A Stand-Alone Input/Output Library

S. R. Eisen

Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

1.1 Motivation

Most stand-alone programs that are supported under UNIX† conform to no input-output standard. They implement their own I/O routines and their own nomenclature for accessing data stored on I/O devices. This library was written with the objective of creating a set of functions that would be used to simulate standard C library functions [1] for a program that is loaded stand-alone into a Digital Equipment Corporation 11-family computer.

1.2 Environment

1.2.1 Compilation and Execution. Normally, a stand-alone program is written in C, using standard library functions found in Sections 2 and 3 of [1]. The program is compiled and the object file is link-edited with the stand-alone library *instead* of the standard UNIX C library. The resulting single object file is loaded by using either the command interpreter that is described in Section 6.2.1 (denoted below by {6.2.1}), or any other standard UNIX bootstrap program.

1.2.2 System Functions. All required services that are usually performed by the operating system, such as input/output, are taken care of by the functions loaded from the stand-alone library. Thus I/O drivers are included in stand-alone executables without any additional work on the part of the user.

Functions such as `fork`, `pipe`, and `exec`, that would simulate system calls that make no sense outside of an operating system environment are *excluded* from the stand-alone library, even to the extent of signifying an error condition. The complete list of excluded functions may be found in {4.3}. Of the routines that were *substituted* for UNIX system calls, all take the same arguments and return the same values as their UNIX counterparts, except as noted in {4.1-4.2}. These functions set the external variable `errno` when an error occurs, so that the C library routine `perror` may be used by stand-alone programs.

The user may call any global functions in the library, including those that would normally be found in an operating system proper, but would not be available to the user in an operating system environment. All such routines, however, have been "disguised" by prefacing their names with the underscore character.

1.2.3 User Interface. UNIX-like file names need not be used, although their use is encouraged. All user functions that require file names, such as `MKNOD` {2.1.2}, `mount` {2.1.3}, and `open` {2.1.4} first pass their file-name arguments through a filter that converts them to a standard form: each element of the path name is separated by a single slash, with a leading slash used only if the file name is non-null.

From the point of view of the user-level program, the environment that is created by the stand-alone library is close enough to a UNIX environment that *a large class of UNIX programs may be compiled for stand-alone execution with little or no revision.* Another class of programs that includes boot programs and other programs that need to be relocated can also be written using the stand-alone I/O library. Specific instructions for compiling and executing programs using the library may be found in {6}.

---

† UNIX is a trademark of Bell Laboratories.

2. I/O PHILOSOPHY

The stand-alone I/O library was designed to provide an environment that is as close to UNIX as possible, while maintaining the generality necessary for the composition of bootstrap programs, disk formatters, and the like. Disk I/O drivers have the capability of handling UNIX file systems, but retain the generality necessary to manipulate disks with other data on them. Because UNIX accesses I/O devices through the file system, and there is no guarantee that a file system (UNIX or otherwise) exists, access to I/O devices must be handled in a special way.

2.1 Block I/O Data Structures

2.1.1 The Configuration Table. All I/O routines operate without interrupt processing; also, the stand-alone implementation of file descriptors differs from the UNIX implementation. The open, close, and strategy (read and write) routines for devices therefore do not strictly resemble the corresponding UNIX routines. The method used to access these routines, however, is very similar; it employs a configuration table that has the form:

```
struct devsw {
        int     (*dv_strategy) ();
        int     (*dv_open) ();
        int     (*dv_close) ();
};
```

The position of a certain device within the `devsw` table is the *device number* for that type of device. The notion of a device number is analogous to the notion of a major number for a UNIX device.

2.1.2 The Device Table. Each family of devices is associated with UNIX-type names by use of a second structure:

```
struct dtab {
        char            *dt_name;
        struct devsw    *dt_devp;
        int             dt_unit;
        daddr_t         dt_boff;
};
```

The `dtab` structure associates a device name with a pointer to the `devsw` structure for that type of device, the unit number of the physical device, and the block offset within the unit at which the logical device should start. The name, in fact, can be any string; by convention, however, a UNIX-type file name, such as */dev/rk1* or */dev/mt4*, is used.

Entries in this table are created by using the `MKNOD` function. Note that although the function of the `MKNOD` routine is similar to the that of the UNIX `mknod` routine, the arguments passed to each routine are not at all alike. `MKNOD` is called using the following synopsis:

```
MKNOD (name, devno, unit, boff)
char *name;
int devno, unit;
daddr_t boff;
```

`MKNOD` associates `name` with the logical device beginning `boff` blocks into the given unit of the device whose device number is `devno`. The value −1 is returned if an illegal argument is passed, the `dtab` table is full, or the given name already exists in the table.

2.1.3 The Mount Table. For mounted file systems, there is yet another structure:

```
struct mtab {
        char            *mt_name;
        struct dtab     *mt_dp;
};
```

The `mtab` structure associates a name with a pointer to the `dtab` structure for a device on which a UNIX file system resides. References to the name will refer to the root file on that device. Entries in this table are created by using the `mount` function. The following synopsis applies:

```
mount (devname, mntname)
char *devname, *mntname;
```

`Mount` announces that a file system has been mounted on `devname`, and that its mounted name will henceforth be `mntname`. `Devname` must be a valid entry in the `dtab` table, and `mntname` must *not* exist in the `mtab` table. If either of these conditions is not met, or if there are no more empty slots in the table, `mount` returns the value −1.

A mount table entry may be deleted by the `umount` function, whose synopsis is the same as the corresponding UNIX routine.

2.1.4 The I/O Block. Each open file is associated with a numerical file descriptor. At the start of program execution, the file descriptors numbered 0, 1 and 2 are each open for reading and writing to the system console, and all other file descriptors are closed (not assigned). File descriptors greater than 2 are available to be assigned to either block devices or UNIX files that reside on mounted file systems by using the `open` function described below.

Each block file descriptor is associated with a structure of the following form:

```
struct iob {
        char            i_flgs;
        struct inode    i_ino;
        time_t          i_atime;
        time_t          i_mtime;
        time_t          i_ctime;
        struct dtab     *i_dp;
        off_t           i_offset;
        daddr_t         i_bn;
        char            *i_ma;
        int             i_cc;
        char            i_buf[512];
};
```

The I/O block contains a data buffer and a block number counter for the device whose `dtab` structure is pointed to by the I/O block. For open UNIX files, the offset within the file and a copy of the inode are included in the I/O block. For open block devices, the inode structure is only partially filled in.

A file descriptor is allocated and an entry is created in the I/O block by the `open` function. The synopsis of the stand-alone `open` function is identical with that of its UNIX counterpart.

`Open` searches the `dtab` table for the given string, and if it is not found, the `mtab` table is searched for the longest path name starting at the beginning of the given string. For example, if `open` is passed the argument `/ab/cd/ef/gh`, it will first look for the argument itself in both the `dtab` and `mtab` tables, then search for `/ab/cd/ef` in the `mtab` table, then `/ab/cd`, and so on.

If the string is found in the `dtab` table, then the named device will be opened for the appropriate operation. If the string or one of its substrings is found in the `mtab` table, the device pointed to by the `mtab` table entry is searched for the remainder of the path name. If found, the file is opened.

At present, files on mounted file systems may only be opened for reading. The reason for this has to do with memory size requirements for a writing capability, the amount of time it would take to implement this capability, and the danger of corrupting file systems unnecessarily. It is likely that the capability of writing files will be included at some time in the future.

The `creat (name, mode)` function is identical to `open (name, 1)`. The `mode` argument is ignored.

The `close` function deallocates the I/O block associated with the named file descriptor.

2.1.5 Summary. The following list contains the definitions of all of the data structures discussed in this section, as they appear in the stand-alone library source code:

```
struct devsw    _devsw[];
struct dtab     _dtab[NDEV];
struct mtab     _mtab[NMOUNT];
struct iob      _iobuf[NFILES];
```

These structures and the corresponding table sizes are all defined in the file `/usr/include/stand.h`.

2.2 Reading and Writing

The `read` and `write` functions are the most primitive I/O routines normally available to the user. The file descriptor argument may refer to either the system console or a block device.

3. I/O DEVICES AND DRIVERS

As was mentioned earlier, file descriptors 0, 1, and 2 all refer to the system console device. The console is the only character device supported. A spectrum of block devices may be defined in the device table by the `MKNOD` function.

3.1 The System Console Driver

The driver for the console terminal is a modified, scaled-down version of the UNIX *tty* driver. Input lines may be up to 255 characters long and there is no read-ahead (i.e., input will not be accepted until the program calls for it). The driver supports programmable options and *erase* and *kill* characters. End of file may be generated in "cooked" mode by typing CTRL-D.

The `stty` and `gtty` functions are implemented and refer to a structure identical with that which is used by UNIX. The only options that have any effect are `RAW`, `CRMOD`, `XTABS`, `ECHO`, and `LCASE`. Initially, the *erase* and *kill* characters are the standard UNIX `#` and `@`, respectively, and the options set are `CRMOD`, `XTABS`, and `ECHO`.

The `isatty` function returns true if the file descriptor argument is in the range 0 to 2.

If, while output is being printed on the console, the ASCII DEL character is typed, a subroutine call to the `_exit` function is immediately effected.

The actual input and output are performed by the functions in the following table:

System Console Driver Routines

Synopsis	Description

`_ttread (buf, n)`
`char *buf;`
`int n;`
	Reads `n` characters from the console into the area pointed to by `buf`.

`_ttwrite (buf, n)`
`char *buf;`
`int n;`
	Prints `n` characters on the console from the area pointed to by `buf`.

The external buffer `_ttstat` contains the current copy of the structure referred to by `stty` and `gtty`. Its synopsis is:

```
# include <stand.h>
struct sgttyb _ttstat;
```

3.2 Block Device Drivers

Block input and output are performed in the stand-alone library in the same manner as *physical* I/O is handled under UNIX; that is, only raw devices are supported.

A particular I/O driver routine is looked up in the `devsw` table and called by one of the following:

```
_devopen (io)       _devclose (io)      _devread (io)       _devwrite (io)
struct iob *io;     struct iob *io;     struct iob *io;     struct iob *io;
```

The external integer variable `_devcnt` contains the number of devices in the `devsw` table.

3.2.1 Disk Drivers. The stand-alone library supports the following disk devices and their equivalents:

RP04/05/06 and RM05(*gd*) RP11/RP03(*rp*) RK11/RK05(*rk*)

Disk device drivers can support file systems that do not start at the beginning of the physical unit. Such file systems are defined by using the MKNOD function {2.1.2}.

The physical I/O operation for disks causes reads and writes to always be started at the beginning of the physical block in which the offset designated in the I/O block {2.1.4} falls. Also, I/O operations that reference a disk address outside of the bounds of either a logical or physical disk will not cause an error to occur.

The synopsis of each of the disk driver functions has the form:

```
_devstrategy (io, func)
struct iob *io;
int func;
```

where *dev* may be `gd`, `rp`, or `rk`.

3.2.2 Tape Drivers. The stand-alone library supports the following magnetic tape devices and their equivalents:

TM11/TU10(*tm*) TU16(*ht*)

For both the *tm* and *ht* drivers, logical units 0 through 7 refer to four 800 bpi magnetic tape transports. For the *ht* driver only, logical units 8 through 15 refer to the corresponding 1600 bpi magnetic tape transports. In each block of eight logical units, the first four units are designated normal-rewind on close, and the other four are no-rewind on close.

`Lseek` is ineffective for tapes. Each `read` or `write` function call reads or writes the next record on the tape. The `dt_boff` entry in the device table is ignored for magnetic tape devices.

The synopses of the tape driver functions have the following forms:

```
_devopen (io)       _devclose (io)      _devstrategy (io, func)
struct iob *io;     struct iob *io;     struct iob *io;
                                        int func;
```

where *dev* may be either `ht` or `tm`.

4. NON-I/O ROUTINES

4.1 Revisions

Several of the system calls that are not required for I/O, but would, however, be useful in a stand-alone environment are included in the library. The operation of some of these functions may differ slightly from the UNIX implementations. These functions, together with the I/O functions described above, form a firm enough basis that the remainder of the C library may be used without modification.

4.1.1 Stat and Fstat. The `stat` and `fstat` functions require the use of an I/O block. In order to execute either one of these functions, the file on which it is operating must be open because the information needed is copied out of the I/O block. For `fstat`, the file is already open, but when the `stat` function is used, first the file is opened, `fstat` is called, and then the file is closed again before returning. Thus, if all I/O blocks are occupied (the maximum number of files are open), `stat` will return an error.

If the argument to `stat` or `fstat` refers to a file that resides on a mounted file system, then the inode is copied verbatim and the routines are completely compatible with the UNIX versions. If the argument refers to a device, the buffer is filled with a reasonable approximation of what may be expected.

4.1.2 Access. The `access` function also requires an open file. If the open succeeds, and either the file is a device or the mode of the file matches the specified mode argument, the value 0 is returned; otherwise, the value −1 is returned. In any case, the I/O block is freed by closing the file before returning.

4.1.3 Time. Because the real-time clock is not supported, the best that can be done is for the `time` function to return the value that was set by the last call of the `stime` function. If `stime` has not been called, `time` returns the value 0.

4.1.4 Break. The `brk` and `sbrk` functions may be used as they are in UNIX. Because memory management is not used, there is no way of detecting if the upward-expanding allocated memory has collided with the downward-expanding stack. The return is therefore always successful, even if the memory allocation request was too large.

4.1.5 Ustat. The `ustat` function takes as its first argument the offset of the device within the `dtab` table. This value is returned by `stat` and `fstat`, when given a device argument, in the `st_dev` and `st_rdev` buffer entries {4.1.1}. The stand-alone `ustat` returns the same information as the UNIX version.

4.1.6 Chdir. The global character pointer `_cdir` is set to the given string, which is prefixed to all file names not beginning with a slash. The string need not be a valid directory name, so `chdir` always returns successfully.

4.1.7 Lseek and Tell. There are no differences between the execution of these stand-alone functions and the operation of the corresponding UNIX routines.

4.1.8 Exit. The functions `exit` and `_exit` have the same meanings as they do in UNIX. The `_exit` function will attempt to return to the bootstrap program directly, and the `exit` function will call the `_cleanup` function first. The user may define his own `_cleanup` function or use the standard `_cleanup` that would be loaded from the library.

4.2 Null Functions

Several functions are included in the stand-alone library that only return zero or error values. These functions were included in the library to resolve external references in some C library functions. The functions that return a value of 0 are:

```
getgid  getegid  getuid  geteuid  nice  umask
```

The `chmod` function returns an error.

4.3 Deletions

The following is a complete list of those C library modules that have *not* been included in the stand-alone library:

```
acct.o      execvp.o    maus.o      sema.o      sync.o
alarm.o     fcntl.o     mktemp.o    setgid.o    syscall.o
cerror.o    fork.o      msg.o       setpgrp.o   system.o
chown.o     fp.o        oldmsg.o    setuid.o    tempfile.o
chroot.o    getpass.o   pause.o     signal.o    times.o
dup.o       getpid.o    pipe.o      sleep.o     ulimit.o
execl.o     getppid.o   plock.o     smclose.o   uname.o
execle.o    ioctl.o     popen.o     smfree.o    unlink.o
execv.o     kill.o      profil.o    smget.o     utime.o
execve.o    link.o      ptrace.o    smopen.o    wait.o
```

Several of these functions may, indeed, be faked rather than excluded; it is likely that the size of this list will be decreased in the future.

5. UTILITY FUNCTIONS

The functions described in this section do not have equivalent functions implemented in the C library.

5.1 User Functions

The following routines are included in the stand-alone library for the convenience of the user.

5.1.1 Getargv. The user has the option of having his stand-alone program invoked by a command interpreter program {6.2.1}, or by another standard UNIX bootstrap program. When a stand-alone program is not invoked by the command interpreter program, there can be no arguments specified on a command line and, consequently, no `argc`, `argv`, or environment are available to be passed to the program. In this case, the start-up code loads a value of 1 into `argc`, a null string into `argv[0]`, and a pointer to a null environment list into `envp`.

The `getargv` functions allows the program to pick up arguments after execution of the main routine has begun. The synopsis is:

```
getargv (cmd, argvp, ff)
char *cmd, *(*argvp[]);
int ff;
```

A prompt and the `cmd` argument are printed on the console and one line is read from the console. The *space* and *tab* characters are considered to be delimiters, and the single quote and double quote characters are properly understood. The arguments are stored in `argv`-format, with `cmd` as `argv[0]`, and the value of `argv` itself is stored into the address pointed to by `argvp`. The value of `argc` is returned.

Note that the area of memory used for the `argv` list is allocated by calling the `malloc` library function. A non-zero value for `ff` causes `getargv` to call the `free` function for `argvp` before calling `malloc`. The value of the `ff` argument would normally be zero on the first and only the first call.

If a typing error is made as the command is being entered, and the *kill* character is typed with the intention of retyping the line, there is a certain temptation to retype not only the arguments, but the command, too. Caveat.

5.1.2 Init. Before the `main` routine is called by the start-up code, the `_init` function is called. Normally, this function does some standard `MKNOD`s and `mount`s, but the user can define his own `_init`, if he does not want the standard one to be loaded. The synopsis is:

```
_init ()
```

5.2 System Functions

The following external functions form that part of the kernel of the stand-alone "system" that were globally defined for the purpose of communication within the modules of the system. Several may be useful to the user, but most will not be, and are included here for the sake of completeness:

System Utility Functions

Synopsis	Description
`_cond (istr, ostr)` `char *istr, *ostr;`	Converts `istr` into the form described in {1.2}, prepends the string given to `chdir`, if any, and places the result in the buffer pointed to by `ostr`.

`ino_t`
`_find (path, io)`
`char *path;`
`struct iob *io;`
	Returns the inode number of the proper path name pointed to by `path` on the file system described in `io`, and fills the appropriate parts of the `io` structure.

`_openi (n, io)`
`ino_t n;`
`struct iob *io;`
	Fills the `inode` structure in `io` with a copy of the disk inode whose number is `n` on the file system described in `io`.

`_prs (str)` `char *str;`	Prints the simple character string `str` on the console immediately.

`daddr_t`
`_sbmap (io, bn)`
`struct iob *io;`
`daddr_t bn;`
	Returns the number of the physical block corresponding to the logical block `bn` of the file on the device described in `io`.

`_trap (ps)` `int ps;`	Prints the type of trap that has occurred, based on the passed value of `ps`.

6. COMPILING AND EXECUTING STAND-ALONE PROGRAMS

6.1 Compilation

Programs are normally prepared for stand-alone execution by the UNIX `scc` command. The syntax of this command is a superset of the standard `cc` command:

`scc [ +[` *lib* `] ] [` *option* `] ... [` *file* `] ...`

The *option* and *file* arguments may be anything that can legally be used with the `cc` command; it should be noted, though, that the `-p` (profiling) option, as well as any object module that contains system calls, will cause the executable not to run.

`Scc` defines the compiler constant, `STANDALONE`, so that sections of C programs may be compiled conditionally for when the executable will be run stand-alone.

The first argument to `scc` specifies an auxiliary library that defines the device configuration of the computer for which the stand-alone executable is being prepared. On the PDP-11, *lib* may be either one of the following; on the VAX-11/780, *lib* may only be `A`:

`A` RP04/05/06 disk (also, RM05 disk on the VAX) and TU16 magnetic tape, or equivalent

`B` RK11/RK05 disk, RP11/RP03 disk, and TM11/TU16 magnetic tape, or equivalent

If no *+lib* argument is specified, `+A` is assumed. If the `+` argument is specified alone, no configuration library is loaded unless the user supplies his own. A manual entry for the `scc` command may be found in [1].

The user may define his own configuration library by loading an object module that defines `_devsw` to be an array of `devsw` structures {2.1.1}, `_devcnt` to be the number of structures in the array {3.2}, and `_init` to be a function that is to be called before the `main` routine {5.1.2}. If the user only wishes to define his own `_init` and not `_devsw` and `_devcnt`, or vice versa, he may do so, but the configuration library must also be loaded in order to resolve the other external reference(s).

6.2 Execution

6.2.1 Sash. Stand-alone programs are normally loaded using a command interpreter which passes the arguments that it reads after its prompt into the loaded program's `argv` list. This command interpreter is called `sash` (for *s*tand-*a*lone *sh*ell). Its implementation is described here, and its use is described more completely in the Appendix.

`Sash` relocates itself up 64K words on a PDP-11, and 320K words on a VAX-11/780. This enables a stand-alone user program to use all of memory below it.

Normally, only programs with execution modes 407 and 410 may be executed (see *a.out*(5) in [1]). On the PDP-11, `sash` turns on memory management in order to relocate itself, and then executes the high-memory copy of itself in *user* mode. It loads the user's program into low memory, copies the argument list to the upper limit of addressability for a non-separate instruction/data space program, sets up a small program beneath the argument list that interfaces from the user's program (which runs in *kernel* mode) to `sash` and sets the *kernel* stack pointer to its initial value, which is just underneath the small interface program; `sash` then manages to begin execution of the user's program in *kernel* mode at physical location 0. The interface program enables the user's program to return (exit) back to `sash` by a simple `rts` instruction. The use of memory management normally allows the user's program about 55.6K words for *text*, *data*, and *bss* segments. If the user wishes to set up his own *bss* segment, then only *text* and *data* are limited to 55.6K words. It should be noted, however, that because memory management is enabled at the outset, the user's program must turn memory management off before changing any memory management-related registers.

To load mode 411 (separate instruction and data space) files, `sash` loads the *data* and *bss* segments at physical address 0 (set to be *kernel data*), and the *text* segment is loaded at the next 64-byte boundary (set to be *kernel text*). `Sash` then turns off memory management, and assumes that the program will restructure itself. It cannot be run without restructuring because the program break can only expand onto the *text* segment, and the stack pointer may contain an address that is in the middle of the *text* segment.

The address space of the VAX-11/780 is sufficiently large that memory management need not be used, and the user's program may be started by a simple subroutine call, and exited by a return from that call.

6.2.2 Other Bootstrap Programs. Alternatively, a stand-alone program may be loaded into memory by some other UNIX bootstrap program. If this is done, the start-up code senses that an argument list is not available, so `argc` will be set to 1, and `argv[0]` will be set to a null string before execution begins, and may be reassigned by `getargv`.

6.3 Relocatable Programs

The stand-alone I/O library may be used with programs than need to relocate themselves at some point during execution. Although this is never a simple task, it is quite a bit easier to do so on the VAX-11/780 than the PDP-11, and somewhat easier on the PDP-11 if memory management need not be used. The user who is considering writing a relocatable program is referred to the source code of the machine-dependent (assembler language) part of the `sash` program {Appendix} for hints.

On the VAX-11/780, the `-T` option may be given to the `ld` program to do the relocation. On the PDP-11, no special processing by `ld` is necessary.

7. OVERHEAD AND PERFORMANCE

On both the PDP-11 and VAX-11/780, a null program will compile to produce an executable object module that has a *text* segment that is slightly larger than 6K bytes, and *data* and *bss* segments that add up to about 8K bytes. This is a good rule-of-thumb calculation for the minimum size of a program that is compiled with the stand-alone library.

Because stand-alone programs run (by definition) without competing against other processes for CPU time, and are never swapped out of memory, a stand-alone program's execution is faster than that of the same program running under UNIX. However, if that program does some I/O operations, it will not benefit from some of the short-cut operations that are implemented in UNIX, such as disk read-ahead, and will therefore actually run more slowly stand-alone than under UNIX.

Acknowledgements

The stand-alone I/O library was originally based on a library written by Charles Haley whom I would like to thank for his comments and suggestions during the course of my work. I would also like to thank Larry Wehr for his explanations of the workings of the UNIX system and device drivers, as well as Ted Kowalski for his help in debugging several stand-alone programs and his suggestions of practical extensions to that which already existed.

References

[1] Dolotta, T. A., Olsson, S. B., and Petruccelli, A. G., eds. *UNIX User's Manual*—Release 3.0. Bell Laboratories, June 1980.

[2] *UNIX Time-Sharing System: UNIX Programmer's Manual*—Seventh Edition. Bell Laboratories, January, 1979.

[3] *UNIX/32V Time-Sharing System: UNIX Programmer's Manual*—Version 1.0. Bell Laboratories, February, 1979.

[4] *Peripherals Handbook.* Digital Equipment Corporation, 1978.

[5] *PDP 11/70 Processor Handbook.* Digital Equipment Corporation, 1976.

[6] *VAX 11/780 Architecture Handbook.* Digital Equipment Corporation, 1977.

[7] *VAX 11/780 Hardware Handbook.* Digital Equipment Corporation, 1978.

Appendix: The Stand-Alone Command Interpreter

The stand-alone command interpreter is called `sash` (for *s*tand-*a*lone *sh*ell). It is a glorified UNIX boot program. `Sash` is begun running through whatever means available. It relocates itself up to high memory and executes there. When it is running, it prompts with `$$`.

`Sash` accepts three types of commands. The most common type is the *program execution* command. Here the user types the name of the stand-alone program to be executed, followed by arguments to be passed to the program. The program name and arguments are separated by spaces or tabs, and the single-quote and double-quote characters are properly understood (for arguments containing special characters within them). For example, if `/stand/ls` is a stand-alone program that does the same as the UNIX `ls` program, then in order to get a long listing of the contents of the directory `/tmp`, the user would type:

```
$$ /stand/ls -l /tmp
```

UNIX itself may be booted by using this method:

```
$$ /unix
```

The second type of command is the `cd` command. `Sash` has a notion of its *current directory*. All programs that are called with names that do not begin with a slash (`/`) are searched for relative to the current directory. When `sash` is begun executing, the current directory is the root directory (`/`). Thus, in the previous paragraph, UNIX could have been booted by typing:

```
$$ unix
```

If the `cd` command is invoked with an argument, then the argument becomes the current directory. The following sequence is equivalent to the `ls` command discussed above:

```
$$ cd /stand
$$ ls -l /tmp
```

The current directory is local to the `sash` program. It is remembered from one `sash` command to the next. It is not, however, passed on to the invoked program. Arguments that are passed to programs must therefore be relative to the root directory.

If `cd` is invoked with no arguments, the value of the current directory is printed.

`Sash` has a default notion of the disk type and unit number for the root file system, as well as for a `/usr` file system. These are generally slices 0 and 1, respectively, of unit 0 of the RP04/05/06 disk. (The defaults may be easily changed by recompiling the `sash` source.) To change `sash`'s ideas of disk type and unit number, the `set` command may be used. There are two basic forms of the `set` command: `set unit` and `set disk`. Their synopses are:

```
set unit {/¦/usr} {0¦1¦...¦7}
set disk {/¦/usr} {rk05¦rp03¦rp04}
```

where `{...¦...}` indicates a mandatory choice. On the VAX-11/780, the `rk05` and `rp03` choices do not exist. For example, in order to execute a stand-alone program in `/usr/steve/saprog` where `/usr` is the file system on slice 1 of RP03 unit 2, the user may type:

```
$$ set disk /usr rp03
$$ set unit /usr 2
$$ /usr/steve/saprog
```

The notions of disk type and unit number are, like current directory, local to `sash`, and are not passed to the invoked program, which has its own idea of where `/usr` (if any) and the root file system are located.