacing the i-node number in each record by its descriptor.

The standard descriptor reading and writing operations are described below. The message formats used are described by the DescriptorRequest and DescriptorReply structures defined in <Vdirectory.h>.

30.7.1. READ DESCRIPTOR and NREAD DESCRIPTOR

requestcode	READ_DESCRIPTOR or NREAD_DESCRIPTOR
nameindex	The byte offset of the name, within the segment specified by the last two long words of the message (NREAD_DESCRIPTOR only).
fileid	File instance id of the file whose descriptor is to be read (READ_DESCRIPTOR only).
dataindex	The byte offset from the start of the specified segment where the returned descriptor is to be placed.
namecontextid	The context id of the context in which the given name is to be interpreted (NREAD_DESCRIPTOR only).
segmentptr	Pointer to a buffer which contains the object name (for NREAD_DESCRIPTOR), and in which the descriptor is to be returned.
segmentlen	Length of the buffer.

replycode	Standard system reply code.
dataindex	Returned unchanged.
segmentptr	Returned unchanged.
segmentlen	Returned unchanged.

These request types provide a way of reading the descriptor (context directory entry) of a single object. READ_DESCRIPTOR specifies the object by file instance id, while NREAD_DESCRIPTOR specifies it by CSname.

30.7.2. WRITE DESCRIPTOR and NWRITE DESCRIPTOR

requestcode	WRITE_DESCRIPTOR or NWRITE_DESCRIPTOR
nameindex	The byte offset of the name, within the segment specified by the last two long words of the message (NWRITE_DESCRIPTOR only).

fileid	File instance id of the file whose descriptor is to be modified (WRITE_DESCRIPTOR only).
dataindex	The byte offset from the start of the specified segment where the new descriptor value begins.
namecontextid	The context id of the context in which the given name is to be interpreted (NWRITE_DESCRIPTOR only).
segmentptr	Pointer to a buffer which contains the object name (for NWRITE_DESCRIPTOR), and the new descriptor value.
segmentlen	Length of the buffer.

---

replycode	Standard system reply code.
dataindex	Returned unchanged.
segmentptr	Returned unchanged.
segmentlen	Returned unchanged.

These request types provide a way of modifying the descriptor (context directory entry) of a single object. WRITE_DESCRIPTOR specifies the object by file instance id, while NWRITE_DESCRIPTOR specifies it by CSname. The server will modify each field in the object's descriptor for which the value written differs from the existing value, if the field is client-modifiable and the new value is legal. A client normally uses one of these operations by first reading the descriptor, then modifying the field(s) of interest, and finally writing it back.

— 31 —
Device Server

The device server provides access to the raw kernel-supported devices via the I/O protocol. It is implemented directly by the kernel as a pseudo-process as opposed to being a normal process like other system servers. Consequently, it is always configured when the V kernel is used. However, the device server behaves as any other I/O server process as far as applications are concerned.

The device server appears as a single process that supports different types of devices using the same I/O protocol. Access to a device is established by sending a create instance request to the pid returned by GetPid(DEVICE_SERVER, LOCAL_PID), or, if the standard context prefix server has been configured, by prefixing the device name with the context name "[device]" in a create instance request or Open() call. Using the standard information returned by the create instance request, the device can then be accessed using I/O protocol messages, either directly or by means of the standard I/O library routines described in chapter . There are also some device-specific operations defined for some devices. The currently supported devices are described below.

31.1. Ethernet

The Ethernet interface is accessed by specifying a device name of the form *enetts*, where *t* is replaced by the Ethernet type, either *3* for 3 Mbit experimental Ethernet, or *10* for standard Ethernet, and *s* is a suffix, which is null for the first Ethernet interface, *a* for the second, *b* for the third, and so forth. Currently only one Ethernet instance may exist at a time and only one Ethernet interface is supported, and the name *ethernet* is defined as an alias for either *enet3* or *enet10*, whichever is present.

The standard header file <Vethernet.h> defines Ethernet-specific information, including the Ethernet packet format and various constants such as ENET_MAX_DATA, the maximum size of the data portion of an Ethernet packet.

In a create instance request, the filemode must be FCREATE. The type of an Ethernet instance is always a readable, writeable, variable block stream.

Read and write instance requests are standard except for the Ethernet block format. The Ethernet is only sensibly accessed as a block (or packet) device, as opposed to a byte stream. The Ethernet block format is exactly that expected by the interface, namely, on the 3 Mbit Ethernet, one byte for destination, one byte for source, two bytes for Ethernet packet type, followed by some number of data bytes, and on the 10 MBit Ethernet, six bytes for destination, six bytes for source, two bytes for packet type, followed by data bytes. The number of bytes specified in a write and returned by a read includes the destination, source and type bytes as well as the data bytes.

An Ethernet-specific QUERY_FILE request is supported that returns the host number, the number of collisions, receiver overflows, CRC errors, receiver synchronization errors, transmission timeouts detected, and the number of valid packets received. The host number should be used as the source address for every packet transmitted. The format for the request and reply messages is given by the QueryEnetRequest struct defined in <Vethernet.h>.

31.2. Mouse: The Graphics Pointing Device

The mouse is a graphics pointing device. It provides a means of indicating a coordinate position plus signalling different states via its three buttons. The device server provides access to the mouse through the I/O protocol, thus viewing it as a file.

The mouse file appears as a 10-byte file divided into 3 major fields. The first two bytes specify the mouse button positions, the three buttons being the low-order three bits of the second byte. A bit with value 0 indicates the button is up, otherwise down. The next 4 bytes specify its current X coordinate. The last 4 bytes specify its current Y coordinate. The kernel updates this file according to the input from the device. These fields are specified in <Vmouse.h> as *buttons*, *xcoordinate* and *ycoordinate* with MBUTTON1, MBUTTON2 and MBUTTON3 specifying the button bit field assignments in the *buttons* field.

A create instance request for a mouse specifies the name *mouse* in the filename field. Only one mouse and one instance of that mouse are currently supported. The *filemode* field of the create instance request must be FCREATE. The mouse file instance created is initialized to have X and Y coordinates of 0. It has type attributes READABLE, WRITEABLE, and FIXED_LENGTH.

Read and write requests must specify block 0 and a byte count of 10 bytes. A read instance request returns 10 bytes specifying the current state of the mouse "file." A read instance request is queued until a change to the mouse file occurs, providing no change has occurred since the last read request. Thus, for instance, a mouse reader process that repeatedly reads from the mouse and updates a cursor is suspended when the mouse is not being moved and no button positions are changing. Conversely, the read returns every time a change does occur.

A write instance operation changes the kernel-maintained record of the mouse button positions and the X and Y coordinates to that specified by the 10 bytes in the buffer. Setting the mouse buttons in the kernel has no significant effect because this record is updated to agree with the actual button positions on the next input (or "squeak") received from the mouse.

There is no need to provide a query function that simply returns the current mouse position because that should always be stored outside the kernel. That is, the application decides where the mouse is; the kernel simply updates the position relative to the absolute position specified.

The kernel does not provide any scaling of mouse movements. That is left to the application.

31.3. Serial Line

The kernel device server provides access to raw serial lines through the serial device. Two serial lines are supported, but only one instance for each may exist at a time.

In a create instance request, the name *serial0* or *serial1* specifies a serial line. The *filemode* must be FCREATE. The instance id returned is used for output; the instance id + 1 is used for input. Parameters for the input instance can be obtained using QueryInstance.

Each serial line is a pair of streams, one readable and one writeable. Characters read from each serial line are buffered in the kernel until a process reads from the device, but the buffer is rather small, so a user who is interested in input from a serial line should keep a process "listening" to it at all times. The serial line device does not provide any echoing of input characters, nor does it convert input editing or conversion of newline characters to a carriage return/line feed sequence on output.

The serial device drivers support QueryFile and ModifyFile operations to allow changing such parameters as the data rate, bits per character, and the state of the modem control outputs DTR and RTS. The necessary message structures and constants for these operations are defined in the standard header file <Vserial.h>. (At this writing, the Query and Modify operations are not implemented in the Sun-1 serial device driver.)

31.4. Console

The kernel console device is intended to provide a measure of hardware independence to programs doing interactive character stream input and output. The console device provides access to the console keyboard and display of the workstation the kernel is running on, independent of the type of workstation. On workstations whose keyboards are connected to serial line 0, reading from the console device reads from serial line 0; on others, it reads from the port to which the keyboard is connected. Likewise, on workstations with frame buffers, writing to the console device draws characters on the frame buffer; for those without, writing to the console sends output to serial line 0. In cases where the console uses serial line 0, instances for serial line 0 and the console may not both exist at the same time.

A create instance request must specify filemode FCREATE, and name *console*. The console device is a pair of streams, one readable and one writeable. As with the serial line device, the instance id returned by a CreateInstance is writeable, and that instance id + 1 is readable. The parameters of the second instance can be obtained using QueryInstance. Both instances are marked INTERACTIVE, but SET_PROMPT and SET_BREAK_PROCESS are not supported.

Console device input is buffered in the same way as serial line input (see above). The console device does not provide any echoing or output conversion, but it does make an effort to sound the workstation's beeper when an ASCII BEL character is output.

The console device is automatically opened by the kernel upon creation of the first team, and is ordinarily never closed.

31.5. Null Devices

Two null devices are available, and are normally configured into all versions of the V kernel. The *nullin* device is a readable stream that returns an end-of-file indication on every read attempt. The *nullout* device is an endless sink for output.

— 32 —
Exception Server

The exception server handles processes that have incurred a processor exception during their execution. It is included in programs that run directly under the V kernel by including a call to **InitExceptionServer()** at the beginning of the program. This call returns the pid of the exception server if successful, else 0. If an exception server already exists, **InitExceptionServer()** will not start another. The pid of the exception server is also returned by

```
GetPid(EXCEPTION_SERVER, LOCAL_PID)
```

The standard V executives automatically start up an exception server.

When a process incurs an exception, it causes a trap which is fielded by the kernel. The kernel effectively causes the process to send a message to the exception server with the contents of the message describing the exception incurred. If there is no exception server, the kernel disables the faulting process by causing it to send to itself, which permanently blocks the process.

The exception server checks to see if another exception handler has registered for this process or an ancestor. If so, it forwards the message to the handler. For ordinary programs, arrangements are made for such messages to be passed on to the V debugger (described in the *V-System Commands Manual*). The format of the exception request and registration messages are defined in <Vexceptions.h>. The only request types supported are EXCEPTION_REQUEST and REGISTER_HANDLER. The REGISTER_HANDLER request code is used both for registering and deregistering handlers. EXCEPTION_REQUEST messages should only be generated by the kernel.

If no process was registered, the exception server prints a message on the screen indicating the type of exception, the pid of the faulting process, and the instruction, program counter and status register at the time the exception occurred. The exception server then destroys the faulting process, thus preventing it from doing further harm. Note: the program counter may have been incremented beyond the actual instruction incurring the exception so it should not be considered exact, although the error message routine attempts to find the correct PC by searching for the opcode of the instruction that was reported in the exception message.

The exception server and its standard message printing routine are included in a special V exceptions library. The loader may be instructed to search this library using the **-lVexcept** option on its command line. The error printing routine is available to other exception handlers as

```
short *StandardExceptionHandler(req, pid, fout)
    ExceptionRequest *req;
    ProcessId pid;
    File *fout;
```

where **req** points to the exception request message, **pid** is the process id of the process that incurred the exception, and **fout** is the file on which the message is to be printed. The routine returns the PC value at the time of the exception, corrected as described above.

— 33 —
Pipe Server

The pipe server is an I/O server that implements a synchronized stream file called a pipe. A pipe is a unidirectional flow-controlled communication channel between two processes using the standard I/O protocol. V pipes are similar to Unix pipes.

A pipe file instance is type STREAM, VARIABLE_BLOCK, and READABLE (for the read end) or WRITEABLE (for the write end).

In response to a CREATE_INSTANCE request, the pipe server creates an instance of a pipe, which is actually two file instances representing the read and write ends of the pipe. The file id returned in the reply to the CREATE_INSTANCE request is the file id of the write end. The file id of the file instance for the read end is one greater than the file id for the write end. The file instances are owned initially by the processes specified in the *readowner* and *writeowner* fields of the CreatePipeRequest. When a pipe is created, it is allocated a fixed number of buffers between 2 and 10 as specified by the *buffers* field of the CreatePipeRequest. Include <Vpipe.h> in a program to define CreatePipeRequest.

Pipe synchronization provides that a request to read a block that has not yet been written is queued until that block is written. Also, a request to write a block when the current buffer limit for the pipe is exceeded is queued until buffer space is available.[14] A request to read from an empty pipe whose write file instance has been released is replied to with an END_OF_FILE reply code. When the read end file instance is released, unread data is discarded and the data of subsequent writes to the write instance are discarded with the write returning successfully. A pipe no longer exists when both the read and write instances are released. The pipe server periodically checks that the owners of both file instances of the pipe exist. When the server determines that the owner of an instance no longer exists, it effectively releases that instance.

The pipe server is located by

```
server_pid = GetPid(PIPE_SERVER,ANY_PID)
```

where the pipe server may be local to the workstation or located on a server node.

The pipe server can be compiled as an independent V program or included in another program. To include the pipe server directly in a V program, call the function **InitPipeServer()** at the start of the program and cause the linker to search the pipe server library when loading the program (i.e., add -lVpipe on the C compilation command line). The standard V command *pipeserver* may be run in the background to provide a local pipe server on any workstation. The V executive automatically starts up a local pipe server if there is not one available when a pipe is needed.

[14] Actually only one reader and one writer are queued; the rest are replied to with a RETRY reply code.

— 34 —
Internet Server

The internet server is an I/O server that provides network communications using any of several protocols. It is essentially a *protocol converter* which allows applications which communicate by means of the V I/O protocol to communicate with hosts which can only (or prefer to) be reached by some other protocol. As such, the server has been structured in a manner which allows easy addition and deletion of protocols as needed. The server consists of a general framework, which is independent of the particular protocols being supported, and one or more protocol-specific modules. Each module implements a particular protocol and must interface that protocol to the requirements and facilities provided by the server's general framework. Currently the DARPA Internet protocols IP and TCP, and the Xerox PUP datagram protocol are supported.

34.1. Running the Internet Server

The internet server can be compiled as an independent V program, or linked into another program.

The standard V command "internetserver" may be run in the background to provide a local internet server on any workstation. The internetserver program by default will only register the server for the logical id INTERNET_SERVER on a local basis. Specifying the -g option to the internetserver program will cause it to register itself globally so that it can create connections for arbitrary hosts in the V system. This facility allows local hosts to avoid spending some 100K of memory for this server.[15] Two additional switches are available with the internet server. -d turns on debugging print-outs; and -q starts up a "query" process which can be used to query the internal state of the server from the user's keyboard. Normal users should not need to concern themselves with these options; they are intended mainly for people who are adding additional protocols to the server.

To include the internet server in another V program, have it create a process which executes the function

```
InitInternetServer(qFlag, localFlag, debugFlag)
    int qFlag;          /* Set up query process for runtime
                           diagnostics if qFlag is true. */
    int localFlag;      /* True if internetserver should be local. */
    int debugFlag;      /* True if debug output should be printed. */
```

and cause the linker to search the V internet library when loading the program (i.e. add -lVinternet on the C compilation command line). It is generally preferable to run the internet server on its own team by invoking the internetserver program described above, rather than linking it into another program.

34.2. Accessing the Internet Server

Once the internet server has been started it can be accessed using the I/O protocol plus the protocol-specific requests and parameters specified in <Vnet.h>.

A CREATE_INSTANCE request to the internet server must specify the mode FCREATE. It results in the creation of two instances, one of type READABLE, VARIABLE_BLOCK, and STREAM, the other of type WRITEABLE, VARIABLE_BLOCK, and STREAM. The parameters of the writeable instance are returned

[15] This can degrade performance however. For bursty applications such as telnet connections it usually not a problem.

in the CreateInstanceReply. The readable instance has an instance id equal to the id of the writeable instance plus 1; its parameters can be obtained using QUERY_INSTANCE.

An internet server connection is owned by the process which requested its creation. If that process should die then the connection is aborted. Ownership of a connection can be passed on to another process by means of the SET_INSTANCE_OWNER request.

34.3. DARPA Internet Protocol (IP)

Possession of an IP network instance provides a process access to the network for sending and receiving IP packets of a specific IP protocol type. Differing IP instances are delineated by the protocol field in the IP packets. Any protocol id value may be specified when creating the instance except for those values already taken. For example, the value for TCP, is already taken by the TCP implementation inside the internet server itself. Creating an instance with protocol 0 yields a "promiscuous" instance that receives all protocol types which have not been specified by any other active IP instances.

IP network instances expect WRITE_INSTANCE to supply completely packaged IP packets. READ_INSTANCE similarly will return complete IP packets. This approach allows IP instances to remain connectionless in concept and thus avoids the overhead of establishing a network connection instance for each different set of IP packet parameters. (Remember that READ and WRITE under the I/O protocol don't allow for specification of parameters.)

To open an IP network instance, use CREATE_INSTANCE and specify the protocol by overlaying the IpParms structure definition in Vnet.h onto the *unspecified* field of the CreateInstanceRequest structure. QUERY_FILE will return the value of the protocol field for an IP instance. MODIFY_FILE has no meaning for these instances. A standard library routine, OpenIp, is provided to allow creating an IP instance and allocating a File structure for it, for use with other I/O library routines.

34.4. DARPA Transmission Control Protocol (TCP)

TCP file instances created by the internet server implement DARPA TCP byte stream connections. There are three minor differences from the specification in the DARPA Internet Handbook. First, the "push flag" is always set -- data written is transmitted over the network as soon as possible. (Buffering of data is performed by the I/O library routines and would thus be redundant.) Second, the urgent data flag is not set as part of a write operation. Instead, a MODIFY_FILE request is used to set the urgent data flag immediately before a write operation containing urgent data. The urgent data flag is reset immediately after the write operation and thus must be set using a MODIFY_FILE request before each urgent data write operation. Third, there is not concept of connection timeout provided. Connections are aborted if their owner process goes away.

Two variants of CREATE_INSTANCE are permitted on instances of type TCP, corresponding to the Active and Passive opens of the Internet Handbook. Note that the foreign host must be specified completely when issuing a CREATE_INSTANCE request with the active bit set. A standard library routine, OpenTcp, is provided to allow creating a TCP instance and allocating a File structure for it, for use with other I/O library routines.

Two types of release mode are supported for RELEASE_INSTANCE requests corresponding to the Close and Abort primitives of the DARPA specification, respectively REL_STANDARD (equal to 0, the normal release mode defined by the V I/O protocol) and REL_ABORT. Releasing the writeable instance closes the client's end of the connection. Data can still be read from the readable instance until the other end closes. It is necessary to release both the readable and writeable instances to deallocate a connection.

Since TCP supports the concept of a byte stream, the READ_INSTANCE and WRITE_INSTANCE operations do not segment the data flow in any way. The presence of unread urgent data in the receive buffer

of a TCP instance is signaled by the UrgentData reply code to READ_INSTANCE and QUERY_FILE requests until the urgent data has been read by the client. Any READ_INSTANCE requests outstanding when a TCP connection closes for whatever reason are replied to with a replycode indicating the reason. An attempt to read from a closed connection is signaled by an END_OF_FILE reply code.

The QUERY_FILE operation may be used on TCP instances to find out the state of the TCP connection. MODIFY_FILE may be used to change various parameters of the connection. The structure TcpParms1 in Vnet.h defines the parameters which can be set both at CREATE_INSTANCE time and by means of a MODIFY_FILE request. The meaning of the fields are defined in the Internet handbook. TcpParms2 defines both parameters which may be set and state variables which may not be set but whose values are returned if QUERY_FILE is executed with TcpParms2 specified. The parameter in TcpParms2 which may be set is sndUrgFlag. This parameter is used to signal urgent data. The rcvUrgFlag field returns whether or not urgent data has been sent from the remote host and not yet received. The bytesAvail field indicates how many bytes of data are waiting to be received by the user. The state field indicates what state the connection is in with respect to being open, listening, established, closed-waiting-for-remote-close, etc. (see the Internet handbook).

34.5. Xerox PUP Protocol

Possession of a PUP network instance provides a process access to the network for sending and receiving PUP packets on a specific local PUP port. Different PUP instances are delineated by the local socket field in the PUP packets. (Net and host fields will be the same for all PUP packets received by the local host, of course.) Opening socket 0 yields a "promiscuous" instance that fields all PUP packets whose local socket numbers have not been explicitly registered for.

PUP network instances expect WRITE_INSTANCE to supply completely packaged PUP packets. READ_INSTANCE similarly will return complete PUP packets. This approach allows PUP instances to remain connectionless in concept and thus avoids the overhead of establishing a network connection instance for each different set of PUP packet parameters.

Since PUP instances are connectionless, MODIFY_FILE has no meaning for these network instances. QUERY_FILE will return the value of the local socket field for an PUP instance. (QUERY_INSTANCE will only return whether an instance is IP, TCP, or PUP.)

A standard library routine, OpenPup, is provided to allow creating a Pup instance and allocating a File structure for it, for use with other I/O library routines.

34.6. Adding New Protocols

This section should be of interest only to persons who wish to add an additional protocol to (or remove one from) the internet server. It describes the specifications governing the interactions between particular communications protocols and the general framework of the internet server.

There are two interfaces that a protocol must deal with: the external interface to clients of the internet server, and the internal interface to the general communications facilities provided by the server's framework. The external interface consists of the operations, message formats, etc. that the protocol must understand in order to interface with a client's V I/O connection. The internal interface consists of the routines, message buffer conventions, etc. that the protocol implementation must respectively use or provide in order to send packets to the network and receive packets from the network.

34.6.1. External Client Interface

The external interface to a protocol is dictated for the most part by the V I/O protocol specification. Interaction between a client and the internet server is by means of a V I/O connection and the only variations that can be effected are by means of the QueryFile and ModifyFile operations. Thus clients open a connection by means of the CreateInstance operation, they read and write data by means of the ReadInstance and WriteInstance operations, they determine the general state of a connection by means of the QueryInstance operation, and they close a connection with the ReleaseInstance operation.

A connection is "owned" by the client process which sent its CreateInstance request, but can be transferred by means of a SetInstanceOwner request. The semantics of ownership are that a connection must be aborted if its owner process dies. One of the general facilities provided by the internet server is monitoring of the existence of connections' owners. However, the protocol implementation module is responsible for providing an abortion routine.

Protocol-specific interactions are handled by means of the QueryFile and ModifyFile operations. Protocol-specific instantiation parameters can also be specified as part of the CreateInstance operation. The QueryFile operation is used by the client to determine the state of protocol-specific connection variables; the ModifyFile operation is used to modify these variables. Thus the manner in which things such as the "Urgent Data Notification" facility in TCP must be implemented is the following:

1. The client's ReadInstance operation returns an exception code indicating that something out of the normal has happened.
2. The client does a QueryFile operation to determine the protocol-specific state of the connection and obtains the "Urgent Data Notification" on return.

Similarly, a client wishing to signal "Urgent Data" on a TCP connection must do so with a ModifyFile operation.[16]

34.6.2. Internal Protocol Interface

Protocol implementations must interface both to the external internet server client and also to the internal environment of the server itself. This internal interface consists of the following components:

1. A network packet buffer module which all protocols must use. This module provides a pool of packet buffers which have a standardized header format so that various general facilities can manipulate them.
2. A process structure specification for the protocol. All protocol implementations must define certain processes and be aware of the existence of certain other processes. Part of this specification is a specification of the message interactions between these processes.
3. A set of protocol-independent routines supplied by the server which all protocol implementations must use for such things as writing packets out to the network, obtaining and returning packet buffers, etc.
4. A set of protocol-specific routines supplied by the protocol implementation which are used by the general server facilities to return incoming network packets to a connection, signal timeout conditions, etc.

These components will be described in more detail in the following subsections.

---

[16] The reason why the V I/O protocol specification has been structured in this manner is for reasons of efficiency. The vast majority of data read and write operations done on a connection are done with "normal" settings for the connection parameters. By removing parameter specification from the read and write operations these operations can be executed more quickly.

34.6.2.1. A Brief Overview Of The Internet Server's Structure

The internet server consists of the following processes:

1. A connection-establishment process. This process registers itself as the internet server logical id and waits for connection creation requests from new clients. For each new connection creation request it invokes a creation routine for the protocol specified in the request. This routine is responsible for setting up a connection and its associated data structures and handling process(es).

2. Connection handling processes. Each protocol connection is handled by one or more separate processes. It is up to the protocol implementation to decide how to structure the connection handling processes for a connection. However, one of these must be designated the "primary" connection process. This process will be responsible for handling all communications with the rest of the internet server.

3. A network reader process. The V kernel allows only one network device instance to exist at any time. The network reader process reads packets from the network device and calls a protocol-specific routine for each protocol being supported. The protocol-specific routines invoked are responsible for determining which connection of their protocol type a packet should be given to. The network reader process runs at the highest priority allowed so that it can read and multiplex incoming network packets before they are overwritten by subsequent packets in the kernel device.

4. Two timer processes. The first timer is a timeout timer which wakes up periodically and invokes a timeout checking routine for each connection. If the timeout check for a connection returns a time which is less than the current time then a message is sent to that connection's primary connection handling process. The timer determines how long to sleep before waking up again by keeping track of the minimum timeout time beyond the current time. The second timer checks whether any connection owners have died. A message is sent to the primary connection handling process of each connection whose owner has died signalling that the connection should be aborted. This second timer wakes up once every 5 seconds.

34.6.2.2. The Packet Buffer Module

The packet buffer module provides a set of routines which manage a pool of packet buffers which are used as the medium of data transmission inside the internet server. These packet buffers are handed between various parts of the internet server by means of pointers (to avoid copy operations) and their header format must be understood by all parts of the internet server.

The header format for packet buffers is the following:

```
typedef struct pbuf
  {
    struct pbuf *next;          /* General purpose link field.*/
    int length;                 /* Length of the data in the buffer. */
    char *dataptr;              /* Location of the start of the
                                   data. */
    unsigned unspecified[2];    /* Scratchpad fields. */
    char data[MAXPBUFSIZE];     /* The actual packet buffer. */
  } *PktBuf;
```

The **next** field allows packet buffers to be placed in various queueing data structures. The **dataptr** field points to the start of the data in the data array. Packets are typically constructed starting from the back of the data array, with various headers progressively added on to the front. The **unspecified** fields are intended for storing various packet-specific items of information. They are used as scratchpad working areas. MAXPBUFSIZE must be large enough to accommodate all packets encountered by the internet server. It is

set to the maximum allowed packet size of the physical network.[17]

The routines provided by packet buffer module are the following:

```
PktBuf AllocBuf();

DeallocBuf(pkt);
    PktBuf pkt;
```

Buffers are handed out one at a time by means of calls to **AllocBuf()**. Buffers are returned to the free pool by calling **DeallocBuf()**. These routines manipulate the buffer pool in an atomic manner; so that they can be used from multiple processes without conflict.

34.6.2.3. Process Interactions

The implementation of a protocol connection must deal with the network reader and the two timer processes in a prescribed manner. In order for these processes to know whom to send messages to each connection must have a "primary" process associated with it. The process ids of these primary processes are stored in a global data structure maintained by the internet server which contains one entry per connection. The details of this data structure will be described in a later subsection.

Network Reader Interactions

The network reader process must run at high priority and cannot afford to do much processing because it must always be ready to accept incoming network packets before they are overwritten in the kernel device by subsequent packets.[18] This has lead to an interface format between the network reader and the various connection handling processes where communication is by means of atomically updated queues of packet buffers. The network reader process enqueues packets for a connection by calling the **EnQueueSafe()** routine, which places a packet in a specified connection queue. This routine is non-blocking (i.e. no message traffic involved) so that the reader process can immediately continue on to process any additional packets that may have arrived from the network. The connection handling processes then remove packet buffers from their queues by calling the **DeQueueSafe()** routine. The definitions for these two routines are as follows:

```
EnQueueSafe(pkt, q)
    PktBuf pkt;
    RingQueue *q;

DeQueueSafe(q)
    RingQueue *q;
```

RingQueues are atomically updated queues which are defined in the general internet server module. They must be initialized with calls to the **InitSafeQueue()** routine:

```
InitSafeQueue(q, ringBufs)
    RingQueue *q;                /* Queue header. */
    RingBufRec ringBufs[];       /* An array of MAX←RING←BUFS queue
                                    records. */
```

RingQueues consist of the following two data types:

---

[17] Note that there is only one packet buffer size for the entire internet server. A single buffer size was chosen primarily for reasons of simplicity. Extending the packet buffer module to handle multiple buffer sizes would not be difficult.

[18] I.e. it must be able to keep up with the (possibly many) hosts that are sending it packets.

```
typedef struct
  {
    RingBuf head;
    RingBuf tail;
  } RingQueue;

typedef struct RingBufType
  {
    PktBuf pkt;
    struct RingBufType *next;
  } RingBufRec, *RingBuf;
```

The RingQueue structure defines a header record for the queue. **RingBufRecs** are the actual queue elements, and are placed in a circular list by the InitSafeQueue() routine.[19] The **pkt** field of a **RingBufRec** is used to point to the packet buffer which is enqueued by it.

Note that at most MAX−RING−BUFS packet buffers can be enqueued in a RingQueue. **EnQueueSafe()** returns 0 if it can't enqueue a packet buffer.

There is one caveat to the above description of how the network reader interacts with individual connections. The primary connection handling process for a connection may be blocked waiting on client requests[20] so that the packet buffer queue cannot be processed until a request message is received. To take care of this case each primary connection process must also set a variable indicating whether it is blocking awaiting client requests or not. The network reader checks this variable when enqueueing a packet for a connection and sends the connection a "wakeup" message if it is blocked. The process receiving the message must reply immediately to this message in order to minimize the time that the network reader is blocked.

Another point to be made here is that the actions for the network reader described above (i.e. invocation of **EnQueueSafe()** and checking to see if a "wakeup" message must be sent) are actually part of the protocol-specific "network reader" routine that each protocol must supply as part of its implementation. This will be described in more detail later.

Timer Interactions

The two timer processes communicate with connections by means of "timeout" messages. Whenever a timeout condition is detected by a timer process it sends a message to the relevant connection process indicating that a timeout condition has occurred. The message format employed is the following:

```
struct timeoutMsg
  {
    SystemCode requestcode;      /* Standard message request code
                                    field. */
    short unused;
    unsigned timeoutCondition;   /* Which timeout has occurred. */
    unsigned unused1[6];
  };
```

The **requestcode** field is the same as that used for all other message requests. However, instead of a "standard" V I/O protocol request code an internet server-specific request code signalling timeout is used. The **timeoutCondition** field specifies which timeout condition has occurred.

---

[19] The reason why a circular queue of this form is needed stems from the problem of maintaining these queues in an atomic manner.

[20] The protocol implementations to date have consisted of a single process per connection which alternately waits on client requests and processes it packet buffer queue.

34.6.2.4. Protocol-Independent Interface Routines and Data Structures

Global Data Structures

There is one global data structure that must be maintained by all active connections in the internet server. This is the NetInstTable, which contains an entry for each connection specifying various V I/O protocol-specific parameter values, the process id of the primary connection handling process, and a pointer to a control block associated with that connection. The V I/O protocol parameter information is used by the **QueryInstance()** routine for answering QueryInstance requests about connections.[21] The process id is used by the network reader and timer processes to find the primary process for a given connection. The control block pointer is used to access connection-specific information. It is intended for use by the protocol-specific network reader and timeout checking routines.

The primary manner in which connections manipulate the NetInstTable is through the following two routines:

```
int AllocNetInst(prot, ownerPid, pid, rblocksize, wblocksize, tcBId)
    int prot;                       /* Connection protocol type
                                       (TCP, PUP, etc.) */
    ProcessId ownerPid;             /* Process id of owner of the
                                       connection. */
    ProcessId pid;                  /* Process id of primary connection
                                       handling process. */
    int rblocksize, wblocksize;     /* Block sizes for resp. read and write
                                       V I/O connection instances. */
    unsigned tcbId;                 /* Pointer to the control block for
                                       this connection. */

DeallocNetInst(index)
    int index;                      /* Index of NetInstTable entry to
                                       deallocate. */
```

AllocNetInst() returns an index into the table where the newly allocated entry has been placed. Individual fields can then be set by indexing through this value into the table. (E.g. SetInstanceOwner requests would be dealt with in this manner.)

Each protocol implementation is expected to employ these routines to manage the NetInstTable in a correct manner. I.e. allocation and deallocation of NetInstTable entries is *not* done automatically by the server's general facilities.

Useful But Not Essential Routines

The internet server provides several generally useful but not essential routines which may be employed by protocol implementations if they so chose. These include the following:

---

[21] These requests are actually directed at the connection handling processes themselves, implying that each connection could employ its own QueryInstance routine. However no benefit would be gained by such duplication.

```
SystemCode QueryInstance(rqMsg)
    QueryInstanceRequest *rqMsg;

Boolean InvalidFileid(rqMsg)
    IoRequest *rqMsg;

ReplyToRead(replycode, pid, packet, bufferPtr, length)
    SystemCode replycode;       /* Reply code to send to a reader. */
    ProcessId pid;              /* Process id of the reader. */
    PktBuf packet;              /* Packet buffer containing data to
                                   return to the reader.  NULL if
                                   there is no data to return. */
    char *bufferPtr;            /* Address of reader's buffer. */
    int length;                 /* Length of data to return. */

QueryProcess()
```

QueryInstance() returns the state of a specified network connection. It is V I/O protocol-specific and hence independent of the particular network protocol being supported by the other end of the connection. It obtains its information from the **NetInstTable** entry for the connection. Connections are specified in the request message in the same manner as with all other V I/O connections, namely by a **fileid**.

InvalidFileid() checks whether the **fileid** field in a client's request message is reasonable; i.e. whether it maps to an existing connection entry in **NetInstTable** which is in use. All incoming client requests should be checked with this routine to avoid corruption of other connections' control blocks.

ReplyToRead() is a generic routine for replying to a client's read request. It performs the MoveTo operation needed to move data from a packet buffer to the client's read buffer and packages an appropriate reply message.

QueryProcess() is a routine which runs in its own process and is used for debugging. It provides a means for examining and changing the state of the internet server while it is in operation.

34.6.2.5. Protocol-Specific Interface Routines and Data Structures

There are two types of protocol-specific routines that a protocol implementation must provide: network-level routines and connection-level routines. Network-level routines are used by the network reader process to multiplex incoming network packets to the correct connection. Connection-level routines are used to initialize a protocol, create a new connection and interface with the connection timeout checking process.

Protocol implementations are usually done for *protocol families* rather than individual protocols. For example, the current internet server implements both the IP and the TCP Internet protocols. However, rather than implementing these two protocols as separate modules, they are implemented together, so that the TCP module can make use of facilities already defined by the IP module. This results in a situation where only the IP module interfaces with the network layer and the TCP module interfaces internally to the IP module. Thus the IP/TCP protocol family implementation has three interfaces to the rest of the internet server rather than four: it has a single network-level interface and a connection-level interface for both IP and TCP respectively.

Protocol-specific interface routines are accessed by the general server facilities through function tables indexed by protocol type. There are two such function tables, one for the network-level routines and one for the connection-level routines. The format of these tables is described below.

Network-level

The network-level function table is called **PnetTable** and is defined as follows:

```
struct PnetBlock
  {
    unsigned prot;              /* Network protocol type. */
    Boolean active;             /* True if a network connection is
                                   active for this protocol. */
    int (*initNetProt() ();     /* Initialization routine for this
                                   protocol. */
    int (*rcv) ();              /* Receiving routine for this
                                   protocol. */
  } PnetTable[NumPnetProtocols];
```

The first two fields are actually not functions. The **prot** field is used to store the network protocol type id so that the network reader process can figure out which table entry to use for a given network packet.

The **active** field is used to allow the network reader process to "short circuit" discarding of broadcast and invalid packets for inactive protocols. Without this field the reader process would have to call the rcv() routine for these packets since it can't tell itself whether they should be discarded. The active field is managed through the following two routines:

```
ActivateNetProtocol(prot)
    int prot;

DeactiveateNetProtocol(prot)
    int prot;
```

prot specifies which table entry to access.

Associated with the **active** field is another table, called **NetLevelProtocol**, which is used to map from connection protocols to the network-level protocols which support them. For example, the IP/TCP protocol implementation described previously would designate both IP's and TCP's network-level protocol as being IP. The definition of the table data structure, along with an example initialization is as follows:

```
int NetLevelProtocol[NumProtocols] =
  {
    0,                          /* IP */
    0,                          /* TCP */
    1,                          /* PUP */
  };
```

The index of each entry corresponds to the index of the corresponding protocol entry in the FuncTable table. The contents of each entry is the index of the corresponding network-level protocol in the PnetTable table. Thus, in the example shown, the FuncTable defines the IP protocol at index 0, the TCP protocol at index 1, and the PUP protocol at index 2. The PnetTable defines the IP network-level protocol at index 0 and the PUP network-level protocol at index 1.[22] The **initNetProt** field specifies an initialization routine for the protocol which is called at server boot time.

The **rcv** field specifies a routine which is called whenever a network packet arrives which has a protocol type equal to that specified in the **prot** field of the entry (and the **active** field is true). This routine is responsible for figuring which connection of its protocol, if any, should receive the packet. If a connection is found then the routine is responsible for enqueueing the packet in that connection's RingQueue (using the **EnQueueSafe()** routine) and for checking to make sure that the connection's process(es) will actually be able to process the enqueued packet buffer. (I.e. if the connection's process(es) are receive-blocked awaiting client requests then the routine must send a message to "wake" them up.) Packets for which no connection is found must be returned to the free buffer pool with a call to **DeallocBuf()**.

---

[22] The actual internet server code uses manifest constants instead of integers to fill these fields - making things much more readable. However, to illustrate the principle, no manifests were employed.

The interface definition for the initNetProt() and rcv() routines is as follows:

```
InitNetProtocol()

ReceiveProtocolPkts(packet)
    PktBuf packet;              /* Ptr to the incoming network
                                   packet. */
```

where InitNetProtocol() and ReceiveProtocolPkts() are example names.

Connection-level

The connection-level function table is called **FuncTable** and is defined as follows:

```
struct FuncBlock
  {
    int (*InitProtocol) ();
    SystemCode (*CreateConnection) ();
    int (*NextTimeout) ();
  } FuncTable[NumProtocols];
```

The **InitProtocol** field specifies an initialization routine for the protocol which is called at server boot time.

The **CreateConnection** field specifies a routine which is called by the connection-establishment process when a client requests the creation of a new connection instance. The routine must create the data and process structures for a new connection and then handle the CreateInstance request from the client.[23] This is usually also the place where a call to the **ActivateNetProtocol()** routine is made to signal that the protocol is active.

The **NextTimeout** field specifies a routine which is called by the timeout checking timer process. This routine returns the time of the next timeout for its connection. If that time is already past then the timer process will send a timeout message to the connection's primary process. The connection's data structures are accessed through the **tcbId** field of the connection's **NetInstTable** entry.

The interface definition for the **InitProtocol()**, **CreateConnection()**, and **NextTimeout()** routines is as follows:

```
InitProt()

CreateProtConnection(reqMsg, clientPid)
    CreateInstanceRequest reqMsg;
                                /* CreateInstance request message sent
                                   by a the client. */
    ProcessId clientPid;        /* Process id of the client. */

NextProtTimeout(tcbId)
    unsigned tcbId;             /* Ptr to the control block for the
                                   connection. */
```

where **InitProt()**, **CreateProtConnection()**, and **NextProtTimeout()** are example names.

---

[23] The method recommended for doing this is to have the routine create the connection handling process(es) and then forward the CreateInstance request to the connection's primary process. This allows the connection handling process(es) to manipulate their own data structures (which are typically kept on the process(es)' stack(s)).

— 35 —
V Storage Server

The V storage server is a file system that implements the V I/O protocol. It is intended to run on a "server" machine with mass disk storage, thus providing file access for users on the network. It provides an alternative to the Unix Server for file storage. It implements a hierarchical name space with a syntax very similar to that of the UNIX file system (i.e. pathname components are separated by a "/"). Additionally, there is no distinction between files and directories in the V storage server (i.e. any file can "act" like a directory in that it can have descendents in the tree structure).

One word of caution is that the V storage server is still at an "experimental" stage, thus providing limited access facilities and no protection. Hence, users requiring robust file access and proteciton should use the file storage provided by the Unix Server. The robustness of the V storage server software is expected to greatly improve in the near future.

35.1. Running the V storage server

One can start up the V storage server from within a V executive by typing

```
storageserver
```

or

```
storageserver devicename
```

If no device name is specified, the storage server attempts to open two devices, [device]disk0 and [device]disk1. Non-existence of a second device does not affect correct operation of the program. Note that the devices must be attached to the workstation from which the command is invoked and the kernel running on the workstation must include the proper disk driver (see the Kernel Section for details on which kernel should be booted).

35.2. Accessing the V storage server

When the V storage server is started it registers itself as VSTORAGE_SERVER. Thus, before a client can communicate with the V storage server it must do a GetPid(VSTORAGE_SERVER, ANY_PID). This function returns a pid to which a client will send its CREATE_INSTANCE request messages.

A CREATE_INSTANCE request causes the server to attempt to open the named file. Files opened in FREAD mode are of type READABLE, FIXED_LENGTH, and MULTI_BLOCK. The modes FCREATE and FMODIFY create instances of type READABLE, WRITEABLE, and MULTI_BLOCK. FAPPEND mode adds the further constraint of APPEND_ONLY. All instances are random access, but operations must start on a block boundary.

If the mode is FCREATE, and the file does not exist, then a new file is created along with the associated instance. The permission bits of the new file will be the same as those of its parent node in the directory tree structure.

If a CREATE_INSTANCE request is successful, a file instance identifier is returned by the server that is used by the client for all subsequent accesses to this instance. In addition, the server returns a *file instance server* pid which is the process to which all subsequent I/O requests will be directed. This pid is different

than that of the main server because one process (namely, the one registered as the VSTORAGE_SERVER) handles CREATE_INSTANCE requests and other processes handle I/O requests.

Once an instance has been created, a client can perform I/O operations on the file represented by the instance using READ_INSTANCE and WRITE_INSTANCE requests. These requests, if legitimate, result in the *file instance server* carrying out the desired tasks. When a client is finished accessing a file, it closes the file by issuing a RELEASE_INSTANCE request.

The V storage server supports many other types of requests including ones to create, remove, and rename files and most other relevant requests associated with the V I/O and naming protocols. Note that many applications need not be concerned with message types and formats as actual message construction usually takes place within V commands and standard library routines. For example, CREATE_INSTANCE, READ_INSTANCE, WRITE_INSTANCE, and RELEASE_INSTANCE requests are encapsulated in the library routines Open(), Read(), Write(), and Close(), respectively.

35.3. Creating a context for the V storage server

In order to provide easy access to the V storage server and its directories, it is convenient to define a context for it using the **define** command. Once this is done, one can simply *cd* to the newly created context and subsequent relative pathnames will be interpreted relative to this context.

Thus, for example,

```
define ss [storage]
```

results in a context being defined for the V storage server, and

```
cd [ss]
```

causes the user's current context to be changed to its root directory.

— 36 —
Unix Server

The V Unix server is a Unix[24] program (and not a V program or command) designed to simulate a V kernel/storage server on a VAX[25]/Unix system. It provides access to some of the Unix system services via the V kernel interprocess communication primitives. To workstations running the V kernel, the Unix server appears as a standard V server, primarily providing Unix file access using the standard V I/O protocol.

`GetPid(UNIX_SERVER, REMOTE_PID)` returns the pid of a Unix server accessible to this workstation. With more than one, `GetPid()` returns the pid of the first Unix server to respond to the request. This is the pid of a *public* Unix server. Public Unix servers also register themselves under the logical pid STORAGE_SERVER. A public storage server is the definitive source for all the standard system files and commands whereas hosts that run non-public storage servers are not required to be kept up-to-date.

36.1. Sessions

The public Unix server provides access to all files on a Unix host that are publically readable (in Unix terminology, "readable by others"). To get access to other files, a client must create a *session* with the Unix server. To create a session, the client sends a CREATE_INSTANCE request to the server, with the *mode* field set to FSESSION+FCREATE. The name field of the request contains a Unix user name and password (separated by a NULL character). The reply message will contain the process id and instance id of the session. The process id allows the client to communicate directly with the session. The session provides several Unix system services, all running under the access privileges of the Unix user specified in the CREATE_INSTANCE request.

The initial owner of a session is specified by a server-specific field in the CREATE_INSTANCE request. The format of this request is defined in the standard header file <Vsession.h>.

The operations SET_INSTANCE_OWNER and RELEASE_INSTANCE are meaningful on session instances. Other I/O protocol operations are currently not supported. Releasing a session instance terminates the session and invalidates its process id.

36.2. File Access

When a CREATE_INSTANCE request is received by the server (or session), and there are no special mode flags set (such as FSESSION), it attempts to open the named file. As was mentioned earlier, the file must have *others* access privileges in order for it to be opened by the main server. Also, the main server does not allow creation of new files, or writing to any file. A session, on the other hand, has the same access privileges as the Unix user that created it.

If the client has the correct permissions, then an instance is created, with the *type* field set according to the request mode. Files opened in FREAD mode are of type READABLE, FIXED_LENGTH, and MULTI_BLOCK. The modes FCREATE and FMODIFY create instances of type READABLE,

---

[24] UNIX is a trademark of Bell Laboratories.

[25] VAX is a trademark of Digital Equipment Corporation.

WRITEABLE, and MULTI_BLOCK. FAPPEND mode adds the further constraint of APPEND_ONLY. All instances are random access, but operations must start on a block boundary. The block size of these instances is equal to the maximum appended segment size for V kernel messages.

If the mode is FCREATE, or it is FMODIFY and the file does not exist, then a new file is created along with the associated instance. Files are created with Unix file protection bits ("mode bits") set to allow reading and writing by the owner, and reading by group and others. A client may change the mode bits using a WRITE_DESCRIPTOR or NWRITE_DESCRIPTOR request.

36.3. Program Execution

A client can execute Unix programs through a V session by sending a CREATE_INSTANCE request with the FEXECUTE flag set in the mode field. The name and arguments of the program to be executed are sent in the segment with the NULL character being a field separator. The last argument need not be null terminated. The context in which the program is to be executed is also specified in the request.

Given a request, the session has a built-in search path that it uses to determine which Unix program to execute.[26] The session tries to find the first file in a directory along the search path that matches the given name. If the name contains a '/', then the search path mechanism is not used and only the context specified in the request is searched. If the program is a shell script, the Bourne shell is invoked explicitly, and it determines which shell should execute the script based on the normal Berkeley Unix conventions. As a side-effect, the shell expands any wild-card characters (such as '*' and '?') found in the arguments. This expansion does not occur if the Unix program is not a shell script.

After all of the preliminary checking is done, the session forks and its child attempts to run the program. The parent process replies to the requestor with an OK status. However, there is no guarantee that the execution will be successful. A failure can occur after the OK reply has been returned, since the program is not loaded until the child has been forked off and the reply is sent asynchronously. If a failure of this nature occurs, then an error message should appear in the program's output.

In the reply message, the session includes an instance id for the running program. If the file mode in the CREATE_INSTANCE request was FREAD, then the instance id specifies an instance of type READABLE, VARIABLE_BLOCK, and STREAM. The client can read the program's standard output using this instance.

If the mode was FCREATE, FMODIFY, or FAPPEND, then the instance returned in the reply message is of type WRITEABLE, VARIABLE_BLOCK, APPEND_ONLY, and STREAM. Data written into this instance is piped into the program's standard input. An instance with id 1 greater than the one returned in the reply is also created, of type READABLE, VARIABLE_BLOCK, and STREAM. Reading from this instance provides access to the program's standard output.

When the program terminates (either normally or abnormally), the session returns an END_OF_FILE reply to any write requests. Read requests will continue to be accepted as long as data is left in the pipe. Write requests will block if the pipe is full and the Unix program is not reading from it. (Unix pipes can buffer up to 4096 bytes of data.)

A client may terminate the program by releasing all instances associated with it. If only one of the instances is closed, then program will not terminate immediately. This allows a client to close the program's input and have it clean up before exiting. One should be careful not to release the readable instance before program termination, because Unix sends a signal to any program that writes to a pipe with only one end. The signal will kill the Unix process, if the process is not catching or ignoring it.

---

[26] To find out the search path used in your installation, execute the Unix command **printenv**. This will display the environment variables that are passed on to programs executed via the session.

36.4. File Descriptors

The server supports the V context directories and descriptor requests. One can open a Unix directory with the FDIRECTORY flag set in the mode field and the server will automatically translate standard Unix directory entries to V Unix file descriptors. Directories are not writeable directly, but descriptors can be modified using a WRITE_DESCRIPTOR or NWRITE_DESCRIPTOR request. The *UnixFileDescriptor* type is defined in the system include file, <Vdirectory.h>.

36.5. Server Name Lookup

A client can get the pid of any Unix server by sending a LOOKUP_SERVER request to another Unix server. The request and reply formats are as follows

requestcode	LOOKUP_SERVER
hostname	Pointer to the character string name of the host on which the server is running.
namelength	Length of the host name.

replycode	Standard system reply code.
serverpid	Process id of the server.

The *hostname* field of the request gives the name of the host machine that the requested server is running on. The server's pid is returned in the *serverpid* field of the reply message. These message formats are defined in the standard include file <Vsession.h>.

— 37 —
Service Server

37.1. Overview

The service server provides a means for managing globally visible servers and services. It provides facilities for registering arbitrary objects (typically entries which describe the state and contact address of a server or service) and also for selecting a subset of these registered objects for retrieval. The selection facilities take a client-specified pattern and match it against the information in each registration entry to determine whether that entry should be included in the retrieval set.

Since any kind of object can be registered, the server is in fact a general "switchboard" service which can be used for arbitrary "rendevous" between two or more clients. However, the primary usage of this server is intended to be for management of global servers and services; and the selection facilities provided for retrieving registered objects have been structured with this goal in mind.

37.2. Registering an Object

Objects are registered with the service server by means of the **RegisterServer()** library routine. This routine packages a registration descriptor into a message and sends it to the service server. Registration is on the basis of an object name and an object type. Object type essentially represents a subcontext within the service server and all objects of a given type must be registered using the same registration entry record structure. Object name distinguishes between the various registered objects within a given object type. All selection and listing of registered objects is done with respect to a given object type.

The service server maintains the concept of an owner for the objects registered with it. Registered objects are unregistered when their owner dies. This is achieved by having the server periodically check each registered object's owner's process id to see if it is still valid. The ownership of a registered object can be changed using the standard **SetInstanceOwner()** library routine.

The format of the registration entry for a particular object type is left to the client. Thus an entry can store arbitrary sorts of information in it. However, in order to be able to perform selections of registered objects on the basis of information contained within their descriptors the formats of the relevant descriptor fields must be known to the service server's pattern matching facilities. To support this, several well-known descriptor formats have been defined in the C include file **Vservice.h**. These record structures are actually descriptor format *prefixes* since the client can append arbitrary numbers of additional fields on the end of the descriptor structure which contain information not used in the selection process.

There are various well-known object types (and associated registration descriptor formats) which are defined in **Vservice.h**. These are utilized by various existing facilities such as the team servers of all hosts throughout the system.

Objects can be unregistered by means of the **UnregisterServer()** library routine. Objects already registered can be reregistered with a new descriptor entry by simply invoking the **RegisterServer()** a second time. The service server will automatically remove the original entry.

The service server has a well-known multicast group associated with it which it uses to send out requests for status update when it first starts up. This allows it to reinitialize itself after crashes and other such events. The well-known multicast address is defined in the **Venviron.h** header file.

37.3. Listing Registered Objects

All registered objects' descriptors of a given type can be listed using the standard V directory listing protocol. Similarly, a single registered object's descriptor can be listed using the NReadDescriptor request defined in this protocol. The format for specifying an object is

object-type: *object-name*

If no object type is specified in the **CreateInstance** request message then all registered objects are returned.

Since the service server understands the V directory listing protocol it is possible to use the **listdir**() and **listdesc**() programs to query it from the exec level. Thus, for example, the status of all running hosts within the system can be found out by typing

```
listdir [service]host
```

to the V exec to query the well-known object type **host.**

37.4. Retrieving Sets of Registered Objects

Sets of registered objects are retrieved from the service server by means of a combination of service server-specific library routines and general V-I/O protocol library routines. The basic idea is to establish a connection instance, just as for a V-I/O protocol connection, through which the descriptors of the selected objects are read as if they constituted a separate file unto themselves. The *selection instance* is created using the **CreateSelectionInstance( )** routine, which specifies which set of objects to retrieve. The instance is subsequently treated just as if it were a standard V-I/O instance; which can be read using standard library routines such as **Read( )** and is released using the standard library routine **Close( )**. The only difference is that the first descriptor associated with the selection instance is immediately returned by the **CreateSelectionInstance** operation.

Since there are many cases where one wants only the first object returned from a set of selected objects (e.g. the first host from a set of hosts eligible as remote execution sites) a means is provided by which a single object descriptor can be retrieved without incurring the cost of establishing a selection instance. One of the parameters to **CreateSelectionInstance** allows one to specify whether one or more than one objects is to be returned. If only one is specified then no connection is established and **CreateSelectionInstance** merely returns the desired descriptor record.

Selection of objects is based on the specification of both a retrieval pattern and a pattern matching function. As mentioned before, all selection is done strictly within a given object type. The pattern matching function to specify is determined by the format of the descriptors for the desired object type. The include file **Vservice.h** contains a list of all available pattern matching functions and descriptions of the descriptor formats they expect to use. This include file also contains a description of the form that retrieval patterns must take as a function of which pattern matching function is to be used.

— 38 —
Exec Server

The exec server is central control facility for all instances of the V system executive on a workstation. Its purpose is to allow sharing of code and data (such as aliases) among all executives. The intention is that while each executive is a separate command stream, all executives on the same workstation should present the same command interface to the user. That includes customized aspects of that command interface, such as aliases. Since the exec server is part of the basic equipment of the V system, such customizations do not vanish even if the terminal agent is replaced, but as long as the user is logged in.

The exec server is located by

```
GetPid(EXEC←SERVER,LOCAL←PID
```

It is present in all the standard configurations of the Vsystem.

The exec server allows programs to have instances of the executive (usually referred to simply as "execs") created and destroyed. An exec is known to the server by its exec id; exec ids are small integers starting at 0. There is currently no concept of ownership of execs-any program can destroy any exec regardless of whether it created it or not.

The following requests are supported.

CREATE−EXEC Creates an executive, with standard i/o and context specified in the request message, and returns the exec id.

START−EXEC Under some circumstances an exec is not started by the CREATE−EXEC request, because the requestor needs to do some **SetInstanceOwner** operations first. START−EXEC then allows the exec to start running. Normally all this is transparent and is handled in **CreateExec.**

DELETE−EXEC Delete an executive. If there is a program running under it, it is abruptly stopped due to the death of its parent process.

KILL− PROGRAM
Kill the program running under an executive. If there was no program running under that executive, nothing happens.

QUERY−EXEC Returns information on an executive: its status (free, loading a program, or running a program), its process id, and the process id of the program running under it, if any.

CHECK−EXEC Makes a check of all executives. If the standard input server or standard output server of an exec has died, the exec is destroyed. This is used mainly when changing terminal agents.

— 39 —
Terminal Agents

Terminal agents are a generic class of sesrver used in the V system. A terminal agent has the duty of mediating between the terminal hardware, the user, and the other programs in the system. It is responsible for line editing functions,e.g. the fact that the back space key does not add a backspace character to the imput stream but deletes a character from the imput stream. It translates the newline character '\n' into a carriage return/linefeed sequence on terminals that require it. It is also responsible for interacting with the exec server to create at least one executive, or providing means for the user to do so. It may, but need not, support multiple i/o streams. Terminal agents may differ for two reasons: because they are designed to offer different services to the user, or because they are designed to run on different types of terminals.

The V system currently contains two different terminal agents, the Simple Terminal Server (sts) and the Virtual Graphics Terminal Server (vgts). The Simple Terminal Server is a minimal terminal agent. It provides a single i/o stream, using the terminal facilities provided by the firmware monitor of the workstation, and creates one executive using that i/o stream. The standard V line editing interface is provided, but no mouse or graphics facilities are available. The Virtual Graphics Terminal Server, in contrast, provides a very large set of facilities: multiple i/o streams in multiple windows, graphics, and mouse-controlled menus. But it supports the same line editing facilities. A large class of programs should be able to run under either of these terminal agents, or any other terminal agent, without any knowledge of which terminal agent is present.

The **newterm** command allows the user to replace the terminal agent on his workstation without rebooting the workstation.

39.1. Implementation of Terminal Agents

These are the requests that should be supported by a terminal agent, at the minimum . It should support the V I/O protocal for INTERACTIVE STREAM files. In simple cases, it may give polite replies to CREATE–INSTANCE and RELEASE–INSTANCE1 without really doing anything, as the sts does. It should also support the MODIFY–FILE request in the fashion expected by **ModifyPad**: it sets the pad mode, with a combination of bits controlling such features as line editing, echoing of input, and translation of \n to carriage-return/linefeed. In particular,**ModifyPad(file,0)** should turn off all such features, giving the client access to the raw, unadorned terminal.

The following conventions should be observed, in order to allow the **newterm** command to work: Upon starting up, a terminal agent should define the context [screen] with itself as the server. It should also support the **GetRawIO** request message, in which the terminal agent tells the client the server and instance id's for its own standard input and output. Presumably these refer to the raw terminal.

— 40 —
Virtual Graphics Terminal Server

The Virtual Graphics Terminal Service (VGTS) allows the display of structured graphical objects on a workstation running the V system. This chapter describes the internal structure of the VGTS. The SDF manager was originally written by "Rocky" Rhodes, incorporated into the **Yale** program by Tom Davis, and converted to use the V kernel by Marvin Theimer. The current VGTS is the work of Bill Nowicki.

40.1. Current VGTS Versions

There are currently two working versions of the VGTS. **sun100vgts** is used on workstations with SMI model 100 framebuffers, while **sun120vgts** is used with the SMI model 120 framebuffer. Users usually will not have to concern themselves with this, since **team1-vgts** (the default first team) automatically loads the correct version of the VGTS. Furthermore, the program **vgts** is a 'bootstrap' program which loads the correct version of the VGTS (in a new team), and then dies. Thus, "vgts" can be given as an argument to **newterm** (see Section 4), regardless of the framebuffer type.

The difference in VGTS versions is important, however, when loading special first teams that have a VGTS already linked in. team1+sun100vgts] will run only with a SMI model 100 framebuffer, and team1+sun120vgts] only with a model 120 framebuffer.

40.2. VGTS Philosophy

The central concept of the VGTS is that application programs should only have to deal with creating and maintaining abstract graphical objects. The details of viewing these objects are taken care of by the VGTS. This is in contrast to traditional graphics systems in which users perform the operations directly on the screen, or on an area of the screen referred to as a viewport or window. The types of objects managed by the VGTS are discussed in more detail in the VGTS chapter of the library manual.

40.3. VGTs, Views, and Instances

Once the VGTS client has defined some graphical objects, it also needs to provide information on which objects can be viewed. Every *VGT* is an item (usually a structured symbol) that is associated with one or more views, that actually appear on the screen. Each VGT can exist in zero or more views, but each view has exactly one VGT associated with it. The "SDF Numbers" can be thought of as separate object *definition* spaces, while the VGTs are object *instance* spaces. Symbol definitions are shared between VGTs, but instances of symbols are not.

The VGTS lets a user view objects in any VGTs anywhere on the screen in *views*. Each view has a zoom factor, a window on the world coordinates of some VGT, and screen coordinates which determine its viewport. Although the SDF client can create default views, the VGTS user can change them with the window manager, and create and destroy more of them. Routines for the client's manipulation of VGTs and views are described in the library manual.

The VGTS maintains an event queue for each instance, and the VGTs associated with the given file instance. Each VGT corresponds to an instance in the V I/O protocol. The mode bits of the instance give the kind of events that will be queued. The details of these functions are defined in the library manual.

40.4. Pad Escape Sequences

Unless otherwise noted, all escape sequences can come with or without the optional left bracket between the escape and the escape command character. Arguments to the escape command are decimal character strings separated by a semicolon. The following subset of the ANSI standard escape sequences is decoded by the SUN VGTS terminal emulator:

BELL	Causes some form of audio feedback (buzzer, bell, etc.) if possible, and flashes all the views of the pad..
TAB	Positions the cursor at next multiple of eight (plus one) columns, erasing characters between the current cursor position and the new position.
FF	Clears the pad.
CR	Returns the cursor to the first column of the current line.
LF	NewLine -- Moves the cursor down one line. If it is at the last line of the pad, all lines move up (scroll).
BS	Cursor moves backwards one space.
SO	Shift Out -- Select the G1 character set. Currently ignored.
SI	Shift Out -- Select the G0 character set. Currently ignored.
NUL	Null -- ignored; may be used for padding.
DEL	Delete -- ignored; may be used for padding.
ESC A	CursorUp -- move the cursor up one line.
ESC [*i* A	CursorUp -- move the cursor up *i* lines.
ESC B	NewLine -- move the cursor down, as with LF.
ESC [*i* B	NewLine -- move the cursor down the *i* lines.
ESC C	CursorForward -- move the cursor forward, but do not overwrite the character at the current position.
ESC [*i* C	CursorForward -- move the cursor forward *i* character positions.
ESC D	Index -- scroll the current scroll region up one line.
ESC [*i* D	CursorBackward -- move the cursor backwards *i* character positions.
ESC E	Next Line -- move the cursor down one line, but if it is at the end of the region, scroll the region up (Index).
ESC [*l*;*c* f	CursorPosition -- Move the cursor to line *l*, column *c*. The lines and columns start from the upper left, which is (1,1). Specifying zero or leaving an argument blank is equivalent to a value of 1. Thus ESC[f alone will "home" the cursor to the upper left.
ESC H	Ignored. Used by some terminals to set tab stops.
ESC [*l*;*c* H	CursorPosition -- same as ESC f.
ESC J	ClearToEOS -- clear from the current cursor position to the end of the pad.
ESC [*n* J	Clear -- if the argument is 2, clear the entire pad. Otherwise, clear to end of pad.

ESC K — ClearToEOL -- clear from the cursor to the end of the current line.

ESC L — InsertLine -- insert a line at the cursor position. All the lines below and including the current one are moved down. The bottom line goes away.

ESC [*n* L — InsertLine -- insert *n* lines at the cursor position.

ESC M — ReverseIndex -- move the scroll region down one line. The top line in the scroll region becomes blank.

ESC [*i* M — DeleteLine -- delete *i* lines starting from the line that the cursor is on, and move all lines below them up.

ESC P — DeleteChar -- delete the character at the cursor position, moving all the rest of the characters in the line to the left one column.

ESC [*i* P — DeleteChar -- delete *i* characters, starting from the one under the cursor.

ESC @ — InsertChar -- move all the characters to the right of the cursor to the right one column. A space appears at the cursor position.

ESC [*i* @ — InsertChar -- Insert *i* characters at the cursor position.

ESC [*i* m — If the value of the argument is non-zero, standout mode is turned on, which will mean characters appear in reverse video. A zero argument resets to normal video.

ESC [*t*;*b* r — Specifies the top and bottom lines of a scroll region. This is used in the Index and ReverseIndex commands.

ESC < — Enter ANSI mode. Currently it is ignored, since VGTS pads are always in ANSI mode.

ESC) *c* — Select G0 character set. Currently it is ignored.

ESC (*c* — Select G1 character set. Currently it is ignored.

The default size of a VGTS pad is 28 lines by 80 columns. This is to be compatible with the "sun" terminal type of the Stanford Unix systems. This terminal type is just a VT-100 with 28 lines, and a few additional escape sequences as described above. For TOPS-20, the command `term VT100` will work. On the SU-AI WAITS system, the `.tty sun 28 80` command can be used for display service.

40.5. VGTS Message Interface

The use of the vgtsexec and view manager is given in the *V-System Commands Manual*. This chapter describes only the internal programmer's interface. The following requests of the I/O protocol are supported:

CREATE_INSTANCE

Causes a new pad to be created. The view manager will let the user decide where to put the upper left corner of the pad by changing the cursor and blocking the process until the user clicks the mouse. The file instances created are READABLE, WRITEABLE, VARIABLE_BLOCK STREAMs. The first two unspecified fields of the message (if non-zero) are the number of lines and columns in the new pad. The filename field of the message is used as the name of the VGT. Usually this is invoked only by the OpenPad routine described in the VGTS chapter of the Library Manual.

QUERY_INSTANCE

Returns the standard values, the same as a Create Instance reply.

WRITE_INSTANCE

Write the bytes to the pad corresponding to the file instance. Output conversions are performed if the appropriate "Cooking" modes are set.

WRITESHORT_INSTANCE
Same as WRITE_INSTANCE.

READ_INSTANCE
Blocks until some characters are entered into the pad. If there are any characters already in the event queue for this pad, they are returned immediately. Note that since the instance is VARIABLE_BLOCK, an unknown number of characters can be returned, up to the blocksize.

RELEASE_INSTANCE
The pad is deleted, along with any views of the pad, and storage is reclaimed.

QUERY_FILE Returns the Cooking mode bits for the pad. These are defined in <Vgts.h> and described below.

MODIFY_FILE The Cooking mode bits are set for this pad. The structure ModifyMsg describes the format of this message.

SET_BREAK_PROCESS
The break process for each instance is the process which will be killed when the Kill Program command is invoked from the View Manager.

SwitchInput The given pad (from the fileid) is selected for input. This is used in the SelectPad routine.

MouseStatusRequest
The position of the mouse is returned immediately. This will be replaced by EventRequest in the future.

MouseEventRequest
The position of the mouse is returned as soon as a significant event occurs, as defined by the Mouse mode bits described in the next section. This will be subsumed by EventRequest in the future.

EventRequest The first item from the event queue is returned to the requester. If the event queue is empty, the requester is blocked until an event comes in for the given VGT

40.6. Internal Organization

The current VGTS implementation consists of the following modules:

- Master Multiplexor. This is the only module which is operating system dependent. Upon initialization, the appropriate process structure is set up. The main loop consists of waiting for a message, dispatching to the appropriate routine in the other modules, and returning a reply. Synchronization problems are avoided by having the data structures accessed only in one process.
- Terminal emulator. This module interprets a byte stream as if it were an ANSI standard terminal. Printable characters are added to text objects, and control and escape codes are mapped into the proper SDF manipulations.
- Input handler. There are various device-dependent input handlers. For example, a single process reads the keyboard and sends typed characters to the multiplexor. Another reads the mouse and tracks the cursor.
- SDF manipulator. This module handles requests of applications to create, destroy, and modify graphical objects in structured display files. These routines maintain bounding boxes for symbols, and

call the appropriate redrawing routines when necessary. There is a hash table to locate items given their client names.

- SDF interpreter. These are the highest level redrawing operations. The structured display files are visited recursively, with appropriate clipping for bounding boxes totally outside the area being redrawn.
- Display operations. These are the graphical operations called by the SDF interpreter. They are device independent, but some of the operations, like viewport clipping, are done in hardware on the IRIS system.
- Drawing primitives. There is one module which implements device dependent graphics primitives. On the SUN workstation this is a simple interface to the RasterOp package. At this level color rectangles are drawn as stipple patterns on monochromatic displays.
- Hit detection. The structured display file is visited, but instead of actually drawing the primitives, the positions are checked to match the cursor's position. A list of possibly selected objects (under other optional constraints) is returned to the application.
- View manager. This module provides a mode in which users can create, destroy, and modify the screen layout. Viewports can be moved rigidly, stretched, or squeezed. Views can be zoomed or panned, all without affecting the applications manipulating the represented objects. On the SUN workstation zooming is by powers of two, and all motions are done in one step. On the IRIS system zooming and moving viewports are smooth, continuous operations.
- Viewport primitives. These are the routines which perform the view-changing operations, invoked by either an application program or the user through the view manager.

40.6.1. Executive Interface

Since the V-System is intended to be modular, the VGTS can be used with an executive other than the standard one. The VGTS module **execs.c** handles the Exec Control part of the view manager command. It starts up new executives as new processes on the same team with the calling sequence: **Exec(in, out, err, cmdin)** where all of the parameters are pointers to Files. These are the input, output, error, and command input files. The Executive then calls the functions **SetVgtBanner(file, banner)** and **SetBreakProcess(file, pid)** as commands are executed.

40.6.2. Frame Buffer Interface

The VGTS was intended to be ported to different graphics devices. Someday someone might actually do it, and then we could have some material for this section. Right now most of the device-dependent routines are in the **draw.c** file.

— 41 —
Simple Terminal Server

The Simple Terminal Server(STS) is a minimal terminal agent. It does not use graphics, and it takes up less memory than the VGTS. Only one i/o stream is supported. A program that wants to do graphics directly on the SUN hardware, not mediated by the VGTS, should be run under the STS.

The STS creates one executive. If this executive is ever destroyed, by encountering end of file or by other means, it will be replaced within a second or so. Such a replacement can be forced by the sequence control-↑ x. A program running under the executive can be killed by control-↑ k. The normal ↑Z and ↑C commands also work, but they can be disabled by ModifyPad requests, while the control-↑ sequences cannot be disabled.

41.1. Input Editing Facilities

The STS provides a superset of the input editing facilities provided by the VGTS. All ModifyPad bits that are not related to the mouse work as they do under the VGTS: **CR—Input**, **LF—Output**, **Echo**, **Linebuffer**, **PageOutput**, **PageOutputEnable**, and **DiscardOutput**.

Printing characters are inserted at the cursor. In addition, the input buffer can be edited with Emacs-style text-editing commands. In the following descriptions, CTRL-x means striking the Control key and the x key simultaneously; ESC-x means striking the Escape key and then the x key. Killing an object means moving the object from the input buffer to the kill buffer.

The STS supports the following text-editing commands:

RETURN	Releases the input buffer, with a newline appended, to the application.
LINEFEED	Same as RETURN.
CTRL-a	Move cursor to beginning of the current screen line.
CTRL-b	Move cursor back one character.
BACKSPACE	Same as CTRL-b.
LEFT ARROW	Same as CTRL-b.
CTRL-c	Kills the Break Process, usually the command running in the current executive.
CTRL-d	Delete character under the cursor.
CTRL-e	Move cursor to the end of the current screen line.
CTRL-f	Move cursor forward one character.
RIGHT ARROW	Same as CTRL-f.
CTRL-g	Abort the command. The input editor will release the input buffer, with a CTRL-g appended, to the application, which is responsible for detecting the CTRL-g and reacting to it.
CTRL-h	Delete the character before the cursor.

DEL	Same as CTRL-h.
CTRL-k	Kill Kill from the cursor to the end of the current line.
CTRL-l	Re-display the input buffer.
CTRL-n	Move cursor down one screen line.
DOWN ARROW	Same as CTRL-n.
CTRL-p	Move cursor up one screen line.
UP ARROW	Same as CTRL-p.
CTRL-q	Quote next character. Control characters are displayed as '↑C'.
CTRL-t	Transpose the two characters preceding the cursor.
CTRL-u	Kill the entire input buffer.
CTRL-w	Kill from the cursor to the beginning of the current word.
CTRL-y	Move the contents of killbuffer into the input buffer, inserting at the current cursor position.
CTRL-z	Causes an End of File indication to be sent to the application reading the input. This will terminate the Executive if no application is running.
CTRL-\	Insert next character with the eighth bit set. Character is displayed as '\nnn', where nnn is the octal representation of the character code.
ESC-,	Move cursor to the beginning of the input buffer.
HOME	Same as ESC-, .
ESC-.	Move cursor to the end of the input buffer.
ESC-b	Move cursor to the beginning of the current word.
ESC-BACKSPACE	Same as ESC-b.
ESC-d	Kill from the cursor to the end of the current word.
ESC-f	Move cursor past the end of the current word.
ESC-h	Kill from the cursor to the beginning of the current word. Same as CTRL-w.
ESC-DEL	Same as ESC-h and CTRL-w.
ESC-t	Transpose the two words preceding the cursor.

41.2. Hardware Environment

The STS communicates with the user via the kernel console device. If the workstation has a framebuffer, characters are sent to the terminal emulator built into the workstation's PROM monitor; otherwise, characters are sent through serial line 0 to a character terminal.

The attached terminal or terminal emulator must understand the escape sequences sent to it by the STS for cursor positioning. The STS currently works properly with the following terminal emulators and terminals:

- Any PROM monitor terminal emulator that supports ANSI standard escape sequences, e.g., the SMI

PROM monitor.

- Cadlinc PROM monitor terminal emulator.
- Any character terminal that supports ANSI standard escape sequences, e.g., VT100 or Heath-19 in ANSI mode.

— 42 —
Context Prefix Server

The V-System naming world is in general a forest, with each tree corresponding to a server. Although the naming protocol provides a way to link this forest into a single, connected graph, we do not anticipate that enough permanent cross-links will be set up to make the graph connected. Note that the simplest implementation of a cross-link requires one server to store the (server-pid, context-id) corresponding to a context on another server. Since server processes (and hence, server pids and context ids) may be relatively short lived compared to the objects they provide access to (e.g., files on non-volatile storage), such a simple implementation is not adequate for a permanent cross-link.

As a partial solution to this problem, each workstation in the V-System contains a local name server process as part of its V executive. The local name server maintains a directory of local aliases for (server-pid, context-id) pairs on servers of interest.

Further, since the present V kernel device server does not provide character string names for the devices it implements, the local name server also performs name mapping on behalf of the device server. The directory of local aliases and the directory of devices are maintained as separate contexts within the name server.

42.1. Name Syntax

When a client issues a CreateInstance request using the standard Open library routine, if the character string name begins with '[', the request is sent to the first process responding to a `GetPid(CONTEXT_SERVER, ANY_PID)`, ordinarily the context prefix server on the client's workstation.

If the name does not begin with a square bracket, the Open routine will send the request to the client process's "current context," a (server-pid, context-id) pair stored in a standard place in the client's stack space (the "per-process area").[27] The context prefix server is a character string name handling server that participates in the naming protocol described in chapter 30, including the ADD_CONTEXT_NAME and DELETE_CONTEXT_NAME requests. It recognizes the character '[' as a special escape which causes the next component of a CSname (up to the next ']' character or end of string) to be interpreted in its context 0 (DEFAULT_CONTEXT). Context 0 is the directory of contexts maintained by the context prefix server, as mentioned above.

The context prefix server maps the name in brackets (context name) to a (server-pid, context-id) pair. If the name consists of more than just the context name, the request is forwarded to the process designated by *server-pid* with *context-id* placed in the name request. The context prefix server adjusts the *nameindex* field so the receiving name server does not look at the context name. If the name consists only of a context, the context prefix server may handle the request itself, depending on the type of request. For example, "DELETE_CONTEXT_NAME [diablo]" deletes "diablo" as the name of a context in the server. On the other hand, "CREATE_INSTANCE [diablo]" would be forwarded to the context "[diablo]" with the name reduced to a null string. This request could be used to read the context directory for "[diablo]".

---

[27] By "sending a request to a context," we mean sending the request to the server specified by the (server-pid, context-id) pair, with the specified context-id placed in the request message. This procedure causes the CSname in the request to be interpreted in the given context.

42.2. Additional Features

The context prefix server provides a few other features which are useful in the present V-System environment.

An entry in the server's context directory includes space for a type indication and some flag bits, as well as an associated instance id and a long word of client-defined information. Space for these is also included in the standard context request and context reply message structures. The only bits of the *entrytype* field which are client-settable are the SESSION and LOGICAL_PID bits. The SESSION bit has no meaning to the context prefix server, but is used by other standard V software to flag the primary name assigned to a session at the time it is created. The instance id of a session is recorded in the *instanceid* directory field.

The LOGICAL_PID bit indicates to the context prefix server that the given server pid is to be interpreted as a logical pid in the ANY_PID scope rather than an actual pid. Every time the server's name mapping algorithm passes through this entry, it will issue a `GetPid()` request to obtain the next pid to use.

— 43 —
Team Server

43.1. Overview

The team server loads and keeps track of teams (usually equivalent to programs -- although a program may consist of more than one team) running on a local host. It accepts requests to load teams and terminate teams, and implements a directory which can be read to find out information about all teams currently running. The team server also registers itself with the exception server as an exception handler "of last recourse." If no other handler registers itself for the process which incurs an exception (or its ancestors), then the team server will receive the exception message and will load a post-mortem debugger to handle matters from there on. (See the command **debug** for a description of the debugger that is used.)

The team server resides on the "first team" on a host, i.e., it is considered to be a server which is always present on a host and is loaded automatically when a host's V-System is booted.

43.2. Team Loading

Teams can be loaded from specific object code files using the library routines `LoadProg()`, `ExecProg()`, or `RunProgram()` in the V library. These package up an appropriate request to the team server and take care of matters such as setting up the initial arguments to a team on its stack. The team server only creates a new team and loads down its object code from a designated open file instance. Setting up parameters and setting initial execution priority and stack size is left to the team load requestor in order to allow control over the order of events. This is necessary for programs such as debuggers which wish to allow users to set breakpoints and examine the code before a team actually starts to run.

Load requests to the team server also specify who the "owner" of a team is. Teams are destroyed if their owner process goes away (same semantics as for processes created by other processes). Teams can optionally be specified to be owned by the team server itself, thus permitting them to outlive their load requestors.

43.3. Team Termination

Teams can terminate by either having their root process destroyed or by sending a termination request to the team server (the library routine `exit()` does this). The latter form also causes the team server to destroy the team's root process; but in addition it allows the team server to immediately update its record of the state of currently running teams. The server uses a timer process to periodically query the state of all teams which the server thinks are still running and remove server entries for those that have unexpectedly gone away.

43.4. Status of Running Teams

The standard context directory listing protocol (see section 30.7) can be used to obtain information on all teams which are currently running under the team server. To obtain information on a specific team only, an NREAD_DESCRIPTOR request can be made. The team of interest is specified by setting the request message's *contextid* field to the team's root process id; the CSname in the message has no significance.

43.5. Remote Execution

The implementation of the team server and team-loading library routines is such that load requests can be made to both local and remote team servers, thus allowing for transparent remote execution of V programs. In order to assure the priority of local requests the team server keeps track of the state of the local host with respect to things such as whether someone is logged in or not, how many applications are running, etc. This state is used to determine whether or not a remote load request will be accepted or not.

Currently the only state information maintained by the team server is whether or not someone has logged into the host. Also, the current policy with respect to remote execution is to accept all requests regardless of the local host's state.

The team server also interacts with the **service server** in order to globally register the current state of its host. An update of the host's status is sent whenever its state changes and whenever the service server requests such an update (e.g. when the service server first starts up and needs to acquire the current state of all hosts in the system).

Part IV:
Kernel

— 44 —
Kernel Overview

The V kernel is a message-based distributed kernel that implements a program environment of many small processes communicating by messages. This program environment is implemented on one or more workstations connected by a local network. The kernel was designed to provide an efficient, real-time process model on which to build sophisticated single-user systems, multi-user systems, network-accessed servers and dedicated real-time applications. These applications may be distributed over one or more network nodes or workstations. The kernel is also designed to be reasonably portable over a large class of machines and local networks.[28] This manual describes the V kernel: its operations, the mechanics of using the kernel, the kernel internal structure, and how to maintain the kernel, namely adding kernel operations and devices. Kernel operations can be broadly divided into three categories: process and memory management, interprocess communication, and device management. The following sections of this chapter provide an overview of the kernel facilities and their intended use.

44.1. Process and Memory Management

The kernel manages memory as entities called *team spaces*, which correspond to an address space or context on the workstation. For example, on the SUN workstation a team space is a context as implemented by the hardware memory management. Operations are provided for creating team spaces, querying the size of a team space, and setting the size of the team space. Team spaces disappear when the last process contained in that space is destroyed, so there is no explicit operation for destroying a team.

A team space is entirely contained on a single workstation. On some machines, the kernel is actually part of the team address space but this fact is transparent to the program. For instance, on the SUN processor board, segments 0 and 1 in every context are kernel space, but protection bits are set to prevent access except in supervisor mode.

A process is a logical activity that sequentially executes instructions. Associated with each process is a priority, state, a team space and a stack. The process priority dictates the preference given to this process with respect to processor allocation. The highest priority *ready* process is allocated the processor. (0 is the highest priority.) The state is essentially the machine state of the processor for that process. The team space is the area of memory to which the process has direct access. The stack is the local memory area contained in the team space that the process uses for local workspace, procedure linkage and return, and the like. All processes with the same team space are said to be on the same team.

The kernel provides support for a *per-process area* by associating a location and value with each process. Whenever a process is activated, the kernel stores its per-process value in its per-process location. By convention, each process on a team uses the same per-process location, and each per-process value is a pointer to a standard per-process data area within the process's stack space.

Processes can be dynamically created and destroyed. When a process is created, it is assigned a unique *process identifier* that is used subsequently to specify that process. Also, it is created as part of the same team as its creator. A process is created in the initial state of *awaiting-reply* from its creating process. (See next section on interprocess communication.) When a process is destroyed, all the processes created by this

[28] Currently, it has only been implemented on the Motorola 68000-based SUN workstations connected by 3 Megabit or 10 Megabit Ethernet. An implementation on a VAX 11/750 is under way.

process are also destroyed.

44.2. Interprocess Communication

Interprocess communication is provided in two forms by the kernel. First, processes may send, receive, reply to, and forward fixed-length synchronous messages. A process sending a message is suspended *awaiting reply* until the message it sent has been received and replied to by the receiving process. Messages are currently 8 full words, where a full word is defined to be the maximum of the space required for a general machine pointer and the space required for a "natural" machine precision integer (32 bits on the MC68000-based SUN workstation).

Second, a process can pass access to a single segment in its team space to the recipient of its message. The recipient process can access this segment for reading or writing, depending on the access specified by the sender, while the process is awaiting reply from the recipient. By convention, the segment start address and size are specified by the last two words of the message by which access to the segment was given. The presence of a segment and its access modes are specified in the first byte of the message.

A process that is blocked awaiting reply from a process that is subsequently destroyed is unblocked with an indication that the receiver of the message does not exist.

44.3. Naming

The kernel implements a low-level naming service that provides efficient access to server processes that implement higher level functions. A process can register its process identifier as corresponding to a particular *logical process identifier*. Processes can then query the kernel as to the process identifier corresponding to a specified logical process identifier. Registration of the logical to real process identifier can be specified as local to a workstation, remote, or both.

44.4. Time Management

The kernel provides operations for reading the time, setting the time, delaying for a time period, and unblocking a delaying process.

44.5. Device Management

Devices managed by the kernel are currently all accessed through the device server pseudo-process inside the kernel. Operations are performed by sending messages to the device server. The protocol used in these messages is the Verex I/O protocol[29] which is described in the *V-System Servers Manual*.

Devices that can be controlled without special kernel support can be handled directly by processes. Special devices that require kernel support but do not fit the I/O model can be handled by adding new kernel operations.

44.6. Initialization

After the kernel has completed its internal initialization, it creates an initial team space and an initial process on this team. It assumes there is a descriptor following it in memory that describes the code and data segments plus the starting instruction for this initial team. Enough physical memory is assigned to the team to

---

29. "Distributed I/O using an Object-Based Protocol" by David R. Cheriton, UBC Computer Science Technical report 81-1.

accommodate its code and data segments.

44.7. Distributed Operation

The kernel supports transparent communication among several workstations running the V kernel. Processes on different workstations may send and receive messages and access segments as though all processes were executing on the same machine. This mode of operation requires a high-speed local network connecting the workstations. Most kernel operations may be performed transparently on non-local processes.

44.8. Application-Level Model

Using the kernel well requires understanding the model of processes and messages that the kernel provides, and how they are intended to be used. Processes represent logical *activities* in the application. They are intended to be sufficiently inexpensive to allow the use of multiple processes to achieve the desired level of concurrency in the application. The *process identifier* is intended to serve as a loose form of *capability* or "ticket." Possession of a process identifier is sufficient to allow the process to send a message to the specified process. Also, because there is no notification facility on the destruction of a process, resources allocated to a process should be associated with its process identifier if they are to be reclaimed. The application can then use "lazy reclamation" of resources by "garbage collecting" resources associated with invalid process identifiers. However, a process may block until another is destroyed using either ReceiveSpecific or Send.

The synchronous message sending is intended to implement communication between processes that looks to the sender essentially like procedure calls. That is, the Send request message sends the parameters of the procedure and the reply message returns the results. The greater flexibility provided to the receiver allows sophisticated scheduling of message handling and replies. Because message sending is totally synchronous, concurrency must be achieved by multiple processes.

The segment access operations follow the procedure paradigm in being used primarily to access what are logically "call-by-reference" parameters. The argument for providing exactly one segment is that at least one is needed, and one is sufficient for the dominant activity, namely file access. It is expensive and difficult to provide arbitrarily many segments -- having just one segment allows a simpler and more efficient implementation. Finally, multiple segments can be linearized to one, so no functionality is lost with this restriction.

There is no form of asynchronous communication between processes. It is intended that process destruction be used for asynchronously interrupting the activity of a process.

Teams are intended to provide fine-grain sharing of code and inexpensive sharing of data between cooperating processes. They separate the idea of program, executable unit, and address space from that of process.

— 45 —
Kernel Operations

The operations provided by the V kernel can be divided into three classes: kernel traps, kernel process operations, and kernel device operations.

The most basic kernel operations, including **Send()**, are implemented as kernel traps. These operations are invoked by executing a trap or system call instruction which invokes the kernel. A number of secondary operations are implemented by a pseudo-process running in the kernel, called the kernel process. Such operations are invoked by sending to the kernel process's pid. Finally, operations on kernel-implemented devices are provided by a second pseudo-process, called the kernel device server. Such operations are invoked by sending messages to the device server's pid, using the standard V-System I/O protocol.

The kernel traps include:

```
Forward()              GetPid()              MoveFrom()
MoveTo()               ReceiveSpecific()     ReceiveWithSegment()
ReplyWithSegment()     RereadMsg()           Send()
```

The kernel process operations include:

```
CreateProcess()        CreateTeam()          Delay()
DestroyProcess()       GetTime()             QueryProcessState()
SetPid()               SetTeamPriority()     SetTeamSize()
SetTime()              Wakeup()              WriteProcessState()
```

These functions are documented fully in the *V-System Program Environment Manual*. Other kernel operations described there, such as **Receive()**, **Reply()**, **ValidPid()**, etc., are implemented as library functions using the basic operations listed above.

— 46 —
Exceptions and Kernel Exception Handling

The V kernel handles exceptions (such as illegal instruction or bus error traps) by forcing the erroneous process to send a message to the *exception server* containing the details of the error. The exception server is a process that has registered as the EXCEPTION_SERVER with the kernel using SetPid. If no exception server exists, the message is sent back to the process that caused the error, blocking it permanently. In either case, other processes in the system can continue to run.

The message from the exception-incurring process to the exception server appears as a normal message and can be received by the standard message primitives. After receiving the message, the exception server can read the faulting process's state using either the kernel primitive QueryProcessState or the library function ReadProcessState. In the Sun implementation of the kernel, the registers in the state record reflect the processor state at the time of the exception, unless it occurred while running in the kernel. In that case, the program counter and status register returned are those that were saved at the time the kernel trap was taken. The other registers reflect the state at the time of the exception. The correct program counter and status register contents can always be obtained from the exception message.

The message also provides read and write access to the segment consisting of the entire team address space of the exception-incurring process. This provides debuggers and exception servers with complete access to the code and data of the process.

The format of the exception message is given by the ExceptionMessage struct in <Vexceptions.h> together with manifest constant definitions for the type of exception. The message format for the Motorola 68000 is given below.

requestcode	EXCEPTION_REQUEST
type	Type of exception. This is encoded as the address of the MC68000 exception vector that was taken.
buserrortype	Additional information on the cause of the error, if *type* is equal to BUS_ERROR.
code	In the case of address errors and bus errors, this contains a code returned by the processor to identify the type of memory reference that caused the exception. Only the low order 5 bits of this word are valid; the others should be masked off. For other types of exceptions, this word contains zero.
accaddr	The access address in the case of address or bus errors, otherwise zero.
instruction	The instruction register in the case of address or bus errors, otherwise zero. This should be the first word of the instruction that caused the error.
status	The status register.
errpc	The program counter.
segment	Starting address of the team's address space. Currently at 0x10000 on the SUN workstation because the kernel uses the first two segments.
segmentsize	Number of bytes in the team's address space.

See the Motorola MC68000 User's Manual for a description of the types of exceptions possible and the meaning of the information returned by the processor and passed on in these exception messages.

Exceptions are always blamed on the currently active process, even if they occur inside the kernel. For instance, it is possible for an exception inside the kernel to be caused by an invalid pointer passed by a process when invoking a kernel operation. However, it is also possible for exceptions to be caused by bugs in the kernel itself, though this is unlikely unless an experimental version of the kernel is being tested.[30] A standard exception server process has been implemented and is described in the *V-System Servers Manual.*

[30] The worst case is when the exception is caused by a bug in a kernel interrupt handler, since in that case there would be no relation between the currently active process and the code that caused the exception; however, in this case, the bug is likely to crash the processor anyway.

— 47 —
Performance

Two measures of performance for the kernel are the speed of various operations and space requirements. For a detailed account of the performance of the V kernel, we refer the reader to *The Distributed V Kernel and Its Performance on Diskless Workstations*, by David R. Cheriton and Willy Zwaenepoel, in Proceedings of the 9th Symposium on Operating System Principles, October 1983 (also available as Technical Report STAN-CS-83-973, Computer Science Department, Stanford University).

47.1. Space Requirements

The space requirements are dependent on the machine, the number and complexity of devices supported, and the maximum number of processes, teams and devices configured. Table 47-1 gives the code segment size for the kernel configured to support distributed operation on the SUN workstation with support for console device, serial interfaces, Ethernet interface and a mouse pointing device. The table also gives the unit space cost of a process descriptor and a team descriptor, and the total space requirements for a kernel configured with a maximum of 64 processes, 16 teams and 16 device descriptors. All measurements are given in bytes.

Table 47-1: SUN Workstation Kernel Memory Requirements

Component	Size in bytes
code segment	31836
process descriptor	202
team descriptor	18
device descriptor	44
total	47990

47.2. Kernel Operation Times

Table 47-2 gives the times for various sequences of kernel operations on the SUN workstation using a 10 Megahertz Motorola 68000 processor. The times for sequences of operations are given instead of times for individual operations to give a better indication of the kernel overhead for higher-level operations. For instance, the Send-Receive-MoveFrom-Reply sequence is indicative of the time to perform a file read operation using the I/O protocol. These sequences are also easier to time in some cases than individual component operations.

Table 47-2: SUN Workstation Times for Kernel Operations (in milliseconds)

	local	remote
GetTime	0.06	Not Applicable
Send-Receive-Reply(0 bytes)	0.77	2.54
Send-Receive-Reply(512 bytes)	1.31	5.56

The table is not intended to be complete but simply indicative of performance. The Send-Receive-Reply time is indicative of the interprocess communication time using the kernel. The column labeled "remote" gives the Send-Receive-Reply time between two processes resident on different workstations. The GetTime time is indicative of the cost of a trivial kernel operation. All other kernel operations can be expected to be faster than the Send-Receive-Reply sequence.

Another measure of interest is the speed at which packets can be read and written over the Ethernet with the overhead of sending read and write requests to the kernel device server to access the network. Table 47-3 gives performance figures for the kernel running on the SUN workstation connected to a 3 Megabit Ethernet.

Table 47-3: SUN Workstation Ethernet Output

Packet Size in bytes	Packets per sec	Throughput in Kbytes/sec
16	2200	35
512	360	180
1024	170	170

Similar throughput figures for a stand-alone Alto are 4, 120 and 140 Kbytes per sec (approximately).

On the input side, with one SUN writing as fast as possible to another, 10000 packets of 1024 bytes can be received in 5.89 seconds with 0 packets lost, yielding a throughput of about 170 Kbytes per second, which is about 40% of the bandwidth of the net. However, for some unknown reason, packets are dropped as the packet size becomes smaller.

47.3. Interrupt Disable Time

The interrupt disable time for the kernel is essentially the maximum of the time required to insert a record in a ordered queue (the ready queue) and the time to load the processor with the state of a new process. Although the former is dependent on the maximum number of ready processes, it is typically very small. For example, on the SUN workstation, adding a lowest priority process to the ready queue when 32 processes are ready to execute is estimated to result in an interrupt disable time of 164 microseconds. Under normal circumstances, an interrupt disable time of about 30 microseconds can be expected.

— 48 —
Kernel Internal Structure

The kernel is implemented as a simple monitor. It executes logically in its own address space in supervisor mode with its own code, data and stack. It is invoked by trap operations and interrupts. When a process executes a kernel operation or an interrupt trap is taken, the kernel executes on the kernel stack.

48.1. Teams

Each team is represented by a team descriptor record (TD) that describes the team space, records the root process of the team, user associated with the team, team priority level, etc. A machine-dependent portion of the team descriptor describes the team's memory space.

48.2. Processes

For each process, the kernel maintains a process descriptor record (PD) that contains the process state and sundry information about the process. When a process is running, a variable Active points at the process descriptor of the currently active process.

48.3. Kernel Synchronization

The kernel is synchronized internally by a combination of scheduling conventions and interrupt masking. The conventions are:

- Both kernel trap-invoked and interrupt-invoked operations only add or remove processes from the list of ready processes. They cannot block a process in the middle of a kernel operation. The clock interrupt routine that may change the state of a process blocked on a remote or nonexistent process only does so if it did not interrupt a kernel operation. Similarly, ethernet interrupts are disabled during the execution of a kernel operation to prevent remote interkernel packets from interfering with the execution of the kernel operation.

- A process switch occurs at the end of a kernel operation if the active or invoking process is no longer the highest priority ready process. The process switch occurs at the point of return from the kernel trap handler after executing the kernel trap.

- A process switch occurs at the end of the execution of an interrupt service routine if the active process is no longer the highest priority ready process AND the interrupt servicing did not interrupt a kernel operation. That is, a process switch cannot occur in the middle of a kernel operation due to an interrupt even though the interrupt can otherwise be serviced.

The net result is that a process executes a kernel operation indivisibly with respect to other processes until it blocks. However, the highest priority ready process is allocated the processor whenever the processor is not in supervisor state. Masking of interrupts is used at crucial points in manipulation of the ready queues and process switching so that interrupt routines do not interfere.

48.4. Interrupt Routines

Interrupts are handled by first invoking a simple assembly language routine that saves some registers and then calls a C procedure associated with that interrupt level, possibly passing some arguments. A macro "Call_inthandler" generates the required assembly language routines that call the C procedure it is passed as an argument. Interrupt-invoked routines are assumed to be short and do little in interacting with processes other than possibly readying a process.

48.5. Kernel Traps

An assembly-language module handles trap instructions, invoking the specified kernel operation and handling the return. On a trap, it moves the arguments onto the kernel stack and calls the specified kernel operation as a C function. On return, it moves the return value back to the process's stack if necessary and checks for a higher priority ready process. If there is one, it switches to the highest priority process. If the active process is still the highest priority ready process, the active process is allowed to continue execution at the instruction after the trap instruction in its code segment.

48.6. Kernel Process

If the specified pid fails to validate on a Send, the Send routine checks whether it is the pid of the kernel process or of the device server process. If the kernel process pid was specified, Send calls the SendKernel routine to perform the requested operation. Thus, the "kernel process" code is executed by the process invoking the operation, not a separate process running in the kernel. The message format and the request codes the kernel process supports can be found in <Venviron.h>. The kernel process identifier is a global variable, Kernel_Process_Pid, set at the beginning of each team's execution.

48.7. Device Server Process

A Send to the device server process results in Send calling the SendDevice routine to perform the requested operation. Thus, the device server code is executed by the process invoking the operation, not a separate process running in the kernel. A process that is forwarded to the device server has its finish-up function (see below) set to SendDevice, and is readied, so that it will begin executing in SendDevice as soon as it reaches the head of the ready queue.

48.8. Process Switching

All process switches occur in the macro function Switch that switches from the currently active process to the process at the head of the ready queue. Each process is created with its state initialized to start it at the initial program counter in its team space when it is readied. Switch relies on there always being a ready process to execute (i.e. non-empty ready queue). This is guaranteed by the presence of an "idle" process that is always ready and executes the processor stop or idle instruction.

Interrupt-invoked routines execute as "involuntary" asynchronous function calls made by the currently active process and thus can also use Switch.

Process switches always occur upon exit from the kernel, never in the middle of a kernel routine. Thus, the kernel only requires one stack, not a separate kernel stack for each process. If there will still be some work to be done on a kernel operation when a process is unblocked, the kernel routine that blocks it sets the *finish-up* function field in the process's state record. If this field is non-zero when a process is unblocked, the specified function is called before the process exits the kernel. A finish-up function can block the process again and set another finish-up function if necessary.

Note: The kernel implementation described so far should support a number of different types of kernels. Also, this basis of trap and interrupt handling plus process switching, device management, and memory management represents most of the machine-dependent code in the kernel.

48.9. Processor Allocation

The strict priority-based processor allocation is implemented efficiently by maintaining a queue of ready processes in order of priority, highest priority first. A state field in the process descriptor indicates the process is *ready* (and thus in this list) or else the state in which it is blocked. Process switching incorporating this priority-based allocation and ready queue management is implemented by two (internal) primitives.

Removeready(pd) Remove the specified process from the ready queue. The active process continues to execute until it exits the kernel even if it has just removed itself from the ready queue.

Addready(pd) Add the specified process (descriptor) to the ready queue in order of priority, after all processes of the same priority as this process.

48.10. Process Creation and Destruction

Unused process descriptors are maintained in a queue. When a process is created, a process descriptor is removed from the queue, assigned a process identifier, and initialized to the specified priority, awaiting reply state, creator's team, etc.

When a process is destroyed, it is removed from any system queues, such as the ready queue or any message queues (one major use of the PD state field is indicating presence in a queue), the process identifier is invalidated and all its descendants are destroyed similarly. The resulting free process descriptors are added to the end of the queue of unused process descriptors. The clock interrupt routine is charged with checking for processes blocked on non-existent processes (one per clock interrupt) so the process destruction mechanism need not worry about this.

48.11. Message Primitives

While a message implementation normally requires independent kernel message buffers, the semantics of the message primitives in this kernel allow the message buffer to be statically associated with the process descriptor so we include it as part of the same C struct. Thus, a message is queued at a receiver by queuing the process descriptor of the sender, saving on extra space for sender identifier, etc. plus time to map to the PD of the sender for unblocking it.

Sending to the kernel device server or to the kernel process is handled by checking the pid of Send to see if it specifies the kernel device server or the kernel process when the pid fails to validate as a real process. The SendDevice or SendKernel routine is then called directly to implement the kernel device server or kernel process.

48.12. Time Primitives

Processes delaying using Delay are maintained in a queue starting at Delayq_head ordered by increasing time to unblock. The time before a process unblocks is stored in its blocked_on field in terms of the number of clock interrupts it must delay after the process before it in the queue is unblocked.

48.13. Distributed Operation

The process identifier contains an indication of the host in its 16 high-order bits. When an operation is invoked that specifies a process identifier that fails to validate locally, it is assumed to be a remote process. The operation then invokes a "nonlocal" version of the operation that formats a network message and transmits it to the workstation host specified by the process identifier. The primary interface to the network is the WriteKernelPacket routine.

In the case of GetPid, a message is broadcast requesting the logical id to pid mapping.

When a process is blocked sending to a remote process, the message is retransmitted periodically by the clock interrupt routine until a reply is received. The Send fails after some number of retransmissions if no "breath of life" packets have been received from the remote host in that time.

A message received on a workstation from a remote process causes a process descriptor to be allocated to store the message and make it appear as a local message to the rest of the kernel. A process descriptor used in this fashion is called an *alien*. Aliens are destroyed an appropriate time interval after the Reply message is sent. (This interval is 0 for idempotent requests.)

This description is far from complete. For a fully detailed discussion of the interkernel protocol, see *The Distributed V Kernel and Its Performance on Diskless Workstations*, by David R. Cheriton and Willy Zwaenepoel, in Proceedings of the 9th Symposium on Operating System Principles, October 1983 (also available as Technical Report STAN-CS-83-973, Computer Science Department, Stanford University).

— 49 —
Kernel Modification and Maintenance

The type of kernel modifications anticipated include: changing the maximum number of processes, teams, or devices allowed, adding or removing kernel operations, and adding support for new devices.

49.1. Kernel Configuration Parameters

The machine-dependent file *config.h* contains the kernel configuration parameters.

MAX_PROCESSES
: Maximum number of processes, which must be a power of 2.

MAX_TEAMS
: Max. number of teams, currently at most 16 on the SUN workstation.

MAX_DEVICES
: Max. number of device instances, which must be a power of 2.

ROOT_PRIORITY
: Priority of root process of first team.

INIT_STACK
: Size of initial stack for root process of first team.

The kernel can be reconfigured with respect to these parameters by changing their definitions in *config.h* within the constraints mentioned above and recompiling the kernel.

49.2. Adding New Device Support

Supporting a new device using the kernel device manager requires writing device-specific initialization, read, write, release, modify and interrupt-handling routines and adding an entry for the device in the DeviceCreationTable defined in *config.c*. There is normally a header file for the new device that defines its device type for this table plus other device-specific information required by users of the device. The existing devices and kernel operations are useful models from which to work.

49.3. Adding Kernel Operations

Adding a kernel operation requires writing the C routines that implement the operation, adding an entry for it to the kernel trap table, **kernelops**, defined in *trap.c* and possibly adding a *stub* for this call to the C environment library for the kernel calls. Adding a new operation to the kernel process requires defining a new request code in <Venviron.h>, handling this request code in the main loop of the kernel process and writing the appropriate code for handling the request. Operations that must be available to remote processes should be implemented as kernel process operations rather than kernel traps.

Certain restrictions apply to kernel operations. They may not execute trap operations or call upon services provided by other processes outside the kernel. However, they can use other routines already available inside the kernel. Kernel operations are passed exactly 5 arguments and allow one return value. A kernel operation cannot take a variable number of arguments unless the number is encoded in the values passed. Operations that access any data modified by interrupt-invoked routines need to mask interrupts if there is any possibility of interference. Finally, operations that block or unblock processes should use the internal primitives

Addready and Removeready.

Part V:
Appendices

— Appendix A — C Programming Style

There has been an effort to use a consistent style in V for writing C programs. The style and the uniformity it encourages are motivated by the desire for readability and maintainability of software. Although style is to a large extent a matter of individual taste, the following describes some general practices with which most of us agree.

A.1. General Format

Recognizing that software is written to be read by other programmers and only incidentally by compilers, the general format follows principles established in formatting general English documents. Take a few more seconds to make things more readable; it is time well spent.

First, software is written to be printed on standard size (8 by 11) paper. This means avoiding lines longer than about 80 columns. In general, there is one statement or declaration per line.

As with other documents, judicious use of white space with short lines and blank lines is encouraged. In particular,

1. At least 2 blank lines between individual procedures.
2. Blank lines surround "large" comments.
3. Blank lines around any group of statements.
4. Blank lines around cases of a switch statement.

A.2. Names

Names are chosen when possible to indicate their semantics and to read well in use, for example:

```
if ( GetDevice(EtherInstance) == NULL ) return( NOT_FOUND );
```

Words should be spelled out, not shortened. A good test is to read your code aloud. You should be able to communicate it over a telephone easily, without resorting to spelling out abbreviations.

In addition, character case conventions are used to improve readability and suggest the scope and type of the name. Global variables, procedures, structs, unions, typedefs, and macros all begin with a capital letter, and are logically capitalized thereafter (e.g. `MainHashTable`). A global variable is one defined outside a procedure, even though it may not be exported from the file, or an external variable. The motivation for treating macros in this way is that they may then be changed to procedure calls without renaming.

Manifest constants either follow the above convention (since they are essentially macros with no parameters) or else are fully capitalized with use of the underscore to separate components of the name. E.g. WRITE_INSTANCE.

Local variables begin with a lower-case letter, but are either logically capitalized thereafter (e.g. `bltWidth`, `power`, `maxSumOfSquares`) or else totally lower case. Fields within structures or unions are treated in this manner also.

Local variables of limited scope are often declared as register, if they are used very often inside inner loops. It is not only more efficient, but usually more readable, to put a pointer to an array of complicated structures (a common occurrence in object-oriented programming) into a register variable with a short name. For example,

```
register struct Descriptor *p = DescriptorTable+objectIndex;
p->count = 0;
Initialize(p->start);
p->usage = p->default;
p->length = p->end - p->start;
```

instead of the inefficient and cluttered:

```
DescriptorTable[objectIndex].count = 0;
Initialize(DescriptorTable[objectIndex].start);
DescriptorTable[objectIndex].usage = DescriptorTable[objectIndex].default;
DescriptorTable[objectIndex].length = DescriptorTable[objectIndex].end
     - DescriptorTable[objectIndex].start;
```

A.3. Comments

There are generally two types of comments: block-style comments, and on-the-line comments or *remarks*. Multi-line, block-style comments have the /* and */ appearing on lines by themselves, and the body of the comment starting with a properly aligned *. The comment should usually be surrounded by blank lines as well. Thus it is easy to add/delete first and last lines, and it is easier to detect the common error of omitting the */ and thus including all code up to and including the next */ in a comment.

```
/*
 * this is the first line of a multi-line comment,
 * this is another line
 * the last line of text
 */
```

On-line comments or remarks are used to detail declarations, to explain single lines of code, and for brief (i.e. one line) block-style descriptive comments.

Procedures are preceded by block-style comments, explaining their (abstract) function in terms of their parameters, results, and side effects. Note that the parameter declarations *are* indented, not flushed left.

```
SystemCode EnetCheckRequest( req )
   register IoRequest *req;
  {
        /*
         * Check that the read or write request has a legitimate buffer, etc.
         */
    register unsigned count;
    register SystemCode r;

        /*  Check length */
    count = req->bytecount;
    if( count <= IO_MSG_BUFFER ) return( OK );

    req->bytecount = 0;          /* To be left zero if a check fails */
    if( count > ENET_MAX_PACKET )
      {
       r = BAD_BYTE_COUNT;
      }
    else
      {
          /*
           * Make sure data pointer is valid.
           *  Check that on a word boundary and not in the kernel area.
           */
```

```
        if( (!CheckUserPointer(req->bufferPointer)) ||
            (Active->team->teamSpace.size < (req->bufferPointer + count)) ||
            ((int) req->bufferPointer) & 1 )
          {
            r = BAD_BUFFER;
          }
        else
          {
            req->bytecount = count;
            r = OK;
          }
      }
    return( r );
  }
```

A.4. Indenting

The above example shows many of the indenting rules. Braces ("{" and "}") appear alone on a line, and are indented two spaces from the statement they are to contain. The body is indented two more spaces from the braces (for a total of four spaces). **else**'s and **else if**'s line up with their dominating **if** statement (to avoid marching off to the right, and to reflect the semantics of the statement).

```
if ( (x = y) == 0 )
  {
    flag = 1;
    printf(" the value was zero ");
  }
else if ( y == 1 )
  {
    switch ( today )
      {
        case Thursday:
            flag = 2;
            ThursdayAction();
            break;

        case Friday:
            flag = 3;
            FridayAction();
            break;

        default:
            OtherDayAction();
      }
  }
else
    printf(" y had the wrong value ");
```

A.5. File Contents

File contents are arranged as follows.

1. initial descriptive comment (see example below) contains brief descriptive abstract of contents. Some programmers add one or more of the following as well:
   a. a list of all defined procedures in their defined order, or alphabetically.
   b. list of recent and major modifications in reverse chronological order with indication (initials) of who made the change.
2. included files (use *relative* path names whenever possible)

3. external definitions (imports and exports)
4. external and forward function declarations
5. constant declarations
6. macro definitions
7. type definitions
8. global variable declarations (use static declarations whenever possible, and group variables with the functions that use them)
9. procedure and function definitions

Here is the beginning of a file as an example.

```
/*
 * Distributed V Kernel - Copyright (c) 1982 by David Cheriton, Willy Zwaenepoel
 *
 * Kernel Ethernet driver
 */

#include "../../libc/include/Vethernet.h"
#include "interrupt.h"
#include "ethernet.h"
#include "ikc.h"
#include "../mi/dm.h"

        /* Imports */
extern Process *Map_pid();
extern SystemCode NotSupported();
extern DeviceInstance *GetDevice();

        /* Exports */
extern SystemCode EnetCreate();
extern SystemCode EnetRead();
extern SystemCode EnetWrite();
extern SystemCode EnetQuery();
extern SystemCode EnetCheckRequest();
extern SystemCode EnetReadPacket();
extern SystemCode EnetPowerup();

unsigned char   EnetHostNumber;         /* physical ethernet address */
InstanceId      EthernetInstance;       /* Instance id for Ethernet */
int             EnetReceiveMask;        /* addresses to listen for */
short           EnetStatus;             /* Current status settings */
int             EnetFIFOempty;          /* FIFO was emptied by last read */
int             EnetCollisions = 0;     /* Number of collision errors */
int             EnetOverflows = 0;      /* Queue overflow errors */
int             EnetCRCerrors = 0;      /* Packets with bad CRC's */
int             EnetSyncErrors = 0;     /* Receiver out of sync */
int             EnetTimeouts = 0;       /* Transmitter timeouts */
int             EnetValidPackets = 0;
char            kPacketArea[WORDS_PER_PACKET*BYTES_PER_WORD+20];
                                        /* Save area for kernel packets */
kPacket         *kPacketSave = (kPacket *) kPacketArea;
                                        /* Pointer to kernel packet area */


        /* Macro expansion to interrupt-invoked C call to Ethernetinterrupt */
CallHandler(EnetInterrupt)
```

A.6. Parentheses

For function calls, the parentheses "belong to" the call, so there is no space between function name and open parentheses. (There may be some inside the parentheses to make the argument list look nice.) When parentheses enclose the expression for a statement (**if**, **for**, etc.), the parentheses may be treated as belonging to the statement (since they are syntactically required by the statement) so there is no space between the keyword and the expression.

```
if( (bytes = req->bytecount) <= IO_MSG_BUFFER )
    buffer = (char *) req->shortbuffer;
else
    return( req->bufferPointer );
```

Alternatively, parentheses may be treated as belonging to the expression, so there is a space between the keyword and the parenthesized expression.

```
if (FuncA())
  {
    FuncB( (a = b) == 0 );
    return (Nil);
  }
else
  {
    FuncC( a, b, c );
    return (ToSender);
  }
```

Note that spaces are used to separate operators from operands for clarity and may be selectively omitted to suggest precedence in evaluation.

A.7. Messages

Although V is a message-based system, most services are available by calling standard routines, so programming at the "message level" is rarely necessary or desirable. However, the programming of new servers and the non-standard use of services or the use of messages within a program require message-level programming. The following conventions have been followed in V.

Space to send or receive a message is declared of type Message, as defined in <Venviron.h>. Standard message formats, as defined in the V header files, declare each message format to be a new data type. Access to the space for the message is made by casting a pointer to the space to be of the type of the message format required. This guarantees that enough space is reserved even when a message format is not as large as the fixed-size message used by the kernel. The following illustrates this style.

```
Read( fad, buffer, bytes )
  File *fad;
  char *buffer;
  int bytes;
   /*
    * Read the specified number of bytes into the buffer from the
    * file instance specified by fad. The number of bytes read is
    * returned.
    */
  {
    Message  msg;
    register IoRequest  *request = (IoRequest *) msg;
    register IoReply    *reply = (IoReply *) msg;
    register unsigned r, count;
    register char *buf;
```

```
for(;;)
  {
    request->requestcode = READ_INSTANCE;
    request->fileid = fad->fileid;
    request->bufferPointer = buffer;
    request->bytecount = bytes;
    request->blocknumber = fad->block;

    if( Send( request, fad->fileserver ) == 0 )
      {
        fad->lastexception = NONEXISTENT_PROCESS;
        return( 0 );
      }
    if( (r = reply->replycode) != RETRY ) break;
  }

fad->lastexception = r;
count = reply->bytecount;

if( count <= IO_MSG_BUFFER )
  {
    buf = (char *) request->shortbuffer;
    for( r = 0; r < count; ++r ) *buffer++ = *buf++;
  }
return( count );
```

— Appendix B — Installation Notes

This document is intended to be an informal collection of information about the problems involved with installing and maintaining the V-System software. The reader should be familiar with the V-System as documented in the V-System manuals, and with the Unix system used for development.

B.1. V-System Distribution

The software should be distributed on a 1600 bpi tar format tape. Licensing information and tapes can be obtained from:

Office of Technology Licensing
105 Encina Hall
Stanford University
Stanford, CA 94305
(415) 497-0651

Please report any bugs you find, or improvements you make. All the software is under copyright protection, so you must get a license for any further distributions. Send comments on the software and documentation to the Arpanet address **vbugs@SU-Pescadero.ARPA.** New versions of the software may be released from time to time.

The first file on the tape is the entire source directory tree for the V-System. Since the first implementation of the V-System is for the Motorola MC68000, our versions of the 68000 C compiler, assembler, and linker are included as the second file on the tape.

Note: This distribution has been booted only on Cadline and SUN MicroSystems Workstations with MC68000s, **not** MC68010s, connected to a VAX by a 3Mb experimental Ethernet, using PUP boot protocols. The next release will support the MC68010, 10 Mb standard Ethernet, and booting via the SMI network disk protocol.

The first step is to run **tar x** to extract the two files into directories in your file system wherever you have room. Remember to use the non-rewinding driver (e.g. **/dev/rmt12**) or the **mt fsf** command if you want to read the second file. Throughout this document the V-System pathnames will be referred to as V/something, and the 68000 directory as sun/something.

B.2. 68000 Tools

We normally put the 68000 tools into /usr/sun. There are a few other required directories that are hardwired into a few of the makefiles. /usr/sun/include is for the include (**.h**) files, and /usr/sun/lib is for libraries. The two major libraries that are needed by some of the V-System servers are **libsfonts.a** for the character fonts, and **libgraphics.a** for the SUN graphics primitives. We put binary versions of the stand-alone bootfiles under /usr/sun/bootfile, and put the V-System commands under /usr/sun/Vboot.

Many of the V-System makefiles invoke the "cc68" command to compile and link. Be sure you have the latest version of the cc68 command, with the **-V** option. Connect to **sun/src/cmd** and do a **make install**. You might want to edit the command file to put the commands in a place other than

/usr/local/bin. Next connect to sun/src/graphics/lib and do a **make install** to make the graphics library. There will be a few warnings issued by the compiler which should be ignored. There are manual entries for the 68000 software in /usr/sun/man68.

Our current V-Server software requires the CMU packet-filtering Ethernet driver for 4.1 or 4.2 Unix. Make sure the maximum packet size (MTU) is large enough to fit all the data bytes in a kernel packet plus the header. This driver and the associated higher-level software is available to people who have purchased Xerox 1100 workstations in a separate distribution.

Users who want to do 68000 development on a 68000-based Unix machine should be able to do so with a small amount of work. Please report your experiences back to us so that any software or information can be included in future releases.

The **ipwatch** family of programs under sun/diag/ipwatch are very useful to debug network problems. The **enwatch** program is used for 3Mb experimental Ethernet, and **ecwatch** for the 3Com interface. Others could be added easily. It keeps a record of the network packets of interest which can be written to a log file. Please include such a log file in all error reports.

B.3. Making the V-System

Edit the shell script under **V/netinstall** to do the appropriate installation procedure for your system. We have it ftp the files to several other machines to automate the installation. This and a few other shell scripts are assumed to be in the search path by the V-System Makefiles. These sources are in V/tools and should be installed into some directory in the search path before making the rest of the system. Each directory contains a file called **buildfile** which is processed by the **buildmake** program to produce a **makefile**. The **buildfile** step includes conditional macro expansion.

Change directory to **V/libc** and do a **make install-includes**. This should copy the V-System specific include files into /usr/sun/include. Then do a **make** and then **make install** under this directory. This should result in **libV.a** and **teamroot.b** being copied into /usr/sun/lib.

Next change directory to **V/servers**, and do a **make** followed by a **make install**. The Vserver is usually installed in **/etc/Vserver** and then a line is added to **/etc/rc** to start it up on system reboot. Give it a large argument on the command line, so that it can put useful information into the area printed by the Unix **ps** command. It should be run as super-user, to allow it to check access protections correctly and setuid to the correct user.

The following are the options available on the Vserver:

-d	Debug flag for the major server code.
-g	Used if your system does not have the simultaneous group feature (4.1a systems and beyond have this).
-k	Kernel debug. Used to debug the kernel simulator.
-n	Network debug. Used to produce a trace of network packets sent and received.
-p	Public mode. If this flag is set then broadcast GetPid requests are answered. The default is to answer only requests directed specificly at this particular host. There must be at least one Vserver running the **-p** option on any given local network.

Then change to the **V/kernel** directory and do a **make** followed by a **make install** to compile the kernel and put the binary into **/usr/sun/bootfile**. You may have to edit the makefile to configure the kernel for your I/O devices. The default is to support the Sun MicroSystems Experimental 3M bit Ethernet interface. The 3Com Multibus Ethernet Interface can also be supported. Other devices such as disk controllers will be supported in the next release.

Change directory to **V/cmds** and again do a **make** followed by a **make install** to compile all the commands. This takes a while, and uses the include files, libraries, and servers.

Finally, change to the V/standalone directory. This directory is for bootstrapping and loading utilities. Currently the kernel and system team are loaded with the PUP EFTP protocol. The **Vload** program is compiled with several different flags. By default it will ask for a first team file, and possible the name of a kernel. By defining the symbol FIRST_TEAM a specific first team file can be used.

It is also possible to use the V protocol itself to do the bootstrapping. In fact, some day we might put such a bootstrap into the PROMs to make the booting process easier.

Index