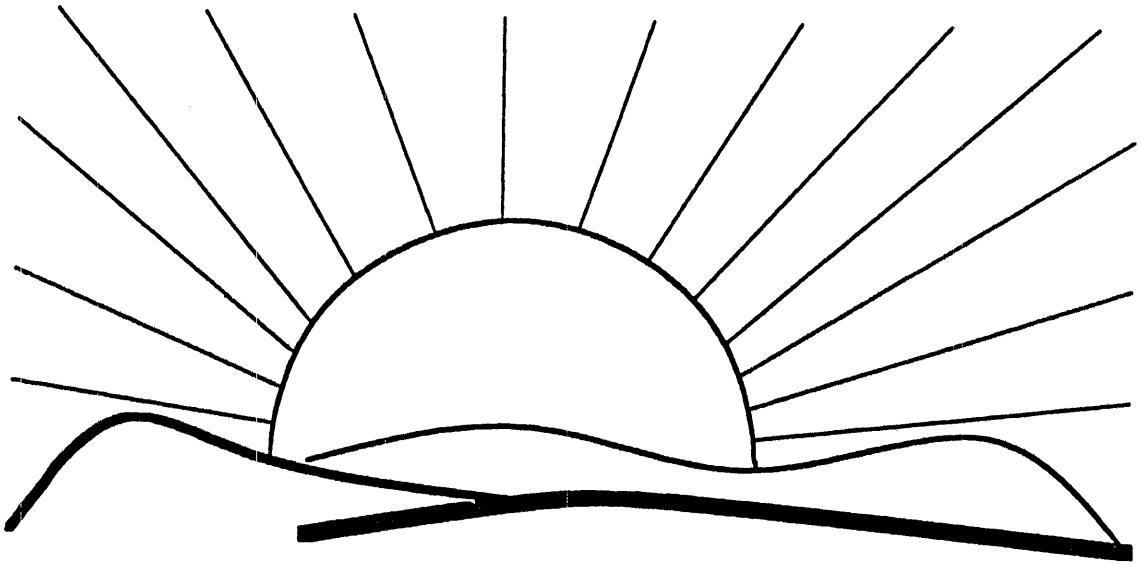
V-System 6.0 Reference Manual



Distributed Systems Group
David R. Cheriton and Keith A. Lantz, Principal Investigators

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

20 June 1986

Preface

The following are a list of some of the major changes evident in Version 6.0 as compared to Version 5.0:

- General changes:
  - Support for Sun2/50's, Sun-3's and VaxStation-II's.
  - Better documentation: as you can see...
- Lots of new commands, including:
  - the C program development environment, including **cc68**, **ld68**, and **build**, an enhanced version of **make**
  - **draw** has been completely redone, including Postscript support
  - the Revision Control System
  - **xlisp**, a variant of Lisp
  - The family of TeX document compilers, including **tex** and **latex** (sorry, no sources)
- New or changed services:
  - **Global authentication services**: Users now authenticate themselves once and need no longer authenticate each "session" independently. This is implemented by a combination of a global authentication server and V-to-UNIX "correspondence tables" maintained by the V servers running under UNIX.
  - **Decentralized object naming**: Local name servers have been eliminated in favor of each manager maintaining the name space of the objects it manages. Objects are located with group IPC.
  - **Group IPC**: Logically, messages are now sent to groups of processes, rather than to individual processes. ("Singleton" groups are, in fact, special cased.) This permits a sender to send one copy of a message, but have it delivered to multiple recipients, in response to which multiple replies may (and can) be received.
  - **"RAM disk"**: A V-storage-server-compatible server whose files are stored in main memory. Useful for temporary files and the like.

This reference manual attempts to be as faithful to the indicated release of the system as possible. Unfortunately, there are almost certainly many errors — most frequently errors of omission. The typical solution to this problem is to read the source code, but this too has its problems. The V-System is the product of a research effort and is constantly undergoing revision. It has not always been possible to keep released and experimental versions strictly separated. Often, the source will include conditionally compiled code, or declarations for constants and data types that are not fully supported in the released version of the system. Therefore, programmers should be wary of using features found in the code that are not documented in this manual. In brief:

> ***Warning:*** Any part of the V-System may change without notice. As a result, this document should be regarded strictly as advisory.

Notes for installing the V-System are to be found in Appendix C.

Contributing authors include Lance M. Berc, Eric J. Berglund, Per Bothner, Kenneth P. Brooks, David R. Cheriton, Stephen E. Deering, J. Craig Dunwoody, Judy L. Edighoffer, Ross S. Finlayson, Cary Gray, Bruce L. Hitson, David R. Kaelbling, Keith A. Lantz, Timothy P. Mann, Thomas Maslen, Robert J. Nagler,

William I. Nowicki, Joseph Pallas, Paul J. Roy, Jay Schuster, Michael Stumm, Marvin M. Theimer, Christopher Zulceg, and Willy Zwaenepoel.

The following are trademarks of Digital Equipment Corporation: DEC, DECSystem-20, TOPS-20, Unibus, VAX, VAXStation, VMS, VT-100, and MicroVAX.

Ethernet is a trademark of Xerox Corporation.

SUN Workstation is a trademark of Sun Microsystems Inc.

UNIX is a trademark of AT&T Bell Laboratories.

V-System is a trademark of Leland Stanford Junior University.

Table of Contents

List of Figures

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 a 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 Hardware Environment

The V-System is targeted for a hardware environment consisting of (see Figure 1-1):

- powerful workstations with:
  - o a high-resolution (e.g. 1024 by 1024) raster display;
  - o a general-purpose 1 MIPS (or better) processor;
  - o 2 Mbytes or more of local memory;
  - o a large (greater than 20 bits) virtual address space;
  - o a graphics input device, such as a mouse; and optionally,
  - o a disk,

  which, typically, will be dedicated to a single user at a time;
- a fast (greater than 1 MHz) communications network that will link the workstations;
- a number of dedicated processors providing printing, file storage, general computation support, and other services; and
- access to time-sharing or special-purpose computers and to long-haul computer networks.

This release of the system runs on Sun and VaxStation workstations interconnected by either 3 or 10 Mb Ethernet. "Guest-level" implementations are available for 4.2BSD and 4.3BSD UNIX systems (with Stanford enhancements).

1.2. The User Model

One of the most important functions for the workstation is to provide state-of-the-art user interface support. 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.

Adhering to these principles, the V-System supports a reasonably sophisticated "window system". Multiple executives or shells may be run simultaneously, each of which may run one application in the "foreground" and any number in the "background" (a la the UNIX C-shell). Applications may run local to the workstation or remote. Each application may be associated with one or more separate *virtual terminals*, each of which may

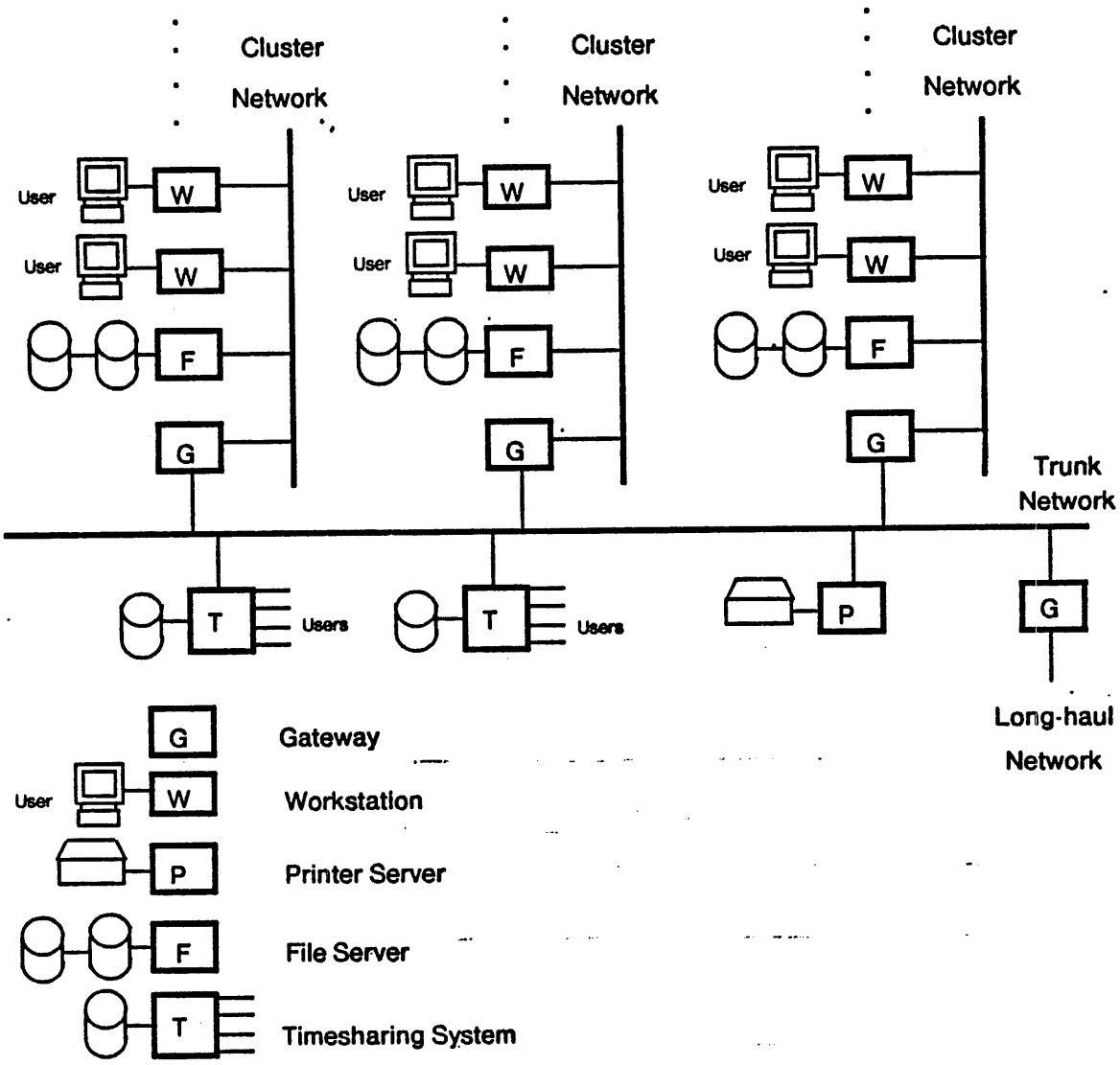


Figure 1-1: A workstation-based distributed system.

be used to emulate either a VT-100 terminal or a 2-D "structured graphics" terminal.

1.3. 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.

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

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

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.

Architecturally, then, the V-System consists of a distributed kernel and a distributed set of server processes.

1.3.1. The Distributed Kernel

The distributed kernel consists of the collection of kernels resident on each participating machine (see Figure 1-2). Each host kernel provides process management, interprocess communication, and low-level device management facilities. All other operating system services are implemented as (collections) of processes outside the kernel. A host kernel may be implemented at a *base level* (as on the SUN workstation) or a *guest level* (as under 4.2BSD).

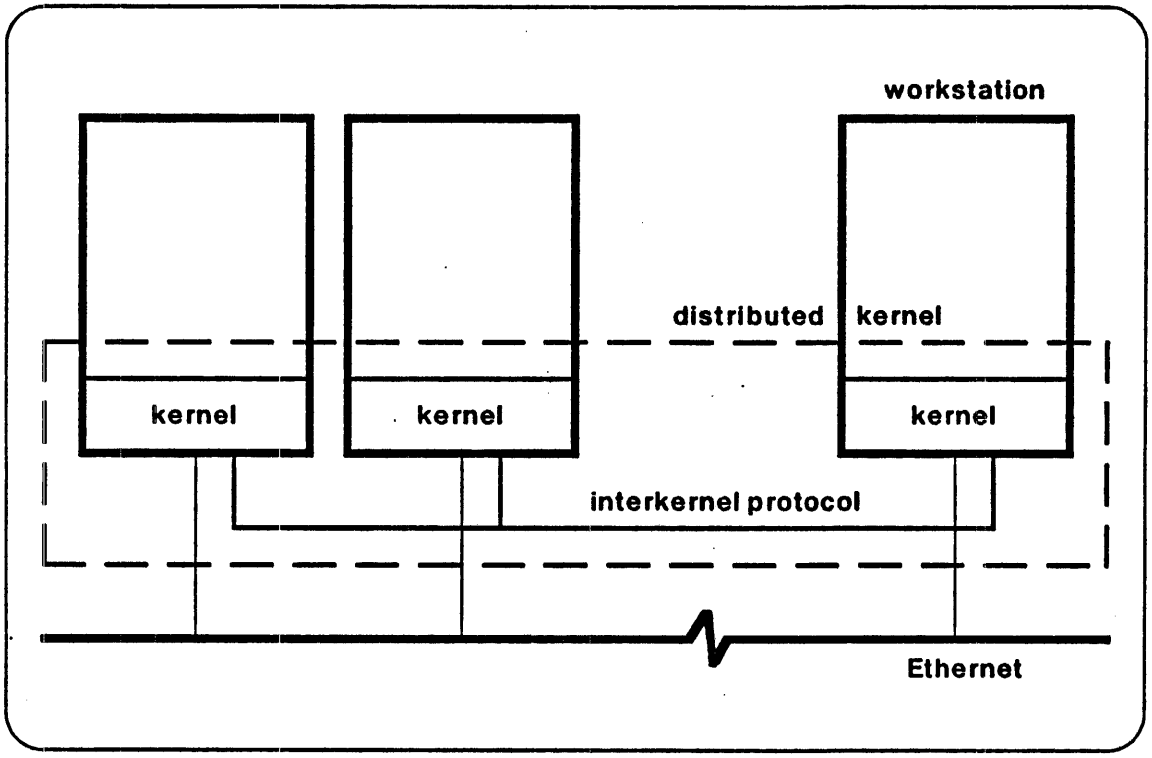


Figure 1-2: The distributed V kernel.

The host kernels are integrated via a low-overhead *inter-kernel* protocol (IKP) that supports transparent interprocess communication between machines. IKP is a reliable request-response protocol, intermediate in complexity between conventional datagram and virtual circuit protocols.

1.3.2. Servers

Servers include:

virtual graphics terminal server
: Provides all terminal management functions, including VT-100 emulation and 2-D graphics. One per workstation.

internet server
: Provides network and transport level support for traditional network architectures, namely, ARPA Internet and Xerox PUP. Higher-level protocols, such as TELNET, are provided as separate packages that interface to the internet server.

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

1.4. The Application Model

In general, it is just as easy to write applications to run under the V-System as it is to write applications to run under any traditional operating system, such as UNIX. A standard program environment is defined, the principal instance of which is the C program library. The C library provides runtime support for standard C and UNIX-like library functions, including both byte-stream and block-I/O facilities (see Figure 1-3). In effect, these libraries can be used to "hide" the underlying V-System kernel calls, thus facilitating the porting of existing C programs.

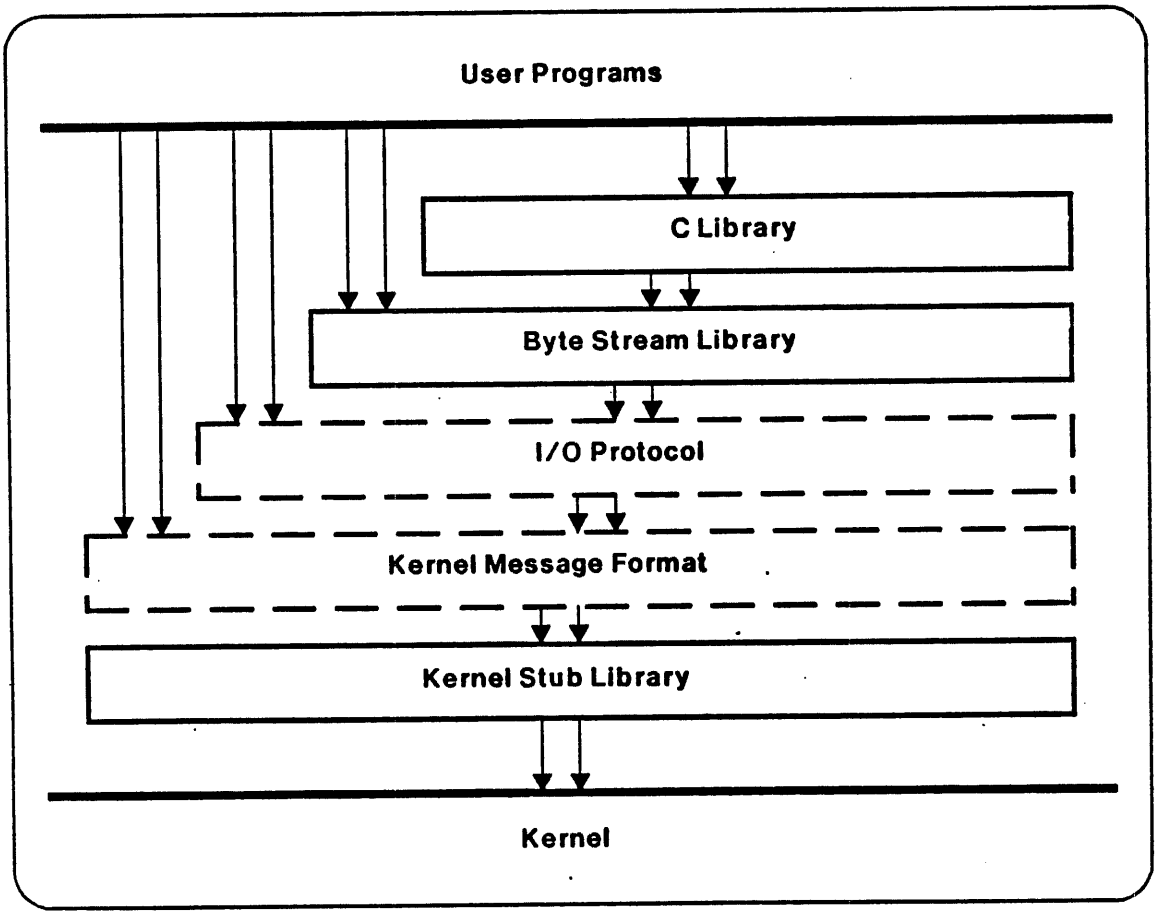


Figure 1-3: Client interfaces to the V-System

On the other hand, an application programmer may choose to take advantage of the enhanced facilities provided by the V-System. These facilities fall in two major categories: user interaction and concurrent programming. Additional advantage accrues from the fact that applications may be distributed across multiple machines.

1.4.1. User Interaction

With respect to user interaction, the V-System provides two principal enhancements over traditional UNIX-like systems. First, a program may manipulate multiple virtual terminals (windows) simultaneously. Second, an application may employ *structured* graphics. Specifically, a graphical object can be defined in terms of other objects, which can in turn be defined in terms of yet other objects. Thus, the VGTS supports *structured display files* rather than the more common *segmented display files*. The resulting *virtual graphics terminal protocol* (VGTP) is a high-level object-oriented protocol that minimizes both the frequency of communication between application and VGTS and the amount of data transmitted at any one time.

1.4.2. Concurrent Programming

Using the distributed 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 also provides process groups and group interprocess communication. Each process can create, join, and leave groups dynamically, and can belong to many groups simultaneously. A message sent to a group is delivered reliably to the first group member to reply, and unreliably to the rest. Replies subsequent to the first may be received (unreliably) by the sender, or ignored, at its option.

Process scheduling is strictly priority-based. The effective priority of a process is the sum of its *process priority* and its *team priority*. Team priorities are dynamically varied by the team server to provide time-slicing.

1.4.3. Classes of Applications

From the previous discussion it should be apparent that applications may run local to the user's workstation or on any other host accessible via the various network protocols. Ultimately, all applications must communicate with the user via the virtual graphics terminal server (VGTS) resident on the user's workstation. The application interface to the VGTS is referred to as the virtual graphics terminal protocol (VGTP).

The VGTP is constant over all applications. However, some applications have no knowledge of the VGTP and some applications are running on machines that do not support the interprocess communication mechanisms underlying the VGTP. The following situations arise (see Figure 1-4, in which each inter-machine arc is labeled with an example *(presentation protocol, transport protocol)* pair):

- Application ***A*** runs on the workstation and communicates via the VGTP. Current examples include text editors, document illustrators, and design aids, many of which are documented here.
- Application ***B*** runs on a machine that supports V kernel services, specifically, network-transparent

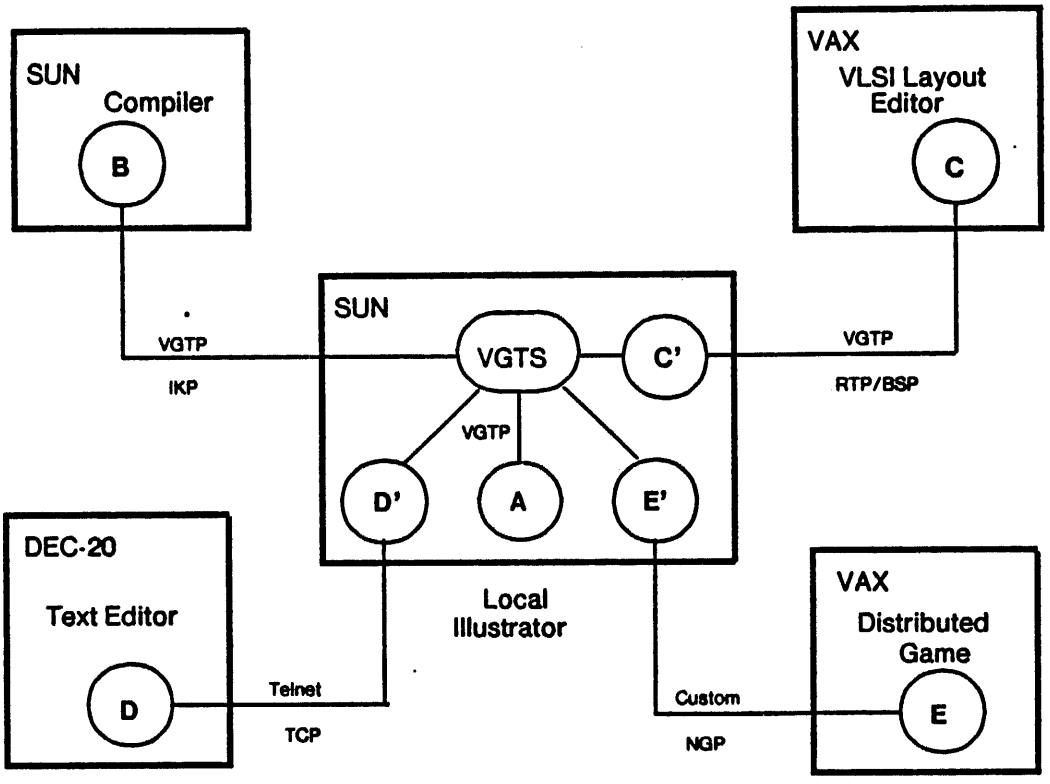


Figure 1-4: Some possible applications.

interprocess communication via IKP. *B* communicates with the VGTS via the VGTP, as in the case of a application *A*.

- Application *C* runs on a machine that does not support IKP, but does support a traditional network architecture such as the Internet protocol family. In addition, a VGTP interface package is available that encapsulates the VGTP within the appropriate transport protocol. Similarly, a local *agent* for the application, *C'*, is created on the workstation to decapsulate the VGTP. Thus, the application may still be written in terms of the VGTP and neither it nor the VGTS have any knowledge that the other is remote. Our VLSI layout editor, for example, can be run in this fashion under VAX/UNIX.
- Application *D* has no knowledge of the VGTS or the VGTP; it wishes to regard the workstation as just another terminal. The local agent, *D'*, is "user TELNET" and performs the appropriate translations between TELNET and VGTP. Any pre-existing application that runs on a remote host falls into this class.
- Application *E* is distributed between the workstation and one or more other machines. The local agent, *E'*, is responsible for representing the multitude to the VGTS. It must perform the appropriate set of protocol conversions indicated above. In addition, it may wish to perform application-specific functions, such as caching. In that case, the protocol used to communicate with the remote applications may require more than simple transport service. The Amaze game documented herein is an example of such an application.

1.5. Outline

The remainder of this manual consists of four parts:

Part 1 **Using V:** describes the user interface and available application programs.

Part 2 **V Programming:** defines the V-System program environment in terms of the existing C program library.

Part 3 **V Servers:** defines the standard message formats, request and reply types, and protocols; presents the various server-specific protocols; and gives some implementation details.

Part 4 **Appendices:** a V-System bibliography, notes on programming style, installation notes, and a list of where the various library functions are defined.

Part I:
Using V

— 2 —
User Interface Overview

This chapter presents an overview of what it is like for the user to interact with the V-System. Details of terminal emulation or graphics support are not discussed, since that is best described by the programmer's interface in Chapter 29. Rather, the basic stylistic conventions are presented, including an overview of how applications' actions are manifested to the user. Also included is a discussion of the the basic architecture of the user interface, in the hope that it will enable the user to better understand the style of interaction and the facilities available to him. The user who is "in a hurry" to get started may skip this discussion, at least on first reading, and begin with Section 2.2. The following chapter discusses command interpretation in some detail.

2.1. The User Interface Architecture

In a typical operating system, the user is presented with the illusion of interacting with a single, unified front end, often referred to as an "executive" or "shell". However, in contemporary workstation-based systems, this front end actually provides three basic levels of interaction:

1. **device I/O**: Manipulation of input devices and generation of output on output devices.
2. **command interpretation**: Command (or argument) specification and response handling, and invocation of applications.
3. **window management**: Management of multiple simultaneous applications (in separate "windows").

Rather than combine these three levels of function in one module, the V-System distinguishes three separate software components — respectively:

1. the *workstation agent*,
2. the *executive*, and
3. the *workstation manager*.

This separation was inspired by a desire to be able to configure each component independent of the others. While this release of the V-System does not reflect the *ideal* realization of this separation, it nevertheless fits the basic framework.

Warning: The workstation agent was originally referred to as the *terminal agent*. The two terms are used interchangeably.

2.1.1. Workstation Agents

The workstation agent provides the lowest-level interface between the hardware and the rest of the system. One of its principal functions is to hide any idiosyncrasies of that hardware — through a virtual terminal interface.

2.1.1.1. Virtual Terminals

Rather than dealing with the "raw" hardware, applications interact with a *virtual terminal*. They request input from a virtual keyboard or mouse, for example, and write output to a virtual store. Depending on the "class" of real terminal (workstation) being emulated, the characteristics of the virtual input and output devices may vary widely. In the simplest implementation, each workstation agent emulates exactly one class of real terminal; emulating a different terminal requires a new workstation agent. More sophisticated workstation agents emulate multiple classes of terminals simultaneously. Note that the number of *classes of terminals* emulated is independent of the number of *virtual terminals* being emulated at any one time.

Historically, the most common class of terminal emulated has been the page-mode (character) terminal — exemplified by the DEC VT-100. Even in this case, the workstation agent can be thought of as emulating different types of terminals, corresponding to the various input and output modes provided by a VT-100 — character-at-a-time versus block transmission, local editing facilities, and the like. In general, the workstation agent, through its virtual terminals, provides a set of facilities that might be referred to as "cooked I/O" — ranging from character echoing to line-editing to page-editing to graphics-editing. These facilities are enabled and disabled on a virtual terminal by virtual terminal basis.

True to its name, a virtual terminal need have no physical, real-world manifestation. In particular, it is possible to write output to a virtual terminal without seeing that output displayed on the screen. Hence, any application may run and change the store of any virtual terminal at any time.

While it is common for multiple applications to be generating output simultaneously, it has historically (if erroneously) been thought less desirable to permit the user to direct input to multiple applications simultaneously. True to this historical bias, the V-System currently restricts input to one application at a time. We refer to the application and associated virtual terminals as being "selected" (for input). Selection for input has no effect on the underlying application's ability to generate output. As with output, however, it *is* possible to generate keyboard input for a virtual terminal without having the virtual terminal mapped to the screen; users should be wary of the possible consequences!

2.1.1.2. Views

In order for the user to actually see the output from or generate graphical input to a virtual terminal, the virtual terminal must be mapped to the screen through a *view*. A view defines the portion of the virtual terminal's store that should be displayed, the area on the screen in which it should be displayed, and the transformation that should be applied when mapping the store to the screen. Using traditional graphics terminology, the store is referred to as the *display file*, the portion of the store is a *window*, the area of the screen is a *viewport*, and the transformation is a *viewing transformation*.[2] Viewports are invariably rectangular, although there is no conceptual reason for this to be the case.

Typically, an application will create one view of a virtual terminal at the same time it creates the virtual terminal. Nevertheless, views are maintained as entities distinct from virtual terminals because, in general, each virtual terminal may have more than one view associated with it. When using the VGTS, for example, the same picture, maintained as one entity by the program, may appear in two separate viewports on the screen, possibly with different viewing transformations. That is, a second view may look upon a different portion of the virtual terminal's store from the first, or at a different magnification.

> Note: Because a view is the physical manifestation of a virtual terminal on the display screen, we will tend to use the term "view" rather than "virtual terminal" when discussing screen management issues. Where necessary to be even more specific, we will use the term "viewport".

So, a virtual terminal may be associated with more than one view. On the other hand, because the virtual terminal is independent of its physical manifestation, there need be no views associated with it. Destruction of all views does not in any way affect the virtual terminal, though it will make it rather difficult for the user to see what is going on.

One common policy is that views are the domain of the user. A program that creates a virtual terminal should create a view of it, so that the user knows that it exists, but after that, in the ordinary course of things, the program should leave the view alone. The program should not depend on the continue existence of that view, nor need it be aware of any other views of the virtual terminal that the user chooses to create. Let the user decide where on the screen he wants views to be, and how big, and with what viewing transformations. That is what the workstation manager is for.

---

[2] Unfortunately, traditional "window system" terminology tends to use the word "window" to mean any or all of window (as just defined), viewport, view, or virtual terminal.

2.1.1.3. V-System Agents

The V-System currently supports two workstation agents, the *simple terminal server* (STS) and the *virtual graphics terminal server* (VGTS). The STS provides basic text terminal emulation by making the workstation appear as a single, traditional, page-mode terminal — compatible with ANSI standard X3.64.[3] Character echoing and line-editing are optional. The STS is used principally to interface to ASCII terminals, but it can also be used over remote terminal connections and as the interface to the normal workstation keyboard and display.

The VGTS provides considerably greater functionality, including support of what is commonly referred to as a *window system* (more on this later). Any "window" may emulate the same type of terminal provided by the STS. Alternatively, a window may emulate a (structured) graphics terminal that provides roughly the facilities available in the ISO standard Graphical Kernel System, together with rudimentary modeling facilities in the form of structured display files. Thus, the VGTS provides simultaneous support for two very different types of real terminal. Each virtual terminal may have any number of views associated with it. Any number of views of any number of virtual terminals can be mapped to the screen at the same time. Applications are unaware of the number of views or what is being displayed in them, except insofar as graphical input events return the appropriate world coordinates. The VGTS is used when it is desired to make the best use of devices typically found on contemporary workstations — such as bit-mapped displays, encoded keyboards, and mice.

While the abstractions for keyboard and mouse are common across (existing) virtual terminals, the abstractions for the store are quite different, as discussed in Chapter 29. When necessary to distinguish the two classes of virtual terminals currently supported, we will refer to the type of virtual terminal that emulates an ANSI standard terminal as an *ANSI virtual terminal* (AVT) and the type of virtual terminal that emulates a structured graphics terminal as a *structured graphics virtual terminal* (SGVT).

> ***Warning:*** The "store" of an AVT is referred to as a *pad*. Unfortunately, that term has been most frequently (and erroneously) used as a placeholder for the complete AVT abstraction. While this manual attempts to use each term where it is appropriate, the code uses "pad" almost exclusively. Consequently, many of the routines described herein also refer to "pad". Similarly, the term "virtual graphics terminal" (or VGT) has been used almost exclusively in the code, rather than the term "structured graphics virtual terminal". In both cases, we trust the reader will be able to make the appropriate semantic substitutions.

The bulk of the discussion to follow assumes use of the VGTS.

2.1.2. Workstation Managers

The workstation manager permits the user to control multiple simultaneous executives — with accompanying applications. Through it executives are created and destroyed, programs are interrupted and killed, and both virtual terminal and views are manipulated. With respect to the last, in particular, the workstation manager is the module that enforces the *constraints* on view management that the user desires. For example, it enforces the precise position and front-to-back "ordering" of viewports.

It is crucial to appreciate the distinction between workstation managers and workstation agents. The manager exerts "control" over the "facilities" provided by the agent, while at the same time using those facilities to interact with the user. Different users may want different styles of control. For example, one user may prefer to specify all views manually, whereas another user may prefer the system to determine the "best" view automatically; one user may prefer his viewports to be tiled, whereas another may prefer them to overlap. These styles are independent of the basic facilities provided by the workstation agent.

In principle, it should be possible to define the ideal workstation manager, independent of all workstation agents. But, just as workstation agents are limited in practice by the classes of real terminals they support, workstation managers are limited in practice by the available classes of workstation agents. For example, the VGTS comes with a large and powerful workstation agent, called the *view manager*, which is accessed via

---

[3] The most widespread example of a terminal adhering to this standard is the DEC VT-100.

popup menus. Many of its commands require the user to select or position viewports on the screen. Such interaction works best in an environment with a mouse, for example, yet some workstation agents may not provide an efficient emulation of a mouse. The STS, on the other hand, has only a trivial vestige of a workstation manager. Its primary function is to guarantee the existence of one executive running on the terminal at all times.

2.1.3. Executives

Workstation agents and managers provide the basic facilities by which the user interacts with the workstation. But little has been said about how the user actually specifies commands and applications. In fact, these functions are provided by executives (or shells).

Some systems permit only one executive, often running only one application at a time, but sometimes capable of running multiple applications at a time — one in the "foreground" and the rest in the "background", for example. Under the STS, the V-System provides one executive (of the latter variety). Under the VGTS, multiple executives are supported simultaneously; the view manager is responsible for creating (and destroying) them.

2.1.4. Summary

We have outlined the basic concepts underlying the user interface to the V-System. The rest of this chapter discusses the more practical details of how the user actually interacts with the system via its workstation agents and workstation managers. Interaction with executives is discussed in the next chapter.

2.2. Getting Started

When you come up to an idle 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 something other than V, is dead, powered down, or the like, it will be necessary to reboot it, as described in the following paragraphs.

2.2.1. Booting the Workstation

As previously noted, the V-System runs on a variety of workstations. Booting procedures vary depending on the manufacturer and on the model. Section 16.1 describes in detail how to boot all of the workstation configurations supported with this release. The following is an overview that should cover most situations.

2.2.1.1. VaxStations

When in the PROM monitor VaxStations display a >>> prompt. If the workstation is not in this state, press the **halt** button twice (once to put it in, once to bring it out). If this doesn't halt the machine, press **reset**. These buttons are on the front panel of the VaxStation CPU. Once at the PROM prompt, the command **b xqa0** will cause the workstation to boot over the ethernet.

2.2.1.2. SMI Workstations

An SMI workstation in a random state can be reset to the PROM monitor by holding down the key in the upper left hand corner of the keyboard, and hitting the "^" key. (The key in the upper left hand corner may be labeled either L1, SET-UP, or ERASE EOF depending on the exact model.) There is no reset button on SMI workstations, so a very serious crash can make it necessary to power-cycle the workstation.

Once you have gotten into the PROM monitor, the next thing to do is to type the **k2** command, which simulates a power-up reset. (Shortcuts are sometimes possible, but **k2** is the safest route.) After the power-up memory test is completed, the workstation will try to boot its default program. Most Sun workstations at Stanford are configured to boot the V-System with VGTS. The bootstrap program first loads the workstation's configuration file (see Chapter 19) to find out what its defaults are. It next prints "V-System" or "xV-System" to indicate whether the production or experimental version of V is being loaded, then prints the filenames of the kernel and initial team as it loads them.

If you notice the workstation is not booting what you want it to, you can interrupt the autoboot with the reset key sequence described above, then type in the exact boot command you want. Most of the time, one of the following simple commands will do the job:

b Boots the default program.

b V Boots the production version of the V-System, with the VGTS.

b xV Boots the experimental version of the V-System, with the VGTS.

General boot commands and the V bootstrap loader are described fully in section 16.

> Note: Most SMI workstations at Stanford use the Sun-2 processor board, but a few Sun-1s have not been upgraded. The above description is correct for SMI workstations with Sun-1 processors as well as Sun-2s, but the details of the general boot command and other PROM monitor commands vary. The V kernel tickles a bug in the current Sun-3 PROMs. Rebooting a Sun-3 usually requires power cycling the workstation.

2.2.1.3. Cadlinc Workstations

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 Cadlinc, either **k1** or **k2** will simulate a powerup reset. You may need to type the command twice for it to take effect. The Cadlinc PROM monitor uses **n** in place of **b** in the simple boot commands above. See section 16 for a full description of Cadlinc boot commands.

All Cadlinc workstations use a version of the Sun-1 processor board.

2.2.1.4. Rack-Mount Suns

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. Some rack-mount Suns at Stanford contain Sun-1 processors; others use the Sun-2. Some boot using the **n** command; others use **b**. In most cases, the boot commands listed above will work, and the STS will automatically be loaded in place of the VGTS, but in general it is best to check with a wizard before rebooting a rack-mount Sun.

2.2.2. An Overview of Subsequent Interaction

> Note: This section is somewhat redundant with Section 2.1 since it is assumed that many people will read it without reading that section!

Once the user has booted 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 similar to the UNIX C-shell. 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 virtual terminals as discussed above.

When the user wishes to initiate a new application concurrent with existing applications, he must first create

a new executive. To do so, the user communicates with the view manager. The executive serves as a command interpreter from which the desired application may be initiated. The user can create a new executive, with associated virtual terminal, at any time, asynchronous to any existing activities. When a particular application requires additional virtual terminals, it is free to create them. These virtual terminals will be deallocated when the application terminates.

A virtual terminal is made "visible" by mapping it to the screen. Each such mapping is termed a *view*. When an application creates a new virtual terminal, the application may specify where on the screen the view should appear or the application may request that the user should specify the view interactively. Thereafter, the user may create as many additional views as he wishes. To some extent, he may manipulate views of the same virtual terminal independent of all other views of that virtual terminal, for example, pan or zoom one view independent of all other views of the same virtual terminals. All such virtual terminal management is performed via the view manager.

2.3. VGTS Conventions

When using the VGTS, views appear as white overlapping rectangles on the screen, with a black border and a "banner" across the top edge. The banner contains the following information:

- a virtual terminal identifier
- a view identifier
- the "name" of the associated application (if any)

Every view of every ANSI virtual terminal displays the text input cursor as a small black box. An additional cursor is associated with the mouse; it may change shape depending on what graphical input event is expected, for example.

2.3.1. Selecting for Input

As discussed above, at most one application is selected for input at one time. At most one of its virtual terminals (usually an AVT) may be selected for keyboard input. Any subset of its virtual terminals may be selected simultaneously for mouse input; the application may selectively enable and disable mouse input for each virtual terminal.

All views of all virtual terminals selected for input (of any kind) display a "blackened" banner — white text on black background. In addition, the virtual terminal selected for keyboard input will display a flashing black box at the cursor. Unselected virtual terminals have whitened banners — black text on white background.

2.3.2. Using the Mouse

There are a few conventions for using the mouse with the VGTS. First, we assume a three-button mouse. In the discussion that follows, the buttons are labeled simply "left", "middle", and "right".

We assume that there is always a cursor associated with the mouse. Wherever the term "cursor" is used without qualification, we are probably talking about the mouse cursor — rather than the keyboard cursor. Also, we will often use the phrase "pointing to" to mean "the mouse cursor is in" or "the mouse cursor is pointing to".

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 virtual terminal).

A "transition" consists of a press *or* a release of *any* combination of buttons. A transition event is sent to the program associated with the view in which the event occurred. Since it typically is acted on immediately, the VGTS provides no feedback and the universal abort feature described above does not work.

2.3.3. The Screen Saver

The VGTS will automatically disable (blacken) the screen if no keyboard or mouse events have occurred within the preceding 10 minutes. This helps protect the screen's phosphor. The screen is reenabled by a subsequent keyboard or mouse event. Note that other VGTS events will not reenable the screen, so the screen save will work even if a program such as **mon** (which periodically updates the screen) is being run.

2.4. Workstation Management

Almost all workstation management functions are accessible via a set of menus managed by the *view manager*. The only exception is selection of an application for input. Many view manager functions are also available via "accelerators" — "appropriate" mouse clicks in "appropriate" places. Some functions are triggered by program-generated requests. We discuss each situation in turn.

2.4.1. Selection for Input

Clicking the left or middle button of the mouse in a view of a non-selected virtual terminal will cause the associated application to be selected for input. The view will be brought to the top.

2.4.2. Using the View Manager Menus

The view manager menus can always be invoked by moving the cursor to the grey background area or to any view not selected for input (except in the banner area) and pressing the right button. The following commands are available from the view manager menus. The commands are presented in alphabetical order, followed by a summary of what each (sub-)menu contains.

2.4.2.1. View Manager Menu Commands

Center Window — Change the "window" associated with a view — without changing the position of the associated viewport. Click any button at the position in the viewport that you want to become the center of the viewport. Doing this to AVT's is almost always a mistake.

Create Executive — Create a new executive, with associated AVT. The cursor changes to the word "Exec". When you press a button, an outline of the new AVT will appear, and 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.

> Note: Comes in two flavors, for two different default sizes of AVT (see Set Alternate Exec Size below).

Create View — Create another view of an existing virtual terminal. The cursor changes to the word "View". 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 virtual terminal that you would like to see with the left or middle button, or hit the right button and select the virtual terminal from the menu. Normally only used with SGVT's.

Delete Executive — Delete an executive. Click any button in any view associated with the executive.

Delete View — Delete a view. Click any button in the associated viewport.

Warning: If you delete the last view of a virtual terminal, it does *not* destroy the virtual terminal or the process associated with it. You can still create views of the virtual terminal by using the right button menu in the Create View command.

Exec Control — Select the submenu for executive and program control functions. A shortcut to the Exec Control menu is obtained by pressing both the middle and right buttons while the cursor is in the gray background or in a view not selected for mouse input.

Expansion Depth — Set the "expansion depth" for a SGVT. For hierarchically defined graphical symbols, this determines how much (how deep) of the hierarchy will be displayed. If this causes some graphical item not to be displayed, its bounding box is displayed, possibly with a text name (if there is room). The default expansion depth is infinity, such that all levels will be expanded. Click any button in the view whose expansion depth you wish to set, then select the new expansion depth from the menu that pops up.

Graphics Commands — Select the submenu for graphics functions. A shortcut to the Graphics Commands menu is obtained by pressing both the left and right buttons while the cursor is in the gray background or in a view that is not selected for mouse input.

Interrupt Program — Interrupt a program, forcing it into the debugger. Click any button in any view associated with the program.

Kill Program — Kill a program. Click any button in any view associated with the program.

Make Bottom — Push a view to the bottom, potentially making visible other views. Click any button in the desired view. A shortcut to this function is obtained by pressing the right button while the cursor is in the banner of the desired view.

Make Top — Bring a view to the top, potentially obscuring other views. Click any button in the desired view. Does *not* select the associated virtual terminal for input. A shortcut to this function is obtained by pressing the left button while pointing to the banner of the desired viewport, which action action *does* select the virtual terminal for input.

Move Edges — Change the viewport associated with a view by moving one or more edges. Scaling is not provided, so this also changes the window (the portion of the object being viewed), but without moving the object relative to the screen. 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.

Move Edges + Object — Similar to Move Edges, but drags the underlying object around with the moved edge or corner.

Move Viewport — Change the viewport associated with a view by changing its position, but retaining its size. Press any button in the desired view. 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. A shortcut to this function is obtained by pressing the middle button while pointing to the banner of the desired view; the viewport outline will follow the cursor until the middle button is released.

Redraw — Erase and redraw the entire screen. Should be necessary only when low-level debugging information trashes the screen.

Reset State — Reset the state of an AVT. Click in any view of the AVT. This is equivalent to pushing the "reset" key on a VT-100 or most other page-mode terminals and is necessary only in extraordinary situations where the AVT appears to be "wedged".

Set Alternate Exec Size — Set the "alternate" size for executives. Type in the size to the VGTS window. Executives of the new size can then be created using the Exec Control submenu.

Toggle Grid - Toggle the background grid in a view of SGVT. 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.

Toggle Paged Output Mode
Enable or disable paged-output mode in a AVT. Click in any view of the AVT.

Zoom Invoke "zoom mode" in an SGVT. The cursor changes 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 an AVT returns from the view manager and selects that AVT 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 to a view not selected for mouse input.

2.4.2.2. Assignment of Commands to Menus

The top-level view manager menu contains the following commands:

Create View
Delete View
Exec Control
Graphics Commands
Make Bottom
Make Top
Move Viewport

The "Exec Control" sub-menu contains:

Create Executive (*two flavors!*)
Delete Executive
Interrupt Program
Kill Program
Reset State
Set Alternate Exec Size
Toggle Paged Output Mode

The "Graphics Commands" sub-menu contains:

Center Window
Expansion Depth
Move Edges
Move Edges + Object
Redraw
Toggle Grid
Zoom

2.4.3. Summary of Accelerators

The workstation management functions available though mouse clicks are listed in Table 2-1. See also section 2.7.

Mouse Buttons L M R	Where	Effect
x . .	In banner	Make Top and select
. x .	In banner	Move Viewport
. . x	In banner	Make Bottom
x . .	In non-selected viewport	Make Top and select
. x .	In non-selected viewport	Make Top and select
. . x	In gray or non-selected viewport	Top-level menu
. x x	In gray or non-selected viewport	Exec Control
x . x	In gray or non-selected viewport	Graphics Commands
x x .	In gray or non-selected viewport	Zoom
x x x	During any workstation management command	Abort

Table 2-1: Accelerators for workstation management functions.

2.4.4. Program-generated Requests

When a program requests the creation of a view, the VGTS enters the same interaction cycle as described for the Create View command above. However, since the virtual terminal will have been specified in the function call, you do not need to select the virtual terminal, and universal abort typically will not work.

When a program requests the creation of an AVT, the cursor will change to the word "Pad" (sorry about that). At this point, hold down any button, and an outline of the viewport that will be created will be tracked on the screen. Position the viewport where desired, and let go of the button.

2.5. Line Editing Facilities

Keyboard input can be edited with Emacs-style line-editing commands. More specifically, the commands listed below are available. CTRL-x means holding down 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. (The standard executive responds to such a line by printing "XXX".)

CTRL-h Delete the character before the cursor. Equivalent to the DEL key.

CTRL-i Insert an appropriate number of spaces, to simulate a TAB character. Equivalent to the TAB 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.

CR Return the line-edited text to the client. Even if struck in the middle of the "line", the entire line will be returned.

Printing characters are normally inserted at the cursor.

2.6. Paged Output Mode

When paged output mode is on, the workstation agent stops writing to an AVT when the AVT fills up with output. The workstation agent then displays the message "Type <space> for next page" in the banner and waits for the user to issue a command that unblocks the AVT.

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 Scroll forward *k* lines [1]

S Scroll forward to the last line

f Scroll forward *k* pages [1]

F Scroll forward to the last page

BS, DEL Erase the last character of the numeric argument

. Repeat the previous command

If the user types a character that 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 2.5) have their usual effect here.

2.7. Sending Mouse Events to Text-oriented Applications

Many applications exist that have not been written expressly for the V-System, but can be accessed via the text terminal emulation protocol. A few minor additions to this protocol permit applications to receive a number of mouse clicks as escape sequences that they may interpret as they wish. The precise escape sequences generated are given in Chapter 29.

A complete list of the events that generate escape sequences, and the use to which the UNIX EMACS text editor puts them, is given in Table 2-2. The actual escape sequences generated are given in Section 29.2.2.

Mouse Buttons L M R	Emacs Interpretation
x . .	Position the cursor at the position of the click.
x x .	Set the mark to the clicked position.
x . x	Delete from the mark to the clicked position.
. x x	Insert the kill buffer at the clicked position.

Table 2-2: Events that generate escape sequences.

2.8. Emulating the Mouse with the Keyboard

For the benefit of hardware configurations without a working mouse, the VGTS can interpret certain keyboard escape sequences as mouse input.

2.8.1. Workstation Management

The input virtual terminal can be changed by using CTRL-↑ (octal 036) followed by a single command character. The only command characters interpreted by the VGTS are 1-9 to select the given virtual terminal for input.

2.8.2. Graphics Events

The VGTS also interprets certain character sequences as mouse movements or button transitions. However, 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 detects the space between characters and passes 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 this one are 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, this type of 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.

Warning: These sequences only work on Sun-100's.

2.9. STS Conventions

The bulk of the discussion thus far has assumed the availability of the VGTS. However, there are occasions when users may have the interact with the STS instead. In that case, the user sees exactly one view of exactly one terminal associated with exactly one executive. That view occupies the entire screen. The user interacts with this executive exactly as he would with an executive running under the VGTS. Line-editing facilities and output paging are provided.

— 3 —
Using the V Executive

3.1. 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. The executive is available as a program and as a service provided by the exec server. Each executive usually runs in a virtual terminal provided by the workstation agent — either the STS or the VGTS. See the description of the STS in section 45 and the description of the Exec Control menu of the VGTS View Manager in section 2.4.2.

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, except when quoted (see section 3.6.6). 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. By default, it looks first for a V program in the current context (i.e., current working directory), then in the [bin] context. You can specify a different search path using the **PATH** environment variable (section 3.6.7). If the command is not found on your search path, the exec will try to execute it 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. The View Manager and STS provide mechanisms for stopping or interrupting a command running in the foreground. A program running in the background may be terminated using the **destroy** command (see chapter 4).

Other exec features are described in section 3.6.

3.2. Naming

A *context* in the V system is similar to the *directories* provided by other systems such as Unix. Each process (and thus each executive) has its own *current context*, i.e., current working directory. A filename is normally interpreted in the current context, unless it begins with a square bracket ('['). The square bracket flags the name as being either an absolute name, or a local alias.

In V, the first component of an absolute name generally specifies the *type* of object or service being named. The second component often specifies a particular server. For example,

```
[storage/pescadero]/etc/passwd
```

names the file **/etc/passwd** on the storage server **pescadero**. The closing square bracket is optional here. Most servers accept it as an alternative to the standard slash character ('/') used to separate name components.

Users can define their own local aliases for contexts, using the **define** and **undefine** commands. For example,

```
define g [storage/gregorio]/user/mann
```

defines **[g]** as an abbreviation for user Mann's home directory on the storage server Gregorio. The command

```
undefine g
```

removes this definition. The command

```
printdef
```

lists the local aliases that are currently defined. Several other standard aliases are automatically defined by the executive when you log in. These include

[home]	The logged-in user's home directory.
[sys]	The directory containing standard V-System files.
[bin]	The directory from which standard V-System commands are loaded. Normally the same as **[sys]bin**.
[V]	The directory containing standard production V-System files. The same as **[sys]** if you are running the production V-System.
[xV]	The directory containing standard experimental V-System files. The same as **[sys]** if you are running the experimental V-System.
[homex]	The "home" server used for program execution. Normally **[team/local]**, the local team server. See section 3.6.9.

When running with the VGTS, aliases defined in any exec created by the view manager are global to all such execs, since all the execs run on the same team. A program run under the exec inherits a copy of the exec's aliases. Later changes to the exec's aliases do not affect it.

3.2.1. 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. When an exec is created, its current context is set to the current value of [home].

The **pwd** command prints the absolute name of the current context.

3.3. Logging In and Out

3.3.1. Login

The **login** command is used to authenticate a user to the V-System. The command format is

login *flags username*

The optional flags are described below.

The login command prompts for a password. The password is not echoed when typed. An error message is printed if the user types an invalid name or password, or cannot be authenticated for some other reason.[4] If authentication is successful, the given user is registered with the exec server and team server as the primary user of this workstation. The exec then forces any guest programs running on the workstation to migrate elsewhere.

The exec next defines the local alias **[home]** to be the user's preferred home context, as recorded in the system authentication database, and undefines all other user-defined aliases. Thus, if user Mann's home context is **[storage/teton]/ds/mann**, after logging in, he can refer to the file

[4] The message "Server not responding" indicates that no authentication server could be contacted. The command **authserver &** will start one locally, after which it should be possible to log in.

/ds/mann/phone-numbers under the name **[home]phone-numbers**. In this case it would be possible to get at the "root" directory on Teton by using a **/** immediately following the alias, for example, **[home]/usr/V/misc/termcap**.

Next, the exec changes its current context to be the new value of **[home]**. Finally, the exec executes a command script from the file **[home].Vinit**, if such a file exists.

The login command's behavior can be modified by specifying one or more of the following flags on the command line:

-v	Verbose flag. Commands in .Vinit are echoed as they are executed.
-q	Quick flag. Prevents .Vinit from being executed.
-x	Exclusive flag. Normally, when a user is logged in to a workstation, that workstation is registered as being unavailable for remote execution of commands, but if some other workstation insists on requesting remote execution there, the team server will still grant the request. If the exclusive flag is given to the login program, the local team server is instructed to refuse all requests for remote execution.
-r	Remote flag. Register the workstation as being fully available for remote execution, as though no one were logged in.
-f	Finish flag. Allow guest programs currently executing on the workstation to finish. If this flag is not given, guest programs are forced to migrate to another workstation.

After a user has logged in to a workstation, all further execs created on that workstation using the View Manager run as that user. The **su** command can be used to authenticate individual execs as some other user. The command format is

su *username*

Like the **login** command, the **su** command prompts for a password, which is not echoed. However, it only authenticates the exec in which the command is typed. It does not affect the exec server's record of the primary logged-in user, or perform any of the other actions of **login**.

Execs created when no one is logged in to a workstation run as the *unknown user*. Most system servers place severe restrictions on what the unknown user is allowed to do.

3.3.2. Logout

Give the **logout** command when you are done using a workstation. The command format is

logout *username* ...

where the user names are optional. If no name is given, the user owning the exec in which the command was typed is logged out. If one or more user names are given, the given users are logged out. If the flag **-a** is given in place of the list of user names, all users with authenticated execs on the workstation are logged out. (The latter two options are restricted to the primary logged-in user.)

Logging out a user destroys all execs authenticated as that user, and hence all foreground programs run by that user. Do not log out if you want such programs to continue running.

3.4. Remote Program Execution on a Unix 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**, **man**, or **ps** 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.

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**, **more**, etc.) do not work in this mode. Such programs can only be run by logging in to the Unix machine, perhaps using the V telnet program to connect to it (see chapter 4).

The V servers do not provide execution of Unix commands to users who are not logged in to V or do not have a Unix account. If the executive tries to execute a Unix command for such a user, the V server returns an "No permission" error.

3.5. Remote Execution on V Hosts

A command can also be executed remotely by designating either a specific remote V host or any remote V host. A specific host can be specified either by the process id of its team server or by its string name (e.g. sun-mj416). (Syntax details are described in 3.6.9.) Remote execution of this type is transparent to the user in that I/O is still directed to the local host.

3.6. Facilities for Command Specification and Modification

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

3.6.1. Line Editing Facilities

Command lines can be edited as described in Section 2.5.

3.6.2. Pattern Expansion

A command argument that contains one or more asterisks (*) is considered a *pattern*, and is replaced by a list of existing filenames that match the pattern, unless it is quoted with "" or '' (section 3.6.6). The asterisk character matches any string of zero or more characters other than slash (/), right square bracket (]), or an initial period (.). Other characters in the pattern match themselves.

3.6.3. Command History References

The executive 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 /tmp/exec.c
```

with the cursor sitting at the end of the line. The user can then hit carriage return to re-execute 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.

3.6.4. 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.

3.6.5. 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.

3.6.6. Quoting Command Arguments

Sometimes it is desirable to include a space in a single argument to a command. To do this, put a pair of either single quotes (') or double quotes (") around the argument. An argument quoted by one of these may contain the other. Unmatched quotes are matched by the end of the line.

3.6.7. Environment Variables

The command

setenv *var value*

sets the environment variable *var* to the character-string value *value*. (The latter should be quoted if it contains spaces.) As with local aliases for context names, environment variables are shared among all execs created by the View Manager, and are inherited by programs run under any exec.

The command

unsetenv *var*

removes the definition of *var*, while the command

printenv *var*

prints its definition. The **printdef** command with no arguments prints all environment variables.

A command argument that begins with a dollar sign ('$') is replaced by the value of the rest of the argument interpreted as an environment variable, if such a variable is defined. Otherwise the argument is left unchanged.

When trying to execute a V command, the exec determines the search path to be used from the environment variable **PATH**, if it is set, as do all programs that use the standard V-System program execution library routines. The value of **PATH** should be a list of context names separated by spaces. The default path, used if **PATH** is not set, is "./ [bin]".

3.6.8. 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. In fact, concurrent commands continue to execute even if the user that initiated them logs out.

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

3.6.9. Execution of Commands on Another Host

Commands can be designated to execute on another host by including

@ *<host-designation>*

on the command line. Here *<host-designation>* can be one of three things:

- The hexadecimal process id of the host's team server. This must given in the form *0xpid*, i.e. as the characters *0x* followed by the hex process id.

- The string name of the host, e.g. **sun-mj430.**
- The string **any**, designating any suitable V host other than the local one. A *suitable* host is defined as a host on which no one is logged in, and whose unused memory and CPU time meet certain minimum requirements.

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).

Another way to specify that commands should be executed remotely is to use the **cx** command to change the exec's *current execution context*—that is, the server (and context) where commands are executed when a remote execution clause is not given. The command format is

```
cx [team/hostname]
```

Giving the **cx** command with no arguments resets the execution context to **[homex]**, the exec's "home" execution context, which is normally **[team/local]**. The command

```
pwx
```

prints the name of the current execution context.

3.7. Support for Heterogeneous Processors

The V-System currently runs on machines with two different types of processor: the Motorola 68000 family and the DEC Vax family. More processor types may be supported in the future. Thus, several versions of the same V program may exist, each compiled for a different processor.

Different versions of the same V program that appear in the same directory may be distinguished by a *machine-specific* file-name suffix. This suffix is "**.m68k**" for Motorola 68000-based machines (in particular, Sun workstations), and "**.vax**" for the Vax. Note, in particular, that the directory for installed V-System command binaries (at Stanford, **/usr/V/bin**) contains two such versions of almost every command. When searching for a command binary, the executive automatically searches for a file name both with and without an appropriate machine-specific suffix. Thus, it is not necessary for the user to enter a machine-specific suffix when typing a command to the executive. This is true even if the command is executed *remotely* (see section 3.6.9) on a host with a different processor type.

In light of the above, the executive's search path mechanism needs to be explained further. When a command is to be executed on a *specific* host (that is, one that is known in advance), then the executive looks down the search path, until it finds a version of the command that can be executed on this particular host. This is the usual case. When a command is to be executed on an *arbitrary* host ("**@ any**"), then a slightly different mechanism is used. In this case, the executive looks down the search path until it finds *any* version of this command. It then determines all versions of the command that exist in this location, and selects a suitable remote host that is able to execute one of these versions. In most circumstances the difference between these two mechanisms will not be noticeable.

— 4 —
Command Summary

4.1. Workstation Commands

The following briefly summarizes the currently available commands for V.

addcorr — Add correspondences from your V user identity to UNIX user accounts. Each V user can correspond to one account on each local UNIX machine that is running a V/UNIX server. The V superuser can add correspondences for other users as well as itself. See Chapter 43 for more information about user correspondences.

amaze — A multi-person distributed game. Does not (yet) run under the VGTS. See Chapter 5.

ar — Constructs library files ("archives"). See the UNIX manual for documentation.

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).

bitcompile — Converts human readable bitmap specifications into initialized C data structures. Usage: bitcompile [options] [file].

The file argument specifies the source, otherwise standard input is used. Output goes to standard output by default.

The following options are interpreted by bitcompile:

-DBIG_ENDIAN
: Order the bytes (and bits) of the bitmap for big endian machines. This is the default.

-DBLACK_IS_1
: Generate bits of 1 for black. This is the default.

-DBLACK_IS_0
: Generate bits of 0 for black.

-DCOLUMN_ORDER
: Generate bitmaps as columns of 16 bit words.

-DLITTLE_ENDIAN
: Order the bytes (and bits) of the bitmap for little endian machines.

-DNOHDR
: Do not place a header before each output bitmap.

-DROW_ORDER
: Generate bitmaps as rows of 16 bit words. This is the default.

-DSUN100FB
: Generate bitmaps for (the current implementation of) SUN 1 frame buffers. This is equivalent to specifying -DCOLUMN_ORDER.

-DSUN120FB
: Generate bitmaps for (the current implementation of) SUN 2 frame buffers. This is equivalent to specifying -DCOLUMN_ORDER.

-DVAX
: Generate bitmaps for (the current implementation of) MicroVax frame buffers. This is equivalent to specifying -DLITTLE_ENDIAN -DBLACK_IS_0.

-o *file*
: Send the output to *file*.

bits
: A program for manipulating (*e.g.* hand-editing) bitmaps and fonts. See Chapter 7.

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 user's name is fetched from the system authentication database.

-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. For printing ordinary text files. The default unless the filename ends in ".dvi" or ".press".

  1
  : DVI. For printing TeX output. The default if the filename ends in ".dvi".

  2
  : Press. Not implemented. The default if the file name ends in ".press".

  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).

-W
: File is in the TeX character set instead of standard ASCII. All characters with the high-order (8th) bit set are treated as printing characters after the high-order bit is stripped. This feature permits access to the printing characters "hidden under" the ASCII codes for carriage return, linefeed, etc. (Line printer mode only).

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

build
: Automatically run programs depending on which files are out-of-date. See Chapter 8. *build* is an extension of the Unix **make** program.

cat
: File concatenation program. Copies each named file to the standard output. A hyphen ("-") represents standard input. If no arguments are given, standard input is assumed. There are no flags.

cc68
: Compiles **C** source programs for running on the m68000 processors. See the Unix **man** page.

cd
: Change directory: change the current context. Built in to the exec.

checkers
: Lets you play a game of checkers against the workstation. This is also a good demonstration of the VGTS's graphics capabilities. See Chapter 6.

checkexecs
: Kill off any exec whose standard input server or standard output server has died.

ci — Part of the Revision Control System. Described by a UNIX manual page.

clear — Clears the AVT.

clock — An analog clock. Understands two flags: -s (with second hand) and -t (without text, in case you want to zoom the clock).

co — Part of the Revision Control System. Described by a UNIX manual page.

cp — If two filenames are given, **cp** copies the first file specified to the second file (or to **stdout** if the second filename is "-"). If more than two filenames are given, or the **-d** flag is given, the last argument is assumed to be a directory name, **cp** copies the first *n*-1 files specified into that directory, forming the name of each new file by appending the last component of the corresponding old file to the directory name. Note that this behavior is not quite identical to that of the Unix **cp** program; the V **cp** program does not attempt to determine whether the last argument "is" a directory.

cpdir — Invoked as:

cpdir *flags fromdir todir*

copies all files in the *fromdir* directory to *todir*. *todir* must previously exist. The **-r** flag specifies that the copy should be recursive: the entire subtree rooted at *fromdir* is copied. The **-y** flag suppresses copying files if a destination file of the corresponding name already exists and is *younger* than the source file, i.e., has a more recent modified date. The **-v** flag causes a 'verbose' message to be printed each time a file is copied.

cx — Changes your current *execution context*—see section 3.6.9. This command is built into the exec.

dale — Distributed version of YALE (Yet Another Layout Editor). This is a VLSI layout editor that provides graphics editing of SILT chip descriptions. YALE is documented in a Stanford CSL Technical Report.

date — Prints the date as maintained by the local workstation kernel, and as maintained by first responding network time server. The kernel-maintained time on a workstation is set from a time server when the workstation is booted. The command **date -s** resets the kernel-maintained time from a network time server.

debug — The V debugger. See Chapter 9.

debugvgts — Allows the user to turn on/off debugging output from the VGTS. See section 46.5 for further details.

define — Defines a local name (alias) for a context. The first argument is the new name to be defined. The last argument is a context name, specifying the value to be given to the new name. Built in to the exec.

delcorr — Delete correspondences from your V user identity to UNIX user accounts. Each V user can correspond to one account on each local UNIX machine that is running a V/UNIX server. The V superuser can delete correspondences for other users as well as itself. See Chapter 43 for more information about user correspondences.

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.

destroy — Takes the name of a team (or any suffix) as an argument, and destroys the root process of 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. The **-i** flag causes the process to be interrupted (with **ForceException**) instead of destroyed.

diff — This command has the same syntax and semantics as under Unix 4.2BSD, with the

addition of the **-n** option of the Revision Control System's **rdiff** program.

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. If the **-v** option is given, then each command line is typed out at the time that it is executed.

domake — A synonym for **doseq** (described below).

dopar — A program similar to **doseq**, except that it allows the executions of its command arguments to take place in parallel on different hosts. For each context, the program prompts for the name of a host on which to execute the command, and pops up an AVT that acts as the command's standard input and output. If "**any**" is entered as the host name, then an *arbitrary* remote host will be selected. The local host can be selected by entering "**0**".

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 buildfiles.

draw — An interactive drawing program that runs under the VGTS. See Chapter 10.

echo — Echos its arguments. The **-n** flag suppresses the newline at the end of the output.

fexecute — Force a command to be executed on the server providing the current context, as described in section 3.4.

freemem — Displays a bar graph showing the current percentage of free memory, and the percentage before the last change.

gftodvi — For producing magnified proofs of fonts created by **mf** (*q.v.*).

gftype — Produces terminal-readable output from a **gf** font file. See **mf**.

grep — This command has the same syntax and semantics as under Unix 4.2BSD.

hack — A **rogue**-like game. See chapter 11.

ident — Part of the Revision Control System. Described by a UNIX manual page.

instances — A diagnostic program that prints out information about any file instances (see chapter 33) that are being maintained by the server that is providing your current context. At present this will work only if your current context is being provided by a Unix server (see chapter 43).

internetserver — A version of the Internet Server. See chapter 39.

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.

killprog — Kill the program, if any, running in the specified executive.

listdir — Lists the names defined in one or more context directories. If the -l flag is given, *listdir* prints one line of information about each object. The output is sorted by default; the -n flag specifies "no sorting." If no argument is given, the current context is assumed.

listdesc — Prints one line of information about each named object, extracted from its object descriptor. If no argument is given, the current context is assumed.

login — Log in to the V system. See section 3.3.1.

logout — Log out of the V system. See section 3.3.2.

mail — The UC Berkeley Unix 4.2 *mail* program, ported to the V-system. Note that this program is merely a front end (for composing, reading and editing mail). In order to use this program, your current context must be on a Unix system. The program's commands are

the same as in the Unix version, with the following exceptions:

1. The ~e command invokes **ved** by default.
2. The new command "Quit" (or "Q" for short) behaves just like the "quit" (or "q") command, except that if new mail has arrived, you will be immediately put back in the "mail" program so that you can read it.
3. The new command "Update" (or "U") is like "Quit", except that you will be put back in "mail" even if no new mail has arrived. (This command is equivalent to exiting the program, and then re-running it.)
4. "**mail -c** *n*" will cause the program to check your mail file, every "*n*" minutes, for the arrival of new mail. If new mail is found, the "Update" command described above will be run automatically (unless another command is in progress at the time).

memserver — A server that allows unused main memory to be used for temporary file storage. See section 40 for more details.

mf — Metafont-84 is Donald Knuth's language for compiling algebraic shape descriptions into bitmap images and fonts (m68k only). See Knuth's book *The MetafontBook* (Addison-Wesley, 1986). Also note the programs **gftodvi** and **gftype**.

migrateprog — Migrate a guest program from the local machine to another machine.. Invoked as

```
migrateprog [-p] [-h <host-name>] [<progs>]
```

The unit of migration is the logical host, *not* individual teams. All remotely executed programs are created in their own logical host, whereas all locally executed programs are run in the system logical host, which never migrates. Thus only remotely executed "guest" programs will ever migrate. Currently there is no way to create a locally invoked program in a separate logical host other than to invoke it remotely from another machine.

If no arguments are specified then all guest programs on a machine are migrated to "lightly loaded" machines, where lightly loaded is defined in the same sense as **any** is defined when initially executing a program remotely. The -p flag specifies that information about what programs are being migrated where should be printed out. The -h flag allows specification of a particular machine to migrate to. The machine to migrate to may be specified by either the hex pid of its team server (in the form 0x(pid)) or by giving its official string name (e.g. sun-mj416). If specific programs are specified then only those programs are migrated. Programs may be specified by the hex pid of one of their processes or by their full invocation name, as stored by the team server.

mon — This program monitors resource utilization of the workstation and presents it graphically. The -d flag specifies a vertical display ("down") instead of horizontal. The default display shows percentage used of memory and processor, and the number of incoming Ethernet packets per second. The flags -m, -p, and -e limit the display to only those specified. The -f flag violates the user interface standards by putting the display in the upper right corner of the workstation instead of requiring the user to position it with the mouse.

name — Prints the login name of the user under whose account the command is running.

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 AVT when the AVT 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 AVT. The user interface for paged output mode is described in section 2.6.

password — Use this program to change your V password, home directory, or personal name. The V superuser can also use it to create new accounts or modify other users' accounts. Uses the **fields** package (section 21). To modify the displayed *Name*, *Fname*, or *Home*, click on the old value with the mouse, use the normal line-editor commands to change the value, then hit return. To make the change take effect, click on *Modify* and supply your old and new passwords.

Other functions: click on *find* to find another user's authentication database entry. Click *add* to add a new user account (a unique user number is selected for you). Click *delete* to delete the displayed account. Click *exit* to leave the program.

pc68 — Compiles **Pascal** source programs for running on the m68000 processors. See the Unix **man** page. The flags are similar to those of cc68.

pwd — Prints the absolute name of the current working directory. Built in to the exec.

pwx — Prints the (absolute) name of your current *execution context*—see section 3.6.9. This command is built into the exec.

Q — This is an experimental interactive functional programming language (m68k only). See Per Bothner (bothner@su-pescadero) for information and a user guide.

query — Prints out the result of performing various 'query' operations. In particular, **query kernel** prints the result of the **QueryKernel()** library routine (see page 27-3), **query config** prints the contents of the workstation's configuration file (see Chapter 19), and **query ethernet** prints the result of querying the "ethernet" device (see section 36.1). **query ?** lists the possible options.

queryexec — Find out the status of the specified executive. Useful mainly for system testing.

ranlib68 — Identical to the UNIX **ranlib** command, but handles archives of either m68k or vax binaries. This command is installed on UNIX under the name **ranlib68**, and on V under both the names **ranlib** and **ranlib68**.

rcs — Part of the Revision Control System. Described by a UNIX manual page.

rcsdiff — Part of the Revision Control System. Described by a UNIX manual page.

rcsmerge — Part of the Revision Control System. Described by a UNIX manual page. Currently, this program must remotely execute the **rdiff3** and **merge** programs of RCS on a UNIX host, since the latter have not been ported to V.

rename — Renames the object specified by the first argument to the name given as the second argument. Will not move objects from one server to another; there are also restrictions on moving objects within one server (for example, from one file system to another under the Unix server).

rlog — Part of the Revision Control System. Described by a UNIX manual page.

rm — Takes one or more filenames as arguments, and removes each file. The **-f** flag suppresses error messages if some of the files do not exist.

Note that in particular, **rm** may be used, as an alternative to the **destroy** command, to destroy one or more teams (by name). For example,

```
rm [team/toto/[bin]internetserver
```

will 'remove' (i.e. destroy) the program "*[bin]internetserver*" that was executing on host "*toto*". Unlike the **destroy** command, the full program name must be given.

sed — Stream editor. Described by a UNIX manual page.

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 workstation 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 creates a menu in the invoking window; commands are normally selected with the mouse. A new window is created for displaying a page from the **.dvi** or **.press** file. You can invoke the program with **show** *filename*, or you can set the filename in the menu. More details are given by the "help" command (type **h** or select **[Help]** with the mouse). TeX generated **dvi** files are handled pretty well, except for possibly missing fonts (and perhaps speed). The **press** support is pretty minimal.

sleep — Delays for a time, then exits. The delay time (in seconds) must be specified as an argument.

sort — This command has the same syntax and semantics as under Unix 4.2BSD.

startexec — Create an exec in a new AVT. 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.

storagestats — Obtains access statistics collected by the V storage server and disk driver. The -c flag causes the statistics to be cleared to zero.

stuffboot — A program that stuffs the file named in argument 2 into the boot block of the disk named as argument 1. The file to be stuffed normally will be Vload and is needed during auto-boot when the boot program is read out of the boot block on the root disk device. If a third argument is present, it is taken as the name of a file to copy into block #0 of the root device (the disk label).

tail — This command has the same syntax and semantics as under Unix 4.2BSD.

talk — Allows interworkstation communication among V hosts similar to talk under Unix. Invoked as

talk [*person*]

Person is optional. If *person* is specified, it can either be in the form **userid** to specify a user on the first host we find that user to be logged in, **userid@host** to specify a particular user on a particular host, or **@host** to get anyone on that host.

Once inside **talk**, you can enter commands by first typing [ESC], and then another letter. The available commands are:

- i — invite a new user. You are prompted for who you want to invite, and you can respond in any of the ways mentioned for *person* above.
- l — log in AVT. This allows you to see in a temporal fashion who says what. You are prompted for a new AVT to write the log in. Giving this command again terminates logging.

q quit. Only you quit; any other persons engaged in the conversation can continue to do so, even if you were the one who initiated the conversation.

r read from file. It reads the file into the conversation such that it looks as though you typed it all. However, there is currently no pagenation. If what was read is very long it will quickly get overwritten.

w write to file. This is the same as "log in AVT" above, except logging is done to a file you specify. Giving this command again terminates writing.

Warning: `talk` runs only under the VGTS. There is currently a compiled in maximum of six talkers in one conversation. `talk` AVTs are fixed at 24 by 80.

Due to current VGTS restrictions, `talk` is forced to initiate a conversation by writing to the VGTS AVT (the small one originally in the lower right hand corner) and then opening a new AVT, which gives the victim a PAD cursor. To accept the invitation the AVT is placed normally; to reject the invitation click all three mouse buttons. This is bad if the VGTS AVT is obscured—the victim will get a beep and a AVT prompt and not know why.

telnet IP/TCP-based telnet implementation. It can run under the STS, or in a VGTS AVT. A destination host name or address may be given as a command argument; if none is given, *telnet* 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).

If the -e flag is given when it is invoked, *telnet* recognizes a set of commands prefixed by ctrl-↑ while connected to a remote host. Ctrl-↑ ? prints a list of all such commands. These functions are not available by default because they sometimes interfere with higher level protocols such as that used by the VGTS.

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

If there is no internet server on your workstation when *telnet* 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 if the -e option is specified on the command line.

The -g flag enables logging mode, which implies the -d debug mode above. A file, "telnet.loigfile" will be created in the current directory. This file will contain a complete transcript of bothe sides of the *telnet* session. Lines preceded by "<" originated from the host while those preceded by ">" originated from the workstation. All non-printing characters are quoted and all telnet protcol commands are printed. Password input is automatically deleted. This mode is transparent to both sides of the connection. Logging mode can be toggled on and off by typing ctrl-↑ g while connected to a remote host if the -e option is specified on the command line.

telnetserver IP/TCP telnet listener. This program listens for incoming telnet connections on the local internetserver, spawning a remote terminal server (RTS) for each connection received.

testexcept Simple interactive program for testing the exception server.

timeipc Performs timing tests of the V interprocess communication primitives. See chapter 13.

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

tsort	Topological sort. Identical to the UNIX program of the same name.
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 (aliases). Built in to the exec.
ved	A text editor, somewhat similar to Emacs, that runs under the VGTS. Described in Chapter 14.
vemacs	A version of the Emacs text editor that can, among other things, make use of the window features of the VGTS. When vemacs is invoked without any arguments, it will display a help file describing how vemacs differs from standard Emacs.
w	Lists logged-in V users throughout the network.
wc	Counts characters, words, and lines in a text file. See the UNIX manual for full documentation.
wh	Lists hosts on the network together with information such as logical host name, free memory, average processor usage, number of free process descriptors, host type, etc., sorted by host name.
whi	Lists currently executing teams for each host. If one or more 'host' arguments are given, then only the teams on the specified host(s) are listed. Such arguments can take the following form: a hostname, a pid (**0x...**), or "**0**" (indicating the local host).

4.2. Commands on Non-V Hosts

There are also several useful commands that can be invoked on non-V 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 3.4).

dale	A version of the Yale layout editor that runs under the VGTS.
draw	An interactive drawing program that runs under the VGTS. See Chapter 10.
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 —
amaze: A Maze Game

Amaze is a game for two to five players which runs under the STS on a workstation with a framebuffer. If you see the letters VGTS in a small window on your screen you are not running the STS. See section 2.2 for instructions on how to start up the STS.

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.

R       Redraw the maze and players.
```

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.

— 6 —
checkers

checkers allows you to play a game of checkers against the computer. The default version of the program executes entirely on the players workstation.

On starting the program, the view manager will prompt you for the position of the SGVT 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 SMI 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 'believes' 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.

A distributed version of the game may be started by specifying the r flag: *checkers -r <NoOfSlaves>*. The program will then try to create up to *NoOfSlaves* slave processes on lightly loaded remote workstations, that help in the search of the alpha-beta search tree. As far as the player in concerned, the only two noticeable differences to the default sequential version of the game, are the possible improved response times and that the computers moves may be nondeterministic. (Note: it is only worthwhile to play the distributed version of the game if the search depth is chosen greater than four.)

Mail comments and/or gripes to stumm@pescadero.

—7—
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 virtual terminal 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 (of a new virtual terminal) in which the actual editing is performed. If you request to view sample text, you will be asked to create a third virtual terminal (see below). These last two virtual terminals are SGVT's and can be zoomed.

(Note: If you are using a Sun-120 framebuffer (including a model 50), you should read the **Bugs** section!)

7.1. 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 virtual terminal. 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.)

7.2. 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.

7.3. Changing Raster Size

To change the *size* of a raster, point at the boundary, hold down the *middle* button and "drag" 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.)

7.4. 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.

7.5. 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.

7.6. 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).

7.7. 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.

7.8. 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.

7.9. [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.

7.10. Making a Copy of the Screen - CURRENTLY NON-WORKING

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 workstation manager commands.

7.11. 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-1 graphics hardware. (Will soon be obsolete.)
- 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/libsfonts.a**.
- Pxl format, which can be generated by MetaFont78, and is used by a lot of the TeX people.
- Gf ("generic font") format, a compact format generated by MetaFont84.

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.

7.11.1. Displaying Fonts

When a character in a font is displayed, there are funny lines sticking out of the bitmap picture. The intersection of the left and top segments indicates the *origin* of the character: The left segment indicates the baseline, and the top segment the starting position. The intersection of the right and bottom segments show the "ending position": the vector from the origin to the ending position is the *width* of the character. The width vector is almost always horizontal, and indicates the spacing between adjacent characters: The "next" character in a string should be positioned so that its origin coincides with the ending position of the current character.

You can select any of these lines (with the middle button), and adjust them with the mouse.

7.11.2. Font parameters

This is a section of the AVT 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. **Resolution** is ratio of pixels per point (vertically and horizontally) at which the font is designed for. (To be compatible with the Altos, we have decided that the resolution of the Sun display should be defined to be 80 pixels/inch. Older Pxl fonts use a *magnification* relative to a default Pxl resolution of 200 pixels/inch.) 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.

[Max. height] and **[descent]** give the maximum total height, and descent below the baseline, of all the characters in the current font. If you change **[descent]**, the baseline of all the characters will be adjusted accordingly.

7.12. 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 virtual terminal. 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, Carriage Return and Line Feed, respectively. **\@**, **\A**, ... **_** are control characters: **↑@**, **↑A**, ... **↑_**.

7.13. 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 (*non-working*) **[Get framebuffer]** command, allows you to print a hardcopy of the screen on a Dover printer.

7.14. 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 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. On Suns with the 120 frame-buffer, bitmaps cannot yet be magnified at all. BUT **bits** still starts up with the working window magnified 8 times!

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.

The whole mechanism for grabbing a copy of (part of) the frame-buffer is very unclean (and currently doesn't work). It should be done by the VGTS, not application programs.

—8—
build: Maintain groups of dependent programs

build is an enhanced version of Feldman's **make** program for Unix. It runs both under V and 4.2bsd Unix.

Except in pathological cases, **build** is meant to be backward-compatible with **make**. See the Unix **man**-page for **make**. In this chapter, we describe only differences between **build** and **make**.

build reads in a file describing dependencies. By default it looks for the files **buildfile**, **makefile** or **Makefile** (in that order) in the current directory.

8.1. Macros

A dependency file can contain lines of the form:

```
OBJECTS=file1.b file2.b
```

. This defines **OBJECTS** as a macro name, which can be used as in:

```
cc68 -r -vV $(OBJECTS)
```

Macro names can also be defined in the command line:

```
build "-DOBJECTS=file1.b file2.b"
```

or equivalently:

```
build "OBJECTS=file1.b file2.b"
```

8.2. Including other dependency files

A line of the form

```
#include filename
```

will parse the *filename* when reading dependency rules. (The *filename* may optionally be surrounded by <...> or "...".)

The *filename* is resolved relative to the directory containing the currently-being-read file (the one containing the **#include**), *not* the current working directory.

8.3. Conditional dependency rules

```
#ifdef name
#ifndef name
#else
#endif
```

These act like C preprocessor directives. For example **#ifdef X** succeeds iff the macro **X** is defined.

NOTE: If there is white space after a '#'-sign, the line is taken as a comment!

8.4. Search paths

A line of the form

```
VPATH=../68k ../m1
```

or, equivalently,

```
VPATH=../68k:../m1
```

causes **build** to search for files first in the current directory, then in the directory **../68k** and finally in **../m1**. The first form is probably preferable as the VPATH macro may then also be used elsewhere in the buildfile for other purposes.

One use of this is for maintaining libraries for multiple machines, where most of the sources are in a machine-dependent directory **../m1**, but some of the sources and all of the binaries are in the current directory.

Another use is for program maintenance: The sources being worked on can be in a private directory, while the remainder can stay in the master directory, if you put the master directory on the search path.

NOTE: After macro-substituting command lines, **build** will look for words (i.e. strings between spaces). If there exists an alias for a word *and* the file is up-to-date, it will be replaced by the alias. An alias exists for a word, if build has searched for the file with that name, failed in the current directory, but found it on the search path.

This simple-minded algorithm will usually do the right thing, but in pathological cases it might lose.

8.5. Dependency patterns

In addition to the old way of expressing dependencies using file suffixes:

```
.SUFFIXES: .c .b
.c.b:
        cc68 -c $*.c
```

you can also use more general pattern-matching:

```
*.b: $*.c
        cc68 -c $*.c
```

That is: a rule for making files can have as its target a pattern containing at most one '*'. The part of the file name matching the '*' defines the value of **$***.

8.6. Suggestion

If there are many files, you can speed up build (quite significantly for the V version at least) by starting out with a empty **.SUFFIXES:** line, and explicitly defining just the suffixes you need. This saves **build** from having to check for the existence of **.y** files etc.

8.7. Bugs

Does not understand RCS.

—9—
debug: The V Debugger

9.1. Synopsis

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

9.2. Description

9.2.1. Invoking the Debugger With a Program

Debug is an assembler-level symbolic debugger for V programs. Versions exist for both the Vax and the 68000.

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 AVT 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.

-o origin — The *origin* is the location where the program to be debugged was linked to load (e.g., 1000 or 2000 in the case of the kernel). The default value is the normal team origin (currently 20000). This option is usually only used by kernel hackers in place of getting a symbol table dump and assembler listing when debugging. They issue the command **debug -o 2000 /xV/kernel/sun2+ec/sun2+ec** (for example), and can find out exactly what is at the address where it crashed. The debugger disables the **g**, **x**, etc. commands when in this mode.

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 AVT is desired (*for VGTS resident environments only*) then one would type

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

9.2.2. Postmortem Debugger

The debugger can also be used as a "postmortem" debugger. The V team server is structured so that if an exception occurs in the program currently being run, the debugger is automatically loaded and given control. The postmortem debugger is always run with the **-d** flag.

9.2.3. 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/PSL
    1010              10172               10
B0> 10174      4880                    main+2C          extw d0
.s
    stack trace
.q
!
```

9.3. 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 omitting 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.

9.3.1. 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 on the 68000:

%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.

The following symbols are recognized on the MicroVAX:

%r0 - %r11	General registers 0 - 11.
%ap	General register 12 (argument pointer).
%fp	General register 13 (frame pointer).

%sp General register 14 (stack pointer).

%pc General register 15 (program counter).

%psl Program status longword.

%psw Program status word.

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. Note that the VAX C compiler prepends an underscore to the names of all global symbols. The debugger attempts to guess the correct symbol if no underscore is typed by the user, but it does get confused sometimes.

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.

9.3.2. 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.

VAX note (not applicable to the 68000): The VAX C compiler uses the *calls* instruction for all function calls. This instruction expects to find a 2-byte register mask at the address specified and actually transfers control to that address plus 2. Therefore you have to add 2 to the address when setting breakpoints at functions in a C program on a VAX. This may or may not apply to subroutine calls in assembly language or in code generated by other compilers, depending on which instruction is used.

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. Note that breakpoints within the subroutines are ignored.

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.

: A synonym for **y**, 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 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. Typing **sp** again makes the debugger go back to its original behavior.

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

9.3.3. 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*. The \ command is not very useful in instruction-typeout mode on the VAX (i.e. after giving the "'i',tt" command) because the VAX uses variable length instructions and almost every byte value is a valid op-code, thus making it impossible to tell where the previous instruction really starts. Similar problems occur less frequently on the 68000.

/

\ 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.

expression,**s** Print out a stack trace describing the chain of subroutine calls and their parameters, to a maximum of *expression* calls. (*expression* defaults to infinity.) 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', *strLength* — type out strings. Set the maximum length of strings to be *strLength*. 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.

9.3.4. Tracing Commands

expression, w — Watch the memory location at *expression*. When program execution resumes, the debugger regains control after *every* instruction and checks whether the contents of the location have changed. If so, a message is printed and the user gets control. Otherwise, the program continues. This causes the program to run several hundred times slower than normal. If *expression* is 0, watching is turned off.

expression, wb — Watch the memory location at *expression*, but only at breakpoints. A breakpoint will not stop the program if the watched location is unchanged.

w — Print information about watched location.

9.3.5. 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.

9.3.6. Help Commands

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

#
: 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**.

9.4. Bugs

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

Currently, the version of the debugger that runs on the Vax can only debug Vax programs, and the version that runs on the 68000 can only debug 68000 programs. This limitation causes little difficultly since the debugger is ordinarily run on the same host as the program to be debugged.

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.

— 10 —
draw: A Drawing Editor

The **draw** program is a document illustrator that can be used to add figures to documents created with programs such as Scribe. This program is loosely based on the Xerox Alto Draw and SIL programs, and the Apple MacDraw program. Many of the same primitive objects are common to all four programs, but there are many features unique to V Draw.

10.1. Conceptual Model

Draw is an "object" oriented graphics editor as opposed to a "bitmap" oriented editor. This means that Draw allows you to freely manipulate the figures that you create after they have been placed on the screen at the expense of being able to do fine freehand sketching and other functions such as seed filling.

The graphics model offered by Draw is very close to that provided by the underlying Virtual Graphics Terminal System (VGTS). All graphic objects manipulated by Draw are variations on three general types: splines, text, and groups.

Splines are b-splines of order 2, 3, or 5. Order 2 splines have straight edges and are thus referred to as **polygons**. Order 3 splines use quadratic interpolation, and are thus referred to as **curves**. **Circles** use order 5 splines which use quartic interpolation. Other shapes such as **ovals**, **rectangles**, and **arrowheads** are special cases of the more general order 2 and 3 splines. Note that the term, *spline*, will be used to collectively denote all of these objects.

Splines can either be **open** or **closed**. Open splines have two ends. Closed splines do not have any endpoints. Any spline can have a border drawn with one of 15 pens or **nibs**. A closed spline may also be filled with any of 27 **patterns**. All splines must have either a border, a fill pattern, or both. Fill patterns can either be **opaque** or **transparent**. Any parts of objects lying behind an opaque object will not be visible. On the other hand, if the object is transparent, then it will act as a screen where the objects underneath will show through in the areas where the upper object is not black.

Text objects allow you to place any type of written message on the page. Text can be in one of various **fonts** . It can also be either left, center, or right **justified**.

Groups do not have any graphic shape of their own, but are used to keep various objects together. Any operation performed on a group is performed on all of its members. Groups can be nested; that is, one group may be a member of another group. This nested relationship is strictly hierarchical. No recursive nesting is allowed.

10.2. 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 VGTS), and the smaller one is the commands window (known as "Draw menu" to the VGTS). The drawing area is zoomable, and the grid spacing available at normal magnification is the same as that used by the program when the right mouse button is pressed. 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. While this rectangle defines the absolute bounds of the page, the default view defines the area which is "safe" to draw on since printers cannot, in general, print in the extreme margins.

The menu window is shown in figure 10-1.

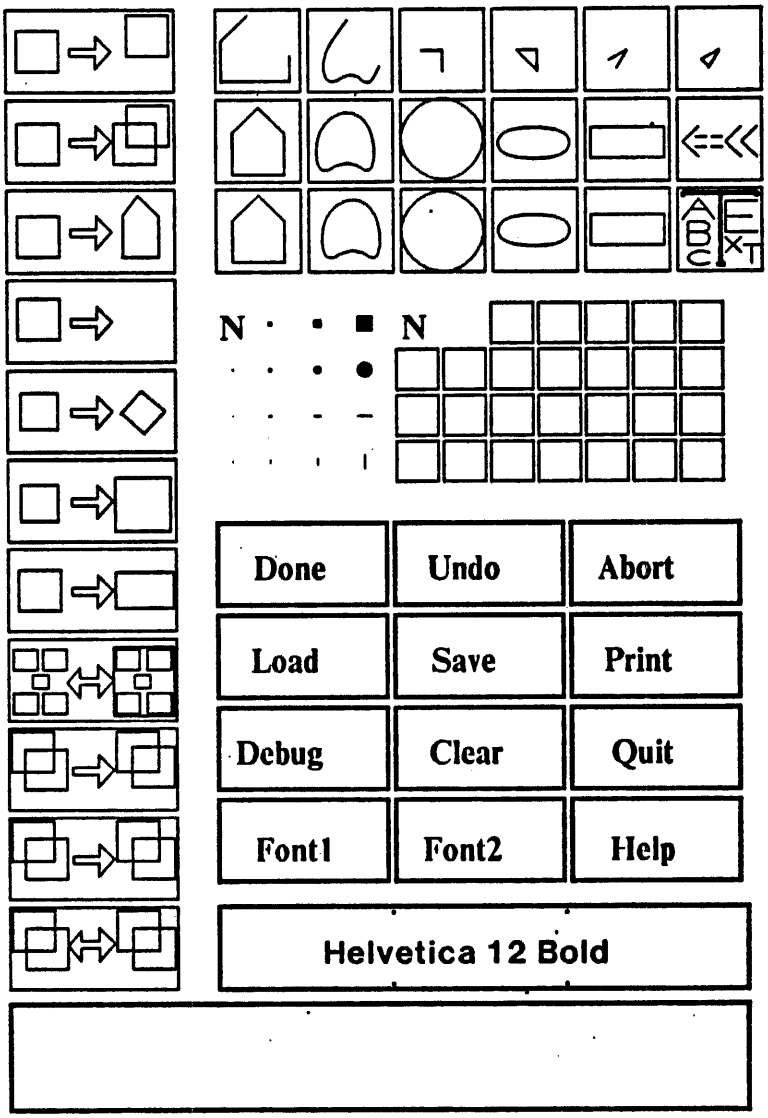


Figure 10-1: The Draw menu

The Menu window is divided into several sections. Near the top, there is a set of square icons known as **nouns**. These define the various primitive objects which can be created with Draw. Along the left hand side are the elongated **verb** icons. These define the types of transformations which can be performed on already existing objects. Below the nouns are the **nib** and **pattern** sections, collectively known as the **attribute** section. These can be used to modify the appearance of all spline objects. Below these are various Draw **commands** which are used to perform various functions. Below the commands is a rectangle which displays the currently selected **font**. This window displays the name of the font in its own typeface and is also used to set and display text justification. The Font1 and Font2 "commands" as well as the font rectangle also fall under the

heading of "Attributes" as these are used to set text attributes like the nib and pattern sections are used to set attributes of spline objects. At the very bottom of the menu is another rectangle where various Draw messages and prompts are displayed.

The original window which you used to run the Draw program will serve as an area where text will be input. It is also used for printing unexpected error messages such as memory full errors.

10.3. General style of interaction

Almost all Draw interaction (except for text and filename input) is done with the mouse. To create an object, click on one of the noun icons in the menu and then click one or more times in the drawing area to set control points for that object. There are three types of objects: indefinite point objects, definite point objects, and text objects. Indefinite point objects, such as curves and polygons will accept input for an arbitrary number of points. To terminate one of these objects, click the **Done** command in the menu. Definite point objects, such as circles, rectangles, and arrowheads require a fixed number of control points. Each point displays a prompt in the message box which indicates what this point will be used for. For example, a circle first asks for a center point, and then for a point along the edge. Text objects ask for a single control point to position the text, and then prompt for the text string from the keyboard.

Draw maintains a **current object**. This object is indicated by framing it with a rectangle. A newly created object becomes the current one. To change the current object, simply click on another one with any single mouse button. Draw will find the object that is closest to the point that you click on and select that object. No object need always be current. To un-select all objects, click in a blank section of the drawing area window. More than one object may be selected at a time by using the **ToggleSelect** or **Range** commands. This is provided primarily in order to create **groups**.

To manipulate an object, first select it, thus making it the current object, and then click on one of the verb or attribute icons. Depending on the operation involved, Draw will request zero or more data points required to perform the given operation. These will be prompted for in the message rectangle. At any time, you can halt an operation with the 'Abort' command. If more than one object is selected, then the verb applies to all of the selected objects.

10.4. Control Points and Sticky Points

When you create a spline object (remember, this also includes polygons), you will be asked to specify its Control Points. 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, in general, 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 it, 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. Circles also have a sticky point at their center. Sticky points for text objects are on the left, right, and center of the baseline (the line which most letters lie on top of, but which letters such as small 'p' descend below.) as well as the line just above the text. Groups don't have sticky points for themselves but include any sticky points of objects within the group. This, in essence, means, that when looking for sticky points, all objects are considered regardless of whether or not they are part of a group, no matter how deeply the groups are nested.

10.5. Mouse Buttons

When the mouse is clicked inside the menu, it is unimportant which buttons you use. (The debug command is a hack and is the only exception.) 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 buttons 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 'Again' command. (see below)
 X - X   The 'ToggleSelect' command. (see groups below)
 - X X   Equivalent to the 'Undo' command.
 X X X   Equivalent to the 'Abort' command.
```

Sticky points were explained above. 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).

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 VGTS. For printed output, pixels are assumed to be distributed at 72 per inch.

The **Again** command allows the previous operation to be repeated. It is equivalent to issuing the **Done** command (if necessary) and then clicking in the icon for the previous operation. The mouse position where you issue the **Again** command is ignored as long as it is anywhere within the drawing area

The easiest way to make fine adjustments to the position of an object is to first click on the **Move** verb icon, and then click on a source and destination data point. If you are not satisfied with the move, click **Again** and repeat the operation without having to go all the way over to the menu. 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 **Again**. 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 **Closed Curve** again.

The **Abort** command is used to cancel the current operation without creating or manipulating any objects. Abort will never throw you completely out of Draw. Use **Quit** for that. Some commands, such as **Raise** and **Lower**, are executed immediately and thus cannot be aborted. Use **Undo** to back out of these.

The **Undo** and **ToggleSelect** functions are described more fully in the sections on Undo and Groups below.

10.6. Verbs

There are eleven verbs in Draw. They are indicated by the set of elongated icons along the left-hand side of the menu. Each is useful for manipulating one or more objects. All verbs require that an object be selected before they are executed. Here they are described as they appear from top to bottom.

Move This verb will permit you to 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 verb is similar to **Move**, except that it leaves behind an image of the object.

Erase This command allows you to delete (erase) the selected object. This requires no extra data

points. If you make a mistake, you can always issue the Undo command.

Alter This verb is useful for changing the characteristics of an existing object. It will permit you to move the control points on splines, change aspects of text objects, etc. **(Not yet implemented)**

Rotate This verb will permit you to 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 always horizontal from left to right.

Scale This verb will permit you to specify a fixed point for the scaling, and two points which define the scaling factor. This command is useful for expanding and contracting objects. X and Y dimensions are scaled equally.

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

Stretch This verb is similar to the Scale command except that X and Y scaling is independent. Thus an object may be made taller or shorter but not wider and vice versa.

Group This verb binds a collection of objects into a group. If a single group is selected, it will un-bind that group back into a collection. Collections are created with the ToggleSelect mouse sequence to allow more than one object to be selected at one time. This, in a sense, creates a temporary group. The Group verb makes this permanent, or makes a permanent group temporary.

Raise This verb 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.

Lower This verb 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.

Opaque This verb will toggle between opaque and transparent filled objects. Opaque objects completely obscure anything they overlap. Transparent objects act like a screen in that they allow what is under them to show through white areas.

10.7. Nouns

There are eighteen icons in the noun section. These are indicated by square icons near the top of the menu.

Polygons and Curves

There are three polygon icons and three curve icons. The three icons in each class correspond to open-unfilled, closed-unfilled, and closed-filled respectively. (It does not make sense to have an open filled shape.) To create one of these types of objects, first select the icon, and then click on as many control points as desired, and then click on the Done command. You can also abort the object by clicking the Abort command either from the menu, or by the (all three buttons down) mouse sequence. Closed unfilled polygons look just like open polygons except that no line is drawn from the last point back to the first. Closed curves are continuous and need not cross any control points. Open curves, however will begin and end at the first and last control point.

Arrowheads There are four types of arrowheads: wide-open, wide-closed, narrow-open, and narrow-closed. All are entered in the same way. First the tip of the arrowhead is requested, and then its root. Arrowheads are separate objects from the main stem of the arrow and are generally placed after the stems have been drawn.

Circles Circles come in two types - filled and unfilled. The two required data points are the center

point and any point along the edge. Circles are the only shapes drawn as order 5 b-splines. Ovals and curves use order 3 b-splines.

Ovals Ovals also come in filled and unfilled varieties. The data points are two opposite corners of the inscribing rectangle. The shape of an oval is exactly the same shape as would be produced by creating a curve, specifying the four corners of the inscribing rectangle as control points.

rectangles These work exactly like ovals but with (Amazingly) straight edges!

Text Creating a text object will first prompt for a single control point. This will specify the left, center, or right side of the text at the baseline. (the bottom line which most characters touch. Letters such as p's and q's descend below the baseline.) Draw will then prompt for the text itself to be entered from the keyboard.

PressEdit symbol (<==<<)
This is a special text object which is used to match a Draw illustration with a Scribe generated document when printing to a Press printer. (see the section below on including Draw-generated illustrations in documents.)

10.8. Attributes

Both text and spline objects have certain attributes. Text objects have font and justifications attributes. Spline objects have filling and border attributes. No attribute currently applies to both of these types nor to groups. Applying an attribute to a group, however, applies it to all of its members.

Various functions in Draw can be used to change these attributes. These same functions set the default attributes for newly created objects. To change the attribute of an existing object, select it, and then click in one of the attribute functions in the menu. To set attributes for a new object, first un-select any selected object by clicking in white space in the drawing area, set the desired attributes, and create the object. Attributes can also be changed while an object is being created. The attributes that are indicated when the object is completed are the ones that stick.

The following attributes are available

Fonts Fonts are changed using the **Font1** and **Font2** commands. Even though these are technically "commands" in that they appear in the commands section, they actually work more like attributes and are thus described here.

Both bring up a pop-up menu with a list of available fonts. Font1 provides some fairly standard fonts while Font2 provides some more exotic ones. Once a font is selected, it is loaded from disk if necessary, and then its name is displayed in the font rectangle in the menu in its own typeface. (Non- Ascii fonts such as Template64 may look weird.)

Text Justification There are three different ways of positioning text: you can specify (with a data point entered via the mouse) either the left-hand corner, the center, or the right-hand corner of the baseline of the text. This provides for left, center, or right justification. Note that the baseline is the bottom line that MOST letters just touch. Small letters with descenders may actually go below the baseline. The current justification is indicated by the position of the name of the current font in the font rectangle. You will notice six small tick marks just inside the font rectangle which divide it into three parts. Clicking in either the left, center, or right area will set the respective justification and move the font name accordingly. Note that the observed action if there is a text object selected in the drawing area is not intuitively obvious. Selecting **left** justification will cause an object to be shifted **right** if it was not already left justified. This is because the object's control point is kept stationary. (Think about it.) If you are still confused about where text should appear, try positioning a few strings, using the exact positioning (leftmost) mouse button.

Nib Nibs select the "brush" that the borders of non-text objects are drawn with. There are 15 different types of nibs arranged in a four-by-four square of four shapes (square, circle, dash, and bar) by four sizes. The sixteenth nib, corresponding to the smallest square is replaced by the letter "N" meaning (N)o border. The square shapes provide sharp corners while the circular shapes provide rounded corners. The largest of these also make nice dots if a polygon or curve is created with just one point. The dashes and bars create interesting calligraphic effects, especially for curves. The no border feature only applies to filled objects. Draw prevents you from accidentally making an object invisible by deleting both its boder and its fill pattern.

Fill Pattern Next to the nibs are a set of 27 fill patterns arranged in a 4 x 7 rectangle. The 28th, at the top-left corner is marked with a letter "N" which stands for (N)o fill pattern. This is different to the one just to its right which looks blank. This is actually a white pattern which can be used to erase parts of objects that the white object overlaps.

By default, all fill patterns are **transparent**. White areas of transparent objects allow objects below them to show through. This feature can be used to create interesting effects such as Venn diagrams. A fill pattern may be made **opaque** by clicking on the **opaque** verb which toggles the opacity of an object. This means that any objects underneath it do not show through. The white pattern when transparent is equivalent to no fill pattern at all.

10.9. Commands

Below the nib and pattern attribute section and above the font rectangle is the **command** section of the Draw menu.

Done This command is used to terminate a curve or polygon which can have an arbitrary number of points. You will notice that the command is outlined in heavy black lines when it is appropriate. At other times, this command is equivalent to the **Abort** command.

Undo This allows you to back-out of the previous operation. There are two levels of Undo in Draw. If you are in the middle of an operation that requires multiple mouse clicks, then Undo will back out of the last mouse-click. Pressing Undo several times will cause more of the command to be undone until you back out completely from the command. There is also a global Undo which works by taking a snapshot of the currently visible objects on the screen just before each command is executed. Ten of these snapshots are saved. Undo will bring back the previous snapshot. The last ten operations can be backed out of in this way. Pressing Undo 10 times is effectively a redo because you return to the top of the circular Undo stack. This is handy in case that you pressed Undo too many times. Note that ALL operations can be undone - even **Clear**! Undo can also be executed by pressing the center and right mouse buttons simultaneously.

Abort This command is used to back out of an operation which requires several mouse clicks completely. The state of Draw will be left as if the operation had never been started. Abort can also be executed by pressing all three mouse buttons simultaneously.

Load This command is used to load files from disk. Anything loaded from disk is actually appended onto what may already be on the screen. To load only what is in the file, use the **Clear** command first. Draw understands how to read several different file formats: its own V Draw files, Alto Draw files, Alto/V SIL files, and journal files. Obviously, V Draw files are the preferred format as they describe all of the information that V Draw is capable of editing. Alto Draw and SIL file support is provided so that users who previously used one of these two drawing editors can port their files over. Unfortunately, the translation is not perfect. For example, Alto Draw dashed lines and the "Arrows" font are not supported. Journal files are discussed below in the section about journalling.

Save Although Draw can read files in the various formats discussed above, it will only write its

own V Draw files. Journal files are created by a completely different mechanism as discussed in "Jounalling" below.

Print Draw supports two different types of printers: **Press** and **Postscript** .

Press printers are somewhat old, but rather fast workhorses which can print a page every second. The Press document format does not allow the full generality available in Draw. In particular, filled spline objects are not supported and hence patterns, opaque, and even raise and lower operations do not affect the final output to Press printers. All of the fonts available in Draw, however, are printable on Press printers. There are three Press printers at Stanford: Dover (Margaret Jacks Hall second floor), Rover (Margaret Jacks Hall fourth floor), and Plover (Durand building basement). The current page can be output to any of these printers directly from the Draw menu.

Postscript printers are more modern, but somewhat slower printers. A typical example is the Apple LaserWriter. Postscript printers are capable of displaying any graphic objects created with Draw. Only the printer's internal fonts are available, though. This includes the Helvetica and Times fonts. Also the Ascii font is mapped onto the Courier font, and the Greek typeface becomes Symbol. These translations are not perfect but they do work most of the time. Postscript printers tend to be owned by specific groups and are not generally publicly available. For this reason, Draw checks to see which printers are available to the local UNIX V server and only displays those printers (if any).

The **Print** menu, besides letting you send files directly to various printers, also allows you to save print files to disk for later printing or for inclusion inside other text documents. This latter operation is described in a section below.

Range This command allows the selection of many objects simultaneously. The program will prompt for two control points which form opposite corners of a rectangle. It will then scan the entire list of objects and issue the ToggleSelect command on any which intersect the given rectangle. Any unselected objects will be selected. Any previously selected objects will be unselected. To select all objects within a given rectangle, first click in a blank area to unselect everything, and then use the Range command to select the desired items. This command behaves exactly like issuing a ToggleSelect on the given items individually.

Clear This provides a method of completely wiping Draw's "slate" and starting from a fresh page. Because this operation is dangerous, Draw requires that you click on the command TWICE before it is actually executed. Even then, however, it can still be backed out of using the **Undo** command.

Quit This is the best way to get out of the Draw program. Like the Clear command, above, it must be clicked on twice to actually be executed. For some strangely bizarre reason, Quit cannot be undone! The Undo button goes away, but so does everything else, for that matter.

Font1 and Font2 These two commands bring up menus which provide a selection of fonts. Font1 provides some more common fonts while Font2 provides more exotic ones. Selecting one of these will make it the current font. If a text object is already selected, it will change to the newly selected font. Otherwise, any newly created objects will use this font.

Help This command will provide a brief description of any other operation you like. To get help on a specific operation, just select that operation 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.

(Debug) This command is hidden. It can only be invoked by pressing all three mouse buttons in the message rectangle. This brings up a menu which provides several internal Draw debugging features which are generally not of interest to the user. One function, "Keep journal," is documented below in the section on journalling. For the curious, you might also try the monkey which generates (pseudo) random events as if a monkey were at the keyboard and

mouse. (Don't worry. It's safe and protected from issuing dangerous commands like Save and Print!)

10.10. groups

Groups provide access to the structured graphics capabilities of the VGTS. A group is a collection of objects. Groups may contain other groups, but a lower level group, may not call a higher level group which called it. Whenever possible, Draw tries to maintain a single copy of a group, even if it is called from many places. The only thing that differentiates two copies of a group when they are drawn on the screen is their absolute position. A group is created out of already existing objects at the top level. To create a group, use the **ToggleSelect** command (push the left and right mouse buttons simultaneously) or the **Range** command (from the commands section of the menu) to select more than one object. Any un-selected object becomes selected, and likewise, any selected object becomes unselected regardless of the number of objects already selected. Use **ToggleSelect** to affect individual objects and **Range** to change the selection status of a number of objects within a given rectangle. A set of selected objects work very much like a temporary group. Any modifications, such as rotation, scaling, or attribute changes, will be applied to all selected objects as if they were one object. Selecting any other object, or selecting nothing (clicking in white space) using the normal selection process undoes this "temporary group" but not any modifications that were made. To make a set of selected objects into a permanent group, use the **group** verb. This makes a set of selected objects into a group, and vice versa. In this way, groups can be created and destroyed. Groups are highlighted with a heavy rectangle around all of the members. Once a group is created, any operations performed on the group are performed on all of its members. Clicking on any of its members selects the entire group, but clicking in white space that happens to lie inside the group rectangle does not. No operation may be performed on a member of a group without either dismantling the group or affecting all other members.

Groups are implemnted internally in a very efficient manner. For example, multiple copies of a group (or any object for that matter) are only pointers to a single part. Only when one of the copies is modified is a true copy made. This is all automatic, however, and the user need not worry about this.

Use of groups may also speed up selection as a group who's bounding box does not contain the given selection point is skipped and all objects within that group are ignored.

10.11. Inserting Draw pictures in text documents

Draw has the capability of creating a file suitable for sending to a Press or Postscript printer, or for inclusion inside a Scribe document. The method for doing this is slightly different for Press than for Postscript.

10.11.1. Press

To insert a picture in a Scribe document, first place a PressEdit symbol (a text item showing "<==<<") in the bottom center of your picture. Note that this symbol is already provided as one of the available pre-made objects in the Draw menu. This actually has some special significance to Draw as it will not allow you to change the font or the justification of this object. It will also be automatically skipped when creating Postscript output.

Choose the "Press file" option from the **Print** menu. You will be prompted for a file name. Because of the limitations of the Dover, filled splines and polygons cannot be printed. These objects will appear unfilled and a warning message will be displayed on the terminal. Some objects are also just too complicated for the Dover to print. In this case, either garbage output will be produced, or the "press file too complicated" message will be printed on the header page with no other output.

Once the Press file has been created, you can now edit your Scribe file to automatically embed the picture in your document. insert the line

```
@libraryfile(picture)
```

near the beginning of your scribe input (.mss) file, and lines like the ones shown below at the point where you want the picture to appear.

```
...like that shown.in figure @ref(press-example).

@Begin (Figure)
@PressPicture(file="example.press", height="3.4inches")
@Caption (An example figure)
@tag (press-example)
@End (Figure)
```

This will produce output like that shown in figure 10-2.

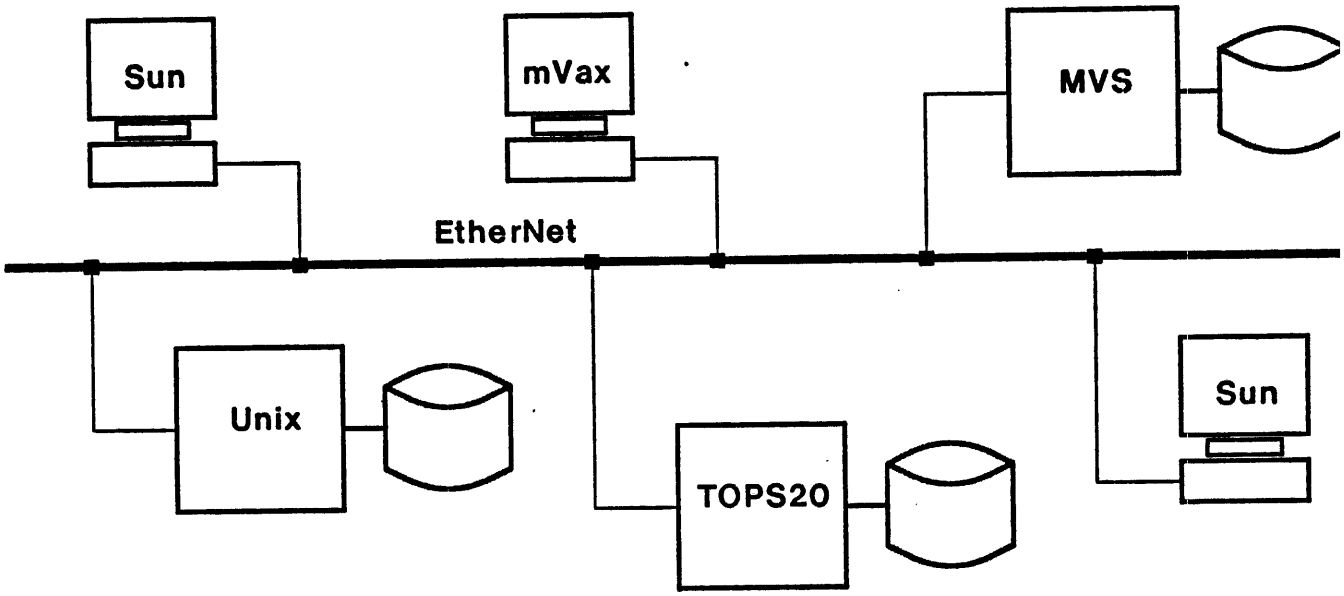


Figure 10-2: An example figure

10.11.2. Postscript

Postscript file inclusion is quite a bit different from Press printing. For starters, the PressEdit symbol is not used. Instead, Draw automatically figures out the extremes of the drawing and centers the picture accordingly. There are two menu items in the **Print** menu which generate Postscript files. The "(print later)" option create a file which is exactly like the one sent directly to the printers. The "(scribe)" version is suitable for inclusion in Scribe generated documents.

Scribe requires a special variation on the normal Postscript device driver file in order to correctly print documents with Draw illustrations. This file, "vposts.dev" must be either in your local directory or in the Scribe database directory. This file differs from the normal Postscript driver in that it contains special header information which defines macros used by Draw pictures. To used this driver, place the line

```
@device(Vpostscript)
```

at the beginning of your scribe input (.mss) file, and lines like the ones shown below at the point where you want the picture to appear.

```
...like that shown in figure @ref(postscript-example).

@Begin (Figure)
@Picture(Postscript="example.psf", size=3.4inches)
@Caption (An example figure)
@tag (postscript-example)
@End (Figure)
```

This will produce output like that shown in figure 10-2.

10.11.3. Both

Often you will want to have a Scribe document which is printable on both types of printers. You would like to be able to have Draw generated illustrations in both Press and Postscript format, and have the Scribe file choose the correct illustration by just changing the @Device command. To do this, add the @LibraryFile command at the top of your document as you would for a Press file, and add the following lines at the position where you want the illustration to appear.

```
@Begin (Figure)
@Case(GenericDevice,
    PRESS "@PressPicture(File=example.press, height=3.4inches)",
    Postscript "@Picture(Postscript=example.psf, size=3.4inches)")
@Caption (An example figure)
@tag (example-figure)
@End (Figure)
```

10.12. Journalling

Whenever Draw starts up, it creates a file called "Draw.journal" in the local directory. In this file, Draw will keep a record of all user input events. If the program should crash in any way, the journal file will be left so that the entire Draw session can be re-constructed. Under normal circumstances, this file will be automatically deleted when you quit Draw using the Quit command. You can explicitly ask that the journal file be kept around by choosing the "keep journal file" from the Debug menu which is found by pressing all three mouse buttons in the message area.

Should you ever find yourself in the V debugger because Draw bombed, the best thing to do is to type "Quit,g" . This will cause Draw to clean up its windows and to ensure that the journal file is closed.

A journal file can be played back by first renaming it so that it does not get clobbered the next time that Draw runs and then using the Load command to read it in like any other file. You will then see your entire previous Draw session performed very quickly before your eyes.

Note that journal files are extremely context sensitive. They depend on everything being exactly as it was when the journal was recorded. For example, if during the session, you loaded a regular file, edited it and saved it, the journal will probably fail because the file being loaded will be the new copy and not the old one.

— 11 —
hack: Exploring The Dungeons of Doom

11.1. Command format

To start up a game of *hack*, use the command

`hack -u` *playername* *-role* `-n -D -d` *directory*

All arguments are optional, and most are normally omitted. You can use the **-u** flag to specify your name on the command line. If this flag is not given, *hack* will use your V login name, or if you are not logged in, it will ask your name after starting. If your name is suffixed by a hyphen and a single letter, the letter specifies your character type. For example **-u fred-t** specifies that Fred wants to play as a Tourist. You can also select a character without changing your name by giving the character type as a flag, e.g., **-t** to play as a Tourist. The **-n** flag suppresses printing of the latest "hack news". (Usually there is no news anyway.) The **-D** flag lets you play in "wizard mode". It is (almost) impossible to die in this mode, and you get a free wand of wishing with 20 charges, but your score is not counted. This mode is mostly good for debugging the game. The **-d** flag specifies the directory *hack* is to use for storing temporary files, the score record, etc. If this flag is omitted, the default directory **[sys]run/hack** is used.

To see the current scores without playing, use the command

`hack -s` *-roles playernames* `-d` *directory*

The **-s** flag is required. It may be followed by one or more *role* flags, or one or more player names to see scores for only those players or roles. If no player or role names are given, *hack* prints only your own scores. The **-d** flag is optional and functions as described above.

11.2. Description

Hack is a display oriented game inspired by the popular Dungeons and Dragons fantasy game. Both display and command structure resemble *rogue*, but *hack* has many more types of monsters, magic items, and so forth.

To get started you really only need to know two commands. The command **?** will give you a list of the available commands and the command **/** will identify the things you see on the screen.

To win the game (as opposed to merely playing to beat other people high scores) you must locate the Amulet of Yendor which is somewhere below the 20th level of the dungeon and get it out. This is easier to do in *hack* than in *rogue*.

When the game ends, either by your death, when you quit, or if you escape from the caves, *hack* will give you (a fragment of) the list of top scorers. The scoring is based on many aspects of your behavior, but a rough estimate is obtained by taking the amount of gold you've found in the cave plus four times your (real) experience. Precious stones may be worth a lot of gold when brought to the exit. There is a 10% penalty for getting yourself killed.

The administration of the game is kept in the directory specified with the **-d** option, or, if no such option is given, a default directory specified at compile time. (Currently **[sys]run/hack**.) This same directory contains several auxiliary files such as lockfiles and the list of top scorers, and a subdirectory **save** where games are saved.

11.3. Options

You may set options using the HACKOPTS environment variable, or the O command within the game. The flag or command is followed by a comma-separated list of options. Available options are echo, terschelp, name, oneline, and passgo. A description of these is available through the ? command. All boolean options default to being false. To set a boolean option true, specify it in the option list. To set your character's name, use the construct **name=***your name*. For example, to set terschelp, passgo, and your name to Yen Goi, the option string would be **tersehelp,passgo,name=Yen Goi**. You cannot change your name once you start playing.

11.4. Authors

Jay Fenlason (plus Kenny Woodland, Mike Thome and Jon Payne) wrote the original *hack*, very much like *rogue* (but full of bugs). Andries Brouwer continuously deformed their sources into the current version—in fact an entirely different game. Ported to the V-System, and additional hacking done, by Tim Mann. The V-System version is based on Andries Brouwer's version 1.0.3.

11.5. Files

Files other than the *hack* program itself are kept in the administration directory mentioned above.

data	Data file for the / command.
help, hh	Data files for the ? help command, respectively the long and terse forms.
news	Hack news, printed whenever a game is started.
rumors	Fortune cookie database.
record	The list of top scorers.
save	A subdirectory containing the saved games.
bones_dd	Descriptions of the ghost and belongings of a deceased adventurer.

11.6. Bugs

Probably infinite. You can mail complaints to games@Pescadero, but we suggest volunteering to fix it yourself if you want it fixed.

This game is a huge time sink.

— 12 —
siledit: A Simple Illustrator

The **siledit** program can be used to edit simple illustrations. It is uses a compatible file format with the Alto **SIL** program, although some obscure features such as macros are not implemented. The main limitation of this format is that only horizontal and vertical lines are supported, with a limited range of fonts. On the other hand, it is simpler and faster than **draw**, and illustrations produced by **siledit** can be easily inserted into other documents or printed out. A remote version of this program will run under UNIX, although users will probably prefer the V-System version of the program if permitted by workstation memory limitations.

12.1. Basic Operation

The **siledit** program is invoked with one argument:

siledit *filename*.**sil**

It first attempts to open the file name given as an argument. If no such file exists, the program continues and allows one to be created. An SGVT is created, and the cursor should change to the "View" prompt indicating the creation of a default view. The default view will be slightly larger than the illustration, or a whole page if the illustration is empty. Press and hold any button an an outline the size of the default view will appear and track the cursor. Position the upper left corner of the desired default view with the cursor, and lift the button up when the view is in the right place. Next the **siledit** program prints out the text fonts that will be used, and tries to load the appropriate fonts into the VGTS. Then the existing illustration is displayed, and the following prompt appears:

Use mouse buttons: Mark, Select, Menu

Thus two mouse buttons are used for the basic commands, with other commands available through combinations of buttons or from the popup menu.

The *Mark*, indicated by an "X" shaped cross, is used as one end of lines and the position of added text. Once objects are added to the illustration, they can be modified by first selecting them and then performing one of the modification commands. Selected objects will appear higlighted in some way, although the exact form of the highlight is dependent on the VGTS implementation. In the SUN implementation, objects are normally black on while, with selected lines appearing as half-tone gray and selected text appearing within a gray box.

12.2. Commands

The commands available on the mouse are as follows:

Left Button — Moves the mark to the point of the click. The "X" shaped cross moves to the new location. The mark is normally moved before drawing lines or placing text.

Middle Button — Selects the single object at or near the click. Any other objects previously selected are no longer selected. The program will echo the kind of object selected, or issue a diagnostic if no objects are found.

Left+Middle Button — Draws a line from the mark to the point of the click. The line is either horizontal or vertical, depending on which difference in position is larger. This is a faster way of drawing lines than using the menu. The current line width is used for the line. The mark

is moved to the point of the click, to facilitate drawing a series of connected line segments.

Middle+Right Button
Adds the object near the click to the selection. This is in contrast to the Middle Button, which causes exactly one object to be selected. Use this command to select several objects.

Right Button Command menu, as described below.

More advanced commands are available on the menu as follows:

Quit Exits without saving the illustration. Usually you want to do the Write command first, so if there have been changes since the last Write command, confirmation is requested first.

Line Width Pops up another menu of default line widths. Select the desired new width from 1 to 8 units. Clicking outside the menu results in no change to the width.

Delete The selected objects are deleted. Currently there is no Undelete, so be careful!

Unselect Another click is requested, and the object near that click will no longer be selected.

Draw Line Another click is requested, and a horizontal or vertical line is drawn between the mark and the position of the click.

Add Text A line of text is requested, and the text is added at the position of the mark in the current font.

Modify Text Selects another menu for modifying text.

Write Writes the illustration back to the file given on the command line.

Stretch Line Position the cursor near one end of the selected line, and hold down a button. The end of the line will move following the cursor until the button is released. (Available only in the native V-System version.)

Move Position the cursor anywhere in any view of the illustration and press any button. The selected objects will follow the cursor until the button is released. (Available only in the native V-System version.)

Copy Position the cursor anywhere in any view of the illustration and press any button. A copy of the selected objects will follow the cursor until the button is released. (Available only in the native V-System version).

Box Move the cursor to one corner of the box, and press any button. While holding down the button, position the opposite corner of the box. The box will be drawn in the current line width. The box can be aborted by pressing all three buttons at the same time. (Available only in the native V-System version.)

Select Area Move the cursor to one corner of the area, and press any button. While holding down the button, position the opposite corner of the area. All objects within the area will be selected. (Available only in the native V-System version.)

Debug Enables several debugging print statements, for maintenance use only. (Available only in UNIX version.)

The following commands are used to modify text:

Edit Text First select some text, then issue this command. The text is stuffed into the VGTS line buffer, and edited by the user.

Default Font Displays a menu of fonts to be chosen to become the new default font. Text added with the Add Text command will use the new default font.

Change Font Changes the font of the selected text. Displays a menu of fonts to be chosen as the new font for the selected text.

12.3. Selecting Alternate Fonts

Only two text font/size combinations are available, but with all of the regular, bold and italic faces. Default fonts are Helvetica7 and Helvetica10, with Helvetica7B, the bold face, Helvetica7I the italic face, etc. A third font, Template64, is used to draw circles and diagonal lines. A one-page chart of the Template64 character set is probably required to use any of these shapes.

Other fonts can replace the two Helvetica fonts by creating a file with the name *filename*`.fonts`. This file should contain the names of the fonts to be used, one per line. Comments in this file are indicated by a # character at the start of a line. The default fonts are acceptable for illustrations to be included in papers, but for slides larger fonts like 12 and 18 point should be used. Thus, for example, the font file:

```
# font file for slides
Helvetica12
Helvetica18
```

could be used when making slides. The command:

```
nm68 -d -g /usr/sun/lib/libsfonts.a
```

can be used to determine what fonts are available. This command lists the defined global symbols in the font library.

12.4. Generating Printed Copy

The `silpress` program is used to produce the printed illustrations from SIL format. Currently this command only runs under UNIX. Alternate fonts can be selected as in the `siledit` program. The command line:

```
silpress filename.sil
```

will convert the named illustration into a Press format file and print it on the Dover. Most of the options available to the CZ program are available in `silpress`. Use the `man cz` command for more details. In particular, the `-p file.press` option can be used to specify the name of a press file and inhibit printing. This is useful if the illustration is to be merged into a document produced with the Scribe or LaTeX document compilers.

When using Scribe, include the command

```
@libraryfile(picture)
```

near the beginning of your manuscript file. Then, for each illustration include the following commands:

```
@Begin(Figure)
@PressPicture( Height = "5.25 Inches", File = "filename.press" )
@Caption(A caption for this illustration)
@End(Figure)
```

Where the height is an estimate of the vertical size of the picture. Then place the character sequence <==<< with `siledit` near the bottom center of the illustration, and run `silpress` to create the Press file. The CZ program of UNIX will insert the figures automatically. It usually several iterations to get the positioning and size right, but it is much faster than using a scissors and paste.

```
! siledit filename.sil
! silpress -p filename.press filename.sil
! cz paper.press
[Inserting filename.press on page 1]
```

— 13 —
timeipc: A V Performance Measurement Tool

The *timeipc* program performs timing tests of the V interprocess communication primitives (**Send**, **Receive[WithSegment]**, **Reply[WithSegment]**, **MoveTo** and **MoveFrom**).

To run the program, simply enter the command **timeipc**. For some tests, *timeipc* invokes a second program named *timeipcserver* to serve as a target for IPC messages; *timeipcserver* need never be invoked directly by users.

13.1. Types of Tests

Timeipc allows you to conduct a number of *tests*. A test consists of a number of *trials*. A trial consists of a number of *message transactions*.

Each test measures one of the following types of message transaction:

Send-Receive-Reply	without segments
Send-Receive-ReplyWithSegment	with short segments
Send-ReceiveWithSegment-Reply	with short segments
Send-Receive-MoveTo-Reply	with long segments
Send-Receive-MoveFrom-Reply	with long segments

Short segments are up to MAX_APPENDED_SEGMENT bytes long (1024 in the current version of the kernel). Long segments are longer than MAX_APPENDED_SEGMENT bytes.

For each trial, the Sender process executes the following code:

```
msgCounter = msgsPerTrial;
<record start time>
do
  {
    msg->timingCode = typeOfTest;
    Send( msg, receiverPid );
    if( msg->timingCode != OK ) <abort trial>
  }
while( msgCounter-- );
<record stop time>
```

The Receiver process executes different code depending on the type of message transaction being tested. For **Send-Receive-Reply** tests, the Receiver executes:

```
msgCounter = msgsPerTrial;
do
  {
    senderPid = Receive( msg );
    if( msg->timingCode != typeOfTest ) <abort trial>
    msg->timingCode = OK;
    Reply( msg, senderPid );
  }
while( msgCounter-- );
```

For **Send-Receive-ReplyWithSegment** tests, the Receiver executes:

```
msgCounter = msgsPerTrial;
do
  {
    senderPid = Receive( msg );
    if( msg->timingCode != typeOfTest ) <abort trial>
    msg->timingCode = OK;
    ReplyWithSegment( msg, senderPid, localSegPtr, msg->segPtr, msg->segSize );
    /* NOTE: lost reply segments are not detected! */
  }
while( msgCounter-- );
```

For **Send-ReceiveWithSegment-Reply** tests, the Receiver executes:

```
msgCounter = msgsPerTrial;
do
  {
    senderPid = ReceiveWithSegment( msg, localSegPtr, &localSegSize );
    if( msg->timingCode != typeOfTest ||
        msg->segSize    != localSegSize ) <abort trial>
    msg->timingCode = OK;
    Reply( msg, senderPid );
  }
while( msgCounter-- );
```

For **Send-Receive-MoveTo-Reply** tests, the Receiver executes:

```
msgCounter = msgsPerTrial;
do
  {
    senderPid = Receive( msg );
    if( msg->timingCode != typeOfTest ) <abort trial>
    MoveTo( senderPid, msg->segPtr, localSegPtr, msg->segSize );
    msg->timingCode = OK;
    Reply( msg, senderPid );
  }
while( msgCounter-- );
```

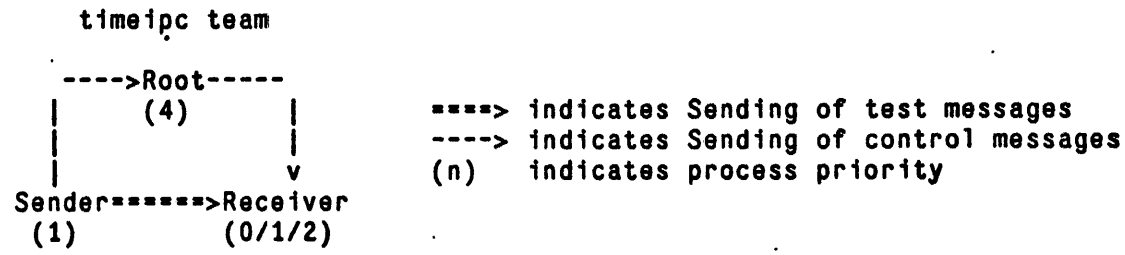
For **Send-Receive-MoveFrom-Reply** test, the Receiver executes:

```
msgCounter = msgsPerTrial;
do
  {
    senderPid = Receive( msg );
    if( msg->timingCode != typeOfTest ) <abort trial>
    MoveFrom( senderPid, localSegPtr, msg->segPtr, msg->segSize );
    msg->timingCode = OK;
    Reply( msg, senderPid );
  }
while( msgCounter-- );
```

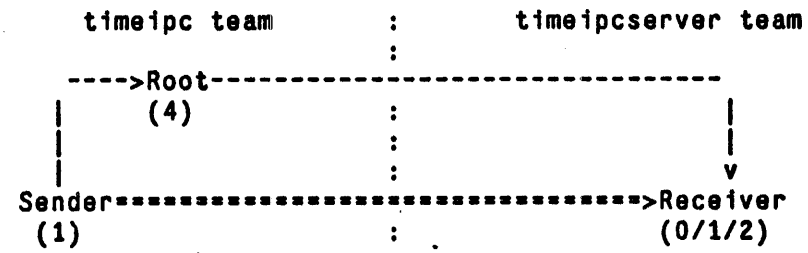
13.2. Process Configurations

Each type of test can be performed in any of the following three process configurations:

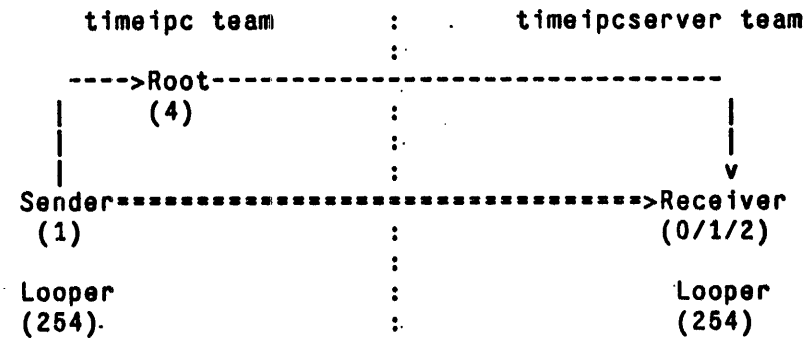
(1) IPC between two processes on the same team.



(2) IPC between two processes on different teams on the same host.



(3) IPC between two processes on different teams on different hosts.



Both *timeipc* and *timeipcserver* execute at REAL_TIME1 team priority, giving their processes precedence over all other processes except the Kernel Process (or other teams executing at REAL_TIME1). For tests within a single host, the Sender and Receiver consume all processor cycles outside of the kernel, for the duration of a trial. For tests between hosts, each testing team runs an additional Looper process which loops forever, consuming and measuring all cycles not used by the Sender or Receiver. Thus, during a trial, the testing workstation(s) will appear to freeze -- the cursor will not track mouse movement and keyboard input will be queued in the kernel. It is also possible that concurrently-running, time-dependent applications may suffer (e.g. TCP connections via the internet server may time out during a trial).

13.3. Input to *timeipc*

Before each test, you must answer a series of prompts to specify the type of test and the process configuration for the test. For each prompt, you may enter a null reply to request the default value specified in brackets, or enter **^z** to terminate the program. **^c** is ignored during prompting.

receiver host name, 'local', 'sameteam', or 'quit'? [sameteam]

Reply with the name of a workstation on which to execute the Receiver team, or enter one of the three keywords. (The Sender team always executes on the workstation where the *timeipc* program was executed.) **local** requests that the test be performed between two teams on the local workstation. **sameteam** requests that only one team be used. **quit** terminates the *timeipc* program. Any of the keywords may be abbreviated to a single letter.

receiver at higher, same, or lower priority than sender? [lower]

This prompt occurs if the test is to be performed within a single host. Reply **higher** or **h** to run the Receiver at process priority 0, **same** or **s** for priority 1, or **lower** or **l** for priority 2.

segment size in bytes, K bytes, or M bytes? [0]

Specify the size of segment to be moved in the test message transaction. Don't leave any space between the number and the optional **K** or **M** suffix. A size of zero requests a simple **Send-Receive-Reply** test.

read or write? [read]

This prompt occurs if a non-zero segment size was requested. **read** or **r** requests a **Send-Receive-ReplyWithSegment** test (if segment size <= MAX_APPENDED_SEGMENT) or a **Send-Receive-MoveTo-Reply** test (if segment size > MAX_APPENDED_SEGMENT). **write** or **w** requests a **Send-ReceiveWithSegment-Reply** test (if segment size <= MAX_APPENDED_SEGMENT) or a **Send-Receive-MoveFrom-Reply** test (if segment size > MAX_APPENDED_SEGMENT).

number of messages per trial? [10000]

Enter the number of times the message transaction is to be repeated within a single trial. Note that **MoveTo/MoveFrom** operations are *not* counted as separate transactions.

number of trials? [10]

Enter the number of trials to be performed. Enter zero to start the prompting all over again.

13.4. Output from *timeipc*

The results of the tests are written to **stdout**. All prompts and error messages are written to **stderr**. Here is an example of test output:

```
Send-Receive-MoveTo-Reply test with 4096 byte segments
between sender host 'nanaimo' and receiver host 'lubbock'
500 messages per trial, 5 trials   Wed Nov  6 16:50:15 1985

                                    sender   receiver   sender   receiver
trial    elapsed   elapsed        overhead  overhead  idle CPU  idle CPU  seg rate
number   seconds   usec/msg       usec/msg  usec/msg  usec/msg  usec/msg  bits/sec

   1      13.030     26060            5         5       15022     14859    1257406
   2      13.320     26640            5         5       15618     15253    1230030
   3      13.490     26980            5         5       15889     15764    1214529
   4      14.800     29600            5         5       18387     18255    1107027
   5      13.020     26040            5         5       14925     14686    1258372

avg.      13.532     27064            5         5       15968     15763    1213473
```

Not all columns are printed for all tests, and the "avg." row is only printed when there is more than one trial per test. The meanings of the various statistics are described here:

elapsed seconds

is the total elapsed time taken for the specified number of message transactions. It is determined by calling the kernel's **GetTime** function before and after the sequence of messages, and thus is only as accurate as **GetTime**. Although it is printed to three decimal places, the current version of the kernel only keeps time to hundredths of a second, so the low-order digit should always be zero.

elapsed usec/msg

is the elapsed seconds divided by the number of message transactions, printed in microseconds.

sender overhead usec/msg
receiver overhead usec/msg

are the number of microseconds of "overhead" instructions expended for each message transaction, in the Sender process and the Receiver process. They are computed by calling **GetTime** before and after 50,000 iterations of the transaction loop with the IPC primitives removed, and dividing by 50,000. These overhead values should be subtracted from the **elapsed usec/msg** to obtain the bare message transaction time.

sender idle CPU usec/msg
receiver idle CPU usec/msg

are only printed for tests between two hosts, and indicate for each host how many microseconds of the **elapsed usec/msg** are spent waiting for the other host or the network. They are measured by having a lower priority process on each machine that loops continuously, incrementing a counter. At the start of a trial, the counter is set to zero. At the end of a trial, the counter value is saved. Then, the looper is allowed to run alone for 1 second to determine how many times per second it can increment the counter. That rate is divided into the saved count to arrive at the printed value.

seg rate bits/sec

is only printed for tests with a non-zero segment size. It is the number of bits of segment data moved during the trial divided by the elapsed time of the trial.

13.5. Warnings and Precautions

- Despite having exclusive access to process-level cycles on the testing workstation(s), the progam can yield different results for repetitions of the same test due to varying kernel overhead and network load. For this reason, a test should consist of more than one trial, in order to observe the variance in the results. A number of steps can be taken to minimize these perturbations, depending on your need for accuracy:
  - Run the test at a time of low network load, even if you are only testing within one workstation -- the kernel expends cycles receiving and handling arriving network packets. The *mon* program is very helpful for discovering the current network load (but not while a test is running!).

    Alternatively, isolate your workstation(s) from the network during the tests. For tests within one workstation, you can unplug the transceiver cable after you have started up the *timeipc* program. For tests between two workstations, you can connect them both to a DELNI which can be isolated from the network at the flick of a switch. (This is also a polite thing to do to avoid loading the network with your test messages.) Note that you will not be able to redirect your test output to a file if you have disconnected from the network.
  - Don't touch your keyboard or your mouse while a test is running -- they cause interrupts that the kernel must handle.
  - Kill off any extraneous process that are running on your workstation(s), such as *mon*, *telnet*, *internetserver*, etc. I haven't figured out why their presence effects the results of the tests, but it does.
  - Be sure that you are logged-in in order to prevent others from remotely executing programs on your workstation(s).
  - Make sure no other REAL_TIME1 teams are running on your workstation(s), such as other instances of *timeipc* or *timeipcserver*.
  - It is possible to run *timeipc* on top of a bare kernel, i.e. without any other teams present. Only tests within a single team can be performed because the services of the team server are not available to set up a separate receiver team. An example boot command to load *timeipc* onto a Sun2 workstation with 3Com Ethernet interface is:

    ```
    b V /usr/V/bin/timeipc.m68k Vkernel/sun2+ec
    ```

    When running in this mode, answers to prompts must be entered using LINEFEED rather than RETURN. You may safely ignore warnings about inability to set the team priority.

    (Currently, a simple **Send-Receive-Reply** transaction on a Sun2 workstation is 6 microseconds faster when performed on top of a bare kernel, compared to running on top of a freshly-booted standard first team and VGTS. Surprisingly, using the STS instead of the VGTS makes the simple message exchange *slower* by 2 microseconds.)
- Be aware that your workstation will appear to freeze up during a trial. You may enter **↑C** to abort the test and return to the first prompt, but the interrupt will not take effect until the end of the current trial. If you don't know how long a particular trial will take, try it first with a small number of messages. However, test results for trials running less than a second or two should not be considered accurate.
- For **Send-Receive-ReplyWithSegment** tests, lost reply segments are *not* detected. Be wary of inter-host results from such a test.
- For **Send-ReceiveWithSegment-Reply** tests, a trial will be aborted if the receiver does not receive the sent segment. This occurs if the receiver is not ready when the segment arrives, for example if the receiver is running at lower priority than a sender on the same host.
- Watch out for VGTS page mode -- you may think you are waiting for a trial to finish when in fact the program is blocked trying to write to your virtual terminal.

- Be aware that inter-host tests consume considerable Ethernet bandwidth (up to 3 megabits/second or more) and you are in danger of becoming unpopular with other users of the network.
- *Timeipc* does not currently measure the performance of **Forward** or any group IPC operations.

— 14 —
ved: A Text Editor

Ved is the V system text editor. 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.

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.

14.1. Starting up

Ved is invoked as follows:

ved {*-number*} {*filename*}

If a file name is given, Ved begins by reading in the file. It then requests an AVT, 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 AVT and click any button. The AVT will appear, and in it the first screenful of text will be displayed. The AVT 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. Typing into this window while not specifically being prompted there for text will buffer those characters until input *is* requested. This is not, in general, the desired result. In normal use it is convenient to shrink this window down to a few lines at the bottom.

The number of lines in the AVT created for displaying a file can be specified with the *-number* option. The default size is 28 lines.

At the top of the editing window, there is a banner. When the banner is inverted (darkened), 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 14.13) and the Vgt number (see section 2.4.2). 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 14.14.2).

14.2. Some Notational Conventions

In the subsequent command descriptions the following notational conventions will be used:

- ↑*k* denotes hitting the CTRL key simultaneously with the *k* key.
- Esc-*k* denotes hitting the ESC key followed by hitting the *k* key.
- ↑*k*-*j* denotes hitting the CTRL key together with the *k* key, followed by hitting the *j* key.
- Some keyboards have function keys that generate sequences beginning with Esc-[. Where these are supported by ved, they will be denoted by Ansi-*k*, meaning the sequence Esc-[-*k*.

In general, there are (roughly) the following categories of key commands:

- Regular key strokes: e.g. *k*.
- "Control" characters: e.g. ↑*k*.
- "Escape" characters: e.g. Esc-*k*.
- "Control-x" characters: e.g. ↑x-*k*.
- "Control-x control" characters: e.g. ↑x-↑*k*.

14.3. Special Commands

↑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, ↑c — Quit the editor. If there are any modified buffers, you will be asked if you want to save them. If any .CKP files (files with .CKP suffixes are checkpoint files) have been created during this ved session, they will automatically be deleted. 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. Typing anything other than an n or y will cause the question to be asked again.

↑l — (CTRL - L) Redraw the display.

↑u — Prefix argument. Typing a number after this causes the number to be used as an input argument to the subsequent editing command. The prefix argument is only used by some commands. The others simply ignore it. ↑u is very similar (in intention at least) to the ↑u repeat factor in Emacs.

14.4. Cursor Motion

↑f, Ansi-c, right arrow
: Move forward (right) one character.

↑b, Ansi-d, left arrow
: Move backward (left) one character.

Esc-f — Move forward to the end of a word.

Esc-b — Move backward to the beginning of a word.

↑p, Ansi-a, up arrow
: Move up one line. A half page is scrolled if the cursor would go off the AVT.

↑n, Ansi-b, down arrow
: Move down one line. A half page is scrolled if the cursor would go off the AVT.

↑a — Move to the beginning of the line.

↑e — Move to the end of the line.

Esc-comma, Ansi-h
: Move to top, left-hand corner of the viewport.

Esc-period — Move to bottom, right-hand corner of the viewport.

Esc < — Move to the beginning of the buffer.

Esc > — Move to the end of the buffer.

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.

14.5. Paging and Scrolling

↑v	Page down 1 page.
Esc-v	Page up 1 page.
Esc-down-arrow, Esc-Ansi-b	Page down 1/2 page.
Esc-up-arrow, Esc-Ansi-a	Page up 1/2 page.
↑z	Scroll one line up. I.e. move the viewport up one line relative to the text.
Esc-z	Scroll one line down. I.e. move the viewport down one line relative to the test.
Esc-!	Scroll current line to the top of the viewport.

14.6. Special Characters

Typing any printing character, or TAB, inserts the character typed. Ved also supports an "auto-linefeed" mode. When auto-linefeed is enabled, typing in characters which would extend beyond the viewport's right-hand edge causes a linefeed character to be inserted before the last word on the current line. The effect is to split the current line into two lines, with the last word of the old line becoming the first word of the new line. This mode can be toggled on or off:

↑x-l	Toggle auto-linefeed option.

Various special characters are handled as follows:

Return	Insert a Linefeed, not a CR character – gets the desired effect.
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.
↑o	Insert a Linefeed, leaving the cursor before it.
↑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.

14.7. The Kill Buffer

Ved provides a special buffer, called the *kill buffer*, that is used to temporarily store text for various operations. Various editing commands specify this buffer as the source or destination of text they manipulate. The buffer should be thought of as a "clipboard" that is used for "cutting and pasting" operations on text.

14.8. Basic Editing Commands

↑d	Delete forward from the cursor – the character under the cursor.
del, backspace, ↑h	Delete backward from the cursor.
Esc-d	Delete word forward.
Esc-h	Delete word backward.

↑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.

↑t — Transpose the two characters before the cursor.

Esc-u — Make the word the cursor is in, or just after, all capital letters.

Esc-l — Make the word the cursor is in, or just after, all lower case.

Esc-c — Capitalize the word the cursor is in, or just after.

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.

Esc-blank, Esc-right-arrow, Esc-Ansi-c
Indent the current line four spaces.

Esc-backspace, Esc-↑h, Esc-left-arrow, Esc-Ansi-d
Decrease the indentation of the current line four spaces.

14.9. 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).

↑w, ↑x-↑k — Kill region. Region vanishes and becomes the killbuffer—so this command can be undone with ↑y.

↑x-↑r — Write region. Prompts for a file name, and writes the region into that file. The buffer is unchanged.

Esc-i — Indent region. Indents all lines in the region by the number of spaces specified by the prefix argument. If no prefix argument was specified, then all lines are indented one space.

14.10. C-Specific Editing Commands

The commands described in this section are specific to the editing of C programs.

Esc-{ — Generate two new lines, the first containing a { indented two spaces from the previous cursor position, the second containing the cursor an additional two spaces indented.

Esc-} — Generate two new lines, the first containing a } indented two spaces less than the previous cursor position, the second containing the cursor indented an additional two spaces less.

14.11. 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. 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.
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.
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.
↑x-t	Tag search. If a **tags** file is present in the current working directory, then this command can be used with it to find keywords in various files.[5]

14.12. File Access

Ved supports various options with respect to file writing operations and checkpointing operations. Files can be backed up and they can be written out using a "verify" option that ensures that what was written out is actually what is in a buffer. These options can be toggled on or off, as described below. Files can also be automatically checkpointed every *n* editing actions. Specification of the checkpoint frequency is done in the **.Ved_pro** initialization file. (See section 14.16.)

When ved's backup option is on, it preserves the previous version of a file by renaming it to its former name followed by ".BAK". Thus *myfile.c* becomes *myfile.c*.BAK . Similarly, if the checkpointing option is on, files are periodically written out to a file whose name consists of the actual filename followed by ".CKP". Thus *myfile.c* becomes *myfile.c*.CKP . The verify option reads files back in after writing them and compares them against the buffer contents. This feature represents an end-to-end check that was implemented at a time when the V-system's file writing operations were not completely reliable.

Upon normal exit from ved (by either typing ↑x-↑z, or ↑x-d to the last window) the .CKP files that were created during the current ved session will be automatically deleted. If ved exits abnormally, these files will contain a copy of your files that are correct as of the last time checkpointing was performed.

Ved filenames can be up to 256 characters long, but filenames of this length are not in general recommended.

↑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.

[5] Those unfamiliar with tags should read the UNIX manual entry for **ctags**. This command creates a file which specifies the location of every C-program function and type definition in a specified set of source files. It provides a means of locating such definitions without having to perform a string search on all source files each time.

↑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-↑m	Write all modified buffers to the files from which they were read. Esc-Return has the same effect.
Esc-~	Forget that the buffer has been modified. This will cause the file not to be written out on exit or when a command is given to write out all modified buffers.
↑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. Typing ↑x-b again will turn it back on.
↑x-v	Toggle the verified write option.
↑x-c	Change current context (working directory). The Ved control window always displays the absolute name of the current context in its banner, while file windows display the absolute path name of the file being edited.

14.13. 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 has its banner inverted (darkened). Window selection can be changed by clicking a mouse button in an unselected window, or by ↑x-*digit*. 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 AVI creation by pressing all three buttons, the command is aborted.
↑x-G	Get file and specify window size. In addition to prompting for a file name, you also get prompted for the number of lines the window should have.
↑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.
↑x-1 - ↑x-9	Select the corresponding window.
↑x-o	Order buffers. Redisplays all the buffers, starting with the highest numbered one. This leaves the buffers "stacked" on top of each other on the screen. This is useful if buffers have been positioned in a "stair-case" order, starting at the lower left and moving to the

upper right, so that the stacked configuration leaves the file name banner of each buffer displayed.

14.14. 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.

14.14.1. 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. **Caution:** The difference between the cursor and a selection which is only one character long is that the cursor flashes, while the selection remains inverted.

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.

14.14.2. 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 any 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.

14.15. 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.
↑l	(CTRL - 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.
t	Tag search. Treats the selection as a tag and searches for its location using the **tags** file of the current working directory.

14.16. Ved Initialization

Various ved features can be initialized to prespecified values using the **.Ved_pro** file, which should reside in the user's home directory. (The existence of this file is optional.) These include:

- Redefinition of key bindings.
- Specification of toggle settings for various options.
- Specification of the checkpointing frequency.

14.16.1. Key Bindings

Ved uses a key table to determine what function should be invoked when a particular key or key sequence (such as ↑x-↑c) is typed. The default settings in this key table have been described. The user can change the key table settings by specifying new bindings in the initialization file. The syntax to use for specifying new key bindings is demonstrated below in the list of default bindings shown. Thus, for example, one could set a new key binding that defined the ↑x-↑m key sequence to denote WriteModifiedFiles instead of the Esc-↑m key sequence, by placing the following line in one's **.Ved_pro** file:

```
↑x-↑m WriteModifiedFiles
```

The default key bindings are the following:

```
\\r InsertReturn
\\n NewlineAndIndent
\\t InsertTab
Esc-\\t IndentLikePreviousLine
↑u ProvidePrefixArgument
↑x-↑z ExitEditor
↑c ExitEditor
↑f ForwardCharacter
Ansi-c ForwardCharacter
rarrow ForwardCharacter
↑b BackwardCharacter
Ansi-d BackwardCharacter
larrow BackwardCharacter
↑a BeginningOfLine
↑e EndOfLine
↑n NextLine
Ansi-b NextLine
darrow NextLine
↑p PreviousLine
Ansi-a PreviousLine
uarrow PreviousLine
↑z ScrollOneLineUp
pf1 ScrollOneLineUp
smi-pf1 ScrollOneLineUp
Esc-z ScrollOneLineDown
pf2 ScrollOneLineDown
smi-pf2 ScrollOneLineDown
Esc-f ForwardWord
Esc-b BackwardWord
Esc-u CaseWordUpper
Esc-l CaseWordLower
Esc-c CaseWordCapitalize
```

```
↑v NextPage
Esc-v PreviousPage
Esc-darrow NextHalfPage
Esc-Ansi-b NextHalfPage
Esc-pf1 NextHalfPage
Esc-smi-pf1 NextHalfPage
Esc-uarrow PreviousHalfPage
Esc-Ansi-a PreviousHalfPage
Esc-pf2 PreviousHalfPage
Esc-smi-pf2 PreviousHalfPage
↑l RedrawDisplay
Esc-, BeginningOfWindow
Ansi-h BeginningOfWindow
Esc-. EndOfWindow
Esc-! LineToTopOfWindow
Esc-< BeginningOfFile
Esc-> EndOfFile
Esc-g GotoRequestedLine
↑x-t RequestTagSearch
↑s RequestStringSearch
Esc-s RepeatStringSearch
pf3 RepeatStringSearch
smi-pf3 RepeatStringSearch
↑r RequestReverseStringSearch
Esc-r RepeatReverseStringSearch
pf4 RepeatReverseStringSearch
smi-pf4 RepeatReverseStringSearch
Esc-q QueryReplace
↑h DeletePreviousCharacter
del DeletePreviousCharacter
↑d DeleteNextCharacter
Esc-d DeleteNextWord
Esc-h DeletePreviousWord
↑t TransposeCharacters
↑o NewlineAndBackup
↑k KillToEndOfLine
↑y YankKillBufferAfterCursor
Esc-y YankKillBufferBeforeCursor
↑x-↑v VisitFile
↑x-↑s SaveCurrentBuffer
↑x-↑w WriteNamedFile
Esc-↑m WriteModifiedFiles
Esc-} MarkUnmodified
↑x-b ToggleBackup
↑x-v ToggleVerifyWrite
↑x-l ToggleAutoLineFeed
↑x-c ChangeContext
↑x-↑i InsertFile
```

```
Esc-{ OpenBrace1
Esc-} CloseBrace1
Esc-  IndentFour
Esc-rarrow IndentFour
Esc-Ansi-c  IndentFour
Esc-↑h OutdentFour
Esc-larrow OutdentFour
Esc-Ansi-d OutdentFour
↑ SetMark
↑x-↑m SetMark
set-up SetMark
↑x-↑x ExchangeDotAndMark
Esc-i IndentRegion
↑x-↑r WriteRegion
↑x-↑k DeleteToKillBuffer
↑w DeleteToKillBuffer
↑x-g VisitFileNewBuffer
↑x-G VisitFileNewBuffer
↑x-d DeleteWindow
↑x-y YankToNewWindow
↑x-a RegionToNewWindow
↑x-m MergeWindows
↑x-1 GoToBuffer
↑x-2 GoToBuffer
↑x-3 GoToBuffer
↑x-4 GoToBuffer
↑x-5 GoToBuffer
↑x-6 GoToBuffer
↑x-7 GoToBuffer
↑x-3 GoToBuffer
↑x-9 GoToBuffer
↑x-o OrderBuffers
```

Several editor functions exist that are not bound to any key by the default definitions. These are the following:

OrderBufferBackwards
: Order the buffers in the opposite order of that used by OrderBuffers. Depending on whether a user prefers to stack their windows from lower left to upper right, or from lower right to upper left, one or the other of these ordering functions should be used.

OpenBrace
: C-program editing command. Generates three new lines, with the first and third line containing matching { and } braces indented two spaces from the original cursor position. The middle line is blank and indented four spaces from the original cursor position.

CloseBrace
: C-program editing command. The same as CloseBrace1, except that the second, blank line is not generated. The cursor is left after the } character.

BackwardHackingTabs
: Same as DeletePreviousCharacter except that tabs are expanded first if they are encountered. Thus, this command will convert a tab into 7 spaces instead of deleting the equivalent of 8 spaces worth of white space.

14.16.2. Specifying Options and Checkpoint Intervals

Various options and the checkpointing frequency (in number of editing actions) can be specified in the initialization file. These include:

```
(checkpoint number)
(defaultrows number)
(autolinefeed on/off)
(backup on/off)
(verifywrite on/off)
```

The *checkpoint* specification expects an integer number between 1 and 2^{32}-1. To turn off checkpointing specify some large number. The default is 500, which corresponds roughly to typing 10 lines of text. The *defaultrows* specification sets the default AVT size. The other specifications expect either an **on** or an **off**, indicating that the option should either be turned on or turned off.

The case of the keywords used is unimportant—everything gets converted to lower case before parsing anyway. However, the parser is unforgiving of extraneous blanks in the specification. No blanks are allowed between the parentheses and the keywords. (I know, this is easy to fix. It just hasn't been done yet.)

14.17. 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 generally drop into the debugger. At this point, the debugger command **Suicide,g** will destroy the process that got the exception. This will usually activate ved's crash recovery facility, signalled by 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 if the backup option has not been turned off, 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.

Remember that if the checkpointing feature was on when ved crashed, that there may be good copies of your files checkpointed in your directory. Take a look at them before you panic, you may end up only losing a few lines of text.

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.

14.18. Some Hints on Usage

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, especially number 9, which is **suicide.**

— 15 —
xlisp: An Experimental Object Oriented Language

This chapter is adapted from the document *XLISP: An Experimental Object Oriented Language*, Version 1.4, January 1, 1985, by David Betz, 114 Davenport Ave., Manchester, NH 03103.

15.1. Introduction

XLISP is an experimental programming language combining some of the features of LISP with an object oriented extension capability. It was implemented to allow experimentation with object oriented programming on small computers. There are currently implementations running on the PDP-11 under UNIX V7, on the VAX-11 under VAX/VMS and Berkeley VAX/UNIX, and on the 8088/8086 under CP/M-86 or MS-DOS. A version is currently being developed for the 68000 under CP/M-68K and for the Apple Macintosh. It is completely written in the programming language 'C' and is easily extended with user written built-in functions and classes. It is available in source form free of charge to non-commercial users. Prospective commercial users should contact the author for permission to use XLISP.

Many traditional LISP functions are built into XLISP. In addition, XLISP defines the objects 'Object' and 'Class' as primitives. 'Object' is the only class that has no superclass and hence is the root of the class heirarchy tree. 'Class' is the class of which all classes are instances (it is the only object that is an instance of itself).

This document is intended to be a brief description of XLISP. It assumes some knowledge of LISP and some understanding of the concepts of object oriented programming.

Version 1.2 of XLISP differs from version 1.1 in several ways. It supports many more Lisp functions. Also, many version 1.1 functions have been renamed and/or changed slightly to follow traditional Lisp usage. One of the most frequently reported problems in version 1.1 resulted from many functions being named after their equivilent functions in the C language. This turned out to be confusing for people who were trying to learn XLISP using traditional LISP texts as references. Version 1.2 renames these functions to be compatible with more traditional dialects of LISP. Version 1.3 introduces many new LISP functions and moves closer to the goal of being compatible with the Common Lisp standard. Version 1.4 introduces user error handling and breakpoint support as well as more Common Lisp compatible functions.

A recommended text for learning LISP programming is the book "LISP" by Winston and Horn and published by Addison Wesley. The first edition of this book is based on MacLisp and the second edition is based on Common Lisp. Future versions of XLISP will continue to migrate towards compatibility with Common Lisp.

15.2. A Note From the Author

If you have any problems with XLISP, feel free to contact me for help or advice. Please remember that since XLISP is available in source form in a high level language, many users have been making versions available on a variety of machines. If you call to report a problem with a specific version, I may not be able to help you if that version runs on a machine to which I don't have access. Please have the version number of the version that you are running readily accessible before calling me.

If you find a bug in XLISP, first try to fix the bug yourself using the source code provided. If you are successful in fixing the bug, send the bug report along with the fix to me. If you don't have access to a C

compiler or are unable to fix a bug, please send the bug report to me and I'll try to fix it.

Any suggestions for improvements will be welcomed. Feel free to extend the language in whatever way suits your needs. However, PLEASE DO NOT RELEASE ENHANCED VERSIONS WITHOUT CHECKING WITH ME FIRST!! I would like to be the clearing house for new features added to XLISP. If you want to add features for your own personal use, go ahead. But, if you want to distribute your enhanced version, contact me first. Please remember that the goal of XLISP is to provide a language to learn and experiment with LISP and object oriented programming on small computers.

15.3. XLISP Command Loop

When XLISP is started, it first tries to load "init.lsp" from the default directory. It then loads any files named as parameters on the command line (after appending ".lsp" to their names). It then issues the following prompt:

```
>
```

This indicates that XLISP is waiting for an expression to be typed. When an incomplete expression has been typed (one where the left and right parens don't match) XLISP changes its prompt to:

```
n>
```

where n is an integer indicating how many levels of left parens remain unclosed.

When a complete expression has been entered, XLISP attempts to evaluate that expression. If the expression evaluates successfully, XLISP prints the result of the evaluation and then returns to the initial prompt waiting for another expression to be typed.

Input can be aborted at any time by typing the CONTROL-G key (it may be necessary to follow CONTROL-G by RETURN).

15.4. Break Command Loop

When XLISP encounters an error while evaluating an expression, it attempts to handle the error in the following way:

If the symbol '*breakenable*' is true, the message corresponding to the error is printed. If the error is correctable, the correction message is printed. If the symbol '*tracenable*' is true, a trace back is printed. The number of entries printed depends on the value of the symbol '*tracelimit*'. If this symbol is set to something other than a number, the entire trace back stack is printed. XLISP then enters a read/eval/print loop to allow the user to examine the state of the interpreter in the context of the error. This loop differs from the normal top-level read/eval/print loop in that if the user types the symbol 'continue' XLISP will continue from a correctable error. If the user types the symbol 'quit' XLISP will abort the break loop and return to the top level or the next lower numbered break loop. When in a break loop, XLISP prefixes the break level to the normal prompt.

If the symbol '*breakenable*' is nil, XLISP looks for a surrounding errset function. If one is found, XLISP examines the value of the print flag. If this flag is true, the error message is printed. In any case, XLISP causes the errset function call to return nil.

If there is no surrounding errset function, XLISP prints the error message and returns to the top level.

15.5. Data Types

There are several different data types available to XLISP programmers.

- lists

- symbols
- strings
- integers
- objects
- file pointers
- subrs/fsubrs (built-in functions)

Another data type is the stream. A stream is a list node whose car points to the head of a list of integers and whose cdr points to the last list node of the list. An empty stream is a list node whose car and cdr are nil. Each of the integers in the list represents a character in the stream. When a character is read from a stream, the first integer from the head of the list is removed and returned. When a character is written to a stream, the integer representing the character code of the character is appended to the end of the list. When a function indicates that it takes an input source as a parameter, this parameter can either be an input file pointer or a stream. Similarly, when a function indicates that it takes an output sink as a parameter, this parameter can either be an output file pointer or a stream.

15.6. The Evaluator

The process of evaluation in XLISP:

- Integers, strings, objects, file pointers, and subrs evaluate to themselves
- Symbols evaluate to the value associated with their current binding
- Lists are evaluated by evaluating the first element of the list
    - If it evaluates to a subr, the remaining list elements are evaluated and the subr is called with these evaluated expressions as arguments.
    - If it evaluates to an fsubr, the fsubr is called using the remaining list elements as arguments (they are evaluated by the subr itself if necessary)
    - If it evaluates to a list and the car of the list is 'lambda', the remaining list elements are evaluated and the resulting expressions are bound to the formal arguments of the lambda expression. The body of the function is executed within this new binding environment.
    - If it evaluates to a list and the car of the list is 'macro', the remaining list elements are bound to the formal arguments of the macro expression. The body of the function is executed within this new binding environment. The result of this evaluation is considered the macro expansion. This result is then evaluated in place of the original expression.
    - If it evaluates to an object, the second list element is evaluated and used as a message selector. The message formed by combining the selector with the values of the remaining list elements is sent to the object.

15.7. Lexical Conventions

The following conventions are followed when entering XLISP programs:

Comments in XLISP code begin with a semi-colon character and continue to the end of the line.

Symbol names in XLISP can consist of any sequence of non-blank printable characters except the following:

```
( ) ' ` , " ;
```

Upper and lower case characters are distinct. The symbols 'CAR' and 'car' are not the same. The names of all built-in functions are in lower case. The names of all built-in objects are lower case with an initial capital. Symbol names must not begin with a digit.

Integer literals consist of a sequence of digits optionally beginning with a '+' or '-'. The range of values an integer can represent is limited by the size of a C 'int' on the machine that XLISP is running on.

Literal strings are sequences of characters surrounded by double quotes. Within quoted strings the '\' character is used to allow non-printable characters to be included. The codes recognized are:

\\	means the character '\'
\n	means newline
\t	means tab
\r	means return
\e	means escape
\nnn	means the character whose octal code is nnn

XLISP defines several useful read macros:

'<expr>	= = (quote <expr>)
#'<expr>	= = (function <expr>)
`<expr>	= = (backquote <expr>)
,<expr>	= = (comma <expr>)
,@<expr>	= = (comma-at <expr>)

15.8. Objects

Definitions:

selector	a symbol used to select an appropriate method
message	a selector and a list of actual arguments
method	the code that implements a message

Since XLISP was created to provide a simple basis for experimenting with object oriented programming, one of the primitive data types included was 'object'. In XLISP, an object consists of a data structure containing a pointer to the object's class as well as a list containing the values of the object's instance variables.

Officially, there is no way to see inside an object (look at the values of its instance variables). The only way to communicate with an object is by sending it a message. When the XLISP evaluator evaluates a list the value of whose first element is an object, it interprets the value of the second element of the list (which must be a symbol) as the message selector. The evaluator determines the class of the receiving object and attempts to find a method corresponding to the message selector in the set of messages defined for that class. If the message is not found in the object's class and the class has a super-class, the search continues by looking at the messages defined for the super-class. This process continues from one super-class to the next until a method for the message is found. If no method is found, an error occurs.

When a method is found, the evaluator binds the receiving object to the symbol 'self', binds the class in which the method was found to the symbol 'msgclass', and evaluates the method using the remaining elements of the original list as arguments to the method. These arguments are always evaluated prior to being bound to their corresponding formal arguments. The result of evaluating the method becomes the result of the expression.

Classes:

```
Object  THE TOP OF THE CLASS HEIRARCHY

    Messages:

        show    SHOW AN OBJECT'S INSTANCE VARIABLES
            returns     the object

        class   RETURN THE CLASS OF AN OBJECT
            returns     the class of the object

        isnew   THE DEFAULT OBJECT INITIALIZATION ROUTINE
            returns     the object

        sendsuper <sel> [<args>...] SEND SUPERCLASS A MESSAGE
            <sel>       the message selector
            <args>      the message arguments
            returns     the result of sending the message

Class   THE CLASS OF ALL OBJECT CLASSES (including itself)

    Messages:

        new     CREATE A NEW INSTANCE OF A CLASS
            returns     the new class object

        isnew [<scls>]  INITIALIZE A NEW CLASS
            <scls>      the superclass
            returns     the new class object

        answer <msg> <fargs> <code>     ADD A MESSAGE TO A CLASS
            <msg>       the message symbol
            <fargs>     the formal argument list
                          this list is of the form:
                            (<farg>...
                             [&optional <oarg>...]
                             [&rest <rarg>]
                             [&aux <aux>...])
                          where
                            <farg>   a formal argument
                            <oarg>   an optional argument
                                       (default is nil)
                            <rarg>   bound to the rest of the
                                       arguments
                            <aux>    a auxiliary variable
                                       (set to nil)
            <code>      a list of executable expressions
            returns     the object

        ivars <vars>    DEFINE THE LIST OF INSTANCE VARIABLES
            <vars>      the list of instance variable symbols
            returns     the object

        cvars <vars>    DEFINE THE LIST OF CLASS VARIABLES
            <vars>      the list of class variable symbols
            returns     the object
```

When a new instance of a class is created by sending the message 'new' to an existing class, the message

'isnew' followed by whatever parameters were passed to the 'new' message is sent to the newly created object.

When a new class is created by sending the 'new' message to the object 'Class', an optional parameter may be specified indicating the superclass of the new class. If this parameter is omitted, the new class will be a subclass of 'Object'. A class inherits all instance variables, class variables, and methods from its super-class.

15.9. Symbols

self	the current object (within a message context)
msgclass	the class in which the current method was found
oblist	the object list
keylist	the keyword list
standard-input	the standard input file
standard-output	the standard output file
breakenable	flag controlling entering the break loop on errors
traceenable	flag controlling trace back printout on errors and breaks
tracelimit	maximum number of levels of trace back information printed on errors and breaks
evalhook	user substitute for the evaluator function
applyhook	(not yet implemented)
unbound	indicator for unbound symbols

15.10. Evaluation Functions

```
(eval <expr>)  EVALUATE AN XLISP EXPRESSION
    <expr>      the expression to be evaluated
    returns     the result of evaluating the expression

(apply <fun> <args>)  APPLY A FUNCTION TO A LIST OF ARGUMENTS
    <fun>       the function to apply (or function symbol)
    <args>      the argument list
    returns     the result of applying the function
                  to the argument list

(funcall <fun> <arg>...)  CALL A FUNCTION WITH ARGUMENTS
    <fun>       the function to call (or function symbol)
    <arg>       arguments to pass to the function
    returns     the result of calling the function
                  with the arguments

(quote <expr>)  RETURN AN EXPRESSION UNEVALUATED
    <expr>      the expression to be quoted (quoted)
    returns     <expr> unevaluated
```

```
(function <expr>)  QUOTE A FUNCTION (THIS IS THE SAME AS QUOTE)
    <expr>      the function to be quoted (quoted)
    returns     <expr> unevaluated

(backquote <expr>)  FILL IN A TEMPLATE
    <expr>      the template
    returns     a copy of the template with comma and comma-at
                  expressions expanded (see the Common Lisp
                  reference manual)
```

15.11. Symbol Functions

```
(set <sym> <expr>)  SET THE VALUE OF A SYMBOL
    <sym>       the symbol being set
    <expr>      the new value
    returns     the new value

(setq [<sym> <expr>]...)  SET THE VALUE OF A SYMBOL
    <sym>       the symbol being set (quoted)
    <expr>      the new value
    returns     the new value

(setf [<place> <expr>]...)  SET THE VALUE OF A FIELD
    <place>     specifies the field to set (quoted):
                    <sym>                the value of a symbol
                    (car <expr>)         the car of a list node
                    (cdr <expr>)         the cdr of a list node
                    (get <sym> <prop>)   the value of a property
                    (symbol-value <sym>) the value of a symbol
                    (symbol-plist <sym>) the property list
                                           of a symbol
    <value>     the new value
    returns     the new value

(defun <sym> <fargs> <expr>...)  DEFINE A FUNCTION
```

```
(defmacro <sym> <fargs> <expr>...)  DEFINE A MACRO
    <sym>       symbol being defined (quoted)
    <fargs>     list of formal arguments (quoted)
                  this list is of the form:
                    (<farg>...
                     [&optional <oarg>...]
                     [&rest <rarg>]
                     [&aux <aux>...])
                  where
                    <farg>  is a formal argument
                    <oarg>  is an optional argument (default nil)
                    <rarg>  bound to the rest of the arguments
                    <aux>   is an auxiliary variable (set to nil)
    <expr>      expressions constituting the body of the
                function (quoted)
    returns     the function symbol

(gensym [<tag>])  GENERATE A SYMBOL
    <tag>       string or number
    returns     the new symbol

(intern <pname>)  MAKE AN INTERNED SYMBOL
    <pname>     the symbol's print name string
    returns     the new symbol

(make-symbol <pname>)  MAKE AN UNINTERNED SYMBOL
    <pname>     the symbol's print name string
    returns     the new symbol

(symbol-name <sym>)  GET THE PRINT NAME OF A SYMBOL
    <sym>       the symbol
    returns     the symbol's print name

(symbol-value <sym>)  GET THE VALUE OF A SYMBOL
    <sym>       the symbol
    returns     the symbol's value

(symbol-plist <sym>)  GET THE PROPERTY LIST OF A SYMBOL
    <sym>       the symbol
    returns     the symbol's property list
```

15.12. Property List Functions

```
(get <sym> <prop>)  GET THE VALUE OF A PROPERTY
    <sym>       the symbol
    <prop>      the property symbol
    returns     the property value or nil
```

```
(remprop <prop> <sym>)  REMOVE A PROPERTY
    <sym>       the symbol
    <prop>      the property symbol
    returns     nil
```

15.13. List Functions

```
(car <expr>)  RETURN THE CAR OF A LIST NODE
    <expr>      the list node
    returns     the car of the list node

(cdr <expr>)  RETURN THE CDR OF A LIST NODE
    <expr>      the list node
    returns     the cdr of the list node

(caar <expr>) == (car (car <expr>))
(cadr <expr>) == (car (cdr <expr>))
(cdar <expr>) == (cdr (car <expr>))
(cddr <expr>) == (cdr (cdr <expr>))

(cons <expr1> <expr2>)  CONSTRUCT A NEW LIST NODE
    <expr1>     the car of the new list node
    <expr2>     the cdr of the new list node
    returns     the new list node

(list <expr>...)  CREATE A LIST OF VALUES
    <expr>      expressions to be combined into a list
    returns     the new list

(append <expr>...)  APPEND LISTS
    <expr>      lists whose elements are to be appended
    returns     the new list

(reverse <expr>)  REVERSE A LIST
    <expr>      the list to reverse
    returns     a new list in the reverse order

(last <list>)  RETURN THE LAST LIST NODE OF A LIST
    <list>      the list
    returns     the last list node in the list
```

```
(member <expr> <list> [<key> <test>])  FIND AN EXPRESSION
                                         IN A LIST
    <expr>      the expression to find
    <list>      the list to search
    <key>       the keyword :test or :test-not
    <test>      the test function (defaults to eql)
    returns     the remainder of the list starting
                  with the expression

(assoc <expr> <alist> [<key> <test>])  FIND AN EXPRESSION
                                         IN AN A-LIST
    <expr>      the expression to find
    <alist>     the association list
    <key>       the keyword :test or :test-not
    <test>      the test function (defaults to eql)
    returns     the alist entry or nil

(remove <expr> <list> [<key> <test>])  REMOVE AN EXPRESSION
                                         FROM A LIST
    <expr>      the expression to delete
    <list>      the list
    <key>       the keyword :test or :test-not
    <test>      the test function (defaults to eql)
    returns     the list with the matching expressions deleted

(length <expr>)  FIND THE LENGTH OF A LIST
    <expr>      the list
    returns     the length of the list

(nth <n> <list>)  RETURN THE NTH ELEMENT OF A LIST
    <n>         the number of the element to return (zero origin)
    <list>      the list
    returns     the nth element
                  or nil if the list isn't that long

(nthcdr <n> <list>)  RETURN THE NTH CDR OF A LIST
    <n>         the number of the element to return (zero origin)
    <list>      the list
    returns     the nth cdr
                  or nil if the list isn't that long

(mapc <fcn> <list1>...<listn>)  APPLY FUNCTION
                                  TO SUCCESSIVE CARS
    <fcn>       the function or function name
    <list1..n>  a list for each argument of the function
    returns     the first list of arguments
```

```
(mapcar <fcn> <list1>...<listn>)  APPLY FUNCTION
                                    TO SUCCESSIVE CARS
    <fcn>       the function or function name
    <list1..n>  a list for each argument of the function
    returns     the list of values returned
                  by each function invocation

(mapl <fcn> <list1>...<listn>)  APPLY FUNCTION TO SUCCESSIVE CDRS
    <fcn>       the function or function name
    <list1..n>  a list for each argument of the function
    returns     the first list of arguments

(maplist <fcn> <list1>...<listn>)  APPLY FUNCTION
                                     TO SUCCESSIVE CDRS
    <fcn>       the function or function name
    <list1..n>  a list for each argument of the function
    returns     the list of values returned
                  by each function invocation

(subst <to> <from> <expr> [<key> <test>])  SUBSTITUTE EXPRESSIONS
    <to>        the new expression
    <from>      the old expression
    <expr>      the expression in which to do the substitutions
    <key>       the keyword :test or :test-not
    <test>      the test function (defaults to eql)
    returns     the expression with substitutions

(sublis <alist> <expr> [<key> <test>])  SUBSTITUTE
                                          USING AN A-LIST
    <alist>     the association list
    <expr>      the expression in which to substitute
    <key>       the keyword :test or :test-not
    <test>      the test function (defaults to eql)
    returns     the expression with substitutions
```

15.14. Destructive List Functions

```
(rplaca <list> <expr>)  REPLACE THE CAR OF A LIST NODE
    <list>      the list node
    <expr>      the new value for the car of the list node
    returns     the list node after updating the car

(rplacd <list> <expr>)  REPLACE THE CDR OF A LIST NODE
    <list>      the list node
    <expr>      the new value for the cdr of the list node
    returns     the list node after updating the cdr

(nconc <list>...)  DESTRUCTIVELY CONCATENATE LISTS
    <list>      lists to concatenate
    returns     the result of concatenating the lists
```

```
(delete <expr> <list> [<key> <test>])  DELETE AN EXPRESSION
                                         FROM A LIST
    <expr>      the expression to delete
    <list>      the list
    <key>       the keyword :test or :test-not
    <test>      the test function (defaults to eql)
    returns     the list with the matching expressions deleted
```

15.15. Predicate Functions

```
(atom <expr>)  IS THIS AN ATOM?
    <expr>      the expression to check
    returns     t if the value is an atom, nil otherwise

(symbolp <expr>)  IS THIS A SYMBOL?
    <expr>      the expression to check
    returns     t if the expression is a symbol, nil otherwise

(numberp <expr>)  IS THIS A NUMBER?
    <expr>      the expression to check
    returns     t if the expression is a symbol, nil otherwise

(null <expr>)  IS THIS AN EMPTY LIST?
    <expr>      the list to check
    returns     t if the list is empty, nil otherwise

(not <expr>)  IS THIS FALSE?
    <expr>      the expression to check
    return      t if the expression is nil, nil otherwise

(listp <expr>)  IS THIS A LIST?
    <expr>      the expression to check
    returns     t if the value is a list node or nil,
                nil otherwise

(consp <expr>)  IS THIS A NON-EMPTY LIST?
    <expr>      the expression to check
    returns     t if the value is a list node, nil otherwise

(boundp <sym>)  IS THIS A BOUND SYMBOL?
    <sym>       the symbol
    returns     t if a value is bound to the symbol,
                nil otherwise

(minusp <expr>)  IS THIS NUMBER NEGATIVE?
    <expr>      the number to test
    returns     t if the number is negative, nil otherwise
```

```
(zerop <expr>)  IS THIS NUMBER ZERO?
    <expr>      the number to test
    returns     t if the number is zero, nil otherwise

(plusp <expr>)  IS THIS NUMBER POSITIVE?
    <expr>      the number to test
    returns     t if the number is positive, nil otherwise

(evenp <expr>)  IS THIS NUMBER EVEN?
    <expr>      the number to test
    returns     t if the number is even, nil otherwise

(oddp <expr>)  IS THIS NUMBER ODD?
    <expr>      the number to test
    returns     t if the number is odd, nil otherwise

(eq <expr1> <expr2>)  ARE THE EXPRESSIONS IDENTICAL?
    <expr1>     the first expression
    <expr2>     the second expression
    returns     t if they are equal, nil otherwise

(eql <expr1> <expr2>)  ARE THE EXPRESSIONS IDENTICAL?
                         (WORKS WITH NUMBERS AND STRINGS)
    <expr1>     the first expression
    <expr2>     the second expression
    returns     t if they are equal, nil otherwise

(equal <expr1> <expr2>)  ARE THE EXPRESSIONS EQUAL?
    <expr1>     the first expression
    <expr2>     the second expression
    returns     t if they are equal, nil otherwise
```

15.16. Control Functions

```
(cond <pair>...)  EVALUATE CONDITIONALLY
    <pair>      pair consisting of:
                    (<pred> <expr>...)
                  where
                    <pred>      is a predicate expression
                    <expr>      evaluated if the predicate
                                is not nil
    returns     the value of the first expression whose predicate
                is not nil
```

```
(and <expr>...)  THE LOGICAL AND OF A LIST OF EXPRESSIONS
    <expr>...    the expressions to be ANDed
    returns      nil if any expression evaluates to nil,
                 otherwise the value of the last expression
                 (evaluation of expressions stops after the first
                  expression that evaluates to nil)

(or <expr>...)   THE LOGICAL OR OF A LIST OF EXPRESSIONS
    <expr>...    the expressions to be ORed
    returns      nil if all expressions evaluate to nil,
                 else the value of the first non-nil expression
                 (evaluation of expressions stops after the first
                  expression that does not evaluate to nil)

(if <texpr> <expr1> [<expr2>])  EXECUTE EXPRESSIONS CONDITIONALLY
    <texpr>      the test expression
    <expr1>      the expression to be evaluated
                   if texpr is non-nil
    <expr2>      the expression to be evaluated if texpr is nil
    returns      the value of the selected expression

(let (<binding>...) <expr>...)  BIND SYMBOLS AND
                                  EVALUATE EXPRESSIONS

(let* (<binding>...) <expr>...)  LET WITH SEQUENTIAL BINDING
    <binding>    the variable bindings each of which is either:
                 1)  a symbol (which is initialized to nil)
                 2)  a list whose car is a symbol and whose cadr
                         is an initialization expression
    <expr>...    the expressions to be evaluated
    returns      the value of the last expression

(catch <sym> [<expr>]...)  EVALUATE EXPRESSIONS AND CATCH THROWS
    <sym>        the catch tag
    <expr>...    expressions to evaluate
    returns      the value of the last expression or
                   the throw expression

(throw <sym> [<expr>])  THROW TO A CATCH
    <sym>        the catch tag
    <expr>       the value for the catch to return (default nil)
    returns      never returns
```

15.17. Looping Functions

```
(do ([<binding>]...) (<texpr> [<rexpr>]...) [<expr>]...)
(do* ([<binding>]...) (<texpr> [<rexpr>]...) [<expr>]...)
    <binding>   the variable bindings each of which is either:
                1)  a symbol (which is initialized to nil)
                2)  a list of the form: (<sym> <init> [<step>])
                    where:
                        <sym>  is the symbol to bind
                        <init> is the initial value of the symbol
                        <step> is a step expression
    <texpr>     the termination test expression
    <rexpr>...  result expressions (the default is nil)
    <expr>...   the body of the loop (treated like a prog)
    returns     the value of the last result expression

(dolist (<sym> <expr> [<rexpr>]) [<expr>]...)  LOOP THRU A LIST
    <sym>       the symbol to bind to each list element
    <expr>      the list expression
    <rexpr>     the result expression (the default is nil)
    <expr>...   the body of the loop (treated like a prog)

(dotimes (<sym> <expr> [<rexpr>]) [<expr>]...)  LOOP FROM ZERO
                                                  TO N-1
    <sym>       the symbol to bind to each value from 0 to n-1
    <expr>      the number of times to loop
    <rexpr>     the result expression (the default is nil)
    <expr>...   the body of the loop (treated like a prog)
```

15.18. The Program Feature

```
(prog (<binding>...) [<expr>]...)  THE PROGRAM FEATURE
(prog* (<binding>...) [<expr>]...)  PROG WITH SEQUENTIAL BINDING
    <binding>   the variable bindings each of which is either:
                1)  a symbol (which is initialized to nil)
                2)  a list whose car is a symbol and whose cadr
                        is an initialization expression
    <expr>      expressions to evaluate or tags (symbols)
    returns     nil or the argument passed to the return function

(go <sym>)  GO TO A TAG WITHIN A PROG CONSTRUCT
    <sym>       the tag (quoted)
    returns     never returns

(return [<expr>])  CAUSE A PROG CONSTRUCT TO RETURN A VALUE
    <expr>      the value (defaults to nil)
    returns     never returns
```

```
(prog1 <expr1> [<expr>]...)  EXECUTE EXPRESSIONS SEQUENTIALLY
    <expr1>      the first expression to evaluate
    <expr>...    the remaining expressions to evaluate
    returns      the value of the first expression

(prog2 <expr1> <expr2> [<expr>]...)  EXECUTE EXPRESSIONS
                                       SEQUENTIALLY
    <expr1>      the first expression to evaluate
    <expr2>      the second expression to evaluate
    <expr>...    the remaining expressions to evaluate
    returns      the value of the second expression

(progn [<expr>]...)  EXECUTE EXPRESSIONS SEQUENTIALLY
    <expr>...    the expressions to evaluate
    returns      the value of the last expression (or nil)
```

15.19. Debugging and Error Handling

```
(error <emsg> [<arg>])  SIGNAL A NON-CORRECTABLE ERROR
    <emsg>       the error message string
    <arg>        the argument expression
                   (printed after the message)
    returns      never returns

(cerror <cmsg> <emsg> [<arg>])  SIGNAL A CORRECTABLE ERROR
    <cmsg>       the continue message string
    <emsg>       the error message string
    <arg>        the argument expression
                   (printed after the message)
    returns      nil when continued from the break loop

(break [<bmsg> [<arg>]])  ENTER A BREAK LOOP
    <bmsg>       the break message string
                   (defaults to "**BREAK**")
    <arg>        the argument expression
                   (printed after the message)
    returns      nil when continued from the break loop

(errset <expr> [<pflag>])  TRAP ERRORS
    <expr>       the expression to execute
    <pflag>      flag to control printing of the error message
    returns      the value of the last expression consed with nil
                 or nil on error

(baktrace [<n>])  PRINT N LEVELS OF TRACE BACK INFORMATION
    <n>          the number of levels (defaults to all levels)
    returns      nil
```

```
(evalhook <expr> <ehook> <ahook>)  EVALUATE AN EXPRESSION
                                     WITH HOOKS
    <expr>      the expression to evaluate
    <ehook>     the value for *evalhook*
    <ahook>     the value for *applyhook*
    returns     the result of evaluating the expression
```

15.20. Arithmetic Functions

```
(+ <expr>...)  ADD A LIST OF NUMBERS
    <expr>...   the numbers
    returns     the result of the addition

(- <expr>...)  SUBTRACT A LIST OF NUMBERS
                 OR NEGATE A SINGLE NUMBER
    <expr>...   the numbers
    returns     the result of the subtraction

(* <expr>...)  MULTIPLY A LIST OF NUMBERS
    <expr>...   the numbers
    returns     the result of the multiplication

(/ <expr>...)  DIVIDE A LIST OF NUMBERS
    <expr>...   the numbers
    returns     the result of the division

(1+ <expr>)  ADD ONE TO A NUMBER
    <expr>      the number
    returns     the number plus one

(1- <expr>)  SUBTRACT ONE FROM A NUMBER
    <expr>      the number
    returns     the number minus one

(rem <expr>...)  REMAINDER OF A LIST OF NUMBERS
    <expr>...   the numbers
    returns     the result of the remainder operation

(min <expr>...)  THE SMALLEST OF A LIST OF NUMBERS
    <expr>...   the expressions to be checked
    returns     the smallest number in the list

(max <expr>...)  THE LARGEST OF A LIST OF NUMBERS
    <expr>...   the expressions to be checked
    returns     the largest number in the list

(abs <expr>)  THE ABSOLUTE VALUE OF A NUMBER
    <expr>      the number
    returns     the absolute value of the number
```

15.21. Bitwise Logical Functions

```
(bit-and <expr>...)  THE BITWISE AND OF A LIST OF NUMBERS
    <expr>       the numbers
    returns      the result of the and operation

(bit-ior <expr...)  THE BITWISE INCLUSIVE OR OF A LIST OF NUMBERS
    <expr>       the numbers
    returns      the result of the inclusive or operation

(bit-xor <expr...)  THE BITWISE EXCLUSIVE OR OF A LIST OF NUMBERS
    <expr>       the numbers
    returns      the result of the exclusive or operation

(bit-not <expr>)  THE BITWISE NOT OF A NUMBER
    <expr>       the number
    returns      the bitwise inversion of number
```

15.22. Relational Functions

The relational functions can be used to compare integers or strings. The functions '=' and '/=' can also be used to compare other types. The result of these comparisons is computed the same way as for 'eq'.

```
(< <e1> <e2>)  TEST FOR LESS THAN
    <e1>         the left operand of the comparison
    <e2>         the right operand of the comparison
    returns      the result of comparing <e1> with <e2>

(<= <e1> <e2>)  TEST FOR LESS THAN OR EQUAL TO
    <e1>         the left operand of the comparison
    <e2>         the right operand of the comparison
    returns      the result of comparing <e1> with <e2>

(= <e1> <e2>)  TEST FOR EQUAL TO
    <e1>         the left operand of the comparison
    <e2>         the right operand of the comparison
    returns      the result of comparing <e1> with <e2>

(/= <e1> <e2>)  TEST FOR NOT EQUAL TO
    <e1>         the left operand of the comparison
    <e2>         the right operand of the comparison
    returns      the result of comparing <e1> with <e2>

(>= <e1> <e2>)  TEST FOR GREATER THAN OR EQUAL TO
    <e1>         the left operand of the comparison
    <e2>         the right operand of the comparison
    returns      the result of comparing <e1> with <e2>
```

```
(> <e1> <e2>)  TEST FOR GREATER THAN
    <e1>        the left operand of the comparison
    <e2>        the right operand of the comparison
    returns     the result of comparing <e1> with <e2>
```

15.23. String Functions

```
(strcat <expr>...)  CONCATENATE STRINGS
    <expr>...   the strings to concatenate
    returns     the result of concatenating the strings

(strlen <expr>)  COMPUTE THE LENGTH OF A STRING
    <expr>      the string
    returns     the length of the string

(substr <expr> <sexpr> [<lexpr>]) EXTRACT A SUBSTRING
    <expr>      the string
    <sexpr>     the starting position
    <lexpr>     the length (default is rest of string)
    returns     substring starting at <sexpr> for <lexpr>

(ascii <expr>)  NUMERIC VALUE OF CHARACTER
    <expr>      the string
    returns     the ascii code of the first character

(chr <expr>)  CHARACTER EQUIVALENT OF ASCII VALUE
    <expr>      the numeric expression
    returns     a one character string
                  whose first character is <expr>

(atoi <expr>)  CONVERT AN ASCII STRING TO AN INTEGER
    <expr>      the string
    returns     the integer value of the string expression

(itoa <expr>)  CONVERT AN INTEGER TO AN ASCII STRING
    <expr>      the integer
    returns     the string representation of the integer value
```

15.24. Input/Output Functions

```
(read [<source> [<eof>]])  READ AN XLISP EXPRESSION
    <source>    the input source (default is standard input)
    <eof>       the value to return on end of file (default nil)
    returns     the expression read
```

```
(print <expr> [<sink>])  PRINT A LIST OF VALUES ON A NEW LINE
    <expr>      the expressions to be printed
    <sink>      the output sink (default is standard output)
    returns     nil

(prin1 <expr> [<sink>])  PRINT A LIST OF VALUES
    <expr>      the expressions to be printed
    <sink>      the output sink (default is standard output)
    returns     nil

(princ <expr> [<sink>])  PRINT A LIST OF VALUES WITHOUT QUOTING
    <expr>      the expressions to be printed
    <sink>      the output sink (default is standard output)
    returns     nil

(terpri [<sink>])  TERMINATE THE CURRENT PRINT LINE
    <sink>      the output sink (default is standard output)
    returns     nil

(flatsize <expr>)  LENGTH OF PRINTED REPRESENTATION USING PRIN1
    <expr>      the expression
    returns     the length

(flatc <expr>)  LENGTH OF PRINTED REPRESENTATION USING PRINC
    <expr>      the expression
    returns     the length

(explode <expr>)  CHARACTERS IN PRINTED REPRESENTATION
                    USING PRIN1
    <expr>      the expression
    returns     the list of characters

(explodec <expr>)  CHARACTERS IN PRINTED REPRESENTATION
                     USING PRINC
    <expr>      the expression
    returns     the list of characters

(maknam <list>)  BUILD AN UNINTERNED SYMBOL FROM
                   A LIST OF CHARACTERS
    <list>      list of characters in symbol name
    returns     the symbol

(implode <list>)  BUILD AN INTERNED SYMBOL FROM
                    A LIST OF CHARACTERS
    <list>      list of characters in symbol name
    returns     the symbol
```

15.25. File I/O Functions

```
(openi <fname>)  OPEN AN INPUT FILE
    <fname>      the file name string
    returns      a file pointer

(openo <fname>)  OPEN AN OUTPUT FILE
    <fname>      the file name string
    returns      a file pointer

(close <fp>)  CLOSE A FILE
    <fp>         the file pointer
    returns      nil

(read-char [<source>])  READ A CHARACTER FROM A FILE OR STREAM
    <source>     the input source (default is standard input)
    returns      the character (integer)

(peek-char [<flag> [<source>]])  PEEK AT THE NEXT CHARACTER
    <flag>       flag for skipping white space (default is nil)
    <source>     the input source (default is standard input)
    returns      the character (integer)

(write-char <ch> [<sink>])  WRITE A CHARACTER TO A FILE OR STREAM
    <ch>         the character to put (integer)
    <sink>       the output sink (default is standard output)
    returns      the character (integer)

(readline [<source>])  READ A LINE FROM A FILE OR STREAM
    <source>     the input source (default is standard input)
    returns      the input string
```

15.26. System Functions

```
(load <fname> [<vflag> [<pflag>]])  LOAD AN XLISP SOURCE FILE
    <fname>      the filename string (".lsp" is appended)
    <vflag>      the verbose flag (default is t)
    <pflag>      the print flag (default is nil)
    returns      the filename

(gc)  FORCE GARBAGE COLLECTION
    returns      nil

(expand <num>)  EXPAND MEMORY BY ADDING SEGMENTS
    <num>        the number of segments to add
    returns      the number of segments added
```

```
(alloc <num>)  CHANGE NUMBER OF NODES TO ALLOCATE IN EACH SEGMENT
    <num>       the number of nodes to allocate
    returns     the old number of nodes to allocate

(mem)  SHOW MEMORY ALLOCATION STATISTICS
    returns     nil

(type <expr>)  RETURNS THE TYPE OF THE EXPRESSION
    <expr>      the expression to return the type of
    returns     nil if the value is nil, else one of the symbols:
                    SYM   for symbols
                    OBJ   for objects
                    LIST  for list nodes
                    SUBR  for subroutine nodes
                            with evaluated arguments
                    FSUBR for subroutine nodes with
                            unevaluated arguments
                    STR   for string nodes
                    INT   for integer nodes
                    FPTR  for file pointer nodes

(exit)  EXIT XLISP
    returns     never returns
```

—16—
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.

16.1. 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.[6] On the Sun-1, the Sun 3 Mbit Ethernet board and Excelan 10 Mbit Ethernet boards are supported as boot devices. On Sun-2s, the 3Com 10 Mbit Ethernet board and the built-in Ethernet interface of the Sun-2/50 are supported. The standard Sun-3 cpu card (not the 3/50) and MicroVaxen with DEUNAs are also 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 19). 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 "[sys]boot" context, unless they begin with a square bracket. In the latter case, the name inside brackets is taken as a machine name. 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 understood: "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 four versions of Vload in use at this writing.

16.1.1. 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 0x20000, and teams are loaded at 0x20000.

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

world V

---

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

team	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 Sun-1 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.

16.1.2. 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 may be removed in the future.

16.1.3. 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 reads the boot file name from the workstation's configuration file; ordinarily, Vload will be specified. Once loaded, Vload 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/sun2+ec
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.

For example, the command

```
b V team1-vgts [pescadero]/user/fred/mykernel.r
```

will load the installed version of team1-vgts as the first team, and a special version of the kernel from Pescadero.

If the workstation being booted has a disk or some other device that the PROM prefers over the Ethernet for booting, specify the boot device **ec()** immediately following the **b** in the boot command, and preceding the first argument. (Some older PROM revisions require **nd()** in place of **ec()**.

16.1.4. Sun-2/50 Ethernet

The Sun-2/50 version of Vload is identical to the 3Com version described above, except that the default kernel is Vkernel/sun50. The boot device name is **ie()**.

16.1.5. Sun-3 Ethernet

The Sun-3 version of Vload is also similar to the 3Com version. The default kernel is Vkernel/sun3+ie, and the boot device is **ie()**. Currently a Sun prom monitor bug requires one to power cycle Sun-3 workstations when rebooting. Our Sun salesman has told us that new proms may be available.

16.1.6. MicroVaxen

There are three switches on the back of the MicroVax CPU. One is obviously the console baud rate selector. The other two have semi-random icons and affect booting.

The flat switch, whose symbol is a triangle inscribed in a circle controls the halt button and auto-reboot. With the switch at the dot-in-circle position the halt button on the face of the CPU halts the machine and forces it into the monitor, leaving you with a >>> prompt. Remember to press it once more to take the machine out of the halt state. When in the other, circle-out-of-dot position, the halt button is disabled and any action which would cause a halt (such as a kernel halt instruction or a power failure) will cause a reset and auto-boot attempt.

The circular knob has three positions. The downward pointing arrow is the normal position. The outline of a face causes the proms to prompt for language and keyboard type. The T-in-circle is a test position.

The commands for booting a MicroVax are:

b Boot according to the config file specifications.

b/1 Boot into the V world.

b/2 Boot into the xV world.

b/3 Boot into V, but prompt for the kernel and first team.

b/4 Boot into xV, but prompt for the kernel and first team.

If the disk drives are enabled then **b xqa0** forces the bootstrap to load over the network.

16.2. Netwatch

netwatch is a standalone tool for examining packets as they are spewn accross the ethernet. It has knowledge of many different protocol formats, including V, IP, XNS, Chaos, and PUP. It maintains packet buffers seperate from those of the ethernet hardware for maintaining packet traces.

We have found this to be the most powerful tool we have for debugging all nature of network protocol and distributed program communication bugs. This includes typical V distributed applications as well as protocol implementations (such as IP/TCP) on other hosts on our networks. There's nothing like silencing a roomfull

of random conjecture with a packet trace printout. The utility of looking at what's actually on the wire cannot be overemphasized.

16.2.1. Booting

netwatch runs standalone, so it must be booted fresh on a bare machine. A typical boot command to fire up the 3com version on a Sun-2 is:

```
b V netwatch-ec2 #
```

The sharp sign tells netwatch to load the first argument at the kernel start address and not to load a first team. See 16.1 for the details on booting other hardware configurations. Other versions of Vload supported are: **netwatch-en** (Sun-1/3Mb), **netwatch-ec** (Sun-1/3Com), **netwatch-ec2** (Sun-2/3Com), **netwatch-50** (Sun-2/50), and **netwatch-ie3** (Sun-3/75).

16.2.2. Operation

The standard train of events is to set up the packet filters, then commence recording packets until a certain event has occured. When recording, packets which pass through the filter are stored in a 127 buffer fifo queue. After recording the queue can be examined and/or written to a file. One may authenticate the **netwatch** process, which runs initially as UNKNOWN. If your storage server allows the unknown user to write to /tmp this may not be needed.

16.2.3. Commands

The commands available at the top level are:

h	Modify host address filter (see 16.2.4.1).
r	Receive packets into buffer (flushes current buffer).
t	Same as r, but prints packets as they are received (may drop packets).
b	Display buffer contents.
s	Same as b, but allows skipping of initial packets.
w	Write current buffer to a file.
l	Login (authenticate).
c	Change default directory for file writing.
!	Print an exclmation mark when a packet is received.
m	Always display the annoying option menu.
q	Quit.
?	Print a list of commands along with the current flag status

16.2.4. Filtering by Packet Type

netwatch understands several protocols, and can filter out packets based on the type field in the packet header. The packet types understood currently are: V, ARP & RARP, Chaos (Symbolics), IP, PUP and XNS.

16.2.4.1. 10meg Address Filter

The ten megabit packet filter is composed of two lists, the primary and secondary host lists. A packet is passed through the filter if its source and destination addresses can be found, one in each list. Ten megabit host addresses are specified using the last four hex digits of the ethernet address. At startup, the primary list is

empty and the secondary list is full (contains all addresses) with multicast turned off. **Note:** At the time of this release the netwatch driver for the Intel 85286 chip (Sun-2/50 and Sun-3) randomizes the first short of the destination address, so filtering on multicast packets doesn't work on those versions. In basic operation, one fills the secondary list with all addresses, and enters the addresses of "interesting" hosts into the primary list. Another typical use, when trying to debug communications between two hosts, is to have the two hosts in the primary list and all but one host (usually the the fileserver) in the secondary list.

16.2.4.2. 3meg Address Filter

The 3 megabit host address filter maintains one list of hosts, and filters in one of two modes. In the first mode, **AND** mode, both the source and destination addresses must be in the list. In the second mode, **OR** mode, only of the source or destination must be present. Hosts addresses are entered in octal form. The entire eight bit address is used.

16.3. Postmortem

The *Postmortem* diagnostic tool is no longer supported. Much of its functionality has been put into the kernel funtions **Aliens()** and **Processes()**. On Suns these functions can be called manually from the monitor using the **g <addr>** command. The address of the function can be gleaned from the kernel's symbol table with either **debug -o 2000 <kernel>** or **nm68**. These functions are not normally compiled into the MicroVax kernel.

16.4. 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 #
```

[7]

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)** the first sector of 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.

One should be aware of the fact that diskdiag's block size is the actual disk sector size, which may be different than the block size used by *fscheck* and the *storage server*.

---

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

Part II:
V Programming

— 17 —
Program Environment Overview

This chapter describes the execution environment provided for C programs written to run in the V-System. The 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, and to provide access to the distributed programming facilities provided by the V-System.

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 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. There has been a strong attempt to provide a superset of this environment for the V-System. Attempts to port C programs have resulted in a slightly more portable program environment than originally used with UNIX. The functions included in the V program environment for C, excluding V and workstation-specific routines, constitute our proposal for 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 workstation-specific, because they are not necessary in standard UNIX on, say, a Vax. For example, the long division routines are in this category.
- Functions that are particular to the V-System, like Create() and Ready().

17.1. Groups of Functions

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

exec	Functions related to the V executive.
fields	Functions that enable an AVT to be used as a menu, similar to a data entry terminal.
io	Input/output related routines.
locking	Routines providing locking for processes in a single team.

math	Numeric and mathematical functions.
mem	Memory management and allocation routines.
naming	Name management functions.
process	Process service functions and V kernel traps.
program	Program execution functions.
user interface	User interface routines
others	Miscellaneous other functions, such as string manipulation and time services.

Subsequent chapters discuss each group of functions in greater detail.

17.2. 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.

Vauthenticate.h	Message formats and definitions for the authentication server.
Vdirectory.h	Defines standard context directory entry formats and message types.
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 Internet server instead.
Vexec.h	Definitions for communication with the exec server.
Vexceptions.h	Exception types and exception request format.
Vfont.h	Standard internal bitmap and font format.
Vgroupids.h	V well-known or static group identifiers.
Vgtp.h	Virtual graphics terminal protocol definitions and message formats. Must be known by the VGTS and stub routines that talk to it, but is not needed by ordinary VGTS applications.
Vgts.h	Virtual graphics terminal server interface. This should be included in any programs that do graphics.
Vikc.h	Manifests and constants relating to the V Interkernel Protocol.
Vinfo.h	Definitions and structs for InfoBase access.
Vinfobuild.h	Structs for building InfoBase.
Vinfoparse.h	Definitions and structs for InfoBase scanner/parsers.
Vio.h	I/O Protocol header file. Types and mode constants for file manipulation functions described in chapter of this manual.
Vioprotocol.h	I/O Protocol message formats.
Vmachine.h	Machine-specific definitions.
Vmigrate.h	Migration-specific definitions.
Vmouse.h	Mouse device-specific header information. Most programs will use the VGTS to handle graphics input.
Vnaming.h	V-System naming manifests, types, and structures.
Vnet.h	Network server definitions. This is included in any programs that use the network.

Vpipe.h	Definitions and structures for V pipes.
Vprocess.h	Processor state structure and other process-specific header information.
Vquerykernel.h	QueryKernel operation manifests and types.
Vserial.h	Manifests for the serial lines.
Vsession.h	Manifests and message structs for the V/UNIX server.
Vspinlock.h	Definitions for spin locks, a cheap mechanism for locking within a team.
Vstorage.h	Definitions and message formats for the V storage server.
Vteams.h	Team header file. Structures used to communicate with the team server and to pass information to teams when they are created.
Vtermagent.h	Information shared by terminal agents and their clients.
Vtime.h	Structures used in time services, primarily for getting time from a session server.

— 18 —
Program Construction and Execution

A V-System C program is constructed and executed similar to a C program on UNIX. Only the differences are discussed here.

18.1. Writing the C Program

An application program on the V-System starts to execute as a single process on a new team. By default, the process is allocated an initial stack area of about 4000 bytes, just above its uninitialized data segment. If this is not large enough (or is larger than necessary), the declaration

```
int RootStackSize = newsize;
```

can be used to set the initial stack size to *newsize* bytes.

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>
```

Programs that use the V input/output library usually need the file **Vio.h**, which corresponds to the UNIX header file **stdio.h**.[8] Other V header files, listed in the previous chapter, are included similarly.

18.2. Compiling and Linking

When an application program is compiled and linked, references to kernel operations and other standard routines must be resolved by searching the library file **libV.a**. Its entry point must be the **_start** routine found in the library, and it must be relocated correctly for the target machine it is to run on. These defaults are automatically selected with the -vV option of the cc68 or ccvax command. The compile command:

```
cc68 -vV programfile.c -o outputfile.m68k
```

or for the Vax,

```
ccvax -vV programfile.c -o outputfile.vax
```

produces an executable file for running with the kernel. The program environment provided by the **libV.a** library is described in the remaining chapters of this part of the manual.

---

[8] In fact, a V program may include **stdio.h** in place of **Vio.h**, since there is a V version of **stdio.h** that simply includes **Vio.h**.

18.3. Program Execution

There are three models for executing V C programs, namely:

1. running them in "bare kernel mode", that is, directly on top of the kernel
2. running them from an executive
3. running them as a subprogram of another program

18.3.1. Bare Kernel Mode

When a program runs in bare kernel mode, none of the standard servers are available, unless the program includes one or more of them itself (as described in Chapter 31). A program written to run in bare kernel mode begins execution at a procedure called **main()**. No arguments may be passed to the program.

A program to be executed in bare kernel mode is loaded by a special loader program called **Vload**. For example, on an SMI workstation:

```
b Vload
```

typed to the PROM monitor causes it to load and execute the loader, which immediately prompts for the name of the file containing the program. The use of this loader is described more fully in Chapter 16.

18.3.2. Execution With the Executive

Use of the V executive is described in Chapter 3. 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, program execution again begins at a procedure called **main()**. This time, however, 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, are passed to the procedure. Moreover, the program is passed standard input, output, and error files, and a variety of other information through the **TeamRoot** message (described below in Section 18.4).

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");
  }
```

18.3.3. Subprograms

A program may run another program as a subprogram by invoking the same library functions employed by the executive. These functions are described in Chapter 28.

18.4. Program Initialization

Along with its compiler-generated code and data segments, a newly created program requires some additional run-time data about its environment before it can start execution. This data includes:

- File instances for standard input, output, and error output, and associated flags.
- Command-line arguments (argv and argc).
- Initial values of environment variables.
- Initial contents of the name cache.
- Initial naming context (working directory).

A program's creator passes this information to the new program in the *team root message* used to start its execution, extended by a *team environment block* of machine-independent format placed in the new team space by the creator. This information is subsequently unpacked by an initialization routine (described below) automatically linked in with the new team.

The format of the team root message and team environment block are given below, using a C-like notation with the following extensions:

- The notation **char s[]** (an array of characters of unspecified size) means a null-terminated string.
- The keyword **repeat** means that the following bracketed structure may be repeated zero or more times.
- The notation **align** *n* means to insert null bytes to align the next field to an address evenly divisible by *n*.

and the following common definitions:

```
typedef Bit32 unsigned long;     /* unsigned 32-bit quantity */
typedef Bit16 unsigned short;    /* unsigned 16-bit quantity */
```

18.4.1. The Team Root Message

The team root message is sent to the new program just prior to its beginning execution (details below). It is formatted as follows:

```
typedef struct
  {
#ifdef LITTLE_ENDIAN
    SystemCode  replycode;
    Bit16       rootflags;              /* Flags; see below */
    InstanceId  stdinfile;              /* Standard I/O instance ids */
    InstanceId  stdoutfile;
    InstanceId  stderrfile;
    Bit16       reserved1;
#else LITTLE_ENDIAN
    Bit16       rootflags;
    SystemCode  replycode;
    InstanceId  stdoutfile;
    InstanceId  stdinfile;
    Bit16       reserved1;
    InstanceId  stderrfile;
#endif LITTLE_ENDIAN
    ProcessId   stdinserver;            /* Standard I/O servers */
    ProcessId   stdoutserver;
    ProcessId   stderrserver;
    Bit32       reserved2;              /* Reserved for expansion of
                                         *  InstanceId to 32 bits */
    TeamEnvironmentBlock *teb;
  }
RootMessage;
```

```
/* Root flags - bit assignments */
#define RELEASE_INPUT       0x0010    /* Release stdin on close */
#define RELEASE_OUTPUT      0x0020    /* Release stdout on close */
#define RELEASE_ERR         0x0040    /* Release stderr on close */
#define STDOUT_APPEND       0x0001    /* Open stdout for append */
#define STDERR_APPEND       0x0080    /* Open stderr for append */
```

18.4.2. The Team Environment Block

The team environment block is formatted as follows:

```
typedef struct
  {
    align 4;
    Bit32 blocksize;    /* Total size of block in bytes */
    Bit32 argc;         /* Number of arguments */
    repeat
      {
        char arg[];
      }
    args;

    align 4;
    Bit32 envc;         /* Number of environment variables */
    repeat
      {
        char name[];
        char value[];
      }
    env;

    align 4;
    Bit32 cachec;       /* Number of cache entries */
    repeat
      {
        align 4;
        ContextPair from;
        ContextPair to;
        Bit16 flags;
        char name[];
        char truename[];
      }
    cache;

    align 4;
    ContextPair ctx;    /* Initial naming context: identifier */
    char ctxname[];     /* Initial naming context: absolute name */
    align 4;
  }
TeamEnvironmentBlock;
```

Note that the team environment block, despite containing variable-length fields, can be unambiguously parsed left-to-right.

The following library routine is available for creating team environment blocks:

```
SystemCode SetUpEnvironment(pid, args, env, cache, where)
    ProcessId pid;
    char *args[];
    EnvironmentVariable *env;
    NameCache *cache;
    char *where;
```

Constructs a team environment block for the specified team, using the given argument vector, environment

variable chain, and name cache, and the caller's current working context. The team environment block is deposited in the new team space at address "where" (rounded up as necessary for alignment).

18.4.3. The Per-Process Area

In addition to sharing the team environment variables (extracted from the team root message and team environment block), each process on a team has a region of team memory reserved for its own use, called its *stack space*. A portion of the stack space, called the *per-process area*, is used to store a few process-global variables. (On the Sun and VaxStation, a process's stack grows downward from the highest address in its stack space, and the per-process area begins at its lowest address.) A team-global variable called **PerProcess** points to the per-process 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.

ctx A ContextPair structure giving the context identifier of the process's current naming context (working directory).

stackSize The size of the process's stack space, in bytes.

ctxname A character string giving the absolute name of the process's current naming context. These strings are allocated on the heap (i.e., by **malloc()**) and are freed by the **ChangeDirectory()** library routine.

env A pointer to the list head for this process's environment variable list.

namecache A pointer to the list head for this process's name cache.

A newly created process on the same team as its parent (if created with the standard **Create()** library routine) inherits a copy of its parent's per-process area, with the exception of the **stackSize** field, which is specified as a parameter to **Create()**, and the **ctxname** pointer, which is modified to point to a fresh copy of the string to avoid the need for reference counting such strings. Thus, each child process inherits its creator's standard I/O, current naming context, name cache, and environment variables.

18.4.4. Initialization Procedure

A new program is created in the *awaiting reply* state, waiting for a (reply) message from its creator. In effect, the kernel simulates a **Send** from the initial process of the program to its creator, in response to which the creator must **Reply** before the program can begin execution. Prior to replying, the creator has access to the new team's entire address space and uses **CopyTo()** to deposit the team environment block of the new program.

> Note: The creator is responsible for passing the 16- and 32-bit fields in the team environment block in the correct byte order for the new team's host machine. This is in accord with the general convention that senders of messages use their native byte order, with receivers being responsible for byte-swapping if necessary. In this case, the new team is logically the sender in its initial transaction.

Finally, the creator invokes **Reply()** to set the new program running; the reply message constitutes the team root message. (See Chapter 27 for details of the interprocess communication primitives used.)

Meanwhile, the new program is blocked in the middle of its initialization code. This code is structured as a small assembly-language routine containing the team's initial entry point (**_start**), plus a machine-independent routine **TeamRoot()**, called from **_start**. **_start** initializes the stack, receives the team root message, zeros the initial data segment (bss) and then calls **TeamRoot()** to do the rest of the

initialization. **TeamRoot()** "unpacks" the team root message and team environment block, placing pieces of data in global variables or on the stack or heap as required by the host machine and programming language. The team environment block is discarded after having been unpacked; generally, the memory it occupied is reused as stack space. When **TeamRoot()** returns, **_start** calls **main()**, the main function of the C program being executed. Finally, if **main()** ever returns, **_start** calls **exit()** with the value returned by **main()**.

— 19 —
The V-System Configuration Database

The V-System maintains a configuration database, containing information about each workstation. The information is organized as sets of keyword/value pairs, one set per workstation.

19.1. Querying the Database

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.

19.2. 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.

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 and 3Com Ethernet interfaces.

bootfile — File to be loaded by ndserver or mvaxbootserver. Defaults to compiled-in /usr/V/boot/Vload10.d or /usr/V/boot/Vload.vax respectively. The former is appropriate for a Sun-2 or Sun-1.5 with 3Com Ethernet interface. Should be an absolute pathname.

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 the 10 Mbit Ethernet, 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 [sys] context. If this keyword is omitted, the Internet server will not forward datagrams through any gateways, i.e., only local traffic will be supported.
boot	Controls whether the boot server (ndserver for Sun Network Disk protocol, mvaxbootserver for MicroVAXes with DEC MOP protocol) will respond to boot requests from this workstation. The server will refuse to respond only if there is no config file (although mvaxbootserver has a "-n" flag to override this) or if the config file contains "boot:no". This field has no effect on Sun-3 RARP/TFTP booting.
ndboot	A synonym for "boot", used only by the ndserver. This is an historical relic and should vanish in the future. Any existing config files which use "ndboot" should be edited to use "boot".
kernel	Filename of the program to be loaded as the kernel, for use by Vload. The name is given relative to the standard [sys]boot context. If this keyword is omitted, Vload uses a compiled-in default.
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. This feature is not in V6.0.
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".
location	Optional location field.
comment	Used to put a comment in the file, such as a description of the workstation.

19.3. 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 present on every publically-available V storage server.[9]

The name of each workstation's configuration file is derived from its hardware Ethernet address (a

[9] Publically-available storage servers are defined as those that respond to GetPid(STORAGE_SERVER, ANY_PID) requests from nonlocal clients.

convenient unique identifier).[10] The files are kept in a subcontext named "config", under the server's [sys] context. 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 "[sys]config/C.02608c019954"; this is in fact how QueryWorkstationConfig() is implemented.

A configuration file is an ASCII text file, 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 "build install" (run as user *ds*) is used to install changes.

19.4. 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 "[sys]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.

[10] 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.

— 20 —
Control of Executives

Instances of the V executive, or command interpreter, are normally created and controlled directly by the user interacting with the system. However, this control is also available to programs through the following functions:

```
int CreateExec(execserver, inserver, infile, outserver, outfile,
                errserver, errfile, cpid, ccid, flags, execpid,
                error)
    ProcessId execserver;
    ProcessId inserver, outserver, errserver;
    InstanceId infile, outfile, errfile;
    ProcessId cpid;
    ContextId ccid;
    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 **CreateExec** is called. Context is specified by two parameters, a server pid and a context identifier relative to the given pid. 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 identifier, 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.

```
SystemCode QueryExec(execserver, execid)
    ProcessId execserver;
    int execid;
```

Inquire about the state of the specified exec. If successful, it returns a code of OK, and the following information: in **execpid** 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.

— 21 —
Fields: Using an AVT as a Menu

These routines allow you to set up a table of *fields* in an AVT. 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 AVT */
    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 AVT the field begins. (row=1 and col=1 is the top left corner of the AVT.) **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 usage is:

```
field = GetField(...);
if (field) (*field->proc))(field->value);
```

or perhaps:

```
if (field) (*field->proc))(field);
```

format is discussed below.

21.1. 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][λ]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**. A type code of **S** (or **ls**) means that the whole input line (including spaces) should be accepted.

21.2. The Field Table as a Menu: Selecting an Action

```
Field * GetField(menu, menuLength, buttons, avt)
    Field *menu;
    int menuLength;
    short buttons;
    File *avt;          /* output AVT */
```

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.

21.3. 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, avt)
    Field *menu;
    int menuLength;     /* see GetField function */
    File *avt;          /* output AVT where fields are to be written */
```

Display in the **avt** 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**.

21.4. 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 AVT 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)**

21.5. 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, one of two fixes shown above 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.

21.6. Limitations

No facilities yet for arrays.

— 22 —
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.

22.1. Standard C I/O Routines

The following standard C I/O routines are available:

clearerr()	closedir()	fclose()	feof()
ferror()	fflush()	fgetc()	fgets()
fopen()	fprintf()	fputc()	fputs()
fread()	freopen()	fscanf()	fseek()
ftell()	fwrite()	getc()	getchar()
gets()	getw()	mktemp()	opendir()
printf()	putc()	putchar()	puts()
putw()	readdir()	rewind()	rewinddir()
scanf()	sprintf()	seekdir()	setbuf()
sscanf()	telldir()	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.

22.2. 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 chapter 33 for more details on the semantics of the various possible file types and modes.

22.3. V I/O Routines

22.3.1. 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 *OpenDuplex(file, mode, error)
        File *file; unsigned short mode; SystemCode *error;
```

Open the "other side" of a duplex file, such as a network connection or terminal. **Mode** and **error** are as in **Open()**.

```
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()**.

```
ProcessId 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 the process id of the first process that replied. If the create instance request was sent to a group, additional replies can be obtained using **GetReply()**.

```
SystemCode CreateDuplexInstance(server, id, mode, req)
        ProcessId server; InstanceId id; unsigned short mode;
        CreateDuplexInstanceRequest req;
```

Open the "other side" of the file specified by the **server** and **id**, but do not set up a File structure for it. A CreateDuplexInstanceReply is returned at the location pointed to by **req**. Return a standard system reply code.

22.3.2. 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 chapter 33 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.

22.3.3. Byte Mode Operations

The purpose of the byte-mode I/O library is to maintain an abstract view of a file instance as an array of bytes with a known (but extensible) length, and the ability to read, write, and (in the case of non-STREAMs) seek, at the byte level. A layer of buffering is imposed between the client and server maintaining the actual file instance, to reduce the amount of actual reading and writing done. The actual file instance is guaranteed to be identical with the local view when the file is first opened, and after a Flush, barring I/O errors. (Note that Close calls Flush before releasing the instance.) At most one block of the local view of the file instance may differ from the actual instance.

The I/O library can be used on any file type, though somewhat confusing results may be obtained with VARIABLE–BLOCK, non-STREAM files, particularly if one attempts to Seek in other than ABS–BLOCK mode.

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.

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;
```

Identical to **ClearEof()**.

```
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 local record of the last exception on the given file, and resynchronize the local view of the associated file instance with that of the server, including the size of the file and (for STREAMs) the next block to read. This function only clears the local record of the exception; it does not affect the circumstances that caused the exception to occur. See **Eof()**.

If the file is not of type STREAM, the contents of the local buffer are discarded even if the buffer was modified and not yet rewritten. If the file is of type STREAM, and the current file position violates the stream condition (always read **nextblock** or write **lastblock**+1), reposition. The contents of the local buffer are discarded only if repositioning is necessary.

```
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.

22.3.4. 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.

The number of bytes returned may be less than the number requested if (1) the file has the type attribute VARIABLE_BLOCK and a short block was being read, (2) end of file was encountered while reading, (3) an error occurred while reading (in this case 0 bytes are returned), or (4) more than one block was requested and either the file does not have the type attribute MULTI_BLOCK, or the server could not return as many blocks as were requested. If the read request cannot be satisfied, the reason is indicated by the standard reply code returned by **FileException()**. If the end of file is encountered while reading, a partial block is returned with the reply code END_OF_FILE. **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. If the number of blocks is unknown, MAXUNSIGNED is returned.

```
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.

22.3.5. 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 the 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.

22.3.6. 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**.

```
int unlink(pathname)
```

```
    char *pathname;
```

Remove (delete) the file specified by **pathname**. Returns 0 on success, -1 on failure. This interface is provided for UNIX compatibility.

```
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.

22.4. Portable binary integer I/O

The following routines can be used to write and read integers to a file in a standard binary representation. This representation stores integers in 1, 2, 3, or 4 bytes, signed or unsigned. The integers are written as binary numbers, 8 bits to a byte, in big-endian order. For the signed routines, twos-complement is used. (This convention is followed by a number of systems, including files read and written by TeX and METAFONT.)

The subroutines are declared as follows, for $N = 1, 2, 3,$ and 4:

```
PutSignedN(i, f)
    long i; File *f;

PutUnsignedN(i, f)
    long i; File *f;

long GetSignedN(f)
    File *f;

long GetUnsignedN(f)
    File *f;
```

— 23 —
Intra-Team Locking

The V kernel provides message-passing as a means of synchronizing processes, and mutual exclusion may be enforced by the use of a server process that executes the critical section for clients. Such an arrangement is not always suitable, however: for processes communicating via a shared data structure the overhead of a message exchange may exceed by an order of magnitude the cost of performing the critical section.

The V library includes support for cheap mutual exclusion among processes in a single team. *Spin locks* ensure mutual exclusion in the presence of contention, but in its absence they introduce very little overhead. Spin locks also maintain a count indicating the level of contention so that the programmer can continue to assess their suitability after they are in use.

Spin locks are essentially binary semaphores without queueing: when a process fails in an attempt to acquire a lock, it simply delays (instead of busy-waiting) before trying again ("spinning" the lock).

Advantages: In the absence of contention, spin locks are *fast.* The optimized macro forms require only from one to six machine instructions each; the procedure forms add only the cost of a single-argument procedure call.

Disadvantages: A process that fails in an attempt to acquire a lock delays one tick before trying again; the locking overhead in the presence of contention is therefore higher than it would be for a message exchange with a server process. Also, spin locks are not fair: the order in which processes acquire the lock is not determined by the order in which they begin their attempts.

Spin locks are best suited for cases in which contention is expected to occur only rarely. The repeated attempts at the lock render them less suitable when the lock is held for long periods of time (several clicks), and the delay period (one click) may be too long for some applications with real-time constraints.

Each spin lock maintains a *contention count*, incremented each time that a process is forced to delay in an attempt to acquire the lock. The counter is incremented without mutual exclusion; its value is therefore not guaranteed precise, but should still provide a rough indication of the level of contention.

The following subroutines are provided in the library, with needed definitions in Vspinlock.h.

```
AcquireSpinLock(lock)
    SpinLockType *lock;
```

Wait until the named lock is acquired before returning. The delay on failure is one click.

```
ReleaseSpinLock(lock)
    SpinLockType *lock;
```

Release the named lock.

Additionally, the macro `SpinLockCount(lock)` provides access to the contention count; it is of type `short integer` and may be assigned to as well as read.

Locks must be initialized to either `SpinLockLocked` or `SpinLockUnlocked`, which also set the contention count to zero.

More efficient macro forms of the locking operations are provided for the common cases of the lock being a

global variable or an argument to the procedure invoking the operation. The costs of these forms of the operations are in the range of one to six machine instructions. The compiler and lint, however, cannot properly check theses forms, which may result in either spurious error messages or failure to detect real errors.

```
AcquireGlobalSpinLock(lockName)
    SpinLockType lockName;
```

Equivalent to `AcquireSpinLock(&lockName)`, where `lockName` is the name of a global (extern) variable. (This will also work if the global variable `lockName` is a struct and the lock is its first component.)

```
ReleaseGlobalSpinLock(lockName)
    SpinLockType lockName;
```

Equivalent to `ReleaseSpinLock(&lockName)`, where `lockName` is the name of a global (extern) variable.

```
AcquireArgumentSpinLock()
```

Equivalent to `AcquireSpinLock(p)`, where `p` is the *first* argument of the containing procedure. (This will also work if the lock is the first component in a struct `*p`.)

```
ReleaseArgumentSpinLock()
```

Equivalent to `ReleaseSpinLock(p)`, where `p` is the first argument of the containing procedure.

— 24 —
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. These routines provide essentially the same functionality as the standard C library. The memory allocation routines are provided on a per-team basis.

Note that there is one pool of free storage for all processes in the team: when using the standard library versions, programmers must be careful to synchronize the processes allocating and freeing this storage. A set of memory management routines with internal locking for mutual exclusion is also available (see **lockedmalloc**, below). These routines run more slowly than the standard versions.

```
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 fast block transfer function. Transfers **count** bytes from **source** to **destination**. Returns **count**. Restriction: the source and destination blocks must not overlap.

```
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()**).

24.1. Use in multi-process teams

The standard library versions of the allocation and deallocation routines do not enforce exclusion among processes within a team; so disastrous things may happen if two or more processes access them simultaneously. A multi-process team may use the routines safely by enforcing its own exclusion (*e.g.*, by having all allocation/deallocation occur in a single process), or by explicitly linking in a provided version of these routines that does provide locking. The routines affected are malloc, realloc, free, calloc, cfree, and GiveToMalloc. (Note, however, that calls to these routines may be hidden in other standard library routines as well.) The locking version may be accessed using the compiler flag **-llockedmalloc**; to include it use, for example:

```
cc68 -V -r otherflags yourfile.o -llockedmalloc otherlibraries
```

This provides full exclusion for all of the routines mentioned, but at a execution-time penalty of up to about 25%.

— 25 —
Naming

The *naming* section of the library includes a number of functions that provide a convenient interface to V-System naming protocol messages, plus other naming-related services. See chapter 34 for an explanation of the naming protocol.

25.1. Current Context

Each process has a *current context* in which object names that do not begin with the root escape character ('[') are interpreted, similar to the *current working directory* of UNIX and other systems. The following functions are provided to query or reset the current context.

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

Change the current context (working directory) 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.

```
int chdir(name)
    char *name;
```

This function is identical to **ChangeDirectory()**, except that it returns 0 to indicate success or -1 to indicate failure. (This interface is provided for UNIX compatibility.)

```
char *getwd(pathname)
  char pathname[];
```

Copies the absolute name of the current context (working directory) into the given character array and returns its address. (This interface is provided for UNIX compatibility.)

25.2. Descriptor Manipulation

V-System servers generally maintain a *descriptor* for each of the objects they implement. Each descriptor contains a *type* field, the associated object's *name* (relative to a particular context), and additional type dependent information such as *size*, *timestamp*, *owner*, etc. The standard header file <Vdirectory.h> defines the descriptor types currently known to the system.

One can read (and in some cases modify) the descriptors of all objects defined in a given naming context by opening the associated *context directory* as a file. A context directory appears as a file of descriptors. A context directory can be opened using the standard system **Open()** routine with the additional bit FDIRECTORY specified as a part of the requested file mode. Context directories are ordinarily opened in FBLOCK mode and read using the standard **Read()** routine.

One caveat is necessary here: an attempt to open a multi-manager context directory in this way will

currently fail with the error code MORE_REPLIES, since such context directories are modelled as multiple files, one per manager. See section 34 for a description of the protocol used to reliably open all partitions of a multi-manager context, or the `listdir` program for a sample implementation.

One can also read or modify an individual object descriptor using the following functions:

```
SystemCode NReadDescriptor(name, desc)
    char *name;
    ArbitraryDescriptor *desc;
```

Read an object's descriptor, specifying the object by name.

```
SystemCode ReadDescriptor(serverpid, instanceid, desc)
    ProcessId serverpid;
    InstanceId instanceid;
    ArbitraryDescriptor *desc;
```

Read the descriptor of the object from which the specified instance was created.

```
SystemCode NWriteDescriptor(name, desc).
    char *name;
    ArbitraryDescriptor *desc;
```

Write an object's descriptor, specifying the object by name.

```
SystemCode WriteDescriptor(serverpid, instanceid, desc)
    ProcessId serverpid;
    InstanceId instanceid;
    ArbitraryDescriptor *desc;
```

Write the descriptor of the object from which the specified instance was created.

25.3. Local Names or Aliases

```
SystemCode DefineLocalName(localname, truename)
    char *localname, *truename;
```

Defines a local alias "`[localname]`" for "`truename`", which must be the name of a context. If `truename` does not begin with a square bracket, it is first mapped in the current context to get an absolute name before the alias is defined. The alias is local to the team defining it, and is inherited by teams it creates.

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

Undefines a local alias.

```
char *ResolveLocalName(name)
    char *name;
```

Returns the stored absolute name for the given local alias. The returned string should not be modified, and

will be freed if the name is later redefined, so beware.

ClearLocalNames()

Undefines all local aliases for this team. It may be useful to call **PrimeCache()** after calling this routine, to reinsert definitions for the system standard aliases.

SystemCode DefineTempArea()

If the local name [tmp] is not already defined, this function selects an appropriate place to store temporary files and defines [tmp] to point to it. The function returns OK if successful, else a standard system code describing the problem.

25.4. Naming Protocol Routines

```
ProcessId NameSend(req)
    NameRequest *req;
```

Sends off the given request message, with the destination determined by the name given in the message. The given name must be a null-terminated string; **NameSend()** neither examines nor sets **req->namelength**. Like **Send()**, **NameSend()** returns the pid of the replier, and modifies its argument to hold the reply message. **GetReply()** can be used if additional replies are anticipated.

```
SystemCode GetAbsoluteName(namebuf, namelength, context)
    char namebuf[];
    unsigned namelength;
    ContextPair *context;
```

Accepts a null-terminated name in namebuf, possibly a relative name or local alias, and modifies it to return the absolute name. The size of namebuf is passed in as namelength. If the name specified an existing context, its context identifier is returned, as with GetContextId(), otherwise context->pid is set to 0. The given name need not correspond to any existing object, as long as it is unambiguous what server would implement such an object if it did exist, and what its absolute name would be.

```
SystemCode GetFileName(namebuf, namelength, serverpid, instanceid)
    char namebuf[];
    unsigned namelength;
    ProcessId serverpid;
    InstanceId instanceid;
```

Returns the absolute name for the specified file instance in **namebuf**. The maximum name length is passed in **namelength**. **GetContextName()** returns OK if the mapping was successful, or a standard system error code if a failure occurred.

```
SystemCode GetContextId(name, context)
    char *name;
    ContextPair *context;
```

Interprets the given name in the current context, and returns a corresponding <process-id, context-id> pair, suitable for caching. 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. .

Callers should recognize that the ContextPair may become invalid at any time, usually due to the server that issued it crashing and restarting with a different pid.

```
SystemCode GetContextName(namebuf, namelength, context)
    char namebuf[];
    unsigned namelength;
    ContextPair context;
```

The inverse of **GetContextId()**. Returns the absolute name for the given context in **namebuf**. The maximum name length is passed in **namelength**. **GetContextName()** returns OK if the mapping was successful, or a standard system error code if a failure occurred.

```
int IgnoreRetry(req, pid, segbuf, segsize, serverpid)
  register MsgStruct *req;
  ProcessId pid, serverpid;
  register char *segbuf;
  register unsigned *segsize;
```

This routine is intended for use only by servers that implement the naming protocol, not for clients that use it. It determines whether the caller is one of the servers that should ignore the given request **req** (probably a CREATE_INSTANCE_RETRY), returning 1 (true) if so, 0 if not. It assumes there is a 0-terminated list of pids beginning at **req->segPtr** in the client's address space, and returns true if the given **serverpid** is on the list. If an appended segment was received, **segbuf** and ***segsize** should indicate its location and size. This routine may read in more of the segment; if so, it alters the **segsize** parameter to reflect what it read. The routine assumes **segbuf** points to an area of at least MAX_APPENDED_SEGMENT bytes.

25.5. Direct Name Cache Manipulation

The following routines are used internally by **NameSend()** and the local alias manipulation functions. They are not ordinarily called directly in user programs.

```
NameCacheEntry *NameCacheAdd(prefix, length, from, to, truename, flags)
    char *prefix, *truename;
    ContextPair from, to;
    unsigned short length, flags;
```

Adds a new entry to the name prefix cache and return a pointer to the new cache record. The cache record format is defined in the standard header file <Vnamecache.h>. If a cache entry already exists for the given prefix, it is deleted. Returns NULL if no memory is available to allocate a cache record.

The **prefix** argument gives the name prefix to be added, and **length** is its length (not counting the terminating null byte, if any). The prefix is interpreted relative to the **from** context and must name the **to** context.

The **flags** may be any combination of the following bits:

- DONT_FLUSH
  The cache entry will never be flushed, even if the specified context-pair becomes invalid.
- ALIAS
  The cache entry specifies a local alias. In this case, **prefix** is the alias, while **truename** is the absolute name to which the alias maps.

- LOGICAL_PID
  The process-id portion of the specified **to** context is a logical pid. **NameSend()** will perform a **GetPid()** with scope ANY_PID each time it attempts to use this cache entry. *NOTE: This feature is provided for backward compatibility with servers that implement the naming protocol of V version 5.1 and earlier. It will be removed in a future V release.*

```
NameCacheEntry *NameCacheLookup(name, context)
    char *name;
    ContextPair context;
```

Checks whether any prefix of the given name matches a cache entry. A pointer to the cache record containing the longest matching prefix is returned. If there is no match, NULL is returned. A prefix match is defined as all the characters in the prefix matching the corresponding characters in the given name, plus the given name containing a delimiter immediately following the match.

```
SystemCode NameCacheDelete(cacheEntry)
    NameCacheEntry *cacheEntry;
```

Deletes the specified name cache entry. NOT_FOUND is returned if **cacheEntry** does not point to a record currently contained in the cache; otherwise OK is returned.

PrimeCache()

Adds a standard set of well-known context names and aliases to the name cache. Normally called only once by the first team, but also useful after a call to **ClearLocalNames()**.

25.6. Environment Variables

The V-System implements character-string environment variables, much like those in UNIX. In V, a process may set variables in its own environment as well as reading environment variables inherited from its creator.

By default, environment variables are global to a team. The root process of a team begins with an environment variable list inherited from its creator (through its team environment block). A newly created process initially shares the environment variable list of its creator. A process may separate its environment variable list from that of its parent by allocating a new list head of type (EnvironmentVariable *), setting PerProcess->env to its address, and assigning it a value—typically either NULL, indicating an empty list, or the result of **copyenv(oldlist)** (see below).

```
char *getenv(var)
    char *var;
```

Returns the value of the given environment variable, or NULL if it is undefined. The returned string should not be modified.

```
setenv(var, value)
    char *var, *value;
```

Sets the given environment variable to the given value, or if value is NULL, sets the variable to be undefined.

```
EnvironmentVariable *copyenv(oldlist)
    EnvironmentVariable *oldlist;
```

Makes a fresh copy of the environment variable chain beginning with **oldlist**, and returns a pointer to the first entry. Useful if a process wants to separate its environment from its parent's. The **oldlist** argument should be the value (not the address) of the parent's environment list head.

```
clearenv()
```

Removes the definitions of all environment variables.

— 26 —
Numeric and Mathematical Functions

26.1. 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.

26.2. 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()
foot()      ceil()      exp()       log()
log10()     pow()       sqrt()
```

— 27 —
Processes and Interprocess Communication

The V kernel supports processes as abstractions, with operations for process management and interprocess communication. Several processes may share an address space on one host. Processes sharing an address space are collectively referred to as a *team.*

The V kernel also supports the concept of a process group. Any process may create a new group and processes may join or leave a group. Most functions that operate on a process also operate on a process group. This is achieved by specifying a group identifier in place of the process identifier. Thus, for example, to destroy all the processes in the group specified by **groupId**, the function **DestroyProcess(groupId)** can be invoked. (A single process can be viewed as a special process group: a group with just one member, with the process identifier serving as the group identifier.)

Similarly, messages may be sent to a group of processes, simply by addressing them to a group identifier. Typical usages of group communication are *notification* (e.g. to notify other processes of some event) and *query* (e.g. to locate a specific server). With local area networks providing broadcast and multicast facilities in hardware, group communication can be significantly more efficient than using repeated one-to-one communication.

The process and interprocess communication-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. These functions provide a convenient interface to kernel-provided services. Some of the functions execute kernel trap instructions, while others send messages to the kernel-server 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.

This chapter is divided into four sections: 1) process-related kernel operations, 2) other process-related functions, 3) process group related kernel functions and 4) interprocess communication-related functions.

27.1. Process-Related Kernel Operations

```
SystemCode ClearModifedPages(pid)
    ProcessId pid;
```

Clears the dirty bits for all pages in the address space in which the process specified by **pid** resides.

```
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 255 inclusive, with 0 the highest priority. (See the discussion on priorities with the description of **QueryProcessPriority()**.) **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, lhost)
    short priority;  char *initialpc, *initialsp; ProcessId lhost
```

Create a new team with initial or root process having the specified priority, initial program counter, and initial stack pointer. **lhost** specifies which (existing) logical host the new team should be placed into. A value of 0 specifies the logical host of the invoker.

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()**. **priority** must be a value between 0 and 255.

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

Warning: **CreateTeam()** will be restricted to the first team in the near future.

```
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.

```
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).[11] **DestroyProcess()** may also be used to destroy a process group by specifying a group identifier with **pid**. **DestroyProcess()** returns OK if **pid** if successful, else a reply code indicating the reason for failure. **DestroyProcess(0)** is suicide. If **pid** specifies a process group, then all processes in that group (and their descendants) are destroyed.

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

```
ProcessId GetObjectOwner(objectPid)
    ProcessId objectPid;
```

Return the process-id of the owner of the specifed object. Currently the only type of object supported is a

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

team. Thus **objectPid** must specify a process on the team whose owner is desired.

```
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.

ANY_PID Return a local or remote process.

If **logicalId** is ACTIVE_PROCESS, the pid of the invoking process is returned. If the scope is *any*, the kernel first looks for a locally registered process; if one is not found, the kernel broadcasts a request for a process identifier registered as this logical id to other workstations running the V kernel on the network. 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.

Note: **GetPid()** and **SetPid** are being phased out. New programs should use the group communication facility instead. The REMOTE_PID scope available in previous releases is no longer supported.

```
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.

```
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 <Vquerykernel.h>.

```
SystemCode QueryProcessorUsage(pid, usage, tusage)
    ProcessId pid; unsigned *usage, *tusage;
```

Return the time allocated so far by the processor to process **pid** in ***usage** and return total for the entire team in ***tusage** if **tusage** is non-zero. Time is returned in "clicks". If **pid** is zero, the time for the invoking process is returned. If **pid** is equal to the logical host name (LHN) shifted left by 16 bits, the time

allocated to the idle process is returned.

The function itself returns a replycode indicating success or otherwise.

```
unsigned short QueryProcessPriority(pid)
    ProcessId pid;
```

Returns the composite priority of the process with **pid**. A **pid** of zero specifies the invoking process. If **pid** does not exist, 0 is returned.

The 16 bit composite priority field of a process effectively consists of two concatenated 8 bit fields. The higher-order field contains the team priority; the lower-order field the process priority within the team. These are initialized when the processes are created (see **CreateProcess()** and **CreateTeam()**) and may be manipulated with **SetProcessPriority()** and **SetTeamPriority()**. The ready process with the lowest number in its composite priority field runs.

```
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 <Vprocess.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.

```
SystemCode ReturnModifiedPages(pid, buffer, bufferlen)
    ProcessId pid; unsigned buffer[256]; unsigned *bufferlen
```

Clears the *modified* bit of each page table entry for the team specified by **pid**. Returns the starting address of each page of memory in the team whose modified bit was on before being cleared. The size of the buffer may not be sufficient to contain the starting address of all dirty pages in the team's address space, although it should be sufficient for most. The operation does not clear the modified bit of any page whose starting address it cannot place in the invoker's buffer. Thus, the invoker need simply reinvoke **ReturnModifiedPages** to continue with the the operation if the return code indicates that more dirty pages might be outstanding. This is indicated with a return code of RETRY instead of OK. The last valid page address returned is delimited by a NULL word in the case where the buffer is not completely filled. **bufferlen** returns the number of modified page addresses returned in **buffer**.

```
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.

```
SetObjectOwner(objectPid, newOwner)
    ProcessId objectPid, newOwner;
```

Set the kernel-maintained owner of the object specified by **objectPid**. Currently the only type of object supported is a team. Thus **objectPid** must be the pid of a process on a particular team. Ownership of a team implies that unrestricted access to the team's address space using **MoveTo** and **MoveFrom** operations without having a process in the team first send a message to the owner.

```
SetPid(logicalid, pid, scope)
    int logicalid, scope;  ProcessId pid;
```

Associate **pid** with the specified logical id within the specified 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 locally.

ANY_PID Register the process globally.

The *local* scope is intended for servers serving only the local workstation. The *any* scope permits both local and remote access.

Note: **GetPid()** and **SetPid** are being phased out. New programs should use the group communication facility instead. The REMOTE_PID scope available in previous releases is no longer supported.

```
SystemCode SetProcessPriority(pid, priority, decay)
    ProcessId pid;  unsigned short priority,  unsigned decay;
```

Set the priority of process **pid** to **priority**. If **pid** is zero the priority of the invoking process is set to **priority**. **priority** may be any integer between 0 and 255. (See **QueryProcessPriority()** for a discussion of process priorities.) If **decay** is nonzero, the priority is incremented every **decay** "clicks" until it reaches 255.

```
SystemCode SetTeamPriority(pid, priority)
    ProcessId pid;  unsigned short priority;
```

Set the team priority of the team associated with **pid** to **priority**. If **pid** is zero the priority of the invoking process's team is set to **priority**. **priority** must be an integer between 0 and 255. (See **QueryProcessPriority()** for a discussion of team priorities.) Teams with **priority** 254 or 255 do not run.

SetTeamPriority() changes the absolute scheduling priority of each process on the team by modifying the team priority field of the composite priority for each process. This operation is intended for implementing macro-level scheduling and is restricted in use to the first team. Other teams should use **ChangeTeamPriority()** to request special scheduling service.

```
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.

```
int ValidPid(pid)
    ProcessId pid;
```

Return true (nonzero) if **pid** is a valid process or group 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 18.4.3).

27.2. Logical Host-Related Functions

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

Create a new team in a separate logical host with initial or root process having the specified priority, initial program counter, and initial stack pointer. This routine is the same as **CreateTeam** except that a new logical host is created for the new team.

```
SystemCode DestroyHost(pid)
    ProcessId pid;
```

Destroy the logical host in which the process specified by **pid** resides. When a process that is frozen is destroyed, any queue local IPC operations on it will be re-executed rather than returned with a failure status code. Also, queued reply messages will be forwarded rather than simply destroyed.

```
SystemCode ExtractHost(pid, optype, buffer, length)
    ProcessId pid; int optype; char *buffer; unsigned *length;
```

Extract the kernel descriptor information for the logical host in which the process **pid** resides and place it in the buffer pointed to by **buffer**. **length** specifies the size of buffer provided on invocation and returns the size of the descriptor information returned. This operation can only be invoked on local logical hosts. **optype** is either the manifest constant **QUERYHOSTCASE** or **EXTRACTHOSTCASE** (specifed in **Vmigrate.h**). The former indicates that only summary information should be returned on the number of processes, teams and memory used by the logical host. The is returned in a **HostResourcesRec** structure, as defined in **Vmigrate.h**. The latter optype indicates that the full kernel descriptor information for the

logical host should be returned. The full descriptor information returned is *not* intended to be interpreted outside the kernel and should only be used by the **TransferHost** operation to be reinstalled into another machine's kernel.

```
SystemCode FreezeHost(pid)
    ProcessId pid;
```

Freeze the logical host in which **pid** resides so that its address spaces and the kernel state associated with its teams and processes are not modified. This will cause all kernel operations on the logical host to be deferred and the priority of the teams in the logical host to be set to non-runnable.

```
SystemCode TransferHost(pid, buffer, length)
    ProcessId pid; char *buffer; unsigned length
```

Takes the kernel descriptor information generated by **ExtractHost** and installs it into an existing logical host that must have an equivalent number of teams already in it. Furthermore, the teams must have only a root process in them. The existing logical host is renamed by this operation to be the logical host specified in the descriptor information, as are all the teams in it. The result is a logical host that is equivalent to the one described in the descriptor information and the effective deletion of the "blank" logical host that was used as an installation base. The owner of the "new" logical host is the invoker of the operation.

```
SystemCode UnfreezeHost(pid)
    ProcessId pid;
```

Unfreeze the logical host in which **pid** resides. All deferred kernel server and IPC operations are executed and the priority of the teams in the logical host are set to their previous runnable values.

27.3. Other Process-Related 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(status)
    int status;
```

Terminate the execution of the team (i.e., program), after closing all open files. The **status** is sent to the creator of the team requesting termination. Thus, 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. This routine can also be called with parameters. If it is, the standard exception handler will interpret the first parameter as a pointer to a **printf**-style format string. The other parameters will be interpreted as values to be printed using that string. In an effort to keep the standard exception handler simple and robust, the number of **%s**'s in the format string must not exceed 8, nor may any of the strings (either the format string or strings to be printed) exceed 128 characters in length.

The format specifier **%z** is included in addition to the usual specifiers. **%z** will interpret its argument as a **SystemCode**, and print the result of running **ErrorString** with that code as its parameter.

```
ChangeTeamPriority(pid, priority)
    ProcessId pid;  short priority;
```

Set the priority of the team in which process **pid** resides to **priority**. If **pid** is 0 then the invoking process is implied. **priority** must be one of the manifest constants: REAL-TIME1 through REAL-TIME4 (of which REAL-TIME1 is the most privileged priority), FOREGROUND, BACKGROUND, GUEST, and STOP-TEAM-PRIORITY. The first team runs at priority REAL-TIME3, locally invoked foreground programs run at FOREGROUND, locally invoked concurrent (&) programs run at BACKGROUND, and remotely invoked programs run at GUEST priority. STOP-TEAM-PRIORITY makes the processes of a team nonrunnable. FOREGROUND, BACKGROUND, and GUEST programs are time-sliced in a round-robin scheme, with lower priority teams only getting the time slice if no higher priority teams exist. Management of team priorities is done by the team server, which uses the privileged **SetTeamPriority** kernel operation to actually change team priorities.

27.4. Process Group Operations

```
GroupId CreateGroup(initial_member, type)
    ProcessId initial_member; unsigned type;
```

Create a new group of the specified **type** (UNRESTRICTED_GROUP_BIT|LOCAL_GROUP_BIT) and make **initial_member** the first member. The invoking process is made the first member of the process group if **initial_member** is 0. If the UNRESTRICTED_GROUP_BIT is set in **type**, then any process may join the group, otherwise only processes of the same user may join. One can specify that only processes

on the same host as **initial_member** may join the group with the LOCAL_GROUP_BIT bit in **type**, thus allowing certain optimizations. **initial_member** may also specify a process group, in which case every member of **initial_member** becomes a member of the newly created group.

Returns the group id of the newly created group if successful, 0 otherwise.

```
SystemCode JoinGroup(groupId, pid)
    GroupId groupId; ProcessId pid;
```

Add the process or process group specified by **pid** to the process group **groupId**. Group **groupid** must exist. Well known groups are defined in the include file <Vgroupids.h>. If **pid** is 0, the invoking process is added to the group. If **pid** specifies a process group, every process of that group joins the group specified by **groupid**. Returns OK if successful, else a reply code indicating the reason for failure.

```
SystemCode LeaveGroup(groupId, pid)
    GroupId groupId; ProcessId pid;
```

pid leaves the process group with group id **groupid**. If **pid** is 0, the invoking process leaves the group. Again, **pid** may either specify a process or a process group.

```
SystemCode QueryGroup(groupId, pid)
    GroupId groupId; ProcessId pid;
```

Query the kernel to see if **pid** (the invoking process if **pid** is 0) would be allowed to join the group with the specified **groupId**. Returns OK if so, otherwise NO_PERMISSION if not allowed, DUPLICATE_NAME if already in, and NOT_FOUND if group does not exist (not at least one member located).

27.5. Interprocess Communication

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

Return true (nonzero) if **awaitingpid** is awaiting reply from **frompid**; otherwise return false.

```
SystemCode ForceException(pid)
    ProcessId pid;
```

Causes process **pid** to send an exception message to the exception server. The exception type is FORCEEXCEPTION.

```
SystemCode ForceSend(msg, fromPid, toPid)
    Message msg; ProcessId fromPid, toPid;
```

Force process **fromPid** to send **msg** to process **toPid**. **ForceSend**cannot be reinvoked on a process until the first invocation is terminated by replying to the process. I.e. there can only be at most a single **ForceSend** in effect for any given process.

```
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. (Namely, the **Send()** returns zero, and the replycode field of the reply message is set to BAD_FORWARD.)

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

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. If the last message received was sent to a process group, **Forwarder()** returns the group identifier the message was sent to.

```
ProcessId GetReply(msg, timeout)
    Message msg;  int timeout;
```

Returns the next reply message from a group **Send()** in **msg** and returns the process identifier of the replying process. If no messages are available within the timeout period, **GetReply()** returns 0. A typical message transaction thus consists of a **Send()** (which returns the first reply) followed by any number of **GetReply()**. However, all replies for a message transaction are discarded when the process sends again, initiating a new message transaction. (Note: Many library routines, such as **printf()** are implemented with message passing primitives, thus ending the last message transaction when they are called.) The timeout is given in clicks.

```
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.

Unless the invoker is the owner of the team in which **srcpid** resides, 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.

Unless the invoker is the owner of the team in which **srcpid** resides, 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.

```
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**. To determine if the message was sent to a process group see **Forwarder()**.

```
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**. (Note: This may be zero even if a segment is specified in the message.) Additional parts of the segment specified in the message may be transferred with **MoveFrom()**.

```
ProcessId ReceiveSpecific(msg, pid)
    Message msg;  ProcessId pid;
```

Suspend the invoking process until a message is available from the process **pid** or from a process in the process group specified by **pid**, returning the pid of this process, and placing the message in the array pointed to by **msg**.

If **pid** is not a valid process or group 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. Note: Messages that have been received but not replied to consume kernel resources until the receiver exits. Therefore, each process should invoke **Reply()** on every message it receives. If no reply is required, then **Reply()** should be invoked with a message whose replycode is set to DISCARD_REPLY. Such a reply message is not delivered to the sender, but releases kernel resources and allows the sender to (eventually) unblock (with a KERNEL_TIMEOUT error reply code if no replies were received at all).

```
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()**).

```
ProcessId Send(msg, pid)
    Message msg;  ProcessId pid;
```

If pid specifies a single process group, 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.)

If **pid** is a process group identifier, the message is sent to all processes in the group on a *best effort basis* and **Send()** blocks until a first process replies. The first reply message overwrites the original message. Further replies of the current message transaction may be received with **GetReply()**. Send initiates a new message transaction, effectively flushing all messages of the last transaction.

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. Some of high-order 8 bits within request and reply codes are assigned special meanings, and currently-unused bits within this subfield are reserved for future use. The bit names given below are defined in the standard header file <Venviron.h>.

REPLY_BIT is reset if a request message is being sent; set if a reply message.

SYSTEM_CODE is set 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.

Several other bits are interpreted with the following special meanings if the message is a request.

READ_BIT is set if read access is provided to a memory segment. If this bit is set, 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**.

WRITE_BIT is set if write access is provided to a memory segment. The segment is specified as described above. Both read and write access can be provided by setting both bits 4 and 5.

DATAGRAM_SEND_BIT
Experimentally, the V kernel currently supports the concept of real-time communication. In this mode, messages are communicated to a single process or a group of processes on a best effort basis. A process will only receive the message if it is receive-blocked waiting for it. The send operation does not block. Thus, one cannot reply to a real-time send. This type of communication is intended for situations, where, for example, a process continuously, in regular intervals, sends update information to a group. This mode of communication is specified by setting the DATAGRAM_SEND_BIT of the requestcode of the message.

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.

The following bits have special meaning in reply codes:

ANONYMOUS_REPLY_BIT
Reply as the forwarder of the message. This feature allows processes to join groups and reply to messages anonymously.

REPLY_SEGMENT_BIT
Reply segment has been specified. If this bit is set in a call to **ReplyWithSeg()**, the kernel interprets the last 2 words of the message as specifying a pointer to the start of the reply segment and the size in bytes of the segment, respectively.

IMMEDIATE_REPLY_BIT
Don't delay this reply, even if it is to a group send. If this bit is not set, replies to group sends are delayed slightly within the replying kernel to avoid swamping the sending kernel with back-to-back packets.

— 28 —
Program Execution Functions

This chapter describes a number of routines related to program execution. Included are routines for program loading and execution, selecting hosts for remote execution of programs, execution of Unix commands remotely on a Unix V server (see section 43), routines that provide compatibility with various Unix program execution routines, and other routines.

28.1. Program Execution

```
ProcessId LoadProgram(argv, hostSpec, rtMsg, path, concurrent, error)
    char *argv[];       /* Program arguments (including name). */
    SelectionRec *hostSpec;
                        /* Specifies the host to execute on.
                           (NULL => default, i.e. local host) */
    RootMessage *rtMsg;
                        /* Root message to use. NULL => default settings */
    char *path;         /* Search path to use for finding the program
                           file.  NULL indicates that the default should
                           be used. */
    int concurrent;     /* Specifies whether the program should be
                           owned by the system (concurrent = 1) or
                           by the user (concurrent = 0). */
    SystemCode *error;  /* Return code. */
```

LoadProgram() interacts with the team server on the specified host, to create a new team and load a program image into the new team space. The program is placed in a separate team and is set ready to run.

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 **hostSpec** argument is used to select a host to execute the new program. If **hostSpec** is NULL, the program is run locally. Alternatively, **hostSpec** can be a pointer to a *selection record*, as defined by the **SelectionRec** structure in **Vteams.h**. In this case, if the TEAMSERVERPID field of the selection record is non-zero, then this value is assumed to be a pid of a team server on the desired host. If, however, the **teamServerPid** field is zero, then an 'arbitrary' remote host is selected, according to the constraints specified in the other fields of the record. *NOTE: This method of host selection is likely to change in future releases of the system.*

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 *team environment block*, and some other information. The fields in the message are described in detail in section 18.4.1. If **rtMsg** is given as NULL, then a 'default' root message is used (see the description of the **DefaultRootMessage()** routine, below).

The **concurrent** argument specifies whether the team is to be "owned" by the process executing the **LoadProgram()** 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.

path specifies the *search path* that is used to locate the code file for the program that is to be executed. A search path is a character string consisting of a sequence of name prefixes separated by spaces. If **path** is NULL, then the value of the "PATH" environment variable is used instead, or, if the "PATH" environment variable is not set, the 'default' search path. This default search path is **"./ [bin]"**, which indicates that the program code file is searched for first in the invoker's current context, and then in the **[bin]** context. (Note that the search path mechanism also involves checking for machine-specific program name suffixes, as described in section 3.7.)

If the named program is not found, then the **fexecute** program is invoked instead, to attempt to execute the program remotely on the server that is providing the invoker's current context. See section 3.4 for further details.

Note: If **argv[0]** is an *absolute name* (that is, beginning with **'['**), then a search path is *not* used, nor is the *fexecute* program used in the case of a failed match.

LoadProgram() returns the pid of the new team's root process, or 0 to indicate an error. A standard system code is return 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 ExecProgram(argv, hostSpec, rtMsg, path, status, error)
    char *argv[];       /* Program arguments (including name). */
    SelectionRec *hostSpec;
                        /* Specifies the host to execute on.
                           (NULL => default, i.e. local host) */
    RootMessage *rtMsg; /* Root message to use. NULL => default settings */
    char *path;         /* Search path to use for finding the program
                           file. */
    int *status;        /* Return code from program executed or NULL
                           if the program is to be run concurrently. */
    SystemCode *error;  /* Return code. */
```

ExecProgram() is like **LoadProgram()**, except that it also starts the new team running (by replying to it). The arguments **argv**, **hostSpec**, **rtMsg**, **path**, and **error** are the same as for **LoadProgram()**.

If the **status** parameter is NULL then the program is run concurrently, otherwise the function waits until the program has terminated and returns its exit status in **status**.

```
Wait(pid, status)
    ProcessId pid;
    int *status;
```

Wait for the team whose root pid is specified by **pid** to expire, and then return its exit status code in the location pointed to by **status**.

```
DefaultRootMessage(rtMsg)
    register RootMessage *rtMsg;
```

This routine sets up the structure pointed to by **rtMsg** to be the 'default' **RootMessage** for any program that the invoker should load. In particular, the **stdin**, **stdout** and **stdin** servers and instance ids are set to be those of the invoker.

28.2. Host Selection

```
int QueryHosts(spec, descArray, numHosts, error)
    SelectionRec *spec; /* Host selection spec. */
    SelectionRec *descArray;
                        /* Array for returning descriptors of selected
                           hosts. */
    int numHosts;       /* Maximum number of selections to return.
                           Also the size of pidArray. */
    SystemCode *error;  /* Status code. */
```

Select a set of hosts for remote execution of programs. Nothing is actually executed -- this routine merely returns candidate hosts for remote execution. **QueryHosts()** returns descriptor records for hosts selected in **descArray** which meet the selection criteria specified by **spec**. At most **numHosts** selections are returned. The number of hosts actually selected is returned as the function value. **error** returns a system status code for the operation.

If **spec** is NULL then the default specification is used (see the description of **DefaultSelectionRec()**, below).

The format of a selection record is specified in **Vteams.h**. The **pid** field of a **SelectionRec** specifies the team server of a candidate host. It is this process-id that should be used with any subsequent calls to **ExecProgram()** or **LoadProgram()**.

QueryHosts() finds candidate hosts by sending a message to the process group containing all team servers in the system (the VTEAM_SERVER_GROUP). Only those hosts which satisfy the requirements specified in **spec** will reply to this message.

```
DefaultSelectionRec(hostSpec)
    SelectionRec *hostSpec; /* assumed to be non-NULL. */
```

Sets up the **SelectionRec** structure pointed to by **hostSpec**, so that it can be used (as an argument to **QueryHosts()**, **ExecProgram()** or **LoadProgram()**) to select an 'arbitrary' remote host. This currently specifies the following minimum resource requirements:

- 1 free team descriptor.
- 10 free process descriptors.
- 200 Kbytes of free memory.
- Less than 50% processor utilization.
- No one logged into the host.

28.3. Remote Execution of Unix Commands

```
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 that provides the invoking process's current context, by opening a file in FEXECUTE mode. Of course, this server must be a Unix V server (see section 43). 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.

28.4. Other Program Execution Routines

```
ProcessId Execl(input, output, errput, status, error, arg0)
char *arg0;
File *input, *output, *errput;
SystemCode *error;
int *status;
```

Execl() calls **ExecProgram()** (and thus waits for the program created to finish executing, if **status** is non-NULL). It returns the program exit status in **status** and a system status code **error** (which indicates the nature of any errors encountered in **Execl** itself). **input**, **output**, and **errput** are used to specify the standard I/O of the program to be loaded and run. The remaining fields of the root message passed to **ExecProgram()** are derived from the invoker's root message. **arg0** actually represents the first of a variable number of parameters that represent the arguments to be passed to the new program. It is the first element of the **argv** array passed to **ExecProgram()**.

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

Invokes an

```
exec -c
```

on the **cmd** string. The program's exit status is returned.

— 29 —
User Interface Functions

This chapter outlines the facilities available to programs for interacting with the user — via the workstation agents. The manner in which this interaction is manifested to the user was discussed in Chapter 2. Implementation details of the various workstation agents may be found in Chapters 44, 46, and 45.

The discussion here is broken down into two basic components: terminal emulation and graphics. The terminal emulation facilities support ANSI virtual terminals and are common to all configurations of the V-System — that is, to both the STS and the VGTS. Indeed, virtually all applications use these facilities, in lieu of or in addition to any graphics facilities they employ. That is, each executive is associated with a separate AVT and any application created by that executive inherits access to the same AVT.

The graphics facilities are provided only by the VGTS. Attempts to use them in conjunction with the STS will fail.

Warning: Take special note of the "warning" in the Preface!

29.1. Virtual Terminal and View Management

Several routines for applications' manipulation of virtual terminals and views follow. All of these routines may be used with respect to any type of virtual terminal, although some are more useful for one type of virtual terminal than for other types. The virtual terminal identifier, **vt**, used in all routines is equal to the value returned by **CreateVGT()** or to the **fileid** field of the file descriptor returned by **OpenPad()** or **OpenAndPositionPad()**. These type-dependent routines, and others, are presented in subsequent sections.

```
int DeleteVGT(vt)
    short vt;
```

Destroy the virtual terminal identified by **vt**. All the views associated with the virtual terminal will also be destroyed.

Note: Badly named, since it may be used with AVT's as well as SGVT's (a.k.a. VGT's).

```
int DefaultView(vt, width, height, wXmin, wYmin,
        zoom, showGrid, pWidth, pHeight)
    short vt, width, height, wXmin, wYmin, zoom, showGrid;
    short *pWidth, *pHeight;
```

Create a view of the virtual terminal identified by **vt**, with the user determining the position on the screen with the graphical input device (mouse). The **width** and **height** parameters give the initial size of the viewport if they are positive; non-positive values indicate that the user should determine the size with the mouse at run-time. Note that these are physical device coordinates, not *normalized* device coordinates. **wXmin** and **wYmin** are the world coordinates to map to the left and bottom edges of the viewport. If the **pWidth** and **pHeight** pointers are non-NULL, then the shorts that they point to receive the selected width and height. Returns negative on error. See Chapter 2 for more information about how this call is manifested to the user.

zoom and **showGrid** are relevant only to SGVT's. **zoom** is the power of two to multiply world coordinates to get screen coordinates; it 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.

Note: In general, this routine is not particularly well-suited to creating views of AVT's, as explained in the following section.

```
int CreateView(vt, sxmin, symin, sxmax, symax,
        xmin, wymin, zoom, showGrid)
    short vt;
    short sxmin, symin, sxmax, symax, wxmin, wymin, zoom;
    BOOLEAN showGrid;
```

Create a view of the virtual terminal identified by **vt** — without interacting with the user. The initial position and size are determined by the **sxmin**, **symin**, **sxmax** and **symax** parameters. **wxmin** and **wymin** are the world coordinates to map to the left and bottom edges of the viewport. Returns negative on error.

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. Again, these parameters are relevant only for SGVT's.

Note: In general, this routine is not particularly well-suited to creating views of AVT's, as explained in the following section.

We now proceed with the description of the terminal emulation and graphics facilities. In the process the differences between the two underlying types of virtual terminals should become clear.

29.2. ANSI Terminal Emulation

ANSI terminal emulation is provided by what we call *ANSI virtual terminals* (AVT). An AVT emulates a (almost complete) subset of ANSI standard X.64 — often equated with the DEC VT-100; the precise subset is given in Chapter 46. An application may use the ANSI terminal protocol to communicate with the workstation agent, including escape sequences for cursor control. Additional V-specific support is provided for graphics input and line-editing, but applications may ignore these features as they wish.

The "store" of an AVT is referred to as a *pad*. Conceptually, a pad may be of infinite size, allowing an application to store arbitrary amounts of data and allowing a user to scroll back and forth through this data. In current practice, a pad provides only enough storage for one "page" of data — one virtual screen- or viewport-full. Consequently, it is not particularly useful to create multiple views of a pad.

Note: Unfortunately, the term "pad" has been adopted to mean both pad and AVT. Hence, most routines specific to AVT employ **Pad** in their names rather than **AVT**. Consequently, in the following discussion the terms are used interchangeably.

29.2.1. Cooking Your AVT's

The following mode bits are maintained for each AVT to indicate the degree of "cooking" to be applied to 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.

DiscardOutput When set, this bit causes all output to an AVT to be ignored. It is automatically set when the user types 'q' to an AVT that is blocked at the end of a page in **PageOutput** mode. It is automatically cleared whenever the workstation agent sends input to a program that is reading from the AVT. The bit may also be cleared "manually" via **ModifyPad()**. In particular, application programs should call **ModifyPad()** to clear this bit before sending a prompt to an AVT, to insure that the prompt is not discarded along with any previous output that was discarded at the user's request.

Echo Echo input characters.

LF_Output Change LF to CR-LF on output. That is, every line-feed operation is preceded by a return.

LineBuffer Wait for a line of input before returning. In addition, the line will be line-edited as described in section 2.5.

NoCursor Do not display a cursor in the indicated AVT.

PageOutput Block the writer each time the AVT fills up with output, and wait for the user to issue a command which unblocks the AVT. The user interface to the this feature is described in section 2.6. This bit is 'on' by default.

PageOutputEnable
Associated with each AVT is an internal flag, which, when 'off', disables turning on the **PageOutput** bit as described above. This internal flag is normally sticky, but can be changed by setting the **PageOutputEnable** bit in a **ModifyPad()** request. In this case, the **PageOutput** bit is also used to set the new value of the internal flag. The **PageOutputEnable** bit should only be used by certain "privileged" programs, as a means of allowing the user to "permanently" disable paged output mode.

ReportClick Report "clicks" of the graphical input device — a press of at least one button, followed the release of *all* buttons — in response to requests for graphical events.

ReportEscSeq Enable the "Emacs hack" described in Section 2.7. The encodings of the associated escape sequences are presented in the next subsection.

ReportTransition
Report "transitions" of the graphical input device — pressing or releasing any combination of buttons — in response to requests for graphical events.

By default, keyboard input is line-buffered and echoed by the workstation agent, with line-editing. More specifically, the following mode bits are set:

```
CR_Input
Echo
LF_Output
LineBuffer
```

29.2.2. Encoding Graphical Input Events

As noted, **ReportEscSeq** indicates that the application is capable of interpreting the associated escape sequences. This allows many useful programs that deal with conventional terminals to be extended to take advantage of the graphical input capability — without major redesign. For example, an EMACS library can be loaded to bind these character strings to commands that position the cursor, set the EMACS mark, delete and insert text. In fact, these sequences were added precisely to support EMACS — which, unfortunately, affected their design somewhat.

The exact encoding of the escape sequences is given in Table 26-1, where *<line>* and *<column>* are the position within the AVT where the mouse button(s) were pressed — encoded as bytes.

> Note: These are escape sequences that the workstation agent generates and the application must interpret. The standard ANSI protocol contains escape sequences that the application generates and the workstation agent must interpret.

29.2.3. Functions

Terminal emulation is implemented in terms of the standard V-System I/O protocol as defined in Chapters 22 and 33. For example, applications may read from and write to AVTs using the standard **Read()** and **Write()** primitives. Consequently, the application interface to an AVT is through a V-System file access descriptor (of type **File**). The following "AVT-specific" routines are also provided:

Mouse Buttons L M R	Escape Sequence
x . .	ESC M *<line><column>*
x x .	ESC M *<line><column>* null
x . x	ESC M *<line><column>* CTRL-W
. x x	ESC M *<line><column>* CTRL-Y

Table 29-1: Encodings for graphical escape sequences.

```
File *OpenPad(name, lines, columns, error)
    char *name;
    short lines, columns;
    SystemCode *error;
```

Create a new AVT *and* interact with the user to create a view of the AVT. **name** is a text name for the AVT. **lines** and **columns** specify the size of the pad. Returns a pointer to a file access descriptor for the pad; NULL on an error. **error** is a pointer to the reply code.

> Note: 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** (= workstation agent) and **fileid** (= virtual terminal / AVT id).

```
File *OpenAndPositionPad( name, x, y, lines, columns, error )
    char       *name;
    short      x, y, lines, columns;
    SystemCode *error;
```

Create a new AVT of size **lines** and **columns**, and place the view of this AVT at **x**, **y**. **name** is a text name for the AVT. Returns a pointer to a file access descriptor for the AVT; NULL on an error. **error** is a pointer to the reply code.

> Note: The note for **OpenPad()** also applies to **OpenAndPositionPad()**.

```
ModifyPad(avt, mode)
    File *avt;
    int mode;
```

Set the cooking **mode** of **avt**. **mode** is some combination of the bits described in the previous subsection.

```
int QueryPad(avt)
    File *avt;
```

Return the cooking mode of **avt**, some combination of the bits described in the previous subsection.

> Note: Rarely used, since its function is subsumed by **QueryPadSize()**.

```
int QueryPadSize(avt, plines, pcols)
    File *avt;
    short *plines, *pcols;
```

Get the size and mode of **avt**. The number of lines and columns are store in the shorts pointed to by **plines** and **pcols**, respectively. The cooking mode is returned as the value of the function.

```
PadFindPoint(avt, nlines, x, y, pline, pcol)
    short avt, nlines, x, y;
    short *pline, *pcol;
```

Convert the world coordinates (x,y) into a a line and column position within **avt**, stored in the shorts pointed to by **pline** and **pcol**, respectively.

Note: The **avt** parameter is currently unused, and the number of lines in the AVT must be specified in **nlines**.

```
RedrawPad(avt)
    File *avt;
```

Redraw the indicated **avt**.

Note: The same functionality should be available for SGVT's, but isn't.

```
SystemCode EditLine(avt, string, count)
    File *avt;
    char *string;
    int count;
```

Enter line-editing mode in **avt**, as defined in Section 2.5. The line-edit buffer is pre-loaded with the first **count** characters of **string**. On return, **string** will contain the line-edited input. Function returns one of the standard system reply codes.

29.3. Graphical Output

The central graphical 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 abstract graphical objects created and manipulated by a program are stored in a *structured display file* (SDF). An SDF is a name space in which graphical *items* and *symbols* are defined; it may be thought of as the "store" of a virtual terminal. The SDF is structured as a hierarchy, a directed acyclic graph of symbols calling other symbols. A symbol is an interior node of the graph, a logical grouping of graphical information. The leaves of the graph consist of graphical primitives such as rectangles, lines, or pieces of text. An item may be either one of these primitives or a call to another symbol — the "call statement" itself, not the symbol definition. Regardless, every item is contained in some symbol.

Note that a symbol call is like a procedure call, not like a macro. Changing the symbol definition changes all instances.

Each symbol is defined within its own 2-dimensional integer world coordinate space — although the dimensions of that coordinate space are the same across all symbols, namely, -32768 to 32767. Translation is the only modeling transformation permitted on "called" symbols. All other transformations, such as rotation or projection from higher dimensions, must be handled by the application.

As discussed in the previous section, defining symbols and filling them with items does not make anything appear on the screen. In order for a symbol to appear, it must be displayed on a *structured graphics virtual terminal* (SGVT). An SGVT may be thought of as a large, two-dimensional, imaginary display surface upon which graphical objects may be displayed. As for symbols, its coordinate space is from -32768 to 32767 in x and y, vastly larger than the actual screen. On this display space, one symbol in the SDF is displayed as the *top-level symbol*. Every item that is in that symbol, or in any symbol called by that symbol, etc., will be

displayed on the SGVT. An item in a symbol that is called several times, will be displayed several times. Thus for example, in our SGVT we might display a bicycle as the top-level symbol. The bicycle symbol contains a call to the frame symbol, and two calls, with different coordinates, to the wheel symbol. The wheel symbol contains several items: a circle for the rim and lines for the spokes. Each of these items will be displayed twice, once for each wheel, though they were defined only once.

29.3.1. SDF Manipulation

29.3.1.1. Item Attributes

Each item has the following attributes, as used in many of the procedures discussed below:

item A 16 bit unique (within the SDF) identifier for this object, or zero. This identifier is assigned by the program, guidelines for which are given in Section 29.3.1.4.

type One of the predefined primitive types described below. Currently eight bits are allocated for this.

typeData Eight bits of type-dependent information, as described in the next section.

xmin, xmax, ymin, ymax
Typically used to define the bounding box (or *extent*) of the item, in world coordinates. Also may be used for additional purposes, as discussed in the next section. Stored as 16-bit signed integers.

> Note: These names are misleading, since the VGTS actually sorts the endpoints and calculates the bounding box correctly.

string A "string's" worth of type-dependent information, as described in the next section.

29.3.1.2. Primitive Item Types

Some of the meanings of the fields above depend on the type of the item. The following are the types of primitive items that occur in a structured display file, with their type-dependent uses of the various attributes:

SDF_CIRCLE A circle, centered at (**xmin**,**ymin**) with a radius given by the **typeData** field.

> Note: This item type is currently supported only for the Sun model 100 framebuffer.

SDF_FILLED_RECTANGLE
A filled rectangle. **typeData** determines the pattern. There are two possible sets of patterns, each influenced by the application for which they were developed. The first set was defined for a VLSI layout editor and is not likely to be of general use; they are defined in **<Vgts.h>**. The second set was defined for a document illustrator and are likely to be of greater interest; they are defined in **<splines.h>**. To use one of them for a filled rectangle, add its index (as defined in **<splines.h>**) to the constant **STIPPLEOFFSET** (defined in **<Vgts.h>**, and use the resulting value as **typeData**.

SDF_GENERAL_LINE
A generalized line, from (**xmin**,**ymin**) to (**xmax**,**ymax**).

SDF_HORIZONTAL_LINE
Horizontal line from (**xmin**,**ymin**) to (**xmax**,**ymin**). **ymax** is ignored.

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 **dale** layout editor.

SDF_OUTLINE Outline for a selected symbol. **xmin**, **xmax**, **ymin** and **ymax** give the box for the outline. **typeData** specifies flag bits to select each of the edges: **LeftEdge**, **RightEdge**,

TopEdge or **BottomEdge**.

SDF_POINT A point, which usually appears as a 2-by-2 pixel square at (**xmin**,**ymin**).

SDF_POLYLINE A poly-line, consisting of a connected set of line segments.. *string* points to an array of points, as in:

```
typedef struct
  {
    short x;
    shor y;
  } SdfPoint;
```

Note: This item type is currently supported only for the Sun model 100 framebuffer.

SDF_RASTER A general raster bitmap with a lower left corner at (**xmin**,**ymin**) and upper right corner at (**xmax**,**ymax**). **typeData** determines if the raster is written with ones as black or white. **string** points to the actual bitmap, in 16 bit-wide swaths.

Note: On the Sun model 100 framebuffer, a raster can be displayed at zoom factors 0, 1, 2, 3, and 4 (only); on the model 120 framebuffer, only zoom factor 0 (no magnification) is currently supported. In all cases, the VGTS only supports the "display" of bitmaps, not any operations on them. An application-level libary containing "RasterOp" routines is, however, available (see Section 29.8).

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_SIMPLE_TEXT
A simple text string employing a fixed-width font (typically 8 pixels wide by 16 pixels high). The lower left corner of the string will be placed at (**xmin**,**ymin**). 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_SPLINE A spline object, of which a special case is a polygon. Splines may be filled with any of a number of different patterns or drawn with any of a number of different "nibs", as defined in **<splines.h>**. **string** points to a **SPLINE** structure as defined in the **<splines.h>**:

```
typedef struct
  {
    short x, y;
  } POINT;

typedef struct
  {
    unsigned short      order;  /* Order of the spline          */
    unsigned short      numvert;/* Number of vertices present.  */
    enum Nib            nib;    /* Nib to be used for drawing.  */
    unsigned short      border; /* Is the border visible?       */
    unsigned short      closed; /* Is this object closed or open?*/
    unsigned short      filled; /* Is this object filled?       */
    unsigned short      opaque; /* Is the filling opaque (solid)?*/
    enum Pattern        pat;    /* Fill (stipple) pattern.      */
    POINT               head;   /* Head of the list of vertices */
  } SPLINE;
```

Note: The patterns used for splines are a subset of those used for filled rectangles. (See the discussion of **SDF_FILLED_RECTANGLE** above.)

SDF_TEXT A string of general text, with the left end at **xmin** and the baseline at **ymin**. **typeData** determines the font number. (To get the actual bounding box (calculated from information in the font file), use **InquireItem()** after the **AddItem()**, as defined in

the following section.) See section 29.3.3 for an example.

SDF_VERTICAL_LINE
Vertical line from (**xmin**,**ymin**) to (**xmin**,**ymax**). **xmax** is ignored.

SDF_VERTICAL_REF
A vertical reference line at (**xmin** + **xmax** / 2).

29.3.1.3. Functions

The following are the currently defined functions used to manipulate an SDF and, hence, generate graphical output. All return values except the actual function value are passed via pointer parameters. If any pointer is NULL, no value is returned for that parameter. For performance reasons, many of these calls are batched (several calls in one request) and/or pipelined (no return values). In either case there are no meaningful return values and any error conditions simply cause the VGTS to drop the call on the floor. The description for each routine indicates whether this is the case.

```
short CreateSDF()
```

Create a structured display file, returning its id. Returns -1 if the VGTS runs out of resources. Must be called before any symbols are defined. Forces all pending calls to be executed.

```
int DeleteSDF(sdf)
    short sdf;
```

Return all the items defined in the given **sdf** to free storage. This includes all data structures associated with items in the SDF. Returns **sdf** or -1 on error. Forces all pending calls to be executed.

```
DefineSymbol(sdf, symbol, textName)
    short sdf, symbol;
    char *textName;
```

Enter **symbol** into the **sdf** and open it for editing. Only one symbol may be open in any given SDF at a time. **textName** is an optional descriptive name for the symbol, used in the hit selection routines for disambiguating selections. Buffered call, but always returns **symbol** for backward compatibility.

```
short EndSymbol(sdf, symbol, sgvt)
    short sdf, item;
    short sgvt;
```

Close **symbol** in **sdf** so no more insertions can be done and cause all views of **sgvt** displaying the symbol to be redrawn. The VGTS ensures that, if only additions have been made since the last **EndSymbol**, only those additions are drawn. Called at the end of a list of **AddItem()** and **AddCall()** calls defining a symbol, started with **DefineSymbol()** or **EditSymbol()**. Forces all pending calls to be executed. Always returns **symbol** for backward compatibility.

> Note: **symbol** is actually redundant, since only one symbol can be "open" in any SDF at a time, but it must be provided.

```
EditSymbol(sdf, symbol)
    short sdf, symbol;
```

Open (already existing) **symbol** in **sdf** for modification. This has the effect of calling **DefineSymbol()** and inserting all the already existing entries. The editing process is ended in the same way as the initial

definition process — a call to **EndSymbol()**. Buffered call, but always returns **symbol** for backward compatibility.

```
short DeleteSymbol(sdf, symbol)
    short sdf,.symbol;
```

Delete **symbol** from **sdf**. More correctly, render the symbol definition "empty" to prevent problems with dangling references (calls) to the definition. The dangling references will be interpreted but will have no effect, since the symbol will no longer contain any items. Returns **symbol** if successful, else 0. Forces all pending calls to be executed.

```
AddItem(sdf, item, xmin, xmax, ymin, ymax,
        typeData, type, string)
    short sdf, item, xmin, xmax, ymin, ymax;
    unsigned char type, typeData;  char *string;
```

Add **item** to the currently open symbol in **sdf**. Remaining parameters as defined above (Sections 29.3.1.1 and 29.3.1.2). Buffered call, but always returns **item** for backward compatibility.

```
AddCall(sdf, item, xoffset, yoffset, calledSymbol)
    short sdf, item, xoffset, yoffset, calledSymbol;
```

Add an instance of the **calledSymbol** to the currently open symbol in **sdf**. The "call statement" itself is given the name **item**. The origin of the called symbol instance is placed at (**xoffset**,**yoffset**) in the coordinate space of the calling symbol. May be called *before* the called symbol is defined, in which case a dummy entry for the symbol is inserted in the SDF; any future attempts to define the symbol will use the dummy entry. Buffered call, but always returns **item** for backward compatibility.

```
DeleteItem(sdf, item)
    short sdf, item;
```

Delete **item** from the currently open symbol in **sdf**. Symbol calls can be deleted just like any other item, but symbol definitions are deleted by the **DeleteSymbol()** function. Buffered call, but always returns **item** for backward compatibility.

```
int InquireItem(sdf, item, xmin, xmax,
        ymin, ymax, typeData, type, string)
    short sdf, item;  short *xmin, *xmax, *ymin, *ymax;
    unsigned char *type, *typeData;  char *string;
```

Read the attributes of **item** in **sdf**. Parameter semantics are defined above (Sections 29.3.1.1 and 29.3.1.2). 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, non-zero. Forces all pending calls to be executed.

```
short InquireCall(sdf, item)
    short sdf, item;
```

Return the name of the symbol called by **item** in **sdf**. Returns zero if the item is not a call, or could not be found. Forces all pending calls to be executed.

```
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 (already existing) **item** in **sdf**. Remaining parameters as defined above (Sections 29.3.1.1 and 29.3.1.2). This is equivalent to deleting an item and then reinserting it, so the item must be part of the open symbol. Buffered call, but always returns **item** for backward compatibility.

29.3.1.4. Naming Items and Symbols

Items and symbols are both identified by 16-bit identifiers, most commonly thought of as unsigned integers. The identifiers are specified 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, so the programmer should be careful not to assign the same item number within the same SDF twice. However, applications may use multiple SDFs for multiple name spaces.[12]

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.

29.3.1.5. Output Modes

By appropriate use of the various functions, programs may achieve the effect of deferral modes for graphical output. First, they may construct graphical objects in their entirety and *then* display them, by executing a **DefineSymbol()** or **EditSymbol()**, followed by many **AddItem()** or **AddCall()** calls, followed by an **EndSymbol()**. This corresponds to creating an "invisible segment" and then displaying it in traditional graphics systems.

Alternatively, an application many construct and display an object "on the fly", that is, display each item as it is added to the object. This is done, for example, by repeatedly executing an **EditSymbol()** - **AddItem()** - **EndSymbol()** sequence, such that each **EndSymbol()** causes the symbol to be redrawn. This corresponds to creating a "visible segment" in traditional graphics systems. (Note the optimization discussed in the description for **EndSymbol()**, which reduces redraw time.)

The first style of output yields higher throughput, whereas the second yields faster response.

29.3.1.6. An Example

To create the bicycle figure of the previous section, we would use code like the following:

[12] The intended use of multiple SDFs is that an application would have both "private" and "shared" graphical data, such that the shared data was stored in an SDF used by multiple (cooperating) applications.

```
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, 3, 0);
    .
    (whoops ... forgot the frame)
    .
EditSymbol(sdf, 4)
    .
    (add frame)
    .
EndSymbol(sdf, 4, 0)
```

29.3.2. SGVT Management

```
int CreateVGT(sdf, type, topSymbol, string)
    short sdf;  int type;  short topSymbol;  char *string;
```

Create an SGVT of the indicated **type**. Put the indicated symbol — **topSymbol** in **sdf** — as the top-level symbol in the SGVT. **topSymbol** can be zero to indicate a blank SGVT. **type** can be some combination of **TTY**, **GRAPHICS**, and **ZOOMABLE**, but programmers are advised to use the terminal emulation functions described above for **TTY**'s. If the **ZOOMABLE** bit is set, the view zooming factor can be changed by the user. Returns the virtual terminal id or negative on errors.

```
DisplayItem(sdf, topSymbol, sgvt)
    short sdf, topSymbol;  int sgvt;
```

Change the top-level symbol for **sgvt** to **topSymbol** in **sdf**. The new symbol is displayed in every view of the SGVT.

29.3.3. 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 font-id returned by this call is used as the **typeData** field for SDFTEXT items. 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.

29.4. Graphical Input

The VGTS maintains an event queue for each virtual terminal — whether AVT or SGVT — on which both graphical (mouse) and keyboard events are queued.

29.4.1. Common Functions

The following functions are (more or less) independent of the type of virtual terminal. To maintain compatibility with the AVT-specific routines and the V I/O protocol, the desired virtual terminal must be bound to a V file access descriptor before calling these functions. Specifically, in the descriptions below, **vt->fileserver** must contain the process id of the workstation agent and **vt->fileid** must contain the id of the virtual terminal. The file descriptor returned from **OpenPad()** is set up precisely in this fashion, but if **CreateVGT()** is used, the application must explicitly construct an appropriate file descriptor, storing the result of **CreateVGT()** in **vt->fileid**. The file pointer **stdin** may be used to receive input from the virtual terminal (usually an AVT) associated with the application's "standard input".

```
GetEvent(vt, px, py, pbuttons, cbuf)
        File *vt;
        short *px, *py, *pbuttons;
        char *cbuf;
```

Wait for any input event in the indicated virtual terminal. This currently means mouse clicks, mouse transitions, or keyboard input — *not* mouse movements. If the virtual terminal is an AVT, then the type of graphical event reported depends on the cooking mode of the AVT. Returns the world X and Y coordinates of the mouse 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 function value is negative on error (in which case, **vt->lastexception** contains the error code), 0 on mouse event, or the number of characters returned on a keyboard event.

> Note: At most IO_MSG_BUFFER (defined in **<Vioprotocol.h>** - currently 20) characters will be returned in **cbuf**. The corresponding actual parameter should be (at least) this size.

```
int GetGraphicsEvent(vt, px, py, pbuttons)
    File *vt;
    short *px, *py, *pbuttons;
```

Wait for a graphical event in the virtual terminal associated with the file descriptor **vt**. Currently, graphical events consist of transitions and clicks of the mouse buttons — *not* movements. If the virtual terminal is an AVT, then the type of graphical event reported depends on the cooking mode of the AVT — which must be set appropriately via **ModifyPad()**. Returns the world X and Y coordinates in the shorts pointed to by **px** and **py**; the state of the buttons is returned both in the short pointed to by **pbuttons** and as the value of the function. The function value is negative on error, in which cast **vt->lastexception** contains the error code.

```
int GetGraphicsStatus(vt, px, py, pbuttons)
        File *vt;
        short *px, *py, *pbuttons;
```

Sample the graphical input device relative to the virtual terminal associated with the file descriptor **vt**. Currently, this means returning the current location and button status of the mouse, whether or not the mouse currently resides in a view of the virtual terminal and *without* waiting for the mouse to move. All events queued for the virtual terminal are flushed prior to sampling. Returns the world X and Y coordinates in the shorts pointed to by **px** and **py**; the state of the buttons is returned both in the short pointed to by

pbuttons and as the value of the function. The function returns negative on error, in which case **vt->lastexception** contains the error code — typically **EOF** to indicate that the mouse cursor was not in a view of the virtual terminal.

29.4.1.1. Antiquated Routines

The following routines pre-date those listed above. The only reason for their continued existence is that they currently are the only graphical input routines that may be employed by applications running on non-V hosts (see Section 29.7.2). However, their days are numbered.

```
short GetMouseClick(x, y, buttons)
    short *x, *y, *buttons;
```

Wait for a mouse click in the virtual terminal corresponding to **stdin**. The world X and Y coordinates are returned in the shorts pointed to by **x** and **y**, and the state of the buttons is returned in the short pointed to by **buttons**. If a key is pressed, a message is printed stating that a mouse click is expected, and the key is ignored. Forces all pending calls to be executed and blocks until a mouse click does come in.

> Note: This function is semantically equivalent to both **GetGraphicsEvent()** and **GetMouseOrKeyboard()** (where any character returned is dropped on the floor).

```
short GetMouseOrKeyboard(c, x, y, buttons)
    short *x, *y, *buttons;
    char *c;
```

Wait for a mouse click or keyboard press in the virtual terminal corresponding to **stdin**. If the mouse is clicked, the world X and Y coordinates are returned in the shorts pointed to by **x** and **y**, the state of the buttons is returned in the short pointed to by **buttons**, and the function returns the identifier of the virtual terminal in which the click occurred. If a key was pressed, the character is returned in the location pointed to by **c**, and the function returns 0.

> Note: This function is semantically equivalent to **GetEvent()**.

29.4.2. SGVT-only Functions

Mouse events often signify an attempt on the part of the user to "select" some graphical object. When such an event is reported to the application, it should respond by calling the following function to determine which, if any, graphical object was so selected.

```
LISTTYPE FindSelectedObject(sdf, x, y, sgvt, searchType)
    short sdf, x, y, sgvt;
    char searchType;
```

Return a list of items that are at or near (x,y) in **sgvt**. 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:

All	Anything will do.
AllLines	Any lines.
JustHoriz	Just horizontal lines.
JustRasters	Just rasters.
JustRects	Just rectangles.
JustSplines	

Just splines.
JustText Just text strings.
JustVerts Just vertical lines.

Usually the constant value **All** will be used. The return value is defined as follows:

```
typedef struct MinElement
  {
    short                       item;
    short                       edgeset;
    struct MinElement           *next;
  } MINREC, *MINPTR;

typedef struct ListInfo
  {
    MINPTR              Header;
    short               NumOfElements;
  } LISTTYPE;
```

29.5. Miscellaneous Functions

The following functions are (more or less) independent of the type of virtual terminal — despite the occurrence of **Pad**, or **Vgt** in their names. As in Section 29.4.1, to maintain compatibility with the AVT-specific routines and the V I/O protocol, the desired virtual terminal must be bound to a V file access descriptor before calling these functions. Specifically, in the descriptions below, **vt->fileserver** must contain the process id of the workstation agent and **vt->fileid** must contain the id of the virtual terminal. The file descriptor returned from **OpenPad()** is set up precisely in this fashion, but if **CreateVGT()** is used, the application must explicitly construct an appropriate file descriptor, storing the result of **CreateVGT()** in **vt->fileid**. The file pointer **stdin** may be used to receive input from the virtual terminal (usually an AVT) associated with the application's "standard input".

GetTTY()

Put the terminal in raw mode. The (remote) 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.

```
short popup(menu)
    PopUpEntry menu[];
```

Display a "pop-up" menu and wait for the user to select an option. The menu argument points to an array of **PopUpEntry** structures:

```
typedef struct
 {
   char *string;                /* String to display. */
   unsigned char menuNumber;    /* Number returned if entry selected. */
  } PopUpEntry;
```

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.

ResetTTY()

Restore the mode before the last **GetTTY()**. Runs under UNIX as well, checking standard input properly.

```
SelectPad(vt)
    File *vt;
```

Cause the virtual terminal associated with **vt** to be selected for input. The (principal) view of the virtual terminal is brought to the top of the stack of views. Only works if the calling program also "owns" the virtual terminal currently selected for input.

```
SystemCode SetVgtBanner(vt, name)
    File *vt;
    char *name;
```

Set **name** to be the banner at the top of each view of the virtual terminal corresponding to **vt**.

29.6. Example Program

The following program can be compiled to run either remotely under Unix or under the V system. The **#ifdef VAX** directives allow the programmer to conditionally compile code for one environment or the other. It first creates an SDF and SGVT, 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, sgvt;

Quit()
  {
     DeleteVGT(sgvt,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 );
     sgvt = CreateVGT(sdf, GRAPHICS+ZOOMABLE, 1, "random objects" );
     DefaultView(sgvt, 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, sgvt );
       }

     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);
  }
```

29.7. Some Logistics

The constants for mouse search types, virtual terminal usage types, etc. are found in the include files **Vgts.h** and **Vtermagent.h**.

29.7.1. Applications Running Under V

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.

29.7.2. Applications Running Under Other Operating Systems

To transparently run programs on a UNIX system, use **-lVGTS** on your **cc** command line. Use **-I/usr/sun/include** to get the file **Vgts.h**.

This package employs escape sequences that can be used through PUP Telnet, IP Telnet, or with the remote command execution facility of the executive. The details of this protocol are explained in Chapter 46.

Note: The following functions are ***not*** currently available to applications in this class:

```
EditLine()
GetEvent()
GetGraphicsEvent()
GetGraphicsStatus()
ModifyPad()
OpenPad()
QueryPad()
QueryPadSize()
RedrawPad()
SelectPad()
SetVgtBanner()
```

29.8. Rolling Your Own

The primitives discussed here have proven suitable to a wide range of applications. Naturally, a few users have found them unsuitable, especially for applications that manipulate large bitmaps, such as image processing applications. Although a raster item type is supported, raster operations are not. Hence, applications must perform the operations themselves and then pass the new bitmaps to the VGTS. Subsequent versions of the VGTS will address these and similar problems.

In the meantime, desperate programmers may, in fact, manipulate the frame buffer directly by using the low-level device-dependent graphics libraries employed by both the VGTS and the STS. There is a separate library for each real device. The libraries and their documentation may be found in the **libc/graphics** directory.

Note: As noted above, these libraries also contain a variety of device-independent routines, including some general-purpose "RasterOp" routines, that may be of use to some applications.

Warning: Directly manipulation of the graphics hardware may be very hazardous to your health. On workstations with the Sun model 100 frame buffer, for example, your manipulation of the frame buffer may conflict with that of the workstation agent, leading to rather odd screen images. That is, both your application and the workstation agent are manipulating the frame buffer registers. Fortunately, in this case, you should be able to avoid most problems by rendering all virtual terminals that are generating output *and* all AVTs that have been selected for input invisible — by burying them under inactive virtual terminals, for example. The latter step is needed in order to disable the blinking cursor.

Finally, if you still are not dissuaded, consider that access to the frame buffer will be prevented in future versions of the system, hopefully coincident with the addition of suitable raster support to the VGTS.

— 30 —
Miscellaneous Functions

30.1. 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.

`unsigned GetTime(clicksptr)`

return the current time in seconds as maintained by the local kernel. The current time is represented as seconds since January 1, 1970 GMT. If clicksptr is not NULL, the number of clicks since the last second is stored in location pointed to by clicksptr. The standard manifest CLICKS_PER_SEC indicates the number of clicks per second for the host.

`SystemCode SetTime(seconds, clicks)`

sets the local kernel time to the specified **seconds** and **clicks**. The time maintained by the kernel is normally set on system boot and need not be changed subsequently.

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.

`unsigned Delay(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 the number of clicks remaining in the delay period. Thus, it normally returns 0. However, if the delaying process is awakened using **Wakeup**, it may return a non-zero value.

`SystemCode Wakeup(pid)`

unblock the process specified by **pid**, returning OK, assuming the process is currently delaying using **Delay** and the invoker is the same user as the specified process, or is a privileged user. Otherwise, the return value is a standard system code indicating the error.

`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.

30.2. Strings

The string-related functions in the V-System C library are described below.

30.2.1. 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()
```

30.2.2. 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 occurrence 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. The function **strsave()** is identical to **Copy_str()**.

```
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**.

30.3. Exception Handling Functions

```
short *StandardExceptionHandler(req, pid, fout)
    register ExceptionRequest *req;
                        /* Exception message. */
    ProcessId pid;      /* Process incurring exception. */
    File *fout;         /* Print out messages on this file */
```

Standard exception handling print routine. Prints out some information about the process incurring the exception and returns the pc at which the exception occurred. **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.

```
PrintStackDump(fout, pid)
    File *fout;  ProcessId pid;
```

Prints out the stack of the process specified by **pid**. The process must be in the same address space as the invoker.

30.4. 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:
V Servers

— 31 —
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 part of the manual describes the various protocols for requesting these services, including the format of request and reply messages, the possible values for the message fields, and the server process that handles the request. A secondary role of this part of the manual is to act as an implementation guide to the various servers; at some future time, these implementation details will be removed to a separate manual.

The information contained in this part of the manual is generally not required by application programmers because most protocols are implemented in the standard C program library described in Part II of the manual. However, more sophisticated use of the system may require the more detailed information in this part of the manual.

This chapter gives an overview of the interactions among the different servers and the kernel. The next three chapters present the standard message formats and codes, and the details of two standard protocols, the V-System I/O Protocol and V-System Naming Protocol. The remaining chapters give the details of the individual servers, describing which of the standard protocols they implement, additional server-specific protocols they provide, and, in many cases, how they are implemented.

31.1. The Basic Servers - In Isolation

Figure 31-1 shows the configuration of servers on a typical workstation. The various interactions indicated are discussed in the following section. Here we discuss the basic functions and structure of each server more or less in isolation from the others.

31.1.1. General Considerations

There are two basic dimensions by which servers may be classified: whether they are implemented as pseudo-processes within the kernel or outside the kernel, and whether an instance of the server exists on each workstation or not. Several servers are implemented internal to the kernel primarily for performance reasons. Naturally, these servers must exist on every workstation. As discussed below, there are several additional servers, including those that manage teams and exceptions, instances of which must also exist on every workstation. Other servers exist, however, that need not be resident on every workstation, the most common example being a storage server.

Regardless of how (or where) servers are implemented, they are always accessible via the usual IPC facilities and standard protocols. The "main" server process typically consists of an infinite loop that receives a request for service, processes it, receives the next request, and so on.

Because all message-passing is synchronous, the main process typically cannot employ the **Send()** primitive, lest it block indefinitely. For this reason and others, servers implemented outside the kernel often employ additional processes, for example, to send messages for them, to service multiple input streams in a responsive fashion, or to manage multiple open "instances" (of objects) without complex multiplexing. These auxiliary processes are generally called *helpers*.

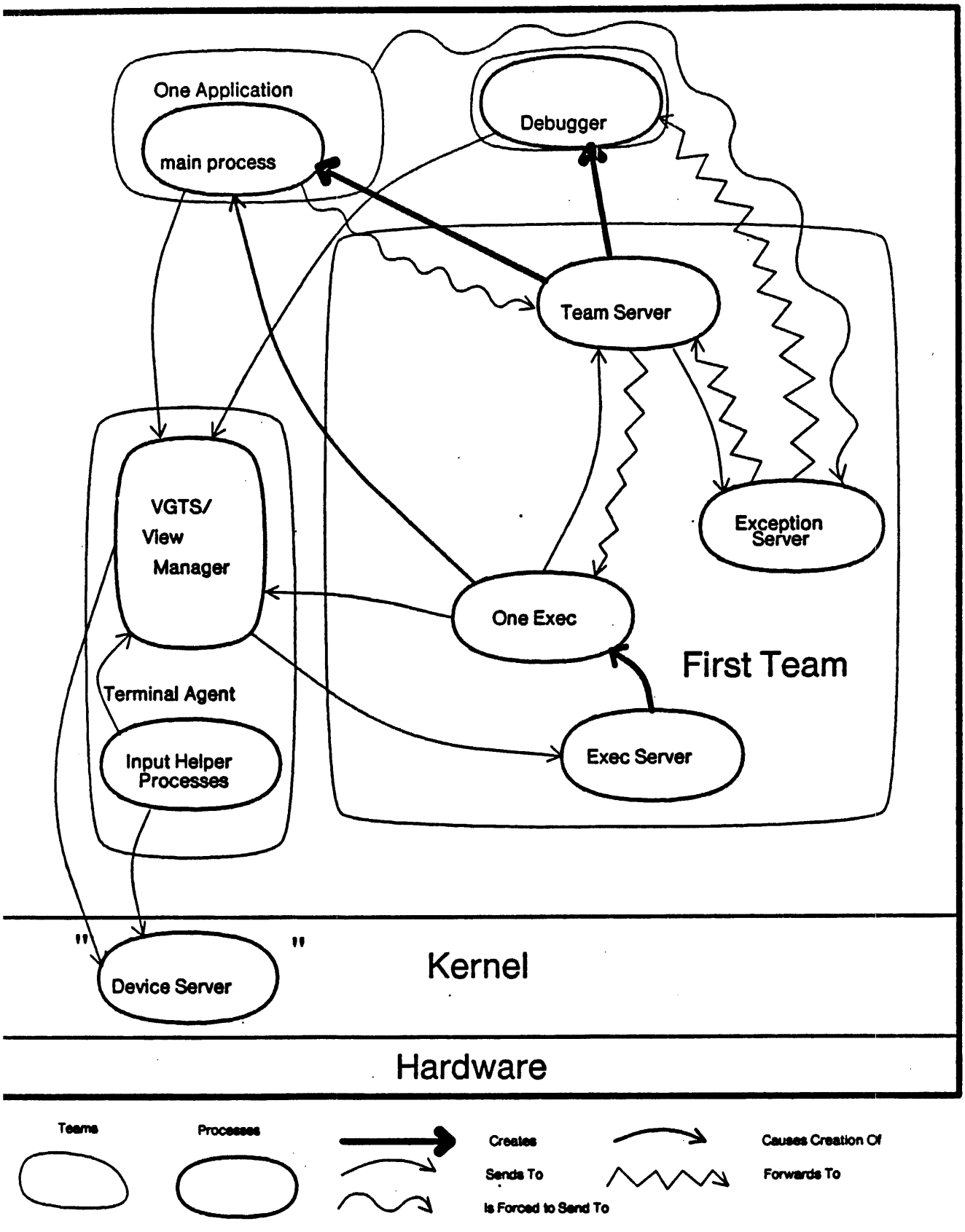


Figure 31-1: The V-System: A single workstation view.

31.1.2. Machine-relative Servers

Machine-relative servers are servers, instances of which exist on every workstation running V.

31.1.2.1. Kernel Server

The kernel server is a pseudo-process embedded in the kernel that handles all requests to manage processes, as well as the requests to create and terminate teams.

31.1.2.2. Device Server

All hardware I/O devices attached to the workstation are serviced by the *device server*, which is a pseudo-process embedded in the kernel. The device server supports the standard I/O and naming protocols discussed in Chapters 33 and 34, respectively. Consequently, it behaves like any other I/O server as far as applications are concerned.

31.1.2.3. Team Server

The team server is the manager of the physical host.[13] It loads, executes, and monitors all teams other than the first. (Recall that a team usually corresponds to a program, although some programs consist of more than one team.) Requests to the team server ask it to load and start a team, to terminate one, or to print the directory of currently executing teams.

The team server also provides the bulk of the remote execution and migration facilities. It implements the policies that determine whether to accept other workstations' programs for execution to begin with and whether to preempt them later on. It also implements the facilities needed to migrate programs between workstations.

31.1.2.4. Exception Server

The exception server is notified whenever a process incurs an exception. If another process has registered itself as the exception handler for the process that incurred the exception, the exception server simply forwards the exception to the registered handler. Otherwise, it prints a message on the screen (using the console device). The latter case does not arise very often in practice, however, because the team server registers itself as the exception handler of last resort for almost all processes.

31.1.2.5. Workstation Agents

Workstation agents were discussed at length in Chapter 2. Here, we merely present the basic implementation of the VGTS as a canonical example of server structure.

The VGTS is structured as one server process with three helper processes (see Figure 31-2). There is one helper process to receive input from the mouse (through the device server), one to read from the keyboard, and a timer process to invoke periodic functions like redrawing the screen. The keyboard and mouse helper send requests to the device server, and block until input arrives. When they receive a reply, they then send the input to the main server process, and request more input from the device server. This is a typical use of helper processes for processing multiple input streams, simultaneously and in a responsive manner.

> Note: Although grouped with all the other machine-relative servers, workstation agents are distinguished by the fact that they need not exist at all. That is, if the workstation does not support any user I/O devices there is no need for it to support a workstation agent.

---

[13] In some documents it is also referred to as the *program manager*.

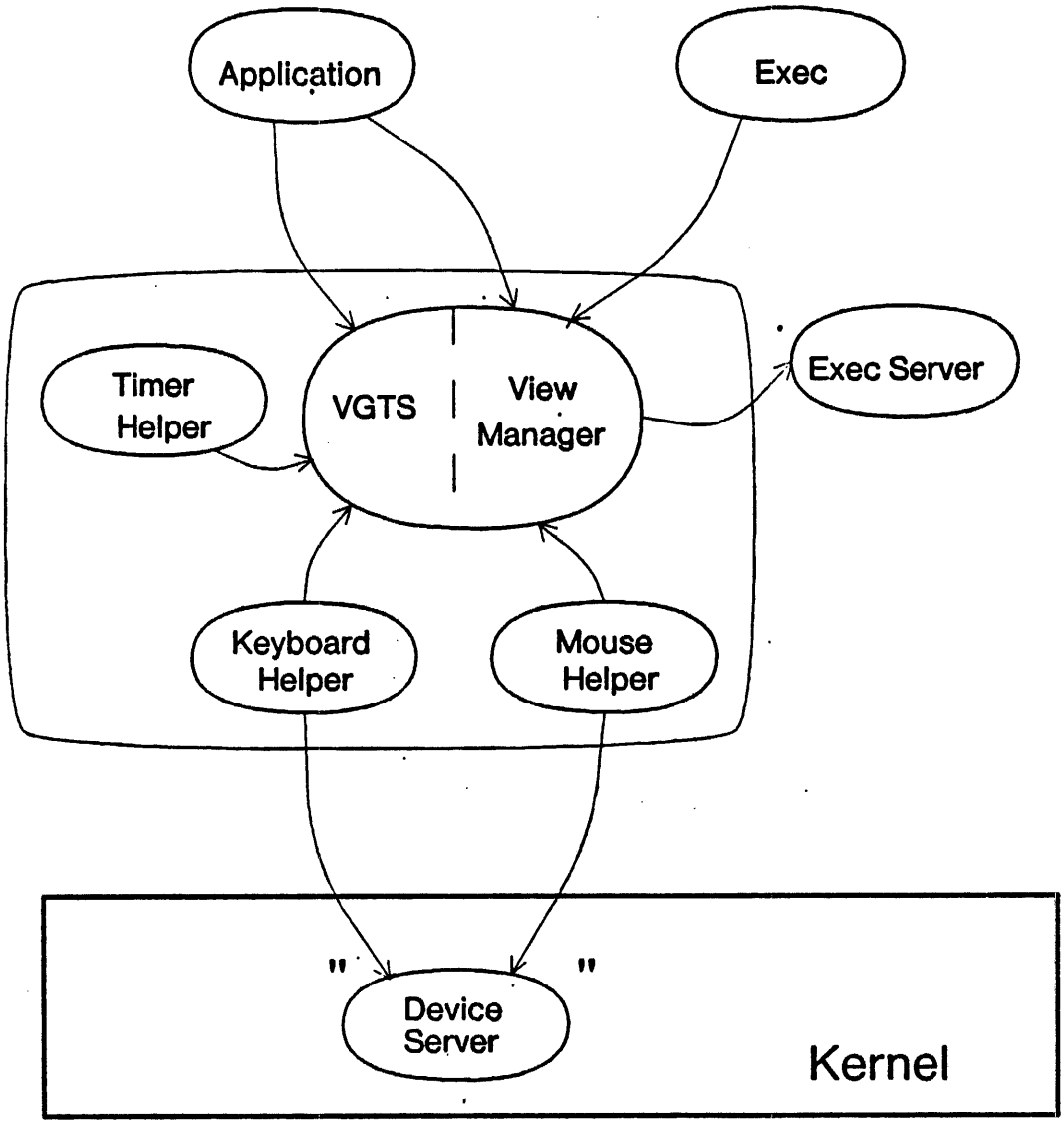


Figure 31-2: VGTS process structure.

31.1.2.6. Executives and the Exec Server

While the workstation agent provides the low-level I/O interface for the user, the executive provides the command processing interface. It corresponds to the Unix shell or the Tops-20 Exec, in that it is a user-level process providing command parsing and convenient access to system services. The basic operation of the executive is documented fully in Chapter 3.

All instances of the executive on a workstation are managed by the *exec server*. Its purpose is to allow sharing of code and data (such as aliases) among all executives.

31.1.3. Global Servers

A *global* server is distinguished by the fact that it is designed to service requests from any workstation, not just from processes running on the workstation on which it happens to be running.

31.1.3.1. Authentication Server

The authentication server provides the basic mechanisms by which users log in to the V-System and by which security is maintained.

31.1.3.2. Storage Server

Storage servers generally provide for long-term information storage. They typically run on workstations will large disks attached, or on VAX/UNIX systems. Each host may support at most one storage server. A "RAM disk" facility is also provided, in the form of the *memory server.*

31.1.3.3. Internet Servers

Internet servers provide for network communications using standard (inter)net protocols, as compared to the inter-kernel protocol implemented by the V kernel. They are essentially protocol converters that allow applications that communicate by means of the V I/O protocol to communicate with hosts that can only (or prefer to) be reached by a protocol other than the inter-kernel protocol.

31.2. The System in Operation

Having summarized the functions of each of the major servers in isolation, we now describe some of the ways in which these servers interact. The intent is not only to help the reader understand the basic structure of the system, but also to understand some basic techniques for multi-process structuring.

31.2.1. System Initialization

When a workstation is booted, its PROM loads a program that loads the V kernel and the *first team.* After the kernel has completed its internal initialization, it creates an initial team space and an initial bootstrap process on this team, and assigns the processor to this process. The bootstrap process starts all the servers necessary to run the system on the workstation: the exception server, the team server, the exec server, and some version of a workstation agent. All but the last are always loaded together on the first team, and thus share a single address space; the workstation agent may or may not be loaded on the first team, at the discretion of the user. The advantage of placing it on its own team is that then it may be replaced dynamically using the **newterm** command.

31.2.2. Loading a Team

Teams other than the first can be loaded from object code files using routines in the V C program library. These routines package an appropriate request to the team server and take care of matters such as initializing the team's data space as discussed in Section 18.4. The detailed message traffic involved is illustrated in Figure 31-3: In its request to the team server (edge 1), a client includes an open file descriptor specifying the file to be loaded. This descriptor references the storage server that manages the file (edge 2) and from which the file will be loaded. After receiving the request, the team server requests the kernel server to create a new team with initial process (edges 3 and 4). Like all processes, it is created in the *awaiting reply* state — waiting for a reply from its creator. In effect, the kernel simulates a **Send()** from the process to the team server (edge 5). The team server forwards this message, and its associated privileges (including access to the entire address space of the new team), to the client that originally asked for the team to be loaded (edge 6). At this point, the client (or the library routine it called) can initialize the team's environment variables and the like, and then **Reply()** (edge 7), thus allowing the new team to begin execution — all as discussed in Section 18.4.

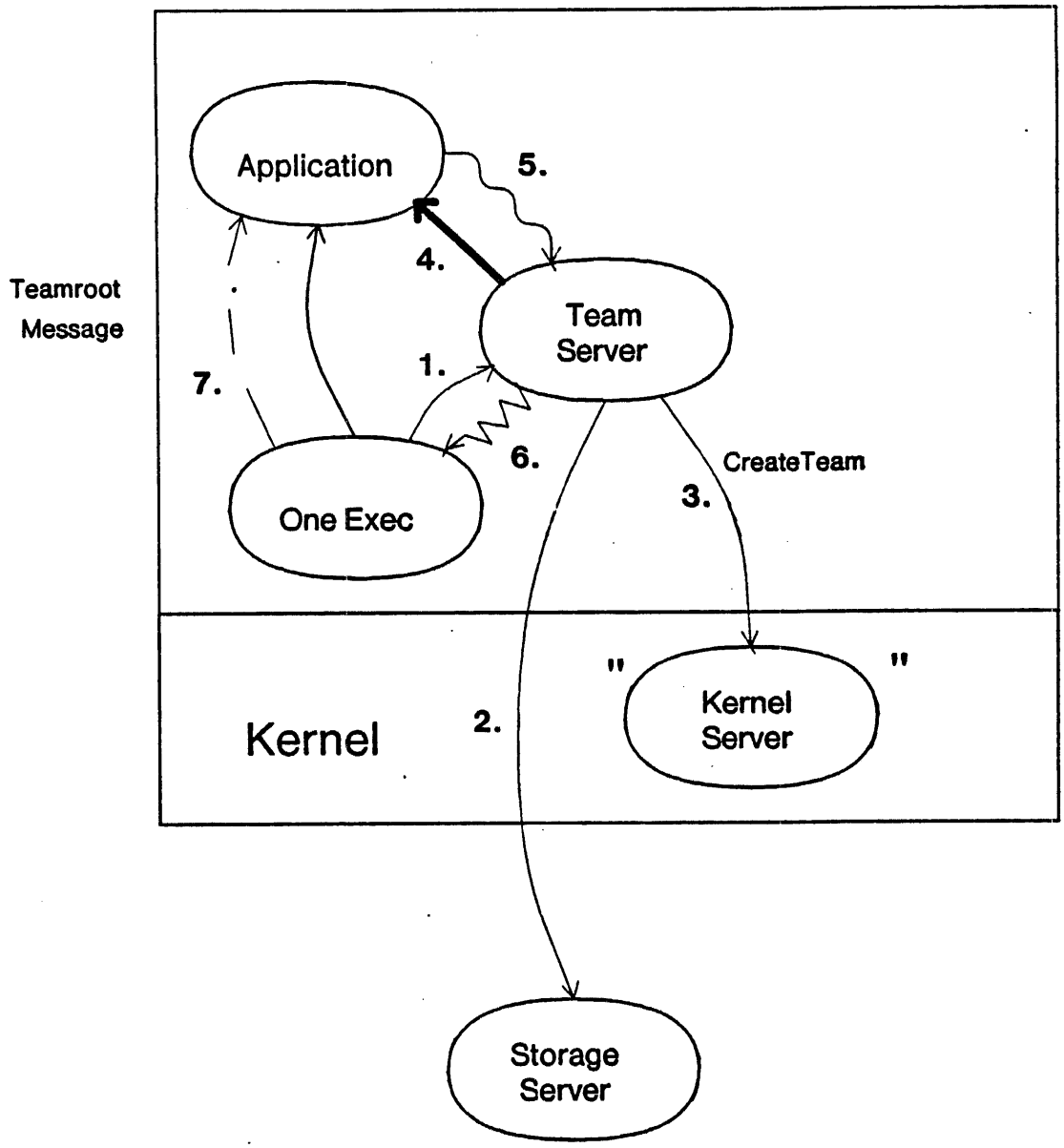


Figure 31-3: Loading a team.

31.2.3. Team Termination and Exit Status

Barring catastrophic failures, the `_start` routine loaded with every team will ensure that the team's owner is always notified of the termination of the team — by sending it an appropriate message. If the team terminated gracefully — by calling `exit()` or returning from `main()` with a valid exit status — the owner will be able to ascertain the team's exit status.

31.2.4. Command Processing

Prior to running an application, the executive must determine what program is to be run! It does so by parsing the command line returned in response to its request for line-edited input from the workstation agent. The executive then opens the file that contains the program (edge 2 in Figure 31-3), and any other files as necessary to handle redirected I/O. Finally, it invokes the procedure discussed in Section 31.2.2 above.

Unless the program is being run in the background, the executive waits for it to complete execution by waiting for a message from it.[14] If the team terminates by calling `exit()` or returning from `main()`, the executive will indeed receive a message from it containing the team's exit status. Otherwise, the team will have died abnormally, in which case the executive will awakened by the kernel and informed of that fact. At this point, it is ready to ask for the next command line.

31.2.5. Exception Handling

As described above, the team server uses only the services of the kernel and a storage server. However, the team server is also the principal client of the exception server. Figure 31-4 illustrates the message flow involved: The team server creates the team to begin with (edges 1 and 2) as described above. It then registers itself as the exception handler for the team (edge 3). If a process on the team incurs an exception, the kernel simulates the effect of the offending process sending an exception message to the exception server (edge 4), which forwards the message to the team server (edge 5). The team server then uses its own facilities to load the V debugger (edge 6), and forwards the exception message to it (edge 7).

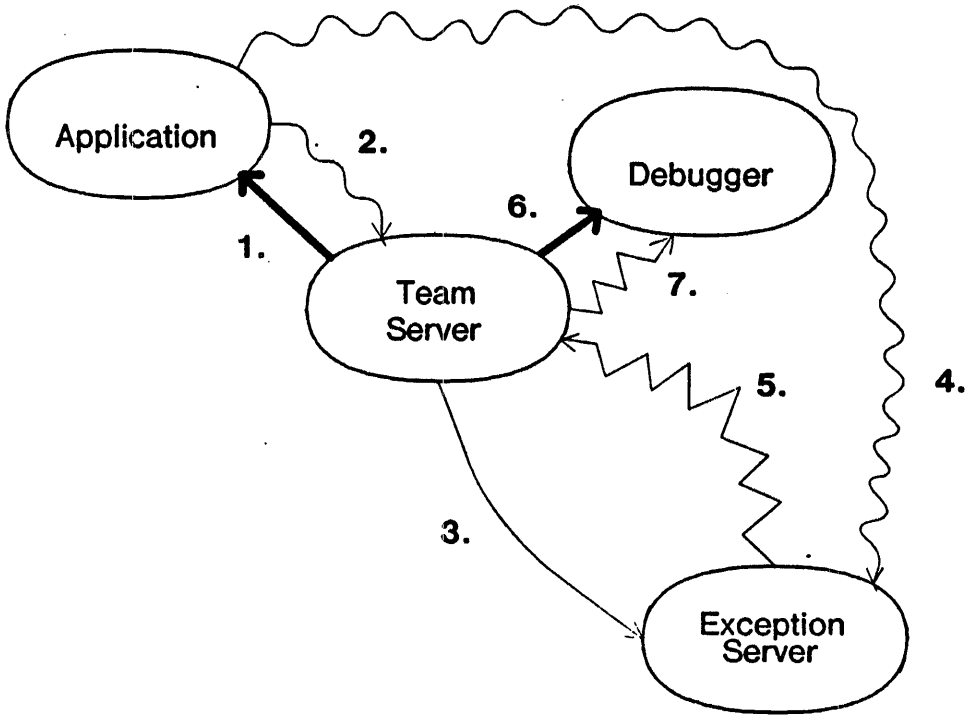


Figure 31-4: Handling an exception.

Upon receipt of the message, the debugger prints the exception data on the screen and registers itself as the

[14] Programs running in the background have the team server as their *owner*. Then, if the executive that started them is deleted by the user, which usually results in the destruction of any program it owns, the background program will continue execution.

(new) exception handler for the offending team. It then handles user commands, one of which may cause the team to be "resumed", in which case the debugger simply replies to the original exception message, freeing the team to continue execution until it either gets another exception or terminates normally. The next exception, if any, will of course be handled by the debugger.

> Note: If a process other than the team server had registered interest in the offending process before it incurred an exception, the exception message would have been forwarded directly to the registered exception handler. The handler can then take any action it deems appropriate, including loading the debugger as discussed above.

31.3. Summary

One of the principles guiding the V-System design has been to place as many the usual operating system functions outside the kernel as efficiency permits. Moreover, functions have been partitioned as far as practical into separate servers. Consequently, the kernel and each server have been kept reasonably small and independent of each other, which has in turn simplified debugging, maintenance, and experimentation with new servers.

— 32 —
Message Codes and Format Conventions

This chapter describes the standard message formats and codes used throughout the V-System.

32.1. Message Format Conventions

System server protocols obey several system-wide conventions. Every request message contains a 16-bit request code indicating the service requested. Similarly, every reply message contains a 16-bit code indicating the successful completion of the request's 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).

32.2. Byte-Ordering Considerations

V may run on a mixture of Suns and VaxStations. The former contain Motorola 680x0 processors which use the first (lowest-addressed) byte of a 16-bit or 32-bit quantity to store the most significant byte of the quantity ("big-endian"), while the latter use VAX-architecture processors which store the least significant byte first ("little-endian"). When processes running on the two architectures exchange messages, some conversion must be done (if messages included floating-point or other highly architecture-specific data, considerably more conversion would be necessary; to date, however, only 16-bit and 32-bit integers have been required).

The kernel, servers, include-files and library routines have been altered to perform the appropriate conversion (byte-swapping) for all code in the V-System distribution. However, anyone who implements a server or who uses message-passing that does not fit the client-server model should be aware of how byte-swapping is done.

The kernel always sends inter-kernel packets in its own byte order. A kernel which receives an IKC packet must determine the byte order of the packet and, if necessary, byte-swap the packet, including the message contained therein (see **IKC_LITTLE_ENDIAN**, **DifferentIKCByteOrder** and **SwapIKPacket** in **Vikc.h**). Currently, the kernel swaps the message as though every message were eight longwords, and treats any segment as a stream of bytes (hence does nothing to the segment).

Any further swapping of the message must be done by a process. We have adopted the policy that a client sends messages and gets replies without regard to byte order. It is then the responsibility of the server to perform any necessary swapping of requests and replies. The server can always determine the byte order of the message's sender because it is encoded in the sender's logical host id (see **LITTLE_ENDIAN_HOST** and **DifferentByteOrder** in **Venviron.h**). The server must, of course, take account of swapping performed by the kernel.

In many cases it is not actually necessary for the process to byte-swap messages at runtime; rather, the struct definition for the message can be #ifdefed for big- and little-endian architectures so that 8- and 16-bit fields automatically end up in the right place. The MicroVAX C compiler #defines **LITTLE_ENDIAN** for this purpose; the 680x0 C compiler does not. The definition of a **Kernelrequest** in **Venviron.h** demonstrates the use of **LITTLE_ENDIAN**.

32.3. 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 Chapter 27. The manifest constant definitions for these request codes are defined in the standard C include file **Venviron.h**.

32.4. 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.

AWOKEN — Returned by the kernel server when a Delay request was terminated by a Wakeup. (It is *not* returned by the Delay library routine, however.)

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_FORWARD
BAD_FORWARD is returned by the kernel when a Send is unblocked due to the receiver issuing an invalid **Forward** kernel operation.

BAD_PROCESS_PRIORITY
The request specified an illegal value for a process priority.

BAD_REPLY_SEGMENT
If a process invokes **ReplyWithSegment()** with a segment to which it does not have write access, the kernel sets the reply code of the message returned to the sender to BAD_REPLY_SEGMENT.

BAD_STATE — Request invalid at this time.

BUSY — The server cannot satisfy the request at this time, probably because a single-user resource is already allocated to another client.

DEVICE_ERROR
File or device-dependent error has occurred.

DUPLICATE_NAME
The request attempted to assign the same name to two different objects.

DISCARD_REPLY
This reply code is used with the **Reply()** primitive when the process receiving a message does not wish to reply. Reply messages containing this reply code are never delivered.

END_OF_FILE Attempt to read beyond file boundaries.

HAS_SUBSTRUCTURE
Returned by the storage server when a client attempts to remove a file that has a son in the directory tree. The attempt fails.

ILLEGAL_NAME
Returned by a server that deems a name to be illegal – for example, the name might be too long.

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_ID
The request contained a context identifier (see chapter 34) 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 when the user hits the BREAK (Ctrl-C) key. It currently isn't.

KERNEL_TIMEOUT
A timeout occurred 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.

MORE_REPLIES An operation request sent to a group was successful, and the client should use `GetReply()` to check for additional replies from other group members.

MULTI_MANAGER
The requested operation is not supported on multi-manager objects.

NO_GROUP_DESC
Returned when the kernel runs out of group descriptors.

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 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.

NO_TDS The server was not able to create a team needed to satisfy the request.

NOT_A_CONTEXT

The request asked the server to perform an operation that is only defined on contexts, but the specified object was not a context.

NOT_HERE The character-string name specified in the request does not specify an object implementing by the receiving server, but may be defined by some other server. This reply code is never returned to a message sent to a process group unless the replier knows that no member of the group implements the name.

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 43) which does not exist, or to an object which is not a session. *Obsolete.*.

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.

RETRY_UNICAST
The request was sent to a group, but the responding server refuses to perform it in parallel with other members of the group. The client should retry the request, this time as as a one-to-one **Send()**, not a multicast.

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 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.

— 33 —
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 that uses the **ReplyWithSegment()** kernel operation to return the data requested in a READ_INSTANCE must store the last block read and allow it to be read again, to provide

duplicate suppression on requests.

WRITE_INSTANCE operations on STREAMs always write to *lastblock*+1, where *lastblock* is a value returned by CREATE_INSTANCE or QUERY_INSTANCE. This block number is incremented after every write operation. The block number specified in the request message is ignored.

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.[15]

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 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

---

[15] 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 41) takes a different approach, creating a separate instance for each end of the pipe, with the connection between them invisible to the protocol.

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 34.10.

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.

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>.

33.1. 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, for servers that do not support character-string naming. This is used, for example, to specify the protocol to the internet server.[16]
filemode	Desired usage mode indicating FREAD, FCREATE, FAPPEND or FMODIFY, plus optionally FDIRECTORY or FEXECUTE.
unspecified	Server-dependent information specifying the file to be created, for servers that do not support character-string naming.
contextid	Specifies the context within the server in which the filename is to be interpreted. (See section 34.3.)
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.

[16] All newly written servers that provide the CREATE_INSTANCE operation should support character string naming and should not use the *type* or *unspecified* fields of the CreateInstanceRequest.

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 *filename* field of a CREATE_INSTANCE request specifies the type and properties of the instance to be created, perhaps by naming some existing permanent object. The request is issued either directly to the server or sent to a group including the server, as described in section 34.

The *fileid* and *fileserver* uniquely identify the file instance created. The file instance exists until released or until the requesting process ceases to exist.

33.2. 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.

33.3. CREATE DUPLEX INSTANCE

requestcode	CREATE_DUPLEX_INSTANCE

fileid	File instance identifier.
mode	Desired usage mode.

replycode	A 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, 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 CREATE_DUPLEX_INSTANCE request message, the server creates (or causes to be created) the "other side" of a duplex file (such as a bi-directional network connection, or a terminal). 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 CREATE_DUPLEX_INSTANCE request indicates the file instance does not exist.

33.4. 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.

33.5. 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. 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.

33.6. 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, 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 written is one greater than the last block in this file instance, regardless of the block number specified. 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.

33.7. 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.

33.8. 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.

33.9. 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.

33.10. QUERY FILE and NQUERY FILE

requestcode	QUERY_FILE
fileid	File instance identifier
unspecified	Server-specific.

requestcode	NQUERY_FILE
nameindex	The index of the first byte in the file name to use in the name mapping.
unspecified	Server-specific.
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.

33.11. 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.
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 device server uses MODIFY_FILE to change the data rate on RS-232 serial interfaces.

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.

— 34 —
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 decentralized, being 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 high-level names, a method for locating the server that implements any given named object, and a small set of request types that must be handled specially by any server that implements the protocol.

In this chapter, we describe the naming protocol in detail and give implementation hints for servers that use it. Refer to Chapter 25 for a description of the naming library routines available to client programs.

34.1. Overview

Conceptually, the V-System naming facility is a system-wide global directory providing reference by high-level (character-string) name to objects implemented by multiple object managers (servers). The global directory contains a (name, object)-tuple for each binding of global name to object.[17] Each client may also have its own directory of bindings from local names (or *aliases*) to global names. The naming facility provides operations for

- Binding names to objects
- Removing name bindings
- Name mapping: finding objects bound to a given name
- Inverse name mapping: finding the name bound to a given object

In the decentralized V naming facility, the global directory is distributed across the object managers such that each object manager stores and maintains that portion of the directory corresponding to the objects it implements. Each client program maintains a cache of bindings from name to object manager, as illustrated in Figure 34-1. When a client invokes an operation using a high-level object name, the client checks its cache for an entry that maps the name to an object manager. If a cache entry for the name is found (as is the case with *name2* in Figure 34-1), the operation and name are then sent to the object manager indicated by the cache entry. Otherwise, a query is multicast to the object managers to determine the correct object manager for the named object (as is the case for *name1* in Figure 34-1). If an object manager responds, a cache entry is created and the processing of the request proceeds as before, with the operation being sent to the responding object manager. Otherwise, the specified object name is assumed to be invalid and an error indication is returned to the client.

Inverse name mapping is simply a lookup in the global directory using an object's low-level identifier (for example, its instance identifier) in place of its high-level name. We assume that the low-level identifier provides enough information to determine which manager implements the object in question, and hence which manager stores the portion of the global directory containing its name. The same (absolute) global

[17] Note that high-level names are bound directly to objects, not to low-level names (such as globally unique numeric identifiers). Our design views high-level names as the only permanent, globally unique identifiers for objects.

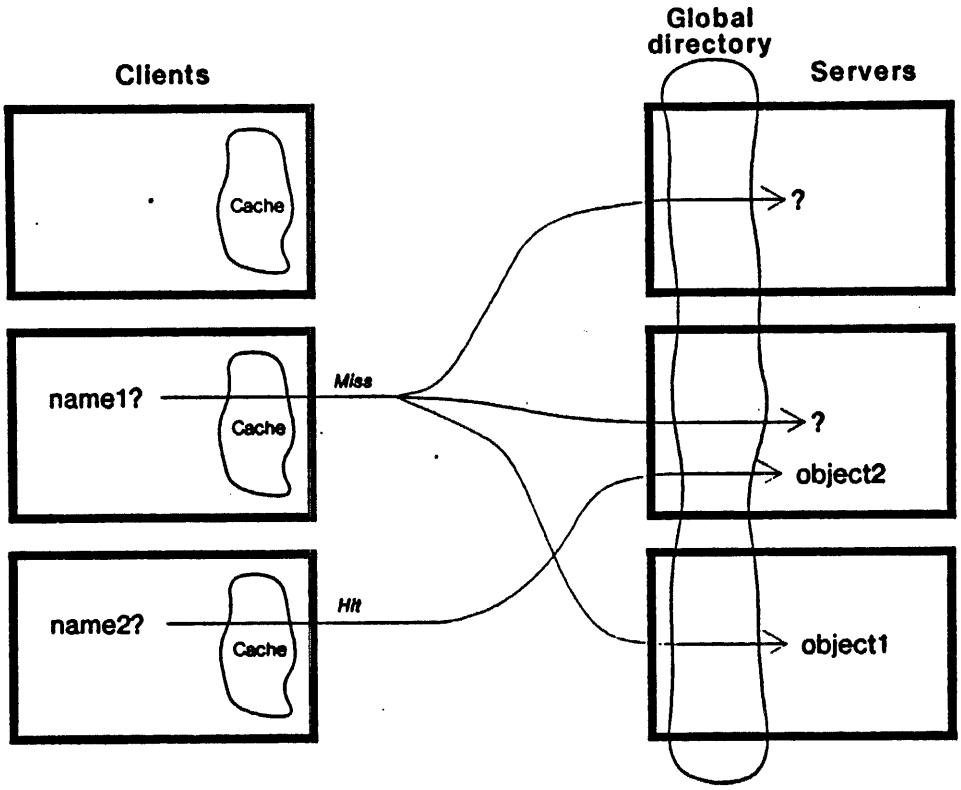


Figure 34-1: Decentralized Global Directory

name is returned for a given object even if the client originally accessed the object using a local name, alias, etc. Low level identifiers are not standardized across all object types, so the inverse name mapping operators provided are type-specific.

34.2. 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.

34.3. Contexts and Context Ids

The V-System name space is hierarchically structured, and we refer to each internal node of the naming hierarchy as a *context* Names are *pathnames* in that they describe a path through the hierarchy, beginning at (i.e., *relative* to) some specific context. Absolute names are those that begin at the root context.

The global directory is divided among object managers using a technique we call *vertical partitioning*. Each

object manager implements a tree of contexts starting *at the root* of the complete name hierarchy, thus storing the absolute names of the objects it implements. Some contexts (the root in particular) are implemented by multiple object managers. Such *multi-manager* contexts are partitioned across the managers that participate in their implementation. Each participating manager stores only that subset of the context needed to name objects it manages.

A context can be referenced by its absolute name, or by its *context identifier*: a compact low-level identifier that is effectively a pointer into the name space, providing direct, efficient access to the object manager(s) implementing the context. Referencing an object using a context identifier plus relative name allows the name lookup to start at the identified context rather than from the global root, thereby reducing the need for multicast and reducing the length of the name that must be looked up by the object managers.

In V, a context identifier is structured as a (*manager-id, specific-context-id*) pair, where the *manager-id* is a process identifier or process group identifier specifying the object manager(s) that implement the context, and the *specific-context-id* is mapped by the identified manager(s) to one of the contexts they implement. The standard system header file <Venviron.h> defines the types **ProcessId**, **ContextId** (corresponding to *specific-context-id*), and **ContextPair**, for these identifiers.

When a context is renamed, its old context identifier becomes invalid and another is assigned. Thus, in effect, a context identifier is bound to a context *name*, not to the context object itself.

Context identifiers are considered *hints*. That is, a context identifier is allowed to become invalid even if the corresponding character-string name is still bound to the same context. For example, if a V object manager crashes and is restarted under a different process identifier, all its old context identifiers become invalid (since they contain the manager's process identifier as a subfield), even if all the objects it manages are recovered.

34.4. Prefix Caching

The client naming library maintains a *prefix cache* mapping from name prefixes to context identifiers. Before sending off any CSname request, the **NameSend()** library routine finds the longest name prefix that can be matched in the cache. (If the matched prefix maps to a multi-manager context, **NameSend()** issues a QUERY_NAME request (see below) to obtain a longer prefix.) The matched prefix is then stripped off, and the resulting relative name is sent off together with the context identifier to which the prefix mapped. If the request fails because the stored context identifier was invalid, **NameSend()** removes the offending cache entry and retries the request, continuing until the request succeeds or the name is known to be invalid.

The naming library also caches the context identifier of each client process's current context ("working directory"). The context's absolute name is also stored, allowing **NameSend()** to recover if its context identifier becomes invalid.

34.5. Static Context Identifiers

Static context identifiers are defined for a few of the most commonly used multi-manager contexts. For each identifier, the specific-context-id portion is defined in <Vnaming.h> and the manager-id portion is defined in <Vgroupids.h>. Static specific context ids are small non-negative integers, less than the manifest constant MAX_WELL_KNOWN_CONTEXTS.

In principle these identifiers need only be known to servers. To improve performance, however, several of the identifiers and corresponding name prefixes are preloaded into client caches by the **PrimeCache()** library routine.

The following static (or *well-known*) context identifiers are defined at this writing.

(VCSNH_SERVER_GROUP, GLOBAL_ROOT_CONTEXT)
Corresponds to the CSname "[".

(VTEAM_SERVER_GROUP, TEAM_SERVER_CONTEXT)
Corresponds to the CSname "[team]".

(VSTORAGE_SERVER_GROUP, STORAGE_SERVER_CONTEXT)
Corresponds to the CSname "[storage]".

(VTIMER_SERVER_GROUP, TIME_SERVER_CONTEXT)
Currently not used for naming.

(VAUTH_SERVER_GROUP, AUTH_SERVER_CONTEXT)
Currently not used for naming.

(VEXCEPTION_SERVER_GROUP, EXCEPTION_SERVER_CONTEXT)
Currently unused.

(VDEVICE_SERVER_GROUP, DEVICE_SERVER_CONTEXT)
Currently unused.

(VINTERNET_SERVER_GROUP, INTERNET_SERVER_CONTEXT)
Currently unused.

(VPRINT_SERVER_GROUP, PRINT_SERVER_CONTEXT)
Currently unused.

(VVGT_SERVER_GROUP, VGT_SERVER_CONTEXT)
Currently unused.

(VPIPE_SERVER_GROUP, PIPE_SERVER_CONTEXT)
Currently unused.

(VEXEC_SERVER_GROUP, EXEC_SERVER_CONTEXT)
Currently unused.

(VTEST_SERVER_GROUP, TEST_SERVER_CONTEXT)
Reserved.

DEFAULT_CONTEXT
Conventionally used for the local root context by many servers that implement a single tree of single-manager contexts. The name is an anachronism, left over from a previous version of the naming protocol.

(VSTORAGE_SERVER_GROUP, PUBLIC_CONTEXT)
Holds publically-available V programs on storage servers. Corresponds to the CSname *[sys]*.

34.6. Generic Names and Group Names

A *group name* is a name that refers to a group (i.e., set) of objects, which need not all be implemented by the same server. A *generic name* refers to one member selected from such a group according to a rule associated with the name. The simplest (and most commonly used) rule is to select one member arbitrarily.

The naming protocol supports generic and group naming by permitting more than one server to respond to a CSname request. In general, the client issuing the name request determines whether the CSname is to be interpreted as a group name or generic name by receiving and processing all the responses (group name), or only the first (generic name with arbitrary selection). Selecting the first response has the pleasant side effect of favoring the most lightly-loaded server.

Servers can also define generic or group names for contexts. In this case the servers determine whether the name can be used as a group name, or only as a generic name. Issuing a **GetContextId** request on a group

name for a set of contexts must return a **ContextPair** that refers to *all* the contexts.[18] Subsequent name lookups that find the given prefix in the cache and substitute the returned context identifier will then correctly refer to all members of the set. In contrast, each response to a **GetContextId** on a generic name for the same set of contexts may return an identifier for just one server's member(s) of the set. Subsequent name lookups that find this prefix in the cache will then map it to the identifier that was returned in the first response.

34.7. 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.
namcindex	The byte offset of the name, within the segment specified by the last two long words of the message.
unspecified	Request-specific information.
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.

The CSname request format includes only the specific-context-id field of the **ContextPair** for the context in which the name is to be interpreted. The manager-id portion is implicitly specified by sending the request to the appropriate manager or group.

34.8. Name Lookup Algorithm

A server receiving a NameRequest performs the following algorithm to look up the name.

1. Set **CurrentNode** to the context specified by the namecontextid. If the context identifier is not recognized, fail with status INVALID_CONTEXT_ID. Go to step 4.
2. While the name still has unmapped components, do
   - Attempt to map the next component of the name, relative to **CurrentNode**.
     - If the name component is not defined locally, but **CurrentNode** is a multi-manager context, fail with status NOT_HERE. Go to step 4.
     - If the name component is not defined, and **CurrentNode** is not a multi-manager context, fail with status NOT_FOUND. (We know the name cannot be defined by any other server.) Go to step 4.
     - If the name component is an upward reference (".."), and the context **C** to which it refers is a multi-manager context implemented by a different (larger) group than **CurrentNode**,

[18] Multi-manager contexts follow this rule, with the context name viewed as a group name for the set of partitions held by the participating servers.

advance *nameindex* to the next component following the upward reference, set *namecontextid* to **C.cid**, and **Forward** the request to **C.pid**. Done.

- Otherwise, the name component is defined. Set **CurrentNode** to the object it maps to and repeat.

3. The entire name has been mapped, and **CurrentNode** is the named object. Done.

4. Fail.

- If the failure status was NOT_FOUND, and the request is of a type that permits automatic creation of an object in this case (for example, CREATE_INSTANCE in FCREATE mode on a file storage server), the object may be created at this point. Its name will be the remaining unmapped portion of the given name, defined relative to the context **CurrentNode**.
- If the request was multicast and the failure status was other than NOT_FOUND, do not reply (that is, invoke **Reply()** with replycode DISCARD_REPLY), since another server in the multicast group may have succeeded in processing it.
- Otherwise, the request was unicast. Reply with the failure status.

34.9. Standard CSNH Server Requests

There are several standard CSNH requests that must be implemented by all CSNH servers, plus a few optional ones. All of the request and reply formats described below are subsets of the ContextRequest structure defined in the standard system header file <Vnaming.h>.

34.9.1. QUERY NAME

requestcode	QUERY_NAME
nameindex	The byte offset of the name relative to *nameptr*.
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.
nameindex	Advanced to indicate the context prefix recognized by the responding server.
context	A **ContextPair** for the recognized context prefix.

Query a name to get information that can be cached, typically to avoid multicast when mapping the name in the future. Implementation required of all CSNH servers.

The query returns the shortest prefix of the given name that specifies a single-manager context, together with a context identifier for that context. If no prefix of the name specifies a valid single-manager context, but the entire name specifies a valid multi-manager context, the entire name is returned together with a context identifier for that context. Otherwise, the query fails. A failing query returns KERNEL_TIMEOUT if multicast, or a more specific error code if unicast. (Multicast is the normal case.)

A server receiving this request performs a variant of the general name-mapping algorithm, as follows.

1. Set **CurrentNode** to the context specified by the namecontextid. If the context identifier is not recognized, fail with status INVALID_CONTEXT_ID. Go to step 4.

2. While **CurrentNode** is a multi-manager context participated in by this server, do
   - Attempt to map the next component of the name, relative to **CurrentNode**.
     - o If the name has no more components, go to step 3.
     - o If the name component is not defined locally, fail with status NOT_HERE. Go to step 4.
     - o If the name component is defined, set **CurrentNode** to the object it maps to and repeat.[19]
3. Succeed. In the reply, set *nameindex* to point to to the first component of the name that was not mapped, or if the entire name was mapped, to just beyond the last character of the name (i.e., to the terminating null byte if there is one). Set *context* to the context identifier of **CurrentNode**. Done.
4. Fail. If the request was multicast, do not reply (that is, invoke **Reply()** with replycode DISCARD_REPLY), since another server in the multicast group may have succeeded in processing it. If the request was unicast, reply with the failure status.

34.9.2. GET ABSOLUTE NAME

requestcode	GET_ABSOLUTE_NAME
nameindex	The byte offset of the name relative to *nameptr*.
namecontextid	Context in which to interpret the given name.
nameptr	Pointer to a buffer containing a symbolic name, and in which the absolute name is to be returned.
namelength	Size of the buffer.

replycode	Standard system reply code.
context	If the given name specified an existing context, a **ContextPair** identifying it is returned in this field.
nameptr	The value provided is returned unchanged.
namelength	Length of the returned name.

Returns an absolute CSname for the object whose (relative) CSname is given. The returned name overwrites the given name. If the name was not bound to a context, *context.pid* is set to 0 in the reply. Implementation required of all CSNH servers.

A server receiving this request performs a slight variant of the general name-lookup algorithm. The named object need not exist, as long as it is clear what its absolute name would be if it were created. If the lookup fails with status NOT_FOUND, but it would be possible to create an object with the given name, the server constructs an absolute name by appending the undefined name suffix to the absolute name for the last **CurrentNode** reached.

34.9.3. 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

[19] Exception: upward references are handled as described in section 34.8

	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.
context	A **ContextPair** identifying the named context.

Given a CSname that names a context, this request returns a (serverpid, contextid) pair that identifies the same context. Implementation required of all CSNH servers.

34.9.4. GET CONTEXT NAME

requestcode	GET_CONTEXT_NAME
context	The **ContextPair** for which a name is to be found.
nameptr	Pointer to a buffer in which the name is to be returned.
namelength	Size of the buffer.

replycode	Standard system reply code.
nameptr	The value provided is returned unchanged.
namelength	Length of the returned name.

Inverse name-mapping for context identifiers. Provides a subset of the functionality of GET_ABSOLUTE_NAME. Implementation recommended for all CSNH servers.

Returns an absolute CSname for the context corresponding to the specified context identifier, if the context identifier is valid and known to the server receiving the request. This request should be sent to the process or group identified by the *pid* component of the **ContextPair**.

34.9.5. 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 is to be returned.
namelength	Size of the buffer.

replycode	Standard system reply code.
nameptr	The value provided is returned unchanged.
namelength	Length of the returned name.

Inverse name-mapping for instance identifiers. Returns an absolute CSname for the file associated with the

specified file instance. Implementation recommended for all CSNH servers.

34.9.6. RENAMEOBJECT

requestcode	RENAME_OBJECT
nameindex	The byte offset of the old name relative to nameptr.
namecontextid	Context in which to interpret the old name.
nameptr	Pointer to the segment containing the old and new names.
namelength	Length of the segment containing the names.

replycode	Standard system reply code.
context	A **ContextPair** identifying the named context.

Given a CSname for an existing object, this request binds a new name to the object and removes the binding of the existing name. Implementation is optional.

The new name must be absolute. (The initial "[" character is omitted.) It follows the old name in the segment, separated by a single null byte. The request fails, returning ILLEGAL_NAME, if the new name is in a portion of the name space not implemented by the object's current manager, and that manager is unable or unwilling to expand its name space as required.

34.10. Context Directories and Object Descriptors

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 a *context directory* for each context that it manages. A context directory appears as a file of records, with each record describing 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 obliged to make all fields presented in a directory modifiable. If a client attempts to change a non-modifiable field, that field is left unaltered, but any other changes indicated in the request are 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.

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 NREAD_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 NWRITE_DESCRIPTOR operation is available for modifying an object's descriptor.

A server need not 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>.

34.10.1. READ DESCRIPTOR and NREAD DESCRIPTOR

requestcode	NREAD_DESCRIPTOR
nameindex	The byte offset of the name relative to *segmentptr.*
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.
segmentptr	Pointer to a buffer that contains the object name and in which the descriptor is to be returned.
segmentlen	Length of the buffer.

requestcode	READ_DESCRIPTOR
fileid	File instance id of the object whose descriptor is to be read.
dataindex	The byte offset from the start of the specified segment where the returned descriptor is to be placed.
segmentptr	Pointer to a buffer in which the descriptor is to be returned.
segmentlen	Length of the buffer.

replycode	Standard system reply code.

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. Implementation of both is recommended for all CSNH servers.

34.10.2. WRITE DESCRIPTOR and NWRITE DESCRIPTOR

requestcode	NWRITE_DESCRIPTOR
nameindex	The byte offset of the name relative to *segmentptr.*
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.

segmentptr Pointer to a buffer that contains the object name and the new descriptor value.

segmentlen Length of the buffer.

---

requestcode WRITE_DESCRIPTOR

fileid File instance id of the file whose descriptor is to be modified.

dataindex The byte offset from the start of the specified segment where the new descriptor value begins.

segmentptr Pointer to a buffer that contains the new descriptor value.

segmentlen Length of the buffer.

---

replycode Standard system reply code.

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.

34.10.3. Multi-ManagerContext Directories

A multi-manager context directory is implemented as multiple context directory files, one per manager participating in the context. To list a multi-manager context directory, the client opens the context directory for each object manager in the context and then merges the object entries into a single list. Merging the lists entails eliminating duplicates, since some objects in the context may themselves be multi-manager contexts, and will thus appear in several managers' directory files. All the context directories for a context are opened in parallel, using a multicast CREATE_INSTANCE request. To compensate for the inherently unreliable delivery of multicast messages and responses, a followup message containing the list of managers from which replies were received can be multicast to the object managers. Only omitted object managers respond to the followup message. For full reliability, additional followup messages can be transmitted until no more replies are received.

The format of a followup message is as follows. The message structure **CreateInstanceRequest**, as defined in <Vioprotocol.h>, is used.

---

requestcode CREATE_INSTANCE_RETRY

filenameindex The byte offset of the start of the actual CSname, relative to *filename*.

type
unspecified
filemode
contextid
Identical to the corresponding fields of the original CREATE_INSTANCE request (chapter 33).

filename Address of a data segment beginning with an array of process ids specifying the managers that should not reply to the request, terminated by a process id of 0. The CSname appears later in the segment, as specified by *filenameindex*.

filenamelen Length of the segment.

The reply format is identical to that for CREATE_INSTANCE.

— 35 —
Authentication and the Authentication Server

Since processes are the active entities in V, the kernel associates each V process with a particular user or account on whose behalf it is acting. Each authorized user within a V domain is assigned a unique *user number*, and each V process bears exactly one user number.[20] A process runs with the privileges associated with its user, and that user is considered responsible for its actions. An *authentication server* maintains a database of information about each user, including login name, personal name, encrypted password, user number, and preferred home directory. The authentication server supports simple queries on this database, which is keyed by user number and by login name. The authentication server will also set the user number of a requesting process if the correct password for that user number is presented in the request.

The V authentication service does not provide a very high level of security. Its main purpose is to provide a sense of user identity to programs that need to exhibit user-specific behavior, and to protect against inadvertent mistakes. Its design is grounded on the belief that the benefits of increased security in a research system like V are very quickly outweighed by its cost in reduced performance and increased complexity.

35.1. Authserver

The authentication server itself is available as a program called **authserver**. Starting the server with the **-d** flag turns on debug output. The **-F** flag, followed by a filename, specifies a non-standard authentication database file.

The V executive automatically starts up an authentication server if none is running when a user attempts to log in.

35.2. User Numbers

In general, a process running with user number *u* has control over other processes running as user *u*, and over server-maintained objects owned by user *u*. Certain special user numbers are exceptions to this rule, however.

A process running with the predefined user number SUPER_USER has total privilege to do anything that the kernel and servers implement. The authentication server runs as super-user.

Somewhat more restricted privileges are associated with the user number SYSTEM_USER. Server processes that need special permissions to enable them to act on behalf of other processes, but do not need the full SUPER_USER privilege level, run as SYSTEM_USER.

User processes running on a workstation in the "not logged in" state have user number UNKNOWN_USER. The UNKNOWN_USER is somewhat more restricted than normal users, since not all processes running as UNKNOWN_USER really belong to the same person. An unknown user on one machine is not allowed to manipulate processes belonging to unknown users on other machines.

When a process is created, it initially has the user number of its parent. The root process of each workstation's initial team is created with user number SYSTEM_USER, allowing server processes on the first team to run as SYSTEM_USER if desired. User numbers can be queried with the **User()** kernel primitive

[20] Processes on the same team need not all run under the same user number.

or changed with the **SetUserNumber()** kernel primitive.

35.3. Authentication Library Functions

The following authentication functions are available in the standard V library.

```
SystemCode AddUser(name, passwd, fullname, home)
   char *name, *passwd, *fullname, *home;
```

Add a new user with the given login name, password, full name, and home directory. Returns OK if successful, else a standard system code indicating the reason for failure. The requesting process must be authenticated as SUPER_USER. Requires that an authentication server be running somewhere on the local network.

```
SystemCode Authenticate(name, passwd)
  char *name, *passwd;
```

Authenticate the calling process as the given user, specified by login name. Returns OK if successful, else a standard system code indicating the reason for failure. Requires that an authentication server be running somewhere on the local network.

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

Delete the user with the given login name from the authentication database. Returns OK if successful, else a standard system code indicating the reason for failure. The requesting process must be authenticated as SUPER_USER. Requires that an authentication server be running somewhere on the local network.

```
DestroyAuthRec(ar)
  AuthRec ar;
```

Free each string in the given AuthRec.

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

Return the full name of the user associated with the given process as a dynamically allocated string. The string should be freed by the caller when no longer needed, using **free**. Requires that an authentication server be running somewhere on the local network.

```
SystemCode MapUID(uid, ar)
  UID uid;  AuthRec *ar;
```

Obtain an AuthRec containing the given user's authentication database entry. The user is specified by user number. Returns OK if successful, else a standard system code indicating the reason for failure. Requires that an authentication server be running somewhere on the local network. Note: this function dynamically allocates several strings to construct the AuthRec. The caller should invoke **DestroyAuthRec(ar)** when the AuthRec is no longer needed.

```
SystemCode MapUserName(name, ar)
  char *name;  AuthRec *ar;
```

Obtain an AuthRec containing the given user's authentication database entry. The user is specified by login name. Returns OK if successful, else a standard system code indicating the reason for failure. Requires that an authentication server be running somewhere on the local network. Note: this function dynamically allocates several strings to construct the AuthRec. The caller should invoke **DestroyAuthRec(ar)** when the AuthRec is no longer needed.

```
SystemCode ModifyUser(ar)
  AuthRec *ar;
```

Modify the given user's authentication database entry to be as specified by the given AuthRec. The user is specified by the **uid** (user number) field of the AuthRec; a user with the given number must exist. Returns OK if successful, else a standard system code indicating the reason for failure. The calling process must be authenticated as the given user or as superuser. Requires that an authentication server be running somewhere on the local network.

```
SystemCode Password(name, passwd)
  char *name, *passwd;
```

Check whether the given password is correct for the given user name. Returns OK if so, else a standard system code indicating the reason for failure. Requires that an authentication server be running somewhere on the local network.

```
SystemCode SetUserNumber(pid, uid)
  ProcessId pid;  UID uid;
```

Set the given process's user number to the given value. Returns OK if successful, else a standard system code indicating the reason for failure. The kernel places the following restrictions on setting user numbers:

1. Any process can set its own user number to be UNKNOWN_USER.
2. Normal user processes are allowed to set the user numbers of descendents to match their own. (This privilege is useful if a parent process must change its user number after having created other processes.)
3. A process running as SYSTEM_USER can set its own user number, or that of any descendent, to match the user number of any process that is awaiting reply from it. (This privilege allows servers to create processes that act on behalf of clients.)
4. The SUPER_USER can set any process's user number to any value.

```
UID User(pid)
  ProcessId pid;
```

Return the user number of the user associated with the given process.

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

Return the login name of the user associated with the given process as a dynamically allocated string. The string can be freed by the caller—using **free()**—when no longer needed. Requires that an authentication server be running somewhere on the local network.

35.4. Adding a New User

The following is the recommended procedure for adding a new V user. See section 35.5 if you are installing V for the first time and need to add many users in one session.

1. If you are using a UNIX host for file service, create a UNIX account for the new user.
2. Under V, use the **su superuser** command to begin running as the V super-user.
3. Run the V **password** program.
   a. Click on the *user name* field, edit it to contain the user's desired login name, and hit return.
   b. Modify the *full name* field to contain the user's personal name, in the same way.
   c. Modify the *home* field to contain the V absolute pathname of the user's home directory.
   d. Click on *add*, and enter the user's desired initial password.
   e. Click on *exit*, or repeat to add more users.
4. Run **addcorr** and answer the prompts. When it requests a password, type the user's UNIX password.[21]

35.5. Authentication Database

There is one Vpassword file per network segment. For each user it contains a username, usernumber, password (using the same format as Unix), full name, and home directory. Each machine providing V fileservice needs a user correspondence file, which lives in /etc/V. It maps between V user numbers and the local Unix account name of the corresponding user.

The authentication server keeps its database in the file **[sys]misc/Vpassword**. This file should be made writeable only by the V super-user. The file format is similar, but not identical, to the UNIX password file. You can convert your UNIX password file to a V password file using the *awk* program provided in **/etc/V/Vpassword.awk**.

The authentication server supports a simple form of password file replication. The first few lines of each file copy should list the absolute names of the master and all slave copies of the password file, as follows:

```
master:[storage/pescadero/usr/V/misc/Vpassword
slave:[storage/gregorio/usr/V/misc/Vpassword
slave:[storage/navajo/usr/V/misc/Vpassword
```

Whenever a new authentication server starts up, it reads [sys]misc/Vpassword, which may come from any public Vserver. When modifications are made, it attempts to first modify the master file. If this file is inaccessible, no password files are updated and the authserver returns the standard reply code POWER-FAILER. If the master file is correctly updated then as many slave sites as possible are also updated.

Changes made when the master site is unavailable are kept in the authserver's in-memory database, so future updates may cause changes made when the master file server is up at a later date. In general, users should refrain from changing the authentication database when the master password file is inaccessible. The design goal is to have a close-to-current password file available if the master site is down when the authserver needs to be restarted. Redundant distribution of the master password file should be carried out to slave sites using *rdist* or similar tools on a regular basis. We do it every night.

Each UNIX system that makes its files accessible from V maintains a *correspondence table* mapping from V user numbers to UNIX login names. See section 43.1.1 for more information about correspondence tables. Another *awk* script, **/etc/V/Vusercorr.awk**, is provided to create this table.

---

[21] The user can run **addcorr** himself if you do not know his UNIX password.

— 36 —
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 like 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 using the standard name cache, 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 22. There are also some device-specific operations defined for some devices. The currently supported devices are described below.

36.1. Ethernet

The Ethernet interface is accessed by specifying a device name of the form **enet***ts*, 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>.

36.2. Disk

The disk interface is accessed by specifying the device name *disk0* or *disk1*. These names correspond to the first and second drives attached to the interface, respectively. Currently, only the Xylogics disk interface is supported.

In a create instance request, the filemode must be FCREATE. The type of the disk instance is always readable, writeable, multi-block.

Upon "opening" a disk device, the disk driver reads the label off of the first sector to obtain disk-specific information (such as the number of cylinders, number of heads, etc.). The disk label must have previously been written to the disk using the *diskdiag* program. The format of a disk label is defined in "/V/kernel/m68k/disklabel.h".

Read and write instance requests are standard and allow a maximum of DISK_MAX_BYTES (as defined in "/V/kernel/m68k/xyl.h", usually 64 kbytes) bytes to be accessed. The disk driver translates from a (block, byte count) pair to a (cylinder, head, sector, sector count) tuple.

A disk-specific QUERY_FILE request is supported that returns device access statistics (e.g., the average seek distance per I/O). A MODIFY_FILE request allows these statistics to be modified (e.g., reset to zero). The format for the QUERY_FILE reply message is given by the QueryStorageReply struct defined in <Vstorage.h>.

36.3. 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.

36.4. 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.)

36.5. 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.

36.6. Framebuffer

There are device drivers available for

36.6.1. Sun framebuffers

The current Sun (and Cadline) drivers allow one to enable and disable video output through modify file requests. The device may be opened and modified, or may be modified directly with a NMODIFY_FILE request. The following routine turns the framebuffer off:

```
#include <Vframebuffer.h>
FbOff()
  {
    ModifyFramebufferMsg *req;

    req.sysCode = NMODIFY←FILE;
    req.request = FB__OFF;
    req.nameindex = 0;
    req.nameptr = "[device]framebuffer";
    req.namelength = sizeof("[device]framebuffer");
    NameSend( &req );
  }
```

36.6.2. MicroVax QVSS Framebuffer

Caution is advised when using this device driver. Opening the device maps the QVSS frame buffer memory into one's address space. One cannot directly access the framebuffer until the device has been opened. To find where the framebuffer has been mapped, perform a QUERY_INSTANCE on the file and look at the field **unspecified**. This driver allows direct user access to QVSS device registers. The device is made up of six sixteen bit (short) blocks referencing the first six device registers. More information can be found in the top secret DEC Engineering specification **VCB01-KP**. **/V/kernel/vax/qvss.h** defines all of the useful control bits.

The registers available are:

```
0 Control Status
1 Cursor X position (not used by the V-System)
2 Mouse Position (x is low byte, y is high byte)
3 Spare
4 CRT controller address pointer
5 CRT data
```

36.7. Null Devices

Two null devices are available, and are normally configured into all versions of the V kernel. The *nullin* device is a readable, 0-length file; it thus returns an end-of-file indication on every read attempt. The *nullout* device is an endless sink for output.

— 37 —
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 prints an error message and 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. 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. EXCEPTION_REQUEST messages should only be generated by the kernel. The REGISTER_HANDLER request code is used both for registering and deregistering handlers.

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 error printing routine used by the exception server is available to other exception handlers as the library routine **StandardExceptionHandler**.

— 38 —
Exec Server

The exec server is the 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 environment of the V system, such customizations do not vanish even if the terminal agent (i.e., the VGTS or STS) is replaced; they remain as long as the user is logged in.

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 exec server is located by

```
GetPid(EXEC_SERVER, LOCAL_PID)
```

It is present in all the standard configurations of the Vsystem.

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 the CreateExec library routine.

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.

The message structure for the requests, the request values and the logical identification of the exec server can be found in the header file Vexec.h.

— 39 —
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. 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 are supported. [22]

39.1. Running the Internet Server

The internet server can be compiled as an independent V program, or linked into another program. As an independent V program, it is often loaded automatically some other V program (e.g., by telnet), so that users usually don't need to invoke it separately.

The standard V command "internetserver" may be run in the background to provide a local internet server on any workstation. The internet server program by default will only register the server for the logical id INTERNET_SERVER on a local basis. There are two optional switches that may be used when starting an internetserver. The **-g** option causes the internet server to register itself globally so that it can create connections for hosts anywhere in the V-System. This facility allows local hosts to avoid spending some 100K of memory for this server.[23] The **-d #** option causes the internet server to enable debug messages up to a severity of "#" (an integer in the range [0..9]; 0 is the default).

To include the internet server in another V program, have it create a process which executes the function

```
InitInternetServer(localFlag, debugFlag)
    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 internet server program described above, rather than linking it into another program.

39.2. Accessing the Internet Server

Once the internet server has been started it can be accessed using the V 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

---

[22] The Xerox PUP protocol is no longer supported (starting with V-System version 5.2). We continue to show the PUP protocol in some of the examples of this section for illustrative purposes only.

[23] Using a global internet server can degrade performance if many connections are being supported simultaneously. For bursty applications such as telnet connections, however, any performance degradation from using a global internet server is typically small enough to go unnoticed.

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 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. Although the internet server does not implement the full naming protocol (see Section 34), it does implement context directories. Thus, commands such as `listdir -l [internet/local]` return useful information.

An internet server connection is owned by the process which requested its creation. Ownership of a connection can be passed on to another process by means of the SET_INSTANCE_OWNER request. If the owner process should die then the connection is aborted.

39.3. 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.

39.4. 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 no 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. (There is one exception: when a packet is received with the urgent flag set, the next READ_INSTANCE receives a BEGIN_URGENT_DATA reply code with zero bytes of data. A similar zero-length reply of END_URGENT_DATA is returned when the point in the data stream indicated by the urgent pointer is reached.) 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).

39.5. 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.

39.5.1. 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 ordinary 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.[24]

39.5.2. 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.

39.5.2.1. 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

[24] 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.

whose owner has died signalling that the connection should be aborted. This second timer wakes up once every 5 seconds.

39.5.2.2. 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 queuing 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.[25]

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.

39.5.2.3. 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.[26] This has led 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

---

[25] 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.

[26] That is, it must be able to keep up with the (possibly many) hosts that are sending it packets.

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:

```
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.[27] 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[28] 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 enqueuing 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.

---

[27] The reason why a circular queue of this form is needed stems from the problem of maintaining these queues in an atomic manner.

[28] The protocol implementations to date have consisted of a single process per connection which alternately waits on client requests and processes its packet buffer queue.

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.

39.5.2.4. 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.[29] 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;                    /* Instance protocol type
                                    (TCP, PUP, ICMP, 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

---

[29] 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.

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:

```
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.

39.5.2.5. 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 */
    0                           /* ICMP */
  };
```

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, the PUP protocol at index 2, and the ICMP protocol at index 3. The PnetTable defines the IP network-level protocol at index 0 and the PUP network-level protocol at index 1.[30] 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 enqueuing 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

[30] 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 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().

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.[31] 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.

---

[31] 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)).

39.6. Monitoring and Debug Facilities

Normally the internet server runs in the background and is accessed using the standard mechanisms discussed in the previous sections. In situations where poor network or protocol behavior is suspected, it is often useful to inspect the internal state of the internet server and to observe the behavior of particular connections.

A simple approach to debugging or monitoring involves starting an internet server in debug mode (e.g., `internetserver -d 5`, where the debug level "5" is useful for debugging or monitoring a wide range of potential problems). Much of the debug information provided details the operation of TCP/IP connections, though some information about the V I/O protocol and other protocols is also reported.

Upon startup, the internet server reads the configuration database for the workstation on which it is running and prints out information about how it will route to various internet addresses.[32] This information typically takes the form shown below (for a workstation with internet address 36.8.x.y):

```
IP Gateway Table:
36.8 -> local
36 -> 36.8.0.4
default -> 36.8.0.1
```

The host at `36.8.0.4` is a gateway that can route to subnets within net `36` (Stanford), while the host at `36.8.0.1` is a gateway that can route to all non-Stanford hosts. This routing information is often useful in determining whether the configuration database for the workstation is set up properly. See Section 19 for a description of the V-System configuration database.

More flexible debugging is possible using a separate V program that is provided specifically for this purpose. There are many advantages to this approach: an internet server that is already running can be examined, and non-local internet servers can be inspected to name two. Typing

```
inquery
```

to a V executive will start a program that can be used for more advanced inspection (and modification) of the internal state of the internet server. Inquery will attempt to find an internet server on the local machine. If none can be found, or if a different internet server is of interest, the user must type additional commands as described below.

Once the inquery program has started, it will prompt for single letter commands. Most commands are intended for low-level debugging by program maintainers and are not described in detail. The commands that may be useful for user-level monitoring are described below in approximate order of usefulness:

`?` list available commands (including brief description).

`A` attach to the debug I/O stream of the internet server. The default is for debug I/O to go to stdout (as defined at the time the internetserver program was invoked). You must always use this option when inspecting non-local internet servers.

`U` unattach from the debug I/O stream of the internetserver (returning it to stdout). One typically uses "`d 0`" to turn of debug output before unattaching (since sending output to stdout is not always what is wanted).

`d` change the verbosity of the debug information that is printed. The user is prompted for a digit in the range [0..9], where 0 (the default) indicates silence and 9 indicates full verbosity. A value of 5 is appropriate for most user-level debugging, as this will cause only the most

[32] This is a temporary mechanism until more complete standards for internet routing in local network environments can be defined and implemented.

"interesting" events (e.g., retransmission of packets, bad packets, unusual events) to be reported. The effect of this command is identical to the **-d #** command line switch that can be given to the internet server at startup.

R reattach to (or relocate) an internet server. User is prompted for the process id of the internet server of interest (or 0 to mean the local internet server). The inquery program can only communicate with one internet server at a time.

V print version informatin about the internet server currently being inspected (workstation name, compilation date and time).

f lists free resources (e.g., buffers and network instances).

n list some basic information about active network instances.

p show detailed information about a particular network instance. Only implemented for TCP instances. Primarily useful to maintainers — see the DARPA Internet Handbook for clues to the meaning of this information. Fields of possible interest at the user-level include counts of retransmissions, out-of-order packets and packet delays (10ms units).

x immediately exit the internet server — abort any existing connections. The inquery program continues to run — use **R** to reattach to an internet server.

Q quit out of the inquery program (leaving the internet server running). The **U** and **d 0** commands are often used before using **Q**.

For typical user-level monitoring of a local internet server, the "**A**" command followed by the "**d 5**" command are the only commands that should needed. They allow a user to observe the frequency of retransmissions, receipt of bad packets, and other unusual events. This may be helpful in identifying the source of poor performance — flakey networks or gateways, incorrect or inefficient TCP/IP implementations, or just long network delays.

— 40 —
Memory Server

The memory server (or **memserver**) simulates a V storage server, storing files in main memory that is otherwise unused. It has no concept of file protection. It otherwise supports the standard V I/O and naming protocols. A file "foo" managed by the mem server can be accessed by referring to *[storage/local]foo* or *[storage/wsname]foo*, where *wsname* is the name of the workstation where the memserver is running.

A memory server is not part of the standard first team, but one can be started by using the **memserver** command (you will usually want to run this command in the background). On startup, the memory server checks that no other memory server or storage server is running on the workstation, to avoid name conflicts.

By default, the memory server will allocate as much main memory as it needs to store the files that it manages. It could thus use up all remaining main memory, if it so desired. The **-m** and **-k** flags can be used to place an upper bound on how much main memory the server can use for file storage. **memserver -m 100K**, for instance, limits the amount of storage space to 100K bytes. **memserver -k 100K**, on the other hand, sets the storage space limit to 100K bytes *less* than the total free memory that is currently available. ('M' can be used in place of 'K' to indicate 'mega' bytes.)

— 41 —
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.[33] 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.

[33] Actually only one reader and one writer are queued; the rest are replied to with a RETRY reply code.

— 42 —
Team Server

42.1. Overview

The team server manages the teams of a host. (Teams usually correspond to programs—although a program may consist of more than one team.) Specifically, it performs the following functions:

- Accepts requests to load teams. Requests can originate both locally and remotely, with the team server deciding whether or not remote execution requests will be accepted.
- Accepts requests to terminate teams.
- Implements a directory of all currently running teams. This directory can be read using the standard directory listing protocol.
- Implements round-robin scheduling for teams. Teams can be run in *foreground*, *background*, or *guest* mode. Typically, locally invoked programs are run either in foreground or background mode. Remotely executed programs are *only* allowed to run in guest mode, which is lower in privilege than either foreground or background mode. The team server also provides 4 real-time priority classes that run ahead of the three round-robin classes, and a "stopped" priority that ensures that no process on a stopped team will run.
- Registers itself as the exception handler "of last resort." The exception server forwards process exception messages to the team server if no one else has registered themselves for them. The team server invokes a postmortem debugger on the team of the process that incurs an exception.
- Responds to host state information requests. This is the mechanism upon which host selection for remote execution of programs is based.
- Acts as an agent for migration of logical hosts (i.e. remotely executed guest programs).

The team server resides on the "first team" of a host, i.e., it is considered to be a system server that is always present on a host and is loaded automatically when a host is booted. Various operations that the team server performs, such as team creation and team execution priority setting, are privileged operations that only processes on the first team may perform.

42.2. Team Loading

The team server is the only process that may create and load new teams. The library routines **LoadProgram** and **ExecProgram** provide the user interface to this function. These package up an appropriate request to the team server and take care of matters such as setting up the team environment block. 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 themselves, thus permitting them to outlive their load requestors.

Teams owned by the themselves are run in background mode, all others are run in foreground mode.

42.3. Team Termination and Exit Status Values

A teams can be terminated by having its root process destroyed using the **DestroyProcess** kernel operation, or it can exit voluntarily by calling the **exit()** library routine or returning from **main()**.

Calling **exit()** or returning from **main()** allows an exit status to be associated with the terminating team. Note: By convention, teams that are destroyed without having called **exit()** or returned from **main()** are considered to have exit status -1.

The team server also terminates any teams whose owners have died. It uses a timer process to periodically query the state of all teams which the server thinks are still running and their owners.

42.4. Host Status

The standard context directory listing protocol (see section 34.10) can be used to obtain information on all teams that are currently running. The command

```
listdir [team/local]
```

lists teams running on the local host, while the command

```
listdir [team/hostname]
```

lists teams running on the named host.

To obtain information on a specific team only, an NREAD_DESCRIPTOR request can be made. The command

```
listdesc [team/local][bin]telnet
```

prints information about a program running locally that was invoked under the name **[bin]telnet**. The team of interest can also specified by setting the request message's *contextid* field to the team's root process id; in this case the CSname (character string name) in the message should be null.

The team server also keeps track of host resource information such as the number of teams running, processor utilization, memory resources available, etc. It returns this information to requests it receives for host status information. These request messages are used primarily to implement host selection for remote execution of programs. Request messages can specify resource requirements and the team server will only reply if its resource state information conforms to the specified requirements. Request messages are typically sent to the well-known process group of all team servers (see include file **Vgroupids.h**), although they can be sent directly to a particular team server. (See the library routine **QueryHosts** for more details on remote host selection.) The command

```
listdir [team/hostname]
```

lists all team servers (and hence all hosts) on the network.

42.5. 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 and uses this information to determine whether or not a remote load request will be accepted.

Currently the system's host selection facilities will not select hosts on which a user has logged in. However, remote execution requests may still be sent to the team servers of such hosts and they will be accepted. This policy allows debugging programs to be executed on a host even when it has "hung" with a user logged in. The policy depends on the goodwill of users to not circumvent the standard host selection facilities. The **-x** option of the **login** command can be used to disallow all remote execution requests.

42.6. Round-Robin Scheduling

The team server implements a round-robin scheduling scheme for all teams except the first team and the workstation agent team (typically either the VGTS or the STS). These are typically run at real-time priority levels 3 and 4 respectively. As mentioned, teams can be run either in foreground, background, or guest mode. Foreground teams have priority over background teams, which have priority over guest teams.

Scheduling actually employs an additional team priority value for its implementation: a higher (i.e. more privileged) running priority. The running priority is used to implement the concept of a "time-slice" so that one team can't block out all other teams of the same priority.

Team priorities are user-settable with the **ChangeTeamPriority** operation, which allows users to request that the priority class of a team be changed, subject to authorization privileges. Users may change the priority of any team on a workstation they are logged in to, even if the **ChangeTeamPriority** request is sent from a remote location. Guest users of a machine cannot change their priority to anything other than *guest* or *stopped.*

42.7. Exception Handling

The team server is the exception handler of "last resort." It invokes the standard debugger in "postmortem mode" on the team of a process that has incurred an exception.

The debugger is invoked with the **-d** flag, so if the VGTS is in use, the debugger will pop up a new window for its command interaction. If the VGTS is not running on the workstation, however, the debugger will use the same standard I/O as the root process of the team that has incurred the exception, and may thus come up in a state where it is competing with an input reader process in the team incurring the exception. This can prevent input from reaching the debugger, in which case the debugger will not be of much use.

42.8. Migration

The team server's duties also include acting as an agent for migration. If a logical host is to be migrated from another machine then the team server must first accept the request and then act as a local agent for its implementation. Implementation includes setting up initial descriptor information in the local kernel and team server, and then participating in the transfer operation of the actual descriptor information.

The team server also implements usage policies with respect to migrating guest logical hosts (i.e. remotely invoked guest programs) away from the local machine. The current usage policy is to migrate guest programs whenever a user logs into the machine.

— 43 —
Unix Server

The V Unix server is a Unix program (and not a V program or command) designed to simulate a V kernel/storage server on a VAX Unix system (currently only Berkeley Unix 4.2 or 4.3). 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. Note: Unix servers are also frequently referred to as *V servers*. (Someday we may even implement such a server for an operating system other than Unix.)

Unix servers, like true V storage servers, implement the V-System naming protocol. The Unix system's complete directory tree is rooted at a node called **[storage/*hostname*]**, where *hostname* is the name returned by the Unix **gethostname()** routine (converted to lower case).[34] A Unix server may be either *public* (if it is started with the **-p** option), or non-public. A public Unix server implements the generic name **[storage/any**, and therefore such a host must maintain the up-to-date versions of all the standard V-System files and commands. On the other hand, hosts that run non-public Unix servers are not required to be kept up-to-date.

43.1. Sessions

If a V server is running on a Unix system, then remote access to the resources of this system is provided by *session* processes. Sessions are 'forked' copies of the main V server, each dedicated to a particular V user number. Like the main V server, each session appears externally as a regular V process. On each Unix host, the main V server, plus all of its sessions, belong to a local V process group. The 'group id' of this process group is usually used to communicate with the Unix server.[35] As mentioned above, each session is dedicated to a particular V user number. Any message that is sent to the common group id will be handled by the particular session that is responsible for that message's user number. (If no such session exists, then the main server will create one automatically.) The distribution of incoming packets to the individual sessions is handled by the *packet filtering* code in the Unix kernel (see the "installation notes" for further details).

> ***Warning:*** As an optimization, the packet filters for each session currently assume that the high-order 16 bits of each user number are zero. Thus, one should beware of using user numbers higher than 65535. This restriction is likely to be eliminated in future releases of the system.

43.1.1. User Correspondences

The main V server always runs as 'root'. The Unix uid of a session, however, is determined by a V-to-Unix *user correspondence* table, which is a mapping from V user numbers to Unix user names. The user correspondence table is maintained as a file on each Unix host. The name of this file is given by the macro USER_CORRESPONDENCE_FILE, defined in the header file **config.h** (in the Unix server source directory). (At Stanford, this file is named **/etc/V/Vusercorrespondence**.) For security, the user correspondence file should be writeable only by "root". The file should contain user correspondences for at least the following V user numbers:

0 (SUPER_USER), and 1 (SYSTEM_USER)

---

[34] *hostname* can also be set by starting the server (**Vserver**) with the **-n** option.

[35] *Individual* pids are used for file instance I/O, however.

These should correspond to whatever Unix account is used to manage V-System files on Unix (although preferably *not* "root").

2 (UNKNOWN_USER)
This should correspond to a Unix user with very few privileges.

43.1.1.1. Adding and Deleting User Correspondences

If a user correspondence for a particular V user number is not present in the user correspondence file, then the user correspondence for the V UNKNOWN_USER is used instead. That is, a session for a V user who does *not* have a user correspondence will run with the same permissions as a non-logged in user. In addition, such users are not permitted to execute programs remotely on the Unix host (see section 43.3).

A V user can use the (V) **addcorr** program to create (or modify) a user correspondence on any Unix host (provided that it is running a V server, of course). **addcorr** repeatedly prompts for a host name, then a (Unix) user name and password on this host. It then attempts to create a new user correspondence. (If this is successful, then any existing correspondence for the V user will be removed.)

The **delcorr** program can be used to delete an existing user correspondence. **delcorr** repeatedly prompts for a host name, and attempts to delete an existing user correspondence for the V user on this host.

The V SUPER_USER can use **addcorr** and **delcorr** to modify user correspondences for *any* V user (not just for SUPER_USER). (In this case these programs will also prompt for a V user name.)

The following additional points should be noted:

- When a user correspondence for a V user on a particular host is added/modified (using **addcorr**) or deleted (using **delcorr**), then any existing session for the V user on this host will be destroyed. (Subsequently, a new session will still be created automatically, if needed.)
- The V UNKNOWN_USER is not permitted to modify his own user correspondences.
- For security, the Unix servers do not allow user correspondences to be made to "root", nor to Unix accounts with a null password.
- If your current context is a Unix V session, then the (Unix) **whoami** program can be executed remotely, in order to show the Unix uid of this session. (Recall that the (V) **name** program can be used to show your exec's V user number.)

43.1.2. Lifetime of Sessions

A session will die automatically if it has been inactive for a certain period of time (defined by MAX_SESSION_INACTIVITY in **config.h**—15 minutes at Stanford), and if it is not maintaining any instances with valid owner pids.

43.2. File Access

When a session receives a CREATE_INSTANCE request, it attempts to open the named file. If the session 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, 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. This protection mode is given by the macro

DEFAULT_CREAT_MODE, defined in **config.h**. A client may change the mode bits using a WRITE_DESCRIPTOR or NWRITE_DESCRIPTOR request.

43.3. 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. This search path is given by the macro PROGRAM_SEARCH_PATH, defined in **config.h**.[36] 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, the 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.

43.4. File Descriptors

The server supports 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

[36] Alternatively, the search path can be found by executing the Unix command **printenv**. This will display the environment variables that are passed on to programs executed via the session.

the system include file **<Vdirectory.h>**.

43.5. Debugging Sessions

It is possible to turn on debugging output from a session (or the main server), by 'killing' it with the SIGTSTP signal. Debugging output is redirected to the file **/tmp/VserverDebug***n*, where *n* is the (Unix) pid of the server or session. To turn off debugging output, kill the process with the SIGTSTP signal once again. *Warning:* Debugging should be turned off as soon as possible, because this file quickly gets to be *very* big. Note that debugging output is likely to be of use only to wizards.

If your current context is a Unix V session, then the (V) **instances** program can be used to find out the status of whatever file instances this session is maintaining at the time.

— 44 —
Workstation Agents

Workstation agents are a generic class of server used in the V-System. A workstation agent has the duty of mediating between the workstation 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 input stream but deletes a character from the input stream. It translates the newline character '\n' into a carriage return/linefeed sequence on workstations 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. Workstation 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 workstations.

The V system currently contains two different workstation agents, the Simple Terminal Server (STS) and the Virtual Graphics Terminal Server (VGTS). The Simple Terminal Server is a minimal workstation agent. It provides a single i/o stream, using the terminal facilities provided by the kernel console device, 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 workstation agents, or any other workstation agent, without any knowledge of which workstation agent is present.

The **newterm** command allows the user to replace the workstation agent on his workstation without rebooting the workstation.

44.1. Implementation of Workstation Agents

These are the requests that should be supported by a workstation agent, at the minimum:

- It should support the V I/O protocol for **INTERACTIVE_STREAM** files. In simple cases, it may give polite replies to CREATE_INSTANCE and RELEASE_INSTANCE without really doing anything, as the STS does.
- It should support the **QueryPadRequest** and **ModifyPadRequest** messages in the fashion expected by **QueryPad()** and **ModifyPad()**. In particular,**ModifyPad(file, 0)** should turn off all "cooking", giving the client access to the raw, unadorned terminal.

In addition, the following conventions should be observed, in order to allow the **newterm** command to work:

- Upon starting up, the workstation agent should join the local workstation agent group.
- It should support the **Die** request message, which is a polite way of asking the workstation agent to expire.

— 45 —
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.

45.1. STS Line Editing Facilities

The STS provides a superset of the line editing facilities that are 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.**

As well as the line editing commands described in section 2.5, the STS also supports the following commands:

CTRL-l	Re-display the input buffer.
CTRL-n	Move cursor down one screen line.
CTRL-p	Move cursor up one screen line.
CTRL-q	Quote next character. Control characters are displayed as '↑C'.
CTRL-y	Move the contents of killbuffer into the input buffer, inserting at the current cursor position.
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.
ESC-.	Move cursor to the end of the input buffer.
ESC-BACKSPACE	Same as ESC-b.
ESC-d	Kill from the cursor to the end of the current word.
ESC-DEL	Same as ESC-h and CTRL-w.
ESC-t	Transpose the two words preceding the cursor.

45.2. 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.

45.3. Remote Terminal Server

The Remote Terminal Server (RTS) supports the same interface as the STS, but encapsulated in the ARPA TELNET Protocol; its standard input and output are normally a TCP connection opened by the telnet server (p. 4.1). Like the STS, the RTS uses execs created by the local exec server; this may lead to difficulties if there is another telnet or local user on the same host, as the exec server assumes it serves only one user at a time.

The RTS violates the standard protocol on two points: it insists on echoing input (under the control of client programs) even if the ECHO option is not successfully negotiated, and it does not send go-aheads as may be required by some hosts to support half-duplex terminals. These violations are typically not a problem in practice, as most user Telnet implementations support these options. All other options are properly refused.

The RTS works with the Heath-19 terminal (in ANSI mode), the VAT provided by the VGTS, the SMI PROM monitor, and possibly others.

— 46 —
Virtual Graphics Terminal Server

The Virtual Graphics Terminal Service (VGTS) allows the display of structured graphical objects on workstations (with appropriate displays) that run the V system. This chapter describes how the standard library routines interface to the VGTS, as well as describing some of the VGTS's internal structure. Applications programmers usually need not concern themselves with the details of this section; instead they should consult the "Graphics Functions" section of the manual (section 29).

46.1. 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. (Sun model 50 workstations also use the model 120 framebuffer.) Users usually will not have to concern themselves with this distinction, since **team1-vgts** (the default first team) automatically loads the correct version of the VGTS shortly after it begins running. 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 workstation's 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.

46.2. AVT 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 AVT.
TAB	Positions the cursor at next multiple of eight (plus one) columns, erasing characters between the current cursor position and the new position. WARNING: this behavior is not VT100 compatible and is subject to change.
FF	Clears the AVT.
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 scrolling region, all lines in the region 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. **WARNING: this behavior is not VT100 compatible and is subject to change.**
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 AVT.
ESC [*n* J	Clear -- if the argument is 2, clear the entire AVT. Otherwise, clear to end of AVT.
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. **WARNING: this behavior is not VT100 compatible and is subject to change.**
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 AVTs 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 an AVT is 28 lines by 80 columns. This terminal type is just a 28 line VT-100, with a few additional escape sequences as described above. On (Stanford) Unix 4.2 systems, this corresponds to the terminal type **vgts** (or **vgts28**). (Other common AVT sizes are also supported in the Unix *termcap* file, namely **vgts24**, **vgts48** and **vgts64**.) 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.

46.3. VGTS Message Interface

This section describes the internal message interface to the VGTS.

46.3.1. I/O protocol requests

The following requests of the I/O protocol (see section 33) are supported:

CREATE_INSTANCE
: Causes a new AVT to be created. The view manager will let the user decide where to put the upper left corner of the AVT 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 AVT. The filename field of the message is used as the name of the virtual terminal. Usually this is invoked only by the **OpenPad()** routine described in section 29.

QUERY_INSTANCE
: Returns the standard values, the same as a Create Instance reply.

WRITE_INSTANCE
: Write the bytes to the AVT 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 AVT. If there are any characters already in the event queue for this AVT, 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 AVT is deleted, along with any views of the AVT, and storage is reclaimed.

SET_BREAK_PROCESS
: The break process for each instance is the process which will be killed if the view manager "Kill Program" command is invoked within the AVT.

SET_INSTANCE_OWNER
: Changes the (process) owner of the AVT.

46.3.2. Workstation Agent Requests

The following request codes (and associated message structures) are defined in **<Vtermagent.h>**:

QueryPadRequest Returns the cooking mode bits for the AVT, as well as the AVT's width and height.

ModifyPadRequest
: The AVT's cooking mode bits and/or size are modified. The structure ModifyMsg describes the format of this message.

SwitchInput
: The specified AVT is selected for input. This is used in the **SelectPad()** routine.

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 virtual terminal.

SetBannerRequest
: The specified virtual terminal's banner string is changed. This request code is used by the **SetVgtBanner** routine.

RedrawRequest
: The specified AVT is redrawn.

LineEditRequest
: The data in the message are treated as line editing commands, rather than simply being output to the AVT. Note, however, that the line editor treats most characters as self-inserting (see section 2.5).

GetRawIO
: The server and instance ids of the VGTS's own stdio are returned to the requester. The **newterm** command uses this code in order to determine what stdio to give the new workstation agent.

Die
: This code requests the VGTS (or other workstation agent) to commit suicide. This is used by the **newterm** command, as a *temporary kludge only* (to circumvent current problems with the system's user number and permission checking policy).

46.3.3. Other requests

SET_DEBUG_MODE
: Sets (or clears) debugging flags within the VGTS. This code is used by the **debugvgts** command.

46.4. Internal Organization

The current VGTS implementation consists (logically) of the following modules ('modules' in this description do not necessarily correspond to procedure names or source files):

- 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 generally device independent.

- Drawing primitives. There is one module which implements device dependent graphics primitives. It is conditionally compiled for different graphics devices.
- 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 allows the user to create, destroy, and modify the screen layout, using the mouse.
- 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.

46.4.1. Executive Interface

The V-System is intended to be modular, so VGTS could conceivably 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, using the **CreateExec()** library routine. The Executive calls the functions **SetVgtBanner(file, banner)** and **SetBreakProcess(file, pid)** as commands are executed.

46.4.2. Frame Buffer Interface

The device-dependent parts of the VGTS currently reside in the files **draw1.c** and **draw2.c**. The **gl__...()** macros form the interface to the underlying graphics device. These macros are defined in the include files **gl__sun100.h** and **gl__sun120.h**. (Which include file is used depends upon which version of the VGTS is being compiled.)

46.5. Debugging the VGTS

The **debugvgts** command allows the user to obtain a trace of certain events within the VGTS. The command syntax is

debugvgts *<debug code> <VGT #>*
or
debugvgts trace *<VGT #>*.

In the first form, the debug code (interpreted as a hexadecimal number) is a disjunction of bit flags taken from those defined in the system header file **<Vgtp.h>**. *<VGT #>* is the number of a text VGT to which debugging output is to be redirected. If this number is not that of a valid text VGT, then debugging output is directed to the VGTS's stdout (the console) instead. Once VGTS debugging has been turned on, it can be turned off again using a debug code of 0. A debug code of 0 is also useful for redirecting trace output as explained below.

In the second form of the **debugvgts** command, the top-level symbol associated with *<VGT #>* is dumped in a symbolic textual form to the current output (as declared in the first form.) This is useful for debugging programs that use the graphics capabilities of the VGTS as well as debugging the internals. if *<VGT #>* is positive, then interactive mode is used, and the trace routine pauses after each item listed. If it is negative, then the top level symbol of the VGT specified by the absolute value of *<VGT #>* is dumped without pausing.

Part IV:
Appendices

— Appendix A —
A V-System Bibliography

[1] E.J. Berglund and D.R. Cheriton.
Amaze: A distributed multi-player game program using the distributed V kernel.
In *Proc. 4th International Conference on Distributed Computing Systems*, pages 248-253. IEEE, May, 1984.

[2] E.J. Berglund and D.R. Cheriton.
Amaze: A multiplayer computer game.
IEEE Software 2(3):30-39, May, 1985.

[3] D.R. Cheriton.
An experiment using registers for fast message-based interprocess communication.
Operating Systems Review 18(4):12-20, October, 1984.

[4] D.R. Cheriton.
Local networking and internetworking in the V-System.
In *Proc. 8th Data Communications Symposium*, pages 9-16. ACM/IEEE, October, 1983.
Proceedings published as *Computer Communication Review* 13(4).

[5] D.R. Cheriton.
The V Kernel: A software base for distributed systems.
IEEE Software 1(2):19-42, April, 1984.

[6] D. R. Cheriton and T. P. Mann.
A Decentralized Naming Facility.
Technical Report, Computer Science Department, Stanford University, February, 1986.
Submitted to *ACM Transactions on Computer Systems.*

[7] D.R. Cheriton and T.P. Mann.
Uniform access to distributed name interpretation in the V-System.
In *Proc. 4th International Conference on Distributed Computing Systems*, pages 290-297. IEEE, May, 1984.

[8] D.R. Cheriton and W. Zwaenepoel.
Distributed process groups in the V kernel.
ACM Transactions on Computer Systems 3(2):77-107, May, 1985.
Presented at the SIGCOMM '84 Symposium on Communications Architectures and Protocols, ACM, June 1984.

[9] D.R. Cheriton and W. Zwaenepoel.
The distributed V kernel and its performance for diskless workstations.
In *Proc. 9th Symposium on Operating Systems Principles*, pages 129-140. ACM, October, 1983.
Proceedings published as *Operating Systems Review* 17(5).

[10] J.L. Edighoffer and K.A. Lantz.
Talicsin: A distributed bulletin board system.
Presented at the 2nd International Conference on Computer Message Systems, IFIP, September 1985. Proceedings to be published by North-Holland.

[11] K.A. Lantz.
An architecture for configurable user interfaces.
Presented at the Working Conference on the Future of Command Languages: Foundations for Human-Computer Interaction, IFIP Working Group 2.7, September 1985. Proceedings to be published by North-Holland.

[12] K.A. Lantz, D.R. Cheriton, and W.I. Nowicki.
Third generation graphics for distributed systems.
Technical Report STAN-CS-82-958, Department of Computer Science, Stanford University, February, 1983.

[13] K.A. Lantz and W.I. Nowicki.
Structured graphics for distributed systems.
ACM Transactions on Graphics 3(1):23-51, January, 1984.

[14] K.A. Lantz and W.I. Nowicki.
Virtual terminal services in workstation-based distributed systems.
In *Proc. 17th Hawaii International Conference on System Sciences*, pages 196-205. ACM/IEEE, January, 1984.

[15] K.A. Lantz, W.I. Nowicki, and M.M. Theimer.
An empirical study of distributed application performance.
IEEE Transactions on Software Engineering SE-11(10):1162-1174, October, 1985.

[16] K.A. Lantz, W.I. Nowicki, and M.M. Theimer.
Factors affecting the performance of distributed applications.
In *Proc. SIGCOMM '84 Symposium on Communications Architectures and Protocols*, pages 116-123. ACM, June, 1984.

[17] W.I. Nowicki.
Partitioning of Function in a Distributed Graphics System.
PhD thesis, Stanford University, 1985.

[18] M.M. Theimer, K.A. Lantz, and D.R. Cheriton.
Preemptable remote execution facilities for the V-System.
In *Proc. 10th Symposium on Operating Systems Principles*, pages 2-12. ACM, December, 1985.
Proceedings published as *Operating Systems Review* 19(5).

[19] W. Zwaenepoel.
Message Passing on a Local Network.
PhD thesis, Stanford University, 1985.

— Appendix B — 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.

B.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.

B.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;
```

B.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;
}
```

B.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 ");
```

B.5. File Contents

File contents are arranged as follows.

1. Initial descriptive comment (see example below), containing a brief descriptive abstract of the contents. Some programmers also add a list of all defined procedures in their defined order, or alphabetically.
2. Included files (avoid the use of absolute path names)
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)
```

B.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 expression, so there is a space between the keyword and the parenthesized expression. This also clearly distinguishes the statement from a function call.

```
if (FuncA())
  {
    FuncB((a = b) == 0);
    return Nil;
  }
else
  {
    FuncC(a, b, c);
    return ToSender;
  }
```

Alternatively, 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;
```

Note that parentheses are not syntactically required around the expression of a **return** statement. Nevertheless, such parentheses may still be included, if so desired.

Note that spaces are used to separate operators from operands for clarity and may be selectively omitted to suggest precedence in evaluation.

B.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 (an array) or MsgStruct (a structure with appropriate fields), 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. Each message format contains enough padding to fill it out to the fixed message size used by the kernel. Where the same space is used for messages of multiple formats (for example, both request and reply messages), access to the space for the message can be made by casting a pointer to the space to be of the type of the message format requires. 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 C — Installation Notes

This document describes the installation and maintenance of 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.

C.1. V-System Distribution Tapes

The software is distributed on a 1600 bpi Unix tar format tape. Licensing information and tapes can be obtained from:

Office of Technology Licensing
Suite 250
350 Cambridge Ave.
Palo Alto, CA 94306
(415) 723-0651

All the software is under copyright protection, so you must get a license from Stanford to have this software. New versions of the software may be released from time to time. Send comments on the software and documentation to the Arpanet address **vbugs@pescadero.stanford.edu**. Please report any bugs you find, or improvements you make.

The full V distribution consists of two tapes, the binary distribution (or binary tape), and the source distribution (or source tape) which has the sources to the V system itself. **V6.0** combines both of these on a single tape, with the "logical binary tape" first. **The combined space requirement for V6.0 is approximately 67 megabytes.**

Note: This V distribution runs on Cadlinc and SUN MicroSystems workstations with 68000s (SUN-1s), SMI workstations with 68010s and 68020s (models 2/{50,100,120,170} & 3/{75,160}), and DEC MicroVAX-II workstations with and without framebuffers. Ethernet drivers are provided for the 3COM and Excelan 10meg boards, the Intel 82586 LAN chip (found in SMI 2/50 and 3/75 models), the SMI 3 meg board, and the DEQNA. There is presently no driver for the SMI Multibus 10 Mbit Ethernet interface or for the AMD 7990 chip found in the Sun-3/50. The VAX server host side must be running 4.2 or 4.3 BSD.

C.2. Binary Distribution Tape

The binary tape contains 3 tar (Unix tape-archive format) files:

enet: The files required to install our ethernet packet filter code in a Vax/Unix 4.2 kernel. The installation.doc and enet.doc files that are part of this tape file describe how to install the driver and how it works, respectively. This code should be included in the official 4.3BSD release.

unix: The binaries and some sources for Unix support programs and servers used with V.

usr.V: Files used by workstations running the V-system.

- V binary images for kernel, servers and programs.
- V libraries for C programs in Unix tar format.

- V header files for C programs, defining standard manifest constants and structures.
- Miscellaneous configuration and documentation files.

The binary tape is structured to be loaded into a single subtree of a Unix file system. Make a subdirectory on a partition with at least 65 megabytes free. We suggest calling it V6.0. Change to the V6.0 directory, make another subdirectory called **enet** and change to it. Extract the packet filter from the tape. Change to V6.0 and run **tar x** repeatedly to extract the remaining files from the tape. (Due to extra end-of-files, you may get a "0 blocks read" message between files.)

Several programs expect to find files in certain directories. Run the **Vlink** shell script to link the distribution into the file structure.

Add the termcap.vgts entries to /etc/termcap.

C.2.1. V Subdirectories

There are several subdirectories of interest under the "V" subdirectory in the installation directory:

bin	Binaries of the V-system commands and servers. These are the programs that run under V on the workstations.
boot	This directory contains standalone programs (programs that do not run under the V kernel) such as **Vload** (the V-System bootstrap), and the **netwatch** family of network monitoring programs, plus initial teams to run under the V kernel. The subdirectory "Vkernel" contains various V kernel configurations.
config	Workstation configuration files. One config file exists per workstation. These files are used by the **ndserver** to determine which version of **Vload** to download, by **Vload** to determine which kernel image to load, and by the internetserver to determine the workstation's IP address.
fonts	Type fonts used by the VGTS and other V-System programs.
include	The include (**.h**) files. Most V-System include files start with an upper case "V".
lib	The V-System libraries. The main run-time library is libV.a Other major libraries are libsun100Vgts.a for the SUN-1 frame buffer and libsun120Vgts.a for the SUN-2 framebuffer, raw character I/O for various hardware configurations, and lowlevel V ike libraries for standalone programs. There is also a library for each of the different servers, e.g. libVinternet.a contains the Internet server, providing primarily IP/TCP service.
misc	Other random files.
run	Various files that are used by programs for runtime support.

C.2.2. Network File Service and Bootloading

The next step is to provide network file service and bootloading service.

1. Provide access to the Ethernet on your VAX/UNIX system (the only configuration fully supported by the distribution tape).
2. Modify the configuration files, under V/config to indicate the desired configuration and network addresses of your workstations.
3. Install and initiate the execution of the Vserver and ndserver on the Vax.

C.2.2.1. Ethernet Filter Code

This code must be added to the Unix kernel, if not already present. It allows a user program to open the Ethernet directly for reading and writing as a special device file. The user program can then specify by a "filter" which packets it wants to receive. See enet/Installation.doc for installation information and enet/enet.doc for a brief description of the code. Note that 4.2 BSD does not currently provide this functionality. However, a group at Stanford has convinced the weenies at Berkeley to include this driver as a standard part of Berkeley Unix, starting with the 4.3 release. Make sure the maximum packet size (MTU) is large enough to fit all the data bytes in a kernel packet plus the header, currently about 1200 bytes. The **V6.0** release packet filter is incompatible with previous packet filter releases.

In addition, the network driver files in_proto.c and ip_input.c should be replaced in /usr/src/sys/netinet. This adds the IPPROTO_ND protocol family (used by SMI boot proms) to the kernel.

C.2.2.2. Ethernet multicast reception

Important: Make sure that your Unix ethernet driver is set up to receive all multicast packets. By default most drivers do not listen to these packets. Multicast is now a fundamental part of the V interkernel protocol.

C.2.2.3. DEC Deuna

Change the initialization line in **/usr/src/sys/vaxif/if_de.c** to include the multicast enable bit:

```
#ifdef  STANFORD
        /* receive all multicast packets to keep V people happy */
        ds->ds_pcbb.pcbb2 = MOD_TPAD|MOD_HDX|MOD_ENAL;
#else
        ds->ds_pcbb.pcbb2 = MOD_TPAD|MOD_HDX;
#endif  STANFORD
```

C.2.2.4. Interlan 1010a

Before setting the Interlan board online send an additional command to enable reception of all multicast packets (in **/usr/src/sys/vaxif/if_il.c**):

```
#ifdef  STANFORD
        /*
         * For V people: receive all multicast packets
         */
        addr->il_csr = ILC_ALLMC;
        while ((addr->il_csr & IL_CDONE) == 0)
                ;
#endif  STANFORD

        /*
         * Set board online.
         * Hang receive buffer and start any pending
         * writes by faking a transmit complete.
         * Receive bcr is not a muliple of 4 so buffer
         * chaining can't happen.
         */
```

C.2.2.5. Configuration Files

The "config" files provide information about individual network nodes or workstations. If a node has Ethernet address AAAAAAAAAAAA (in hex) then its configuration file should be config/C.AAAAAAAAAAAA. In general, C.* files describe a network node, G.* files describe routing to be used by a gateway and S.* files are scripts for servers to execute on initialization. Config files contain

information fields of the form "name:value". Several exapmles are included in **V/config**. See chapter 19 for a full description of the keywords and their appropriate values.

The fields needed for booting are:

name — Name to be used as a user-specified designation of network node.

bootfile — Which version of **Vload** to download. Sun-2/50s must have this field set to Vload50.d. It defaults to Vload10.d for Suns and Vload.vax for microVaxen.

alt-ether-addr — Sun-2 workstations that have 3COM ethernet interfaces use the SMI address (0800.2001.xxxx) for booting and the 3COM address (0260.8c00.xxxx) when running the V-System. To configure such a workstation, name the config file after its 3COM address and put the SMI address in the alt-ether-addr field. SMI workstations generally print their SMI-assigned ethernet address on the screen during powerup. You can determine your workstation's 3COM address by running the following program under SMI Unix:

```
#include <sys/file.h>

main()
  {
    int fd, i;
    unsigned char addr[6];

    fd = open("/dev/mbmem", O+RDONLY, 0);
    lseek(fd, 0xE0400, 0);
    for (i=0; i<6; i++)
        read(fd, &addr[i], 1);
    printf("%02x%02x.%02x%02x.%02x%02x\n",
            addr[0], addr[1], addr[2],
            addr[3], addr[4], addr[5]);
  }
```

C.2.2.6. Initiation of V Servers on Unix

The directory unix/etc contains the Unix server programs needed to boot diskless workstations and serve remote sessions. This directory is symbolically linked to **/etc/V**.

The Vserver uses the file **V/run/Vhosttab** to map from hostnames to V logical host ids. Since this file is installation dependent you'll have to generate it by editing the SERVERHOSTS variable in (source) V/servers/unix/buildfile, run buildmake, then make install.

The line "sh /etc/V/rc" should be added to **/etc/rc.local** to fire up these server programs whenever the system is booted. (The "rc" file also provides a reasonable description of how to hand start these servers.) Note: the Vserver expects to be run as root, so that it can fork "sessions" that setuid to the appropriate user.

There must be at least one *public* Vserver running in any given local network. A Vserver is made public by starting it with the **-p** flag. See the *Unix Server* section of the manual for further information.

C.2.3. V Authentication Files

V6.0 requires two files for authentication: the Vpassword file, which is used by the Vauthentication server to authenticate (log in) users, and the Vusercorrespondence file, which maps V user numbers to Unix user names. For a complete description see chapter 35, *Authentication and the Authentication Server*. The section 35.5 explains the use and needs of the files involved in V authentication.

Two *awk* scripts are provided to generate the Vusercorrespondence and Vpassword files from the Unix /etc/passwd file. These scripts make each V user's V password the same as their password under Unix, and makes their V home directory on the correct Unix host. These scripts are aimed at sites which have only one

Unix host providing V file service. Section C.2.3.4 explains what to do if you have more than one such Unix host.

C.2.3.1. Creating the Vpassword File

Follow the directions in **/etc/V/Vpassword.awk**, then create a rough password file by executing

```
awk -f /etc/V/Vpassword.awk < /etc/passwd > /tmp/rough
```

Edit /tmp/rough to remove password entries for such Unix users as root, uucp, etc. Copy /tmp/rough to /usr/V/misc/Vpassword.

C.2.3.2. Creating the Vusercorrespondence File

Follow the directions in **/etc/V/Vusercorr.awk**, then create a rough correspondence file by executing

```
awk -f /etc/V/Vusercorr.awk < /etc/passwd > /tmp/rough
```

Edit /tmp/rough to remove the same extraneous entries deleted from the password file. Copy /tmp/rough to /etc/V/Vusercorrespondence.

C.2.3.3. The Unknown User and the V administrator

A workstation which has nobody logged in is authenticated to a special unknown user. We assume that you may want people to be able to run V programs (such as telnet and netwatch) without logging in. To do so, create an account called "Vunknown" on your unix system, make its home /tmp and give it minimal privileges. In addition you should create ,"Vadmin" an account for the administration of V system files and *chown* /usr/V/misc/Vpassword to that user.

C.2.3.4. V Authentication and Multiple Vservers

If you have more than one machine serving V, you must create a single password file that contains entries for the users from all of the machines. To do this first create a Vpassword file on each offsetting the V user numbers so that there is no overlap between machines. Then form a rough master password file by merging all of them together on the master password site. Massage the rough master by editing out duplicate entries, keeping the entries that correspond to each users "home" machine. Sorting the rough password file before duplicate deletion makes this chore much easier.

It is a bit more difficult to automatically generate individual correspondence files when building a merged Vpassword file. This is because there may be little relation between a given user's V user number and the user's Unix uids on multiple Unix hosts. We suggest starting with a minimal correspondence table on each machine and having users run **addcorr** the first time they log in to the V-System. A minimal file sufficient to boot and test the system is as follows:

```
0       Vadmin
1       Vadmin
2       Vunknown
```

C.2.4. The Boot Sequence

This section explains what happens during the bootstrap process. See chapter 16 for a detailed description of workstation boot commands. The boot process consists of three steps: loading **Vload**, the V-system bootstrap, loading the kernel and first team, and initializing the system (which may include downloading more programs such as the **vgts**).

Vload is downloaded over the network using the diskless booting protocol contained in the workstation's proms. There are many brands of workstations, and even more boot protocols. Once running, **Vload** locates and connects to a "public" Vserver to load the appropriate kernel and first team. After **Vload** has been loaded, all subsequent network file I/O is performed through a V server using V interkernel protocols.

C.2.4.1. SUN-1

Owners of Sun-1s are pretty much on their own. At Stanford we use our own proms which boot using PUP EFTP. You may have another protocol, or none at all. We suggest upgrading to a Sun-2.

C.2.4.2. SUN-2

Booting SUN-2 workstations with the ND protocol requires a running **ndserver**. The actual boot sequence is as follows. When an SMI workstation boots, if it contains no disk interface, it attempts to boot over the Ethernet. This is done using the "ND" boot protocol which asks for the first 18 512-byte blocks of a virtual disk. (Several of these blocks are thrown away. The boot program must be less than 0x1E00 bytes of text and initialized data). The ND server "intercepts" these requests and replies with **Vload**.

In general, the ndserver is a modest "hack" to allow one to run V on SMI SUN workstations without modifying the PROM monitor as it comes from the manufacturer. If a Sun is to run SMI Unix the field "boot:no" should be placed in the workstation's .config file to prevent the **ndserver** from downloading **Vload**.

C.2.4.3. SUN-3

Sun-3 workstations require running **rarpd** and **tftpd** servers. A Sun-3 boots using IP RARP (reverse address resolution protocol) to determine its IP address, then uses TFTP (trivial file transfer protocol) to download a bootstrap program.

The rarpd uses a simple database file, rarpdb.<pseudo-net>, to map physical ethernet addresses to IP addresses. Edit the example file to include the Sun-3s that will be running V6.0. Make sure that the <pseudo-net> extension matches the enet device corresponding to the 10 meg interface that the workstations are on. Typically this is "enet" or "eneta".

Once the Sun-3 knows its IP address it uses TFTP to download a file named by its hexadecimal IP address. For instance, if the rarpd responds with "36.8.1.3", the workstation will attempt to load a file named "24080103". This file should be linked to **Vload3+1e.d**, the Sun-3 version of **Vload**. These files (including a copy of **Vload3+1e.d**) should be placed in your tftp daemon's home directory. The **tftpd** distributed in 4.2BSD was buggy. We've included a patched version with V6.0, along with a rarp daemon. Remember to start the rarp and tftp daemons from /etc/rc.local.

Note: The Sun-3 boot process doesn't involve a workstation's configuration file until after **Vload** is running, so neither the 'bootfile' nor the 'boot' config file fields affects booting. To run Sun Unix prevent the tftpd from downloading **Vload** by removing the file linked to **Vload3+1e.d**.

C.2.4.4. MicroVAX

Booting MicroVAX workstations requires the **mvaxbootserver** to be running on a Unix host. The **mvaxbootserver** services MicroVAX network boot requests just as the **ndserver** responds to SMI ND requests. A "boot:no" entry in the workstation's config file will prevent the **mvaxbootserver** from responding to boot requests.

C.2.5. Debugging Suggestions

If the system fails to boot after following the above sequence, it is suggested that you try booting a "standalone" program like **netwatch** to check the initial portion of boot sequence and also (assuming you have multiple workstations), monitor the network activity when you try a full system boot. See the "Standalone" chapter of the reference manual (chapter 16) for details on how to load and use standalone programs. Section 16.2 gives an overview of how to use the **netwatch** family of programs. In general, the **netwatch** family of programs are very useful for debugging network problems. They keep a record of network packets which can be written to a log file. Please include such a log file in bug reports that relate to the network.

If the workstation fails to load a standalone program, check that your ethernet connection is working correctly. This is easiest if you have other means of monitoring Ethernet activity. You should also check that the **ndserver** or **mvaxbootserver** is properly instructed to respond to requests from the workstation's host address by having the correct config files present. The **ndserver** or **mvaxbootserver** can be executed with the "d" flag to put it into debugging mode. This should give a clear indication as to whether or not it is receiving the workstation boot request packets and what it is doing in response to the packets it receives.

Assuming you can load standalone programs, you should be able to load **Vload**. The error codes generated by **Vload** (e.g. "C017") can be decoded by looking the the header file **V/include/mi/Venviron.h**. Besides using netwatch to monitor network activity, one can run the **Vserver** in debug mode. Option "A" gives a verbose account of the Vservers's activities. See chapter 43 for the details of other debugging options.

C.3. Source Distribution Tape

The source distribution tape contains the V-System sources. Unless one is modifying the standard software or recompiling for some reason, there should be no reason to keep these sources on-line.

The procedure for extracting the files of the source tape is identical to that used with the binary distribution tape. **Warning:** Do not extract the source tape into the same directory as the binary tape. Some subdirectories have identical names. At Stanford, we keep V sources under **/V**.

C.3.1. Structure of the V Sources

The V source directories are structured by function and by machine dependency. For convenience each division is in a separate tar file. The major functional divisions are:

cmds	Standard command programs. A subdirectory for each command program (with some exceptions that we plan to eliminate).
kernel	V Kernel sources. The machine-independent source is under mi, 680X0 source under m68k, Microvax source under vax, and configuration-specific files under the other subdirectories. For example, sun2+ec configures the kernel for the SUN-2 with a 3COM Ethernet interface.
libc	Standard C run-time library for V. Subdirectories for different functional parts of the library. Machine-specific directories occur at various levels if there are machine-specific files.
servers	Server programs. A subdirectory for each separate server. The two Unix server programs **ndserver** and **Vserver** are here, though they don't run under V.
standalone	Standalone programs. A subdirectory for each separate program.
fonts:	Sources for some of the fonts used in the V-System.
doc:	The V6.0 manual in Scribe format.

C.3.2. Recompiling V Sources

Many of the V-System makefiles invoke **cc68** to compile and link. Be sure you have the latest version (included on the tape) of **cc68**, with the **-V** option.

Edit the shell script under **V/netinstall** to perform the appropriate installation procedure for your system. Some possibilities are for it to copy the binaries to other V hosts on your network (thus automating the installation and causing changes to take network-wide effect immediately), copy binaries to hosts in the local V-domain only, or copy to the local host only (a good choice if you have only one host running a Vserver

or update local hosts with regular rdists). It's important to keep all of the V binary directories synchronized within a domain since binaries are often served by the first **Vserver** to answer a request. 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 have been installed into some directory in the search path by **Vlink** before making the rest of the system. Each directory contains a file called **buildfile** which is processed by the **build** program, an enhanced version of **make**. The sources to **build** are included.

The following describes the steps (and order) to completely remake the V-System binaries (and libraries).

Change directory to **V/libc** and do a **build install-includes**. This should copy the V-System specific include files into **/usr/V/include**. Then do a **build** and then **build install** under this directory. This should result in **libV.a** being copied into **/usr/V/lib**.

Next, change to the **V/standalone** directory. This directory is for bootstrapping and loading utilities.

Next, change to **/V/kernel**. There is a subdirectory corresponding to each of the hardware combinations currently supported by the V-System. cd to the directory corresponding to your hardware configuration, then do a **build** followed by a **build install** to compile the kernel and put the binary into **V/boot/Vkernel**. For instance, **V/kernel/sun2+ec** and **V/kernel/sun1+en** correspond to the configurations "sun2 cpu with 3COM ethernet" and "sun1 (old sun, cadline etc.) with sun 3meg ethernet". You may have to create a new subdirectory and edit the buildfile to configure the kernel for your I/O devices.

Next change directory to **/V/servers**, and do a **build** followed by a **build install**.

Next, change directory to **/V/cmds** and again do a **build** followed by a **build install** to compile all the commands. This takes a while, and uses the include files, libraries, and servers.

Do the same for **/V/config** (after making some config files for your workstations) and **/V/standalone**.

C.3.3. Source Distribution Summary

The source tape provides all the files required to regenerate the binary distribution tape files (we believe). Any omissions were unintentional or forced upon us by lawyers.

— Appendix D —
List of Library Functions defined in libc

ASSERT	mem/mi/malloc.c
AcquireSpinLock	locking/mi/spinlock.c
AddUser	auth/mi/adduser.c
AllocFont	graphics/mi/allocraster.c
AllocRaster	graphics/mi/allocraster.c
ArbLoadProgram	exec/mi/arbloadprog.c
Attention	drivers/m68k/enet50.c
Authenticate	auth/mi/authenticate.c
AwaitKernelPacket	drivers/m68k/enet3.c
AwaitingReply	ipc/mi/awaitingrepl.c
BlksInFile	io/mi/blksinfile.c
BlockPosition	io/mi/blkposition.c
BlockSize	io/mi/blocksize.c
BoundingBox	graphics/mi/rasterbbox.c
BufferEmpty	io/mi/bufferempty.c
BufferModified	io/mi/seek.c
BufferValid	io/mi/fillbuffer.c
BytePosition	io/mi/byteposition.c
ByteSwapLongCopy	mem/mi/swablong1.c
ByteSwapLongInPlace	mem/mi/swablong2.c
ByteSwapShortCopy	mem/mi/swabshort.c
ByteSwapShortInPlace	mem/mi/swabshort.c
ChangeDirectory	io/mi/chdir.c
ChangeTeamPriority	exec/mi/changeteampr.c
CheckExecs	execserver/mi/checkexecs.c
ClearEof	io/mi/clearcof.c
Close	io/mi/close.c
ColToRowRaster	graphics/m68k/columnorder.c
CompB	graphics/vax/rasterop.c
CompImmedB	graphics/vax/rasterop.c
CompImmedL	graphics/vax/rasterop.c
CompImmedW	graphics/vax/rasterop.c
CompImmedX	graphics/vax/rasterop.c
CompIncr	graphics/vax/rasterop.c
CompL	graphics/vax/rasterop.c
CompLit	graphics/vax/rasterop.c
CompLitL	graphics/vax/rasterop.c
CompLoad	graphics/vax/rasterop.c
CompOp	graphics/vax/rasterop.c
CompReg	graphics/vax/rasterop.c
CompW	graphics/vax/rasterop.c
CopyDownwards	mem/mi/bcopy.c
CopyField	auth/mi/atoar.c
CopyMsg	drivers/m68k/enet3com.c
CreateDuplexInstance	io/mi/crtdupinst.c

CreateExec	execserver/mi/createexec.c
CreateGroup	ipc/mi/creategroup.c
CreateInstance	io/mi/createinst.c
CreatePipeInstance	io/mi/createpipe.c
Debug_BDL	drivers/vax/deqna.c
Debug_Deqna	drivers/vax/deqna.c
DefaultRootMessage	exec/mi/execprogram.c
DefaultSelectionRec	exec/mi/defaultselrec.c
DeleteExec	execserver/mi/deleteexec.c
DeleteUser	auth/mi/deleteuser.c
DestroyAuthRec	auth/mi/destroyar.c
DiscardDataPacket	drivers/m68k/enet3.c
EnetFlushReceiver	drivers/m68k/enet3.c
EnetInterrupt	drivers/m68k/enet50.c
EnetPowerup	drivers/m68k/enet3.c
EnetReset	drivers/vax/deqna.c
Eof	io/mi/eof.c
EqString	graphics/mi/getfont.c
ErrorString	exceptions/mi/error.c
ExecProgram	exec/mi/execprogram.c
Execl	exec/mi/system.c
Execv	exec/mi/system.c
FileException	io/mi/fileexcept.c
FileId	io/mi/fileid.c
FileServer	io/mi/fileserver.c
FileType	io/mi/filetype.c
FillBuffer	io/mi/fillbuffer.c
FindMatch	exec/mi/lookup.c
FindMatchingProgs	exec/mi/lookup.c
Flush	io/mi/flush.c
FlushBuffer	io/mi/flushbuffer.c
ForceException	exceptions/mi/forceexcept.c
ForceSend	exceptions/mi/forcesend.c
Forwarder	ipc/mi/forwarder.c
FreezeHost	exceptions/mi/freezehost.c
FullUserName	auth/mi/fullusername.c
GetFakePid	drivers/m68k/enet3.c
GetFont	graphics/mi/getfont.c
GetFontEntry	graphics/mi/getfont.c
GetFormatString	exceptions/m68k/stdexcept.c
GetKernelPid	exec/mi/kernelpid.c
GetMoreMallocSpace	mem/mi/mallocaux.c
GetNumberOfParams	exceptions/m68k/printstack.c
GetNumberofParams	exceptions/m68k/stdexcept.c
GetParams	exceptions/m68k/stdexcept.c
GetProcessorType	exceptions/vax/stdexcept.c
GetSigned1	io/mi/getbigendian.c
GetSigned2	io/mi/getbigendian.c
GetSigned3	io/mi/getbigendian.c
GetSigned4	io/mi/getbigendian.c
GetStringParams	exceptions/m68k/stdexcept.c
GetTeamPriority	exec/mi/getteampr.c
GetUnsigned1	io/mi/getbigendian.c
GetUnsigned2	io/mi/getbigendian.c

GetUnsigned3	io/mi/getbigendian.c
GetUnsigned4	io/mi/getbigendian.c
GiveToMalloc	mem/mi/malloc.c
INCR	graphics/vax/rasterop.c
InstructionFetch	exceptions/m68k/stdexcept.c
Interactive	io/mi/interactive.c
InterruptBoard	drivers/m68k/enetxln.c
JoinGroup	ipc/mi/joingroup.c
KillProgram	execserver/mi/killprogram.c
LeaveGroup	ipc/mi/leavegroup.c
List128	graphics/mi/bitreverse.c
List16	graphics/mi/bitreverse.c
List2	graphics/mi/bitreverse.c
List256	graphics/mi/bitreverse.c
List32	graphics/mi/bitreverse.c
List4	graphics/mi/bitreverse.c
List64	graphics/mi/bitreverse.c
List8	graphics/mi/bitreverse.c
LoadFileRegion	graphics/mi/loadfile.c
LoadGenericFont	graphics/mi/loadgfont.c
LoadProgram	exec/mi/execprogram.c
LoadTeamFromFile	exec/mi/loadteamfile.c
LoadXmitDescriptor	drivers/vax/dcqna.c
LogOutExecs	exec/mi/logoutexecs.c
LookupFont	graphics/mi/lookupfont.c
MASKOP	graphics/mi/rasterclear.c
MapTeamName	exec/mi/mapteamname.c
MapUID	auth/mi/mapuid.c
MapUserName	auth/mi/mapusername.c
ModifyUser	auth/mi/modifyuser.c
MoveFrom	ipc/mi/movefrom.c
MoveTo	ipc/mi/moveto.c
NewRaster	graphics/mi/newraster.c
Offset	drivers/m68k/enet50.c
Open	io/mi/open.c
OpenDuplex	io/mi/openduplex.c
OpenFile	io/mi/openfile.c
OpenIp	io/mi/openip.c
OpenProgFile	exec/mi/lookup.c
OpenStr	io/mi/openstr.c
OpenTcp	io/mi/opentcp.c
ParseLine	exec/mi/parseline.c
Password	auth/mi/password.c
PatternOp	graphics/vax/rasterop.c
PrintError	exceptions/mi/printerror.c
PrintFile	io/mi/printfile.c
PrintStackDump	exceptions/m68k/printstack.c
PutSigned1	io/mi/putbigendian.c
PutSigned2	io/mi/putbigendian.c
PutSigned3	io/mi/putbigendian.c
PutSigned4	io/mi/putbigendian.c
PutUnsigned1	io/mi/putbigendian.c
PutUnsigned2	io/mi/putbigendian.c
PutUnsigned3	io/mi/putbigendian.c

PutUnsigned4	io/mi/putbigendian.c
QueryExec	execserver/mi/queryexec.c
QueryGroup	ipc/mi/querygroup.c
QueryHosts	exec/mi/queryhosts.c
QueryHostsViaCS	exec/mi/queryhostscs.c
QueryHostsViaMulticast	exec/mi/queryhostsm.c
QvssDisable	graphics/vax/useqvss.c
QvssEnable	graphics/vax/useqvss.c
QvssInit	graphics/vax/qvssinit.c
QvssRectangle	graphics/vax/qvssrectangle.c
REG	graphics/vax/rasterop.c
RasterClear	graphics/mi/rasterclear.c
RasterCompileDummyL	graphics/m68k/rastercompile.c
RasterCompileDummyW	graphics/m68k/rastercompile.c
RasterInvert	graphics/mi/rasterinvert.c
RasterOp	graphics/vax/rasterop.c
RasterOpS	graphics/vax/rasterop.c
RasterPrint	graphics/mi/rasterprint.c
RasterSet	graphics/mi/rasterset.c
Read	io/mi/read.c
ReadAccess	exceptions/m68k/stdexcept.c
ReadDataPacket	drivers/m68k/enet3.c
ReadGenericCharacter	graphics/mi/readgfont.c
ReadGenericFont	graphics/mi/readgfont.c
Receive	ipc/mi/kernellib.c
ReceiveSpecific	ipc/mi/kernellib.c
ReceiveWithSegment	ipc/mi/kernellib.c
ReleaseFileRegion	graphics/mi/loadfile.c
ReleaseInstance	io/mi/releaseinst.c
ReleaseSpinLock	locking/mi/spinlock.c
RemoteExecute	exec/mi/remoteexec.c
RemoveFile	io/mi/removefile.c
Reply	ipc/mi/kernellib.c
ReplyWithSegment	ipc/mi/kernellib.c
ResetReceive	drivers/m68k/enet75.c
RestrictRaster	graphics/mi/restrictrast.c
Resynch	io/mi/resynch.c
RowToColRaster	graphics/m68k/columnorder.c
SearchBit	graphics/mi/rasterbbox.c
SearchPathMatch	exec/mi/lookup.c
Seek	io/mi/seek.c
SeekBlock	io/mi/seekblock.c
SetBitPtr	graphics/mi/setbitptr.c
SetBreakProcess	io/mi/setbreak.c
SetInstanceOwner	io/mi/setowner.c
SetMode	drivers/m68k/enetxln.c
SetUpEnvironment	exec/mi/setupenv.c
ShortString	exceptions/mi/error.c
SimpleText	graphics/vax/drawtext.c
SkipToBoc	graphics/mi/readgfont.c
SpecLoadProgram	exec/mi/specloadprog.c
SpecialClose	io/mi/close.c
StandardExceptionHandler	exceptions/m68k/stdexcept.c
TeamOwner	exec/mi/teamowner.c

TextBBox	graphics/mi/textbbox.c
TryEnetTransmit	drivers/m68k/enet3com.c
UnaryOp	graphics/vax/rasterop.c
UnaryOpS	graphics/vax/rasterop.c
UnfreezeHost	exceptions/mi/freezehost.c
UseQvss	graphics/vax/useqvss.c
UserName	auth/mi/username.c
ValidMagicNum	exec/mi/validmagicnum.c
WORDOP	graphics/mi/rasterclear.c
Wait	exec/mi/wait.c
Write	io/mi/write.c
WriteGfFont	graphics/mi/writegfont.c
WriteKernelPacket	drivers/m68k/enet3.c
WriteText	graphics/vax/gentext.c
Zero	mem/mi/zero.c
_Open	io/mi/_open.c
abort	exceptions/m68k/abort.c
acos	math/mi/asin.c
align	exec/mi/loadteamfile.c
allock	mem/mi/malloc.c
artoa	auth/mi/artoa.c
asin	math/mi/asin.c
asympt	math/mi/j0.c
atan	math/mi/atan.c
atan2	math/mi/atan.c
atoar	auth/mi/atoar.c
bcopy	mem/mi/bcopy.c
botch	mem/mi/malloc.c
cabs	math/mi/hypot.c
calloc	mem/mi/calloc.c
ceil	math/mi/floor.c
cfree	mem/mi/calloc.c
chdir	io/mi/chdir.c
clear	drivers/m68k/enetxln.c
clearerr	io/mi/clrerr.c
coffee_break	drivers/vax/dcqna.c
compileW	graphics/m68k/rastercompile.c
cos	math/mi/sin.c
cosh	math/mi/sinh.c
debug	drivers/m68k/enet3.c
enetaddress	drivers/m68k/enetxln.c
enopen	drivers/m68k/enetxln.c
exp	math/mi/exp.c
fabs	math/mi/fabs.c
floor	math/mi/floor.c
free	mem/mi/malloc.c
get_Ap_Fp	exceptions/vax/printstack.c
gf_new_row	graphics/mi/writegfont.c
gf_paint	graphics/mi/writegfont.c
hvTest	graphics/mi/textbbox.c
hypot	math/mi/hypot.c
if	graphics/vax/gentext.c
j0	math/mi/j0.c
j1	math/mi/j1.c

jn	math/mi/jn.c
log	math/mi/log.c
log10	math/mi/log.c
malloc	mem/mi/malloc.c
mc68kenread	drivers/m68k/enetxln.c
mc68kenwrite	drivers/m68k/enetxln.c
mktemp	io/mi/mktemp.c
one_sip	drivers/vax/deqna.c
pack2	exec/mi/setupenv.c
pack4	exec/mi/setupenv.c
packpair	exec/mi/setupenv.c
packstr	exec/mi/setupenv.c
pow	math/mi/pow.c
print_pc	exceptions/vax/printstack.c
realloc	mem/mi/malloc.c
returns	graphics/vax/gentext.c
s_getc	io/mi/getbigendian.c
satan	math/mi/atan.c
sin	math/mi/sin.c
sinh	math/mi/sinh.c
sinus	math/mi/sin.c
sizeof	drivers/m68k/enet3.c
sqrt	math/mi/sqrt.c
swab	mem/mi/swabshort.c
swap1	drivers/m68k/enet50.c
switch	graphics/vax/gentext.c
system	exec/mi/system.c
tan	math/mi/tan.c
tanh	math/mi/tanh.c
u_getc	io/mi/getbigendian.c
ungetc	io/mi/ungetc.c
unlink	io/mi/unlink.c
unlockAndGetSpace	mem/mi/malloc.c
unlockedFree	mem/mi/malloc.c
unlockedGiveToMalloc	mem/mi/malloc.c
unlockedMalloc	mem/mi/malloc.c
unlockedRealloc	mem/mi/malloc.c
xatan	math/mi/atan.c
y0	math/mi/j0.c
y1	math/mi/j1.c
yn	math/mi/jn.c
AddCList	packages/mi/clist.c
AddDList	packages/mi/dlist.c
AddQueue	packages/mi/queue.c
AddSList	packages/mi/slist.c
Any	strings/mi/any.c
AwaitSendReply	sa/mi/ikc.c
BackSpace	sa/mi/flushfill.c
BufferValid	sa/mi/flushfill.c
CTRL	termlib/mi/tgoto.c
ClearLocalNames	naming/mi/clearlocal.c
ClearModifiedPages	process/mi/clearpages.c
Concat	strings/mi/concat.c
Convert_num	strings/mi/convertnum.c

Cooked	sa/mi/flushfill.c
CopyMsg	sa/mi/ikc.c
Copy_str	strings/mi/copystr.c
Create	process/mi/create.c
CreateHost	process/mi/createhost.c
CreateProcess	process/mi/createproc.c
CreateSelectionInstance	service/mi/select.c
CreateTeam	process/mi/createteam.c
Creator	process/mi/creator.c
DefineLocalName	naming/mi/deflocal.c
DefineTempArea	naming/mi/deftemparea.c
Delay	time/mi/delay.c
Destroy	process/mi/destroy.c
DestroyHost	process/mi/destroyhost.c
DestroyProcess	process/mi/destroyproc.c
DisplayFields	vgts/mi/fields.c
Echo	sa/mi/flushfill.c
EditField	vgts/mi/fields.c
EditLine	vgts/mi/editline.c
EditStdFld	vgts/mi/fields.c
EmptyCList	packages/mi/clist.c
EmptyDList	packages/mi/dlist.c
EmptyQueue	packages/mi/queue.c
EmptySList	packages/mi/slist.c
EmptyStack	packages/mi/stack.c
Equal	strings/mi/equal.c
ExtractHost	process/mi/extracthost.c
FillBuffer	sa/mi/flushfill.c
FirstCList	packages/mi/clist.c
FirstDList	packages/mi/dlist.c
FirstSList	packages/mi/slist.c
FlushBuffer	sa/mi/flushfill.c
FormatFormat	vgts/mi/fields.c
Forward	saconsole/mi/dummyikc.c
FreezeHost	process/mi/freezehost.c
GetAbsoluteName	naming/mi/getabsname.c
GetBufferedLine	sa/mi/flushfill.c
GetContextId	naming/mi/namesend.c
GetContextName	naming/mi/getctxname.c
GetEvent	vgts/mi/usemouse.c
GetField	vgts/mi/fields.c
GetFileName	naming/mi/getfilename.c
GetGraphicsEvent	vgts/mi/usemouse.c
GetGraphicsStatus	vgts/mi/usemouse.c
GetHostPid	naming/mi/gethostpid.c
GetMoreMallocSpace	sa/m68k/getmoremalloc.c
GetMouseEvent	vgts/mi/usemouse.c
GetMouseStatus	vgts/mi/usemouse.c
GetObjectOwner	process/mi/objectowner.c
GetPid	naming/mi/getpid.c
GetRemoteTime	time/mi/remotetime.c
GetReply	saconsole/mi/dummyikc.c
GetTTY	vgts/mi/vtty.c
GetTeamRoot	process/mi/getteamroot.c

GetTeamSize	process/mi/getteamsize.c
GetTime	time/mi/gettime.c
GotAnInt	sa/vax/initints.c
Hex_value	strings/mi/hexvalue.c
IgnoreRetry	naming/mi/ignoreretry.c
InitCList	packages/mi/clist.c
InitDList	packages/mi/dlist.c
InitQueue	packages/mi/queue.c
InitSList	packages/mi/slist.c
InitStack	packages/mi/stack.c
IsCooked	sa/mi/flushfill.c
IsEcho	sa/mi/flushfill.c
IterCList	packages/mi/clist.c
IterDList	packages/mi/dlist.c
IterSList	packages/mi/slist.c
K_call	rawio/m68k/sun1rawio.c
K_getchar	rawio/m68k/sun1rawio.c
K_getconfig	rawio/m68k/sun1rawio.c
K_getcontext	rawio/m68k/sun1rawio.c
K_getmemsize	rawio/m68k/sun1rawio.c
K_getsegmap	rawio/m68k/sun1rawio.c
K_mayget	rawio/m68k/sun1rawio.c
K_proctype	rawio/m68k/sun1rawio.c
K_putchar	rawio/m68k/sun1rawio.c
K_puts	rawio/m68k/sun1rawio.c
K_setcontext	rawio/m68k/sun1rawio.c
K_setsegmap	rawio/m68k/sun1rawio.c
K_ticks	rawio/m68k/sun1rawio.c
K_version	rawio/m68k/sun1rawio.c
LastCList	packages/mi/clist.c
LastDList	packages/mi/dlist.c
LastSList	packages/mi/slist.c
Lower	strings/mi/lower.c
MakeHexDigit	sa/mi/makehexdigit.c
Meditate	unix-compat/mi/signal.c
ModifyPad	vgts/mi/openpad.c
NReadDescriptor	naming/mi/nreaddesc.c
NWriteDescriptor	naming/mi/nwritedesc.c
NameCacheAdd	naming/mi/namecache.c
NameCacheDelete	naming/mi/namecache.c
NameCacheLookup	naming/mi/namecache.c
NameSend	naming/mi/namesend.c
NoEcho	sa/mi/flushfill.c
Null_str	strings/mi/nullstr.c
OpenAndPositionPad	vgts/mi/openpad.c
OpenConfigFile	query/mi/qwconfig.c
OpenPad	vgts/mi/openpad.c
ParseFormat	vgts/mi/fields.c
PopStack	packages/mi/stack.c
PrimeCache	naming/mi/primecache.c
PushStack	packages/mi/stack.c
PutField	vgts/mi/fields.c
PutUntilConversion	vgts/mi/fields.c
QueryKernel	process/mi/querykern.c

QueryPad	vgts/mi/openpad.c
QueryPadSize	vgts/mi/openpad.c
QueryProcessPriority	process/mi/priority.c
QueryProcessState	process/mi/queryprocess.c
QueryProcessorUsage	process/mi/queryusage.c
QueryWorkstationConfig	query/mi/qwconfig.c
Raw	sa/mi/flushfill.c
RawGetchar	sa/mi/flushfill.c
ReadDescriptor	naming/mi/readdesc.c
ReadProcessState	process/mi/readprocess.c
ReadStdFld	vgts/mi/fields.c
Ready	process/m68k/ready.c
RealFillBuffer	sa/mi/flushfill.c
RealFlushBuffer	sa/mi/flushfill.c
RedrawPad	vgts/mi/usemouse.c
RegisterObject	service/mi/register.c
RegisterServer	service/mi/register.c
RemoveCList	packages/mi/clist.c
RemoveDList	packages/mi/dlist.c
RemoveQueue	packages/mi/queue.c
RemoveSList	packages/mi/slist.c
Rename	naming/mi/rename.c
ReplyWithSeg	saconsole/mi/dummyikc.c
ReportNamingStats	process/mi/exit.c
ResetTTY	vgts/mi/vtty.c
ResolveLocalName	naming/mi/resolvelocal.c
ReturnHostStatus	service/mi/hoststatus.c
ReturnModifiedPages	process/mi/returnpages.c
SameTeam	process/mi/sameteam.c
SelectPad	vgts/mi/openpad.c
Send	sa/mi/ikc.c
SetObjectOwner	process/mi/objectowner.c
SetPid	naming/mi/setpid.c
SetProcessPriority	process/mi/priority.c
SetTeamPriority	process/mi/setteamprio.c
SetTeamSize	process/mi/setteamsize.c
SetTime	time/mi/settime.c
SetUserNumber	user/mi/setusernumber.c
SetVgtBanner	vgts/mi/setbanner.c
Shift_left	strings/mi/shiftleft.c
Size	strings/mi/size.c
SpecialSprintf	vgts/mi/fields.c
StrToFormat	vgts/mi/fields.c
Suicide	process/mi/destroy.c
TeamRoot	process/mi/teamroot.c
TransferHost	process/mi/transferhost.c
UndefineLocalName	naming/mi/undeflocal.c
UnfreezeHost	process/mi/unfreezehost.c
UnregisterObject	service/mi/register.c
UnregisterServer	service/mi/register.c
UpdateHostStatus	service/mi/hoststatus.c
Upper	strings/mi/upper.c
User	user/mi/user.c
ValidPid	process/mi/validpid.c

Vsasu	sa/vax/Vsasu.c
Wakeup	time/mi/wakeup.c
WatchForBreak	unix-compat/mi/signal.c
WriteDescriptor	naming/mi/writedesc.c
WriteProcessState	process/mi/writeprocess.c
_Receive	saconsole/mi/dummyikc.c
_doprnt	stdio/mi/doprnt.c
_doscan	stdio/mi/doscan.c
_getccl	stdio/mi/doscan.c
_innum	stdio/mi/doscan.c
_instr	stdio/mi/doscan.c
_start	sa/m68k/_start.c
_strout	stdio/mi/strout.c
abort	sa/mi/abort.c
abs	numeric/mi/abs.c
align	process/mi/teamroot.c
asctime	time/mi/ctime.c
atof	strings/mi/atof.c
atoi	strings/mi/atoi.c
atol	strings/mi/atol.c
chmod	unix-compat/mi/unix-io.c
clearenv	naming/mi/clearenv.c
close	unix-compat/mi/unix-io.c
closedir	naming/mi/closedir.c
control	sa/mi/flushfill.c
copyenv	naming/mi/copyenv.c
creat	unix-compat/mi/unix-io.c
crypt	strings/mi/crypt.c
ct_numb	time/mi/ctime.c
ctime	time/mi/ctime.c
cvt	strings/mi/ecvt.c
debug	sa/mi/ikc.c
decprint	saconsole/mi/printf.c
dysize	time/mi/ctime.c
ecvt	strings/mi/ecvt.c
encrypt	strings/mi/crypt.c
exit	process/mi/exit.c
fbmode	rawio/m68k/sun1rawio.c
fclose	stdio/mi/fclose.c
fcvt	strings/mi/ecvt.c
feof	stdio/mi/ferror.c
ferror	stdio/mi/ferror.c
fflush	stdio/mi/fflush.c
fgetc	stdio/mi/fgetc.c
fgets	stdio/mi/fgets.c
fopen	stdio/mi/fopen.c
fprintf	stdio/mi/fprintf.c
fputc	stdio/mi/fputc.c
fputs	stdio/mi/fputs.c
fread	stdio/mi/rdwr.c
freopen	stdio/mi/freopen.c
frexp	numeric/vax/frexp.c
fscanf	stdio/mi/scanf.c
fseek	stdio/mi/fseek.c

fstat	unix-compat/mi/unix-io.c
ftell	stdio/mi/ftell.c
ftime	time/mi/ctime.c
fwrite	stdio/mi/rdwr.c
gcvt	strings/mi/gcvt.c
getenv	naming/mi/getenv.c
gets	saconsole/mi/gets.c
getw	stdio/mi/getw.c
getwd	naming/mi/getwd.c
gmtime	time/mi/ctime.c
hexprint	saconsole/mi/printf.c
index	strings/mi/index.c
initints	sa/vax/initints.c
initstate	numeric/mi/random.c
linecontrol	rawio/m68k/sun1rawio.c
linedata	rawio/m68k/sun1rawio.c
lineget	rawio/m68k/sun1rawio.c
linereadyrx	rawio/m68k/sun1rawio.c
link	unix-compat/mi/unix-io.c
localtime	time/mi/ctime.c
lseek	unix-compat/mi/unix-io.c
nevercalled	sa/m68k/_start.c
octprint	saconsole/mi/printf.c
open	unix-compat/mi/unix-io.c
opendir	naming/mi/opendir.c
pBreathOfLife	sa/mi/ikc.c
pRemoteForward	sa/mi/ikc.c
pRemoteFoward	sa/mi/ikc.c
pRemoteMoveFromReq	sa/mi/ikc.c
pRemoteMoveToReq	sa/mi/ikc.c
pRemoteReceiveSpecific	sa/mi/ikc.c
pRemoteReply	sa/mi/ikc.c
pRemoteSend	sa/mi/ikc.c
printf	saconsole/mi/printf.c
puts	saconsole/mi/puts.c
putw	stdio/mi/putw.c
qs1	numeric/mi/qsort.c
qsexc	numeric/mi/qsort.c
qsort	numeric/mi/qsort.c
qstexc	numeric/mi/qsort.c
rand	numeric/mi/rand.c
random	numeric/mi/random.c
read	unix-compat/mi/unix-io.c
readdir	naming/mi/readdir.c
rename	naming/mi/rename.c
return_K_ticks	sa/vax/initints.c
rewind	stdio/mi/rewind.c
rindex	strings/mi/rindex.c
sbrk	unix-compat/mi/sbrk.c
scanf	stdio/mi/scanf.c
seekdir	naming/mi/seekdir.c
setbuf	stdio/mi/setbuf.c
setecho	rawio/m68k/sun1rawio.c
setenv	naming/mi/setenv.c

setkey	strings/mi/crypt.c
setstate	numeric/mi/random.c
signal	unix-compat/mi/signal.c
sleep	time/mi/sleep.c
sprintf	stdio/mi/sprintf.c
srand	numeric/mi/rand.c
srandom	numeric/mi/random.c
sscanf	stdio/mi/scanf.c
stat	unix-compat/mi/unix-io.c
stdinit	unix-compat/mi/unix-io.c
stime	time/mi/ctime.c
strcat	strings/mi/strcat.c
strcatn	strings/mi/strcatn.c
strcmp	strings/mi/strcmp.c
strcmpn	strings/mi/strcmpn.c
strcpy	strings/mi/strcpy.c
strcpyn	strings/mi/strcpyn.c
strlen	strings/mi/strlen.c
strncat	strings/mi/strncat.c
strncmp	strings/mi/strncmp.c
strncpy	strings/mi/strncpy.c
strsave	strings/mi/strsave.c
sunday	time/mi/ctime.c
tdecode	termlib/mi/termcap.c
telldir	naming/mi/telldir.c
tgetent	termlib/mi/termcap.c
tgetflag	termlib/mi/termcap.c
tgetnum	termlib/mi/termcap.c
tgetstr	termlib/mi/termcap.c
tgoto	termlib/mi/tgoto.c
time	time/mi/ctime.c
timezone	time/mi/timezone.c
tnamatch	termlib/mi/termcap.c
tnchktc	termlib/mi/termcap.c
tputs	termlib/mi/tputs.c
tskip	termlib/mi/termcap.c
umask	unix-compat/mi/unix-io.c
unix_errno	unix-compat/mi/unix-io.c
vdir_to_stat	unix-compat/mi/unix-io.c
wordchar	sa/mi/flushfill.c
write	unix-compat/mi/unix-io.c

Index

This work was supported by the Defense Advanced Research Projects Agency under contracts MDA903-80-C-0102 and N00039-83-K-0431, by Digital Equipment Corporation, by the National Science Foundation under grant DCR 83-52048, by the National Aeronautics and Space Administration under contract NAGW-419, by Bell-Northern Research, ATT Information Systems, Philips Research and NCR, and by Graduate Fellowships from IBM, the National Science Foundation, Shell, and TRW.

20 June 1986