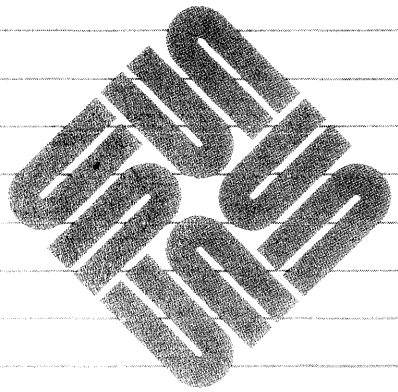
sun®
microsystems

C Programmer's Guide



Part Number: 800-1771-10
Revision A, of 9 May 1988

Contents

Tables

Figures

Preface

This manual describes how to write C programs that interface with the SunOS operating system in a nontrivial way. This includes programs that use files by name, that use pipes, that invoke other commands as they run, or that catch interrupts and other signals during execution.

There is no attempt to be complete; only generally useful material is dealt with. It is assumed that you will be programming in C, so you must be able to read C roughly up to the level of language as described in *The C Programming Language*, by Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, 1978. You should also be familiar with SunOS itself, at least as far as being familiar with getting around in the *SunOS Reference Manual.*

1

Using The Sun C Compiler

1

Using The Sun C Compiler

This chapter describes how to compile C programs on Sun Microsystems' workstations under the SunOS version of the UNIX† operating system.

If you are already familiar with using `cc`, (the UNIX C compiler), either on Sun workstations or on other UNIX systems, you can probably ignore or skim the rest of this chapter without regretting it later.

If you need to learn about programming in C, or about SunOS programming tools, you should refer to one or more of the introductory books available that address the topic.

1.1. Basics — Compiling and Running C Programs

This section shows how to compile and run a minimal C program. Consider this C program that just displays a message and exits:

```
#include <stdio.h>

main()
{
    printf("Real Programmers Hack C!\n");
    exit(0);
  /* But they can do it in hexadecimal if necessary */
}
```

Using your preferred text editor, save the text of this program in a file called `hackers.c`. After you have saved the file, compile it with the `cc` command:

```
tutorial% cc hackers.c
tutorial%
```

`cc` works silently unless there are errors in the program: In this case, there are no errors, and `cc` compiles the program and saves an executable version of it in a file named `a.out`.

† UNIX is a registered trademark of AT&T.

When you want to run the program, type the name of the executable file:

```
tutorial% a.out
Real Programmers Hack C!
tutorial%
```

1.2. C Compiler

This section describes the compiler options supported by Sun Microsystems' C compiler. Later sections cover specific dependencies and features of Sun C under SunOS.

```
cc [options] filename
```

`cc` translates programs written in C into executable load modules, (or into relocatable binary programs for later linking with `ld`), and optionally links (or binds) the result with object files generated by `cc` or other language processors.

`cc` accepts a list of C source files and various object files contained in the list of files specified by *filename....* The resulting executable is placed in the file *a.out*, unless the (`-o`) option is specified (see below).

`cc` lets you compile and link any combination of the following:

- C source files (with a `.c` suffix)
- C preprocessed source files with a `.i` suffix
- SunOS system object-code files with `.o` suffixes.
- Assembler source files with `.s` suffixes.

After successfully linking, `cc` places the product of linking those files in the file `a.out`, or in the file specified by the `-o` option.

1.3. `cc` Options

`-a` Option

`-a` directs `cc` to insert code to count how many times each basic block in a program is executed. This creates a `.d` file for every `.c` file compiled that accumulates execution data for its corresponding source file. On the Sun-2, -3, and -4 you can then run `tcov`(1) on the source files to generate statistics about the program.

`-align` *_block_* Option

This option directs `cc` to page-align the uninitialized FORTRAN common `block`: This increases its size to a whole number of pages, and places its first byte at the beginning of a page. Multiple `-align` options may be given.

`-c` Option

`-c` directs `cc` to suppress linking with `ld` and produce a `.o` file for each source file.

NOTE *You should use the `-o` option to explicitly name a single object file.*

`-C` Option

`-C` prevents the C preprocessor, `cpp`, from removing comments.

`-dryrun` Option

`-dryrun` directs `cc` to show but not execute the commands constructed by the compilation driver.

`-D`*name*`[=`*def*`]` Option

This option defines a symbol *name* to the C preprocessor `cpp`. This is equivalent to a `#define` directive at the beginning of the source. If you don't use *=def*, *name* is defined as '1'. Multiple `-D` options may be given.

`-E` Option

`-E` runs the source file through `cpp` only. It sends the output to either `stdout`, or to a file named with the `-o` option (which must end with `.i`) and includes the `cpp` line numbering information. (See also, the `-P` option.)

Floating-Point Options

Sun supports several ways to perform floating-point calculations, both in hardware and software. The floating-point point options provided by `cc` permit you to choose the way that gives you the best performance and portability for your programs.

NOTE *There are no floating point options for the Sun-4. On the Sun386i, only the `-fsingle` option is legal, but it has no effect.*

The floating-point code generation options that you use can be any of the following:

`-f68881` This directs `cc` to generate in-line code for the Motorola MC68881 floating-point coprocessor (supported only on Sun-3 systems).

`-ffpa` This directs `cc` to generate in-line code for the Sun Floating-Point Accelerator (supported only on Sun-3 systems).

`-fsky` This directs `cc` to generate in-line code for the Sky floating-point processor (supported only on Sun-2).

`-fsoft` This directs `cc` to generate software floating-point calls (this is the default for all Sun-2 and Sun-3 workstations).

`-fswitch` This directs `cc` to generate runtime-switched floating-point calls. The compiled object code is linked at runtime to routines that support one of the above types of floating-point code. This option exists mainly for compatibility with earlier releases of `cc` on Sun-2's. Floating-point-intensive programs on Sun-3's should use either the `-ffpa` or `-f68881` options instead.

`-fsingle` This directs `cc` to use single-precision arithmetic in computations involving only `float` expressions — that is, do not convert everything to `double`, which is the default. Note that floating-point *parameters* are still converted to double precision, and *functions* returning values still return double-precision values.

Although this is not standard Kernighan and Ritchie C, some programs run much faster using this option. Be aware that some significance can be lost due to lower-precision intermediate values.

-g Option

`-g` produces additional symbol table information for *dbx*(1) and *dbxtool*(1) and passes the `-lg` flag to `ld`.

This option suppresses the `-O` and `-R` options.

-go Option

`-go` produces additional symbol table information for `adb`. When this option is given, the `-O` and `-R` options are suppressed.

-help Option

`-help` displays information about `cc`.

-I*pathname* Option

This option adds *pathname* to the list of directories which are searched for `#include` files with relative filenames (those not beginning with slash /).

The preprocessor first searches for `#include` files in the directory containing the sourcefile, then in directories named with `-I` options (if any), and finally, in `/usr/include`.

-J Option

`-J` generates 32-bit offsets in `switch` statement branches. Not supported on the Sun386i.

-l *lib* Option

This option directs `cc` to link with object library *lib* (for `ld`).

-L *dir* Option

This option adds *dir* to the list of directories containing object-library routines (for linking with `ld`).

-M Option

`-M` runs only the macro preprocessor on the named C programs, requesting that it generate makefile dependencies and send the result to the standard output (see *make*(1) for details about makefiles and dependencies).

-o *outfile* Option

This option names the output file *outfile*. *outfile* must have the appropriate suffix for the type of file to be produced by the compilation. *outfile* cannot be the same as *sourcefile* since `cc` will not overwrite the source file.

-O Option

`-O` directs `cc` to optimize the object code. It is ignored when either `-g` or `-go` is used.

-p Option

`-p` prepares the object code to collect data for profiling with `prof`. `-p` invokes a run-time recording mechanism that produces a *mon.out* file at normal termination.

-pg Option

`-pg` prepares the object code to collect data for profiling with `gprof`(1). It invokes a run-time recording mechanism that produces a `gmon.out` file at normal termination.

-pipe Option

`-pipe` directs `cc` to use pipes, rather than intermediate files, between compilation stages. (Very CPU-intensive.)

-P Option

-P runs the source file through the C preprocessor, cpp, without putting *cpp*-type line-number information in the output. It puts the output in a file with a .i suffix.

-Qoption *prog opt* Option

This option passes the option *opt* to the compiler phase *prog*. The option must be appropriate to that program and may begin with a minus sign. *prog* can be one of: as(1), cpp(1), inline, or ld(1).

-Qpath *pathname* Option

This inserts a directory *pathname* into the compilation search path. This lets you choose whether or not to use default versions of programs invoked during compilation.

-Qproduce *sourcetype* Option

This option produces source code of the type *sourcetype*. *sourcetype* can be one of the following:

.c C source (from bb_count).

.i Preprocessed C source from cpp.

.o Object file from as.

.s Assembler source (from *ccom*, *inline* or *c2*).

-R Option

-R directs cc to merge the data segment with the text segment for as. Data initialized in the object file produced by this compilation is read-only, and (unless linked with ld -N) is shared between processes. This option is ignored when either -g or -go is used.

-S Option

-S directs cc to produce an assembly source file but not to assemble the program.

-temp= *dir* Option

This sets the directory for temporary files to be generated during the compilation process to be *dir*.

-time Option

-time directs cc to report execution times for the various compilation passes.

-U*name* Option

This removes any initial definition of the cpp symbol *name*. This option is the inverse of the -D option. Multiple -U options may be given.

-v Option

-v directs cc to print the name of each program it executes.

-w Option

-w directs cc to not print warnings.

2

Accessing a Program's Environment

2

Accessing a Program's Environment

This chapter discusses two basic topics:

- How to get the arguments from the command line used to run a program.
- How to access environment variables.

2.1. Basics — Accessing Command Line Arguments

Assuming that you have written a C program, you might like to be able to get information from the command line when the user starts up the program. Although many SunOS system programs are run as *filters* — they obtain input from the standard input and send output to the standard output — sometimes you might like to be able to specify alternative files to operate upon, or to specify *options* on the command line to control the program's behavior.

When a C program is run as a command, the arguments on the command line are made available to the program's function `main` as an argument count `argc` and an array `argv` of pointers to character strings that contain the arguments. By convention, `argv[0]` is the command name itself, so `argc` is always greater than 0.

The following program illustrates the mechanism: it simply echoes its arguments back to the terminal — this is essentially the `echo` command.

```
#include <stdio.h>

main(argc, argv)    /* echo arguments */
    int argc;
    char    *argv[];
{
    int arg_count;

    for (arg_count = 1; arg_count < argc; arg_count++)
       printf("%s%c", argv[arg_count], (arg_count<argc-1) ? ' ' : '\n');
    exit(0);
}
```

`argv` is a pointer to an array whose elements are pointers to arrays of characters; each is terminated by `\0`, so they can be treated as strings. The program starts by printing `argv[1]` and loops until it has printed `argv[argc-1]`.

The argument count and the arguments are parameters to `main`, so if you want to keep them around for other routines to use, you must copy them to external variables.

2.2. Basics — Accessing Environment Variables

The next topic is how to obtain values from a running program's environment.

You can 'tailor' your SunOS system environment by setting *environment variables*, and these environment variables are accessible from a program.

When a C program is started, three arguments are passed to its `main` function. In addition to `argc` and `argv` as described above, there is an array of pointers — named `envp` — to the character strings that comprise the environment.

Each environment variable is a null-terminated character string of the form *name* = *value* that can be manipulated like any other character string.

Here is a short program to display all the environment variables:

```
#include <stdio.h>

main(argc, argv, envp)
    int argc;
    char    *argv[];
    char    *envp[];
{

    int env_count = 0;

        while (envp[env_count] != NULL) {
        printf("%s\n", envp[env_count]);
        env_count++;
    }
        exit(0);
}
```

If you save the above text as `environ.c`, you can compile and run it as follows:

```
tutorial% cc environ.c
tutorial% a.out
HOME=/usr/henry
SHELL=/bin/csh
PATH=/usr/doctools/bin:/usr/local:.:/usr/ucb:/bin:/usr/bin
TERM=sun
USER=henry
EXINIT=set noai wrapmargin=16 para=IPLPPPQPLSLEDSDETSTEKSKEPSPEEQENLIpplpipbp
WINDOW_PARENT=/dev/win0
WINDOW_ME=/dev/win8
WINDOW_GFX=/dev/win8
tutorial%
```

Accessing Environment Variable Using `getenv()`

While `environ.c` is somewhat useful, parsing the *name* = *value* pairs is rather tedious, so there is a C library function called `getenv()` whose purpose is to get values from the environment. Here is the interface definition for `getenv()`:

```
char    *getenv(name)
char    *name;
```

Now we can compose a program that displays the value of a variable supplied as an argument on its command line:

```
   /*  getenv().c -- obtain specified variable from environment */

#include <stdio.h>

char    *getenv();

main(argc, argv)
    int     argc;
    char    *argv[];
{

    char    *variable;

                        /*  Check any argument supplied  */
    if (argc < 2) {
    printf("Usage: %s name\n", argv[0]);
        exit(1);
    }

                        /*  Search for the variable  */
    if ((variable = getenv(argv[1])) == NULL)
    printf("%s: no variable %s\n", argv[0], argv[1]);
    else
        printf("%s = %s\n", argv[1], variable);
        exit(0);
}
```

After compiling and running this program, you can use it like this:

```
tutorial% a.out PATH
PATH = /usr/doctools/bin:/usr/local:.:/usr/ucb:/bin:/usr/bin
tutorial% a.out nonesuch
a.out: no variable nonesuch
tutorial% a.out
Usage: a.out name
tutorial%
```

3

Processes

3

Processes

The following section describes how to execute one program from within another. This makes it possible to use existing programs rather than always having to write new ones.

3.1. The `system()` Function

The easiest way to execute a program from another is to use the standard library routine `system()`. `system()` takes one argument, a command string exactly as typed at the terminal (except for the newline at the end) and executes it — for instance, to timestamp the output of a program:

```
main( ) {
    system("date");

    /* rest of processing */
}
```

The in-memory formatting capabilities of `sprintf()` are useful if you must build the command string from pieces.

3.2. Low-Level Process Creation — `execl()` and `execv()`

If you're not using the standard library, or if you need finer control over what happens, you will have to construct calls to other programs using the more primitive routines that the standard library's `system()` routine is based on[1].

The most basic operation is to execute another program *without returning*, by using the routine `execl()`. For example, you can display the date as the last action of a running program:

```
execl("/bin/date", "date", NULL);
```

[1] `system()` uses */bin/sh* (the Bourne Shell) to execute the command string, so syntax specific to the C-Shell will not work.

The arguments that you pass to `execl()` are:

1. The *filename* of the command that you want executed; you have to know where it is found in the file system.
2 The second argument is conventionally the program name (that is, the last component of the file name), but this is seldom used except as a placeholder.
3. If the command takes arguments, they are strung out in order after the program name (or its position).
4. Following the arguments, the end of the list is marked by a `NULL` argument.

The `execl()` call overlays the existing program with the new one, runs that, then exits. There is *no* return to the original program.

More commonly, a program falls into two or more phases that communicate only through temporary files. Here it is natural to start the second pass simply by an `execl()` call from the first.

The one exception to the rule that the original program never gets control back occurs when there is an error in performing the `execl()` call itself, for example if the file can't be found or is not executable. If you don't know where `date()` is located, you might try

```
execl("/bin/date", "date", NULL);
execl("/usr/bin/date", "date", NULL);
fprintf(stderr, "Someone stole 'date'\n");
```

A variant of `execl()` called `execv()` is useful when you don't know in advance how many arguments there are going to be. The call is

```
execv(filename, argp);
```

where `argp` is an array of pointers to the arguments; the last pointer in the array must be `NULL` so `execv()` can tell where the list ends. As with `execl()`, `filename` is the file in which the program is found, and `argp[0]` is the name of the program. (This arrangement is identical to the `argv` array for program arguments.)

Neither of these routines provides the niceties of normal command execution. There is no automatic search of multiple directories — you have to know precisely where the command is located. Nor do you get the expansion of metacharacters like `<`, `>`, `*`, `?` and `[ ]` in the argument list. If you want these, use `execl()` to invoke the shell `sh`(1), which then does all the work. Construct a string `commandline` that contains the complete command as it would have been typed at the terminal, then say

```
execl("/bin/sh", "sh", "-c", commandline, NULL);
```

The shell is assumed to be at a fixed place, `/bin/sh`. Its argument `-c` says to treat the next argument as a whole command line, so it does just what you want. The only problem is in constructing the right information in `commandline`.

3.3. Process Control — `fork()` and `wait()`

So far what we've talked about isn't really all that useful by itself. Now we will show how to regain control after running a program with `execl()` or `execv()`. Since these routines simply overlay the new program on the old one, to save the old one requires that it first be split into two copies; one of these can be overlaid, while the other waits for the new, overlaying program to finish. The splitting is done by a routine called `fork()`:

```
proc_id = fork( );
```

This call splits the program into two copies, both of which continue to run. The only difference between the two is the value of `proc_id`, the *process id.* In one of these processes (the *child*), `proc_id` is zero. In the other (the *parent*), `proc_id` is nonzero; it is the process number of the child. Thus the basic way to call, and return from, another program is

```
if (fork( ) == 0)
    execl("/bin/sh", "sh", "-c", cmd, NULL); /* in child */
```

And in fact, except for handling errors, this is sufficient. The `fork()` makes two copies of the program. In the child, the value returned by `fork()` is zero, so it calls `execl()` which does the `command` and then dies. In the parent, `fork()` returns nonzero so it skips the `execl()`. If there is any error, `fork()` returns –1.

More often, the parent wants to wait for the child to terminate before continuing itself. This can be done with the function `wait()`:

```
int status;

if (fork( ) == 0)
    execl(...);
wait(&status);
```

This still doesn't handle any abnormal conditions, such as a failure of the `execl()` or `fork()`, or the possibility that there might be more than one child running simultaneously. The `wait()` returns the process id of the terminated child, in case you want to check it against the value returned by `fork()`. Finally, this fragment doesn't deal with any unusual behavior on the part of the

child (which is reported in `status`). Still, these three lines are the heart of the standard library's `system()` routine, which we'll show in a moment.

The `status` returned by `wait()` encodes in its low-order eight bits the system's idea of the child's termination status; it is 0 for normal termination and nonzero to indicate various kinds of problems. The next higher eight bits are taken from the argument of the call to `exit` which caused a normal termination of the child process. It is good coding practice for all programs to return meaningful status.

When a program is called by the shell, the three file descriptors 0, 1, and 2 are set up to point to the right files (see Appendix A.1), and all other possible file descriptors are available for use. When this program calls another one, correct etiquette suggests making sure the same conditions hold. Neither `fork()` nor `exec` affect open files in any way. If the parent is buffering output that must come out before output from the child, the parent must flush its buffers before the `execl()`. Conversely, if a caller buffers an input stream, the called program will lose any information that has been read by the caller.

3.4. Pipes

A *pipe* is an I/O channel intended for use between two cooperating processes: one process writes into the pipe, while the other process reads from the pipe. The system looks after buffering the data and synchronizing the two processes. Most pipes are created by the shell, as in

```
tutorial% ls | pr
```

which connects the standard output of `ls` to the standard input of `pr`. Sometimes, however, it is most convenient for a process to set up its own plumbing; in this section, we illustrate how the pipe connection is established and used.

The system call `pipe()` creates a pipe. Since a pipe is used for both reading and writing, two file descriptors are returned; the actual usage is like this:

```
int     fd[2];

stat = pipe(fd);
if (stat == -1)
    /* there was an error ... */
```

`fd` is an array of two file descriptors, where `fd[0]` is the read side of the pipe and `fd[1]` is for writing. These may be used in `read`, `write()` and `close()` calls just like any other file descriptors.

If a process reads a pipe which is empty, it waits until data arrives; if a process writes into a pipe which is too full, it waits until the pipe empties somewhat. If the write side of the pipe is closed, a subsequent `read` will encounter end of file.

To illustrate the use of pipes in a realistic setting, let us write a function called `popen(cmd, mode)`, which creates a process `cmd` (just as `system()` does),

and returns a file descriptor that will either read or write that process, according to mode. That is, the call

```
fout = popen("pr", WRITE);
```

creates a process that executes the pr command; subsequent write() calls using the file descriptor fout will send their data to that process through the pipe.

popen() first creates the pipe with a pipe() system call; it then fork()'s to create two copies of itself. The child decides whether it is supposed to read or write, closes the other side of the pipe, then calls the shell (via execl()) to run the desired process. The parent likewise closes the end of the pipe it does not use. These closes are necessary to make end-of-file tests work properly. For example, if a child that intends to read fails to close the write end of the pipe, it will never see the end of the pipe file, just because there is one writer potentially active.

```
#include <stdio.h>

#define READ    0
#define WRITE   1
#define tst(a, b)       (mode == READ ? (b) : (a))
static  int     popen_pid;

popen(cmd, mode)
char    *cmd;
int     mode;
{
        int p[2];

        if (pipe(p) < 0)
           return(NULL);
        if ((popen_pid = fork( )) == 0) {
           close(tst(p[WRITE], p[READ]));
           close(tst(0, 1));
           dup(tst(p[READ], p[WRITE]));
           close(tst(p[READ], p[WRITE]));
           execl("/bin/sh", "sh", "-c", cmd, 0);
           _exit(1); /* disaster has occurred if we get here */
        }
        if (popen_pid == -1)
           return(NULL);
        close(tst(p[READ], p[WRITE]));
        return(tst(p[WRITE], p[READ]));
}
```

The sequence of close()'s in the child is a bit tricky. Suppose that the task is to create a child process that will read data from the parent. Then the first close() closes the write side of the pipe, leaving the read side open. The lines

```
close(tst(0, 1));
dup(tst(p[READ], p[WRITE]));
```

are the conventional way to associate the pipe descriptor with the standard input of the child. The `close()` closes file descriptor 0, that is, the standard input. `dup()` is a system call that returns a duplicate of an already open file descriptor. File descriptors are assigned in increasing order and the first available one is returned, so the effect of the `dup()` is to copy the file descriptor for the pipe (read side) to file descriptor 0; thus the read side of the pipe becomes the standard input[2]. Finally, the old read side of the pipe is closed.

A similar sequence of operations takes place when the child process is supposed to write to the parent instead of reading. You may find it a useful exercise to step through that case.

The job is not quite done, for we still need a function `pclose` to close the pipe created by `popen()`. The main reason for using a separate function rather than `close()` is that it is desirable to wait for the termination of the child process. First, the return value from `pclose` indicates whether the process succeeded. Equally important when a process creates several children is that only a bounded number of unwaited-for children can exist, even if some of them have terminated; performing the `wait()` lays the child to rest. Thus:

```
#include <signal.h>

pclose(fd)      /* close pipe fd */
int fd;
{
     register r, (*hstat)( ), (*istat)( ), (*qstat)( );
     int      status;
     extern int popen_pid;

     close(fd);
     istat = signal(SIGINT, SIG_IGN);
     qstat = signal(SIGQUIT, SIG_IGN);
     hstat = signal(SIGHUP, SIG_IGN);
     while ((r = wait(&status)) != popen_pid && r != -1);
     if (r == -1)
               status = -1;
     signal(SIGINT, istat);
     signal(SIGQUIT, qstat);
     signal(SIGHUP, hstat);
     return(status);
}
```

The calls to `signal()` make sure that no interrupts, etc. interfere with the waiting process; this is the topic of the next section.

[2] Yes, this is a bit tricky, but it's a standard idiom.

The routine as written has the limitation that only one pipe may be open at once, because of the single shared variable `popen()_pid`; it really should be an array indexed by file descriptor. A `popen()` function, with slightly different arguments and return value is available as part of the standard I/O library discussed later. As currently written, it shares the same limitation.

4

Signals — Interrupts and All That

4

Signals — Interrupts and All That

This chapter is concerned with how to deal gracefully with signals from the outside world (like interrupts), and with program faults. Since there's nothing very useful that can be done from within a C program about program faults, which arise mainly from illegal memory references or from execution of peculiar instructions, we'll discuss only the outside world signals: *interrupt* and *quit*, which are generated from the keyboard, *hangup*, caused by hanging up the phone on dialup lines, and *terminate*, generated by the *kill* command. When one of these events occurs, the signal is sent to *all* processes which were started by the corresponding user — the signal terminates the process unless other arrangements have been made. In the *quit* case, a core image file is written for debugging purposes.

`signal()` is the routine which alters the default action. `signal()` has two arguments: the first specifies the signal to be processed, and the second argument specifies what to do with that signal. The first argument is just a numeric code, but the second is either a function, or a somewhat strange code that requests that the signal either be ignored or that it be given the default action. The include file `signal.h` gives names for the various arguments, and should always be included when signals are used. Thus

```
#include <signal.h>
 ...
signal(SIGINT, SIG_IGN);
```

means that interrupts are ignored, while

```
signal(SIGINT, SIG_DFL);
```

restores the default action of process termination. In all cases, `signal()` returns the previous value of the signal. The second argument to `signal()` may instead be the name of a function (which must be declared explicitly if the compiler hasn't seen it already). In this case, the named routine will be called when the signal occurs. Most commonly this facility is used so that the program can clean up unfinished business before terminating, for example to delete a temporary file:

```
#include <signal.h>

main( )
{
    int onintr( );

    if (signal(SIGINT, SIG_IGN) != SIG_IGN)
        signal(SIGINT, onintr);

    /* Process ... */

    exit(0);
}

onintr( )
{
    unlink(tempfile);
    exit(1);
}
```

Why the test and the double call to `signal()`? Recall that signals, like interrupts, are sent to *all* processes started from a particular user. Accordingly, when a program is to be run non-interactively (started with `&`), the shell turns off interrupts for it so it won't be stopped by interrupts intended for foreground processes. If this program began by announcing that all interrupts were to be sent to the `onintr()` routine regardless, that would undo the shell's effort to protect it when run in the background.

The solution, shown above, is to test the state of interrupt handling, and to continue to ignore interrupts if they are already being ignored. The code as written depends on the fact that `signal()` returns the previous state of a particular signal. If signals were already being ignored, the process should continue to ignore them; otherwise, they should be caught.

A more sophisticated program may wish to intercept an interrupt and interpret it as a request to stop what it is doing and return to its own command processing loop. Think of a text editor — interrupting a long display should not terminate the edit session and lose the work already done. The outline of the code for this case may be written like this:

```
#include <signal.h>
#include <setjmp.h>
jmp_buf sjbuf;

onintr( )
{
    printf("\nInterrupt\n");
    longjmp(sjbuf); /* return to saved state */
}

main( )
{
    int (*istat)( ), onintr( );

    istat = signal(SIGINT, SIG_IGN);  /* save old status */
    setjmp(sjbuf);  /* save current stack position */
    if (istat != SIG_IGN)
        signal(SIGINT, onintr);

    /* main processing loop */
}
```

The include file `setjmp.h` declares the type `jmp_buf` — an object in which a process's state can be saved. `sjbuf` is such an object. The `setjmp()` routine then saves the state. When an interrupt occurs the `onintr()` routine is called, which can display a message, set flags, or whatever. `longjmp()` takes as argument an object set by `setjump()`, and restores control to the location following the call to `setjump()`, so control (and the stack level) will pop back to the place in the main routine where the signal is set up and the main loop entered. Notice, by the way, that the signal gets set again after an interrupt occurs.

Some programs that want to detect signals simply can't be stopped at an arbitrary point, for example in the middle of updating a linked list. If the routine called when a signal occurs sets a flag and then returns instead of calling `exit()` or `longjmp()`, execution continues at the exact point it was interrupted. The interrupt flag can then be tested later.

There is one difficulty associated with this approach. Suppose the program is reading the standard input when the interrupt is sent. The specified routine is duly called; it sets its flag and returns. If it were really true, as we said above, that 'execution resumes at the exact point it was interrupted,' the program would continue reading `stdin()` until the user typed another line. This behavior might well be confusing, since the user might not know that the program is reading; he presumably would prefer to have the signal take effect instantly. The method chosen to resolve this difficulty is to terminate the read when execution resumes after the signal, returning an error code which indicates what happened.

Thus programs which catch and resume execution after signals should be prepared for 'errors' which are caused by interrupted system calls.

The ones to watch out for are `read()`, `wait()`, and `pause()`. A program whose `onintr()` routine just sets `intflag`, resets the interrupt signal, and returns, should usually include code like the following when it reads the standard input:

```
if (getchar() == EOF)
    if (intflag)
        /* EOF caused by interrupt */
    else
        /* true end-of-file */
```

A final subtlety to keep in mind becomes important when catching signals is combined with executing other programs. Suppose a program catches interrupts, and also includes a method (like '!' in *ex* and *vi*) whereby other programs can be executed. Then the code should look something like this:

```
if (fork( ) == 0)
    execl(...);
signal(SIGINT, SIG_IGN);     /* ignore interrupts */
wait(&status);           /* until the child is done */
signal(SIGINT, onintr);      /* restore interrupts */
```

Why is this? Again, it's not obvious, but not really difficult. Suppose the program you call catches its own interrupts. If you interrupt the subprogram, it will get the signal and return to its main loop, and probably read from `stdin`. But the calling program will also pop out of its wait for the subprogram and read from `stdin`. Having two processes reading the same input is very unfortunate, since the system figuratively flips a coin to decide who should get each line of input. A simple way out is to have the parent program ignore interrupts until the child is done. This reasoning is reflected in the standard I/O library function `system`:

```
#include <signal.h>

system(s)   /* run command string s */
char *s;
{
    int status, pid, w;
    register int (*istat)( ), (*qstat)( );

    if ((pid = fork( )) == 0) {
        execl("/bin/sh", "sh", "-c", s, 0);
        _exit(127);
    }
    istat = void(SIGINT, SIG_IGN);
    qstat = void(SIGQUIT, SIG_IGN);
    while ((w = wait(&status)) != pid && w != -1)
        ;
    if (w == -1)
        status = -1;
    void(SIGINT, istat);
    void(SIGQUIT, qstat);
    return(status);
}
```

As an aside on declarations, the function `void()` obviously has a rather strange second argument. It is in fact a pointer to a function, and this is also the type of the signal routine itself. The two values `SIG_IGN` and `SIG_DFL` have the right type, but are chosen so they coincide with no possible actual functions. For the enthusiast, here is how they are defined for the Sun system — the definitions should be sufficiently ugly and nonportable to encourage use of the include file.

NOTE *Before SunOS release 4.0,* `void()` *was named* `signal()`.

```
#define SIG_DFL     (void (*)())0
#define SIG_IGN     (void (*)())1
```

5

The Standard I/O Library

5

The Standard I/O Library

Input and output are, strictly speaking, not an intrinsic part of the C programming language. Rather, the input and output functions are supplied by a library which comes with each implementation.

This chapter describes the Standard I/O Library available to C programmers on Sun workstations.

5.1. The Standard I/O Library

The standard I/O library was designed with the following goals in mind:

1. It must be as efficient as possible, both in time and in space, so that there will be no hesitation in using it, no matter how critical the application.
2. It must be simple to use, and also free of the magic numbers and mysterious calls whose use mars the understandability and portability of many programs using older packages.
3. The interface provided should be applicable on all machines, whether or not the programs which implement it are directly portable to other systems, or to non-Sun machines running a version of UNIX.

5.2. Using the Standard I/O Library

The `stdio.h` routines are in the normal C library, so no special library argument must be declared in your program for linking. All names in the include file intended only for internal use begin with an underscore _ to reduce the possibility of collision with a user name. The names intended to be visible outside the package are

- **`stdin()`**
- **`stdout()`**
- **`stderr()`**
- **`EOF`**
- **`NULL`**
- **`FILE`**
- **`BUFSIZ`**
- **`getc()`**, **`getchar()`**, **`putc()`**, **`putchar()`**, **`feof()`**, **`ferror()`**, and **`fileno()`**, are defined as macros. Their actions are described below; they are mentioned here to point out that it is not possible to redeclare them

and that they are not actually functions; thus, for example, they may not have breakpoints set on them.

The routines in this package offer the convenience of automatic buffer allocation and output flushing where appropriate. The names `stdin()`, `stdout()`, and `stderr()` are constants and may not be assigned to.

Any program which uses the Standard I/O Library must have the following line in the program source text, before using any of the functions in the library.

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

Putting this `include` statement in your program defines some macros and variables for the program.

The routines made available through the above `include` statement are in the standard C run-time library, so no other special actions are needed when compiling and linking.

All names in the include file which are used internally to the library, start with the underline character (_) to reduce the probability of conflict with user-defined names.

Names which are intended to be visible to user programs outside the package are as follows:

Table 5-1 *Standard I/O Library Names Accessible to User Programs*

Name	*Description*
stdin()	The name of the **standard input** file. This file is automatically connected at program startup time, and is the place from which a program reads its input.
stdout()	The name of the **standard output** file. This file is automatically connected at program startup time, and is the place to which a program writes its output.
stderr()	The name of the **standard error** file. This file is automatically connected at program startup time, and is the place to which a program writes any error or diagnostic responses which should not clutter up the standard output.
EOF	is actually the value –1. EOF is returned by the read routines upon encountering end-of-file, or error conditions.
NULL	is a notation for the null pointer. Functions whose values are pointers return NULL to indicate an error.
FILE	is an abbreviation for the declaration: `struct _iob` and is a useful notation when declaring a pointer to a stream.

Table 5-1 *Standard I/O Library Names Accessible to User Programs—Continued*

Name	*Description*
BUFSIZ	is a number of the size suitable for a user-supplied input-output buffer. BUFSIZ is usually 1024. See the `setbuf()` function described below.

`getc()`, `getchar()`, `putc()`, `putchar()`, `feof()`, `ferror()`, and `fileno()` are all defined as macros. Their descriptions appear later in this chapter. They are mentioned here to indicate that they cannot be redeclared. In addition, because they are macros and not functions, they cannot be passed as arguments to other functions, nor can their addresses be taken.

The 'Standard I/O Library' is a collection of routines intended to provide efficient and portable I/O services for most C programs. The standard I/O library is available on each system that supports C, so programs that confine their system interactions to its facilities can be transported from one system to another essentially without change.

This chapter describes the basics of the standard I/O library. Following chapters contain a fuller description of the capabilities and calling conventions of the functions in it.

You could do I/O by calling the system routines directly. However, there is a 'standard I/O package' that provides a high-level I/O access mechanism. This and the following chapters discuss the functions available in the standard I/O package. (An appendix discusses the raw interface to the operating system.) In general, you can get by using the standard I/O package and never need to use the raw system calls.

The standard I/O package provides access to files in the system through a collection of *file descriptors* that refer to structures for managing I/O buffering. The first part of the discussion in this chapter describes those file descriptors that are defined automatically. Later sections describe how to get your own descriptors connected to files in the system.

5.3. The 'Standard Input' and 'Standard Output'

When a SunOS program starts up, three files are connected automatically. These files are called the *standard input* (`stdin()`), the *standard output* (`stdout()`), and the *standard error* (`stderr()`).

The very simplest standard I/O call for output is to use `putchar(c)` to put the character *c* on the standard output, which is normally the user's screen.

If the user redirected the standard output by using the > syntax on the command line, the standard output is *redirected.* For example, if you typed:

```
tutorial% prog > outputfile
```

on the command line, the standard output from `prog` is written to *outputfile* and the program is unaware that the standard output is going to a file instead of the

keyboard. *outputfile* is created if it doesn't exist; if it already exists, its previous contents are overwritten.

Similarly, you can send the standard output from a program through a *pipe* with the command line:

```
tutorial% prog | otherprog
```

and the standard output of `prog` goes into the standard input of `otherprog`.

Reading Standard Input and Writing Standard Output

The simplest input mechanism is to read from the 'standard input,' which is generally the user's keyboard. The function `getchar()` returns the next input character each time it is called. A file may be substituted for the keyboard by using the < convention (input redirection): if `prog` uses `getchar()`, the command line

```
tutorial% prog < filename
```

makes `prog` read from the file specified by *filename*, instead of from the keyboard. `prog` itself need know nothing about where its input is coming from. This is also true if the input comes from another program through the *pipe* mechanism:

```
tutorial% otherprog | prog
```

provides the standard input for `prog` from the standard output (see above) of `otherprog`.

`getchar()` returns the value `EOF` when it encounters the end of file (or an error) on whatever you are reading. The value of `EOF` is normally defined to be `-1`, but it is unwise to take any advantage of that knowledge. As will become clear shortly, this value is automatically defined for you when you compile a program, and need not be of any concern.

The function `printf()`, which formats output in various ways, uses the same mechanism as `putchar()` does, so calls to `printf()` and `putchar()` may be intermixed in any order; the output appears in the order of the calls.

Similarly, the function `scanf()` provides for formatted input conversion. `scanf()` reads the standard input and breaks it up into strings, numbers, etc., as desired. `scanf()` uses the same mechanism as `getchar()`, so calls to them may also be intermixed.

Many programs read only one input and write one output; for such programs I/O with `getchar()`, `putchar()`, `scanf()`, and `printf()` may be entirely adequate, and it is almost always enough to get started. This is particularly true if the SunOS pipe facility is used to connect the output of one program to the

of another. For example, the following program strips out all ASCII control characters from its input (except for `newline` and `tab`).

```
#include <stdio.h>

main()  /* ccstrip: strip non-graphic characters */
{
    int c;
    while ((c = getchar()) != EOF)
        if ((c >= ' ' && c < 0177) || c == '\t' || c == '\n')
            putchar(c);
    exit(0);
}
```

The line

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

should appear at the beginning of each source file which does I/O using the standard I/O functions described in section 3(S) of the *System Interface Manual* — the C compiler reads a file (*/usr/include/stdio.h*) of standard routines and symbols that includes the definition of `EOF`.

If it is necessary to treat multiple files, you can use `cat` to collect the files for you:

```
tutorial% cat file1 file2 ... | ccstrip > output
```

and thus avoid learning how to access files from a program. By the way, the call to `exit()` at the end is not necessary to make the program work properly, but it assures that any caller of the program will see a normal termination status (conventionally 0) from the program when it completes. Section 3.3 discusses returning status in more detail.

5.4. Error Handling — `stderr()` and `exit()`

`stderr()` is assigned to a program in the same way that `stdin()` and `stdout()` are. Output written on `stderr()` appears on the user's terminal even if the standard output is redirected, unless the standard error is also redirected. For example, the command *wc* writes its diagnostics on `stderr()` instead of `stdout()` so that if one of the files can't be accessed for some reason, the message finds its way to the user's terminal instead of disappearing down a pipeline or into an output file.

The argument of `exit()` is made available to whatever process called the process that is exiting (see Section 3.3), so the success or failure of the program can be tested by another program that uses this one as a subprocess. By convention, a return value of 0 indicates that all is well; nonzero values indicate abnormal situations.

`exit()` itself calls `fclose()` for each open output file, to flush out any buffered output, then calls a routine named `_exit()`. The function `_exit()` terminates the program immediately without any buffer flushing; it may be called directly if desired.

5.5. Miscellaneous I/O Functions

The standard I/O library provides several other I/O functions besides those illustrated above.

Normally, output with `putc()` and such is buffered — use `fflush(fp)` to force it out immediately.

`fscanf()` is identical to `scanf()`, except that its first argument is a file pointer (as with `fprintf()`) that specifies the file from which the input comes; it returns `EOF` at end of file.

The functions `sscanf()` and `sprintf()` are identical to `fscanf()` and `fprintf()`, except that the first argument names a character string instead of a file pointer. The conversion is done from the string for `sscanf()` and into it for `sprintf()`, and no input or output is done.

`fgets(buf,size,fp)` copies the next line from stream `fp`, up to and including a newline, into `buf`; at most `size-1` characters are copied; it returns `NULL` at end of file. `fputs(buf,fp)` writes the string in `buf` onto file `fp`.

The function `ungetc(c,fp)` 'pushes back' the character `c` onto the input stream `fp`; a subsequent call to `getc()`, `fscanf()`, and so on will encounter `c`. Only one character of pushback is guaranteed to work.

6

Accessing Files Through Standard I/O

6

Accessing Files Through Standard I/O

The above programs have all read the standard input and written the standard output, which we have assumed are magically predefined. The next step is to write a program that accesses a file that is *not* already connected to the program. One simple example is *wc*, which counts the lines, words and characters in a set of files. For instance, the command

```
tutorial% wc x.c y.c
```

displays the number of lines, words and characters in `x.c` and `y.c` and the totals.

The question is how to arrange for the named files to be read — that is, how to connect the filenames to the I/O statements which actually read the data.

The rules are simple — you have to *open* a file by the standard library function `fopen()` before it can be read from or written to. `fopen()` takes an external name (like `x.c` or `y.c`), does some housekeeping and negotiation with the operating system, and returns an internal name which must be used in subsequent reads or writes of the file.

This internal name is actually a pointer, called a *file pointer*, to a structure which contains information about the file, such as the location of a buffer, the current character position in the buffer, whether the file is being read or written, and the like. Users don't need to know the details, because part of the standard I/O definitions obtained by including `stdio.h` is a structure definition called `FILE`. The only declaration needed for a file pointer is exemplified by

```
FILE    *fp, *fopen();
```

This says that `fp` is a pointer to a `FILE`, and `fopen()` returns a pointer to a `FILE`. `FILE` is a type name, like `int`, not a structure tag.

The actual call to `fopen()` in a program has the form:

```
fp = fopen(name, mode);
```

The next thing needed is a way to read or write the file once it is open. There are several possibilities, of which `getc()` and `putc()` are the simplest. `getc()` returns the next character from a file; it needs the file pointer to tell it what file. Thus

```
c = getc(fp)
```

places in `c` the next character from the file referred to by `fp`; it returns `EOF` when it reaches end of file. `putc()` is the inverse of `getc()`:

```
putc(c, fp)
```

puts the character `c` on the file `fp` and returns `c` as its value. `getc()` and `putc()` return `EOF` on error.

When a program is started, three streams are opened automatically, and file pointers are provided for them. These streams are the standard input, the standard output, and the standard error output; the corresponding file pointers are called `stdin()`, `stdout()`, and `stderr()`. Normally these are all connected to the terminal, but may be redirected to files or pipes as described in Section 5.3. `stdin()`, `stdout()` and `stderr()` are predefined in the I/O library as the standard input, output and error files; they may be used anywhere an object of type `FILE *` can be. They are constants, however, *not* variables, so don't try to assign to them.

With some of the preliminaries out of the way, we can now write `wc`. The basic design is one that has been found convenient for many programs: if there are command-line arguments, they are processed in order. If there are no arguments, the standard input is processed. This way the program can be used standalone or as part of a larger activity.

```
#include <stdio.h>

main(argc, argv)    /* wc: count lines, words, chars */
int argc;
char *argv[ ];
{
    int c, i, inword;
    FILE *fp, *fopen();
    long linect, wordct, charct;
    long tlinect = 0, twordct = 0, tcharct = 0;

    i = 1;
    fp = stdin;
    do {
        if (argc > 1 && (fp=fopen(argv[i], "r")) == NULL) {
            fprintf(stderr, "wc: can't open %s\n", argv[i]);
            continue;
        }
        linect = wordct = charct = inword = 0;
        while ((c = getc(fp)) != EOF) {
            charct++;
            if (c == '\n')
                linect++;
            if (c == ' ' || c == '\t' || c == '\n')
                inword = 0;
            else if (inword == 0) {
                inword = 1;
                wordct++;
            }
        }
        printf("%7ld %7ld %7ld", linect, wordct, charct);
        printf(argc > 1 ? " %s\n" : "\n", argv[i]);
        fclose(fp);
        tlinect += linect;
        twordct += wordct;
        tcharct += charct;
    } while (++i < argc);
    if (argc > 2)
        printf("%7ld %7ld %7ld total\n", tlinect, twordct, tcharct);
    exit(0);
}
```

The function `fprintf()` is identical to `printf()`, save that the first argument is a file pointer that specifies the file to be written.

The function `fclose()` is the inverse of `fopen()`; it breaks the connection between the file pointer and the external name that was established by `fopen()`, freeing the file pointer for another file. There is a limit on the number of files that a program may have open simultaneously, so you should free things when they are no longer needed. There is another reason to call `fclose()` on an output file — it flushes the buffer in which `putc()` collects output. `fclose()` is called automatically for each open file when a program terminates normally.

6.1. Accessing Files

Several `stdio` routines, needed to perform file I/O housekeeping and access functions are described below:

`fopen()` — Open a File

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

opens the file and, if needed, allocates a buffer for it. `filename` is a character string specifying the name. `type` is a character string (not a single character) indicating the access mode. It may be `"r"`, `"w"`, or `"a"` to indicate intent to read, write, or append. In addition, each mode may be followed by a + sign to open the file for reading and writing. `r+` positions the stream at the beginning of the file, `w+` creates or truncates the file, and `a` positions the stream to the end of the file. Both reads and writes may be used on read/write streams, with the limitation that an `fseek`, `rewind()`, or reading end-of-file must be used between a read and a write or vice versa. The value returned is a file pointer. If it is `NULL` the attempt to open the file failed.

Figure 6-1 *Example of Using* `fopen()`

```
demo ()
{

    FILE  *fp;

                    /*  open the file  */
    if ((fp = fopen ("/usr/lib/tmac.tmac.e", 'r')) == NULL)
        printf ("Can't open /usr/lib/tmac/tmac.e\n");
    else
        . . . go ahead and work with the file

}     /*  end of the demo function  */
```

The first argument of `fopen()` is the name of the file, as a character string. The second argument is the mode, also as a character string, which indicates how you intend to use the file. The allowable modes are read (`r`), write (`w`), or append (`a`). In addition, each *mode* may be followed by a + sign to open the file for reading and writing. `"r+"` positions the stream at the beginning of the file, `"w+"` creates or truncates the file, and `"a+"` positions the stream to the end of the file. Both reads and writes may be used on read/write streams, with the limitation that an `fseek`, `rewind`, or reading end-of-file must be used between a read and a write or vice versa.

If a file that you open for writing or appending does not exist, it is created (if possible). Opening an existing file for writing discards the old contents. Trying to read a file that does not exist is an error, and there may be other causes of error as well (like trying to read a file without read permission). If there is an error, `fopen()` returns the null pointer value `NULL` — defined as zero in `stdio.h`.

`freopen()` — Reopen a File

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

The stream named by `ioptr` is closed, if necessary, and then reopened as if by `fopen()`. If the attempt to open fails, `NULL` is returned; otherwise `ioptr` is returned, which now refers to the new file. Often the reopened stream is `stdin()` or `stdout()`. The `filename` and `type` parameters are as for `fopen()`.

filename is a character string that specifies the name of the file.

type is a character string (not a single character) that specifies the access mode of the file. `type` can be one of:

- *r* reopen the file for reading,
- *w* reopen the file for writing,
- *a* reopen the file for appending.

ioptr is a pointer to the existing stream which is to be closed.

The value of the `freopen()` function is a file pointer. If the value of the file pointer is `NULL`, the attempt to open the file failed.

Figure 6-2 *Example of Using* `freopen()`

```
demo ()
{

    FILE  *fp;

                    /*  re-open the file  */
    if ((fp = freopen ("/lib/ftncterrs", 'r', fp)) == NULL)
        printf ("Can't open /lib/ftncterrs\n");
    else
        . . . go ahead and work with the file

}     /*  end of the demo function  */
```

`fflush()` — Flush Stream Buffer

The `fflush()` function flushes the stream buffer for a given file. The interface to `fflush()` is:

```
fflush(ioptr)
    FILE  *ioptr;
```

Any buffered information on the output stream designated by `ioptr` is written out to the file.

Output files are normally buffered if they are not directed to a screen. The `stderr()` file usually starts off unbuffered, and remains unbuffered unless the `setbuf()` function is used, or unless the file is reopened.

`fclose()` — Close A File

The `fclose()` function closes an open file. The interface definition is:

```
fclose(ioptr)
    FILE  *ioptr;
```

The file designated by `ioptr` is closed, after any buffers associated with that file have been written out.

Any buffers allocated to the file are freed.

When a C program terminates normally (in a controlled fashion), `fclose()` requests are issued automatically.

`setbuf()` — Set Buffer for File I/O

The `setbuf()` function sets up a buffer for an open file. The user can designate a buffer different from the one which the run-time library chooses, or the user can select no buffer at all. The interface to `setbuf()` is:

```
setbuf(ioptr, buf)
    FILE  *ioptr;
    char  *buf;
```

The `setbuf()` function is used after a file is opened, but before any I/O transfers have been made to that file.

If the `buf` parameter is `NULL`, the stream is unbuffered. Otherwise, the buffer supplied is used. The buffer `buf` must be a sufficiently large character array. The usual way to assure this is to declare the buffer:

```
char  buf[BUFSIZE];
```

Here's an example of `setbuf()` usage:

Figure 6-3 *Example of Using* `setbuf()`

```
demo ()
{

    FILE  *fp;

                          /*  open the file  */
    if ((fp = fopen ("/lib/pascterrs", 'r')) == NULL)
        printf ("Can't open /lib/pascterrs\n");
    else
                        /*  make the file unbuffered  */
        setbuf (fp, NULL);

}     /*  end of the demo function  */
```

`fileno()` — Obtain File Descriptor

The `fileno()` function returns an integer value which is the file descriptor associated with the file.

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

Here's an example of `fileno()` usage:

Figure 6-4 *Example of Using* `fileno()`

```
demo ()
{

    FILE  *fp;
    int  file_num;

                          /*  open a file  */
    if ((fp = fopen ("/etc/passwd", 'r')) == NULL)
        printf ("Can't open /etc/passwd\n");
    else
                          /*  get the file number  */
        file_num = fileno (fp);

}     /*  end of the demo function  */
```

`rewind()` — Rewind a Stream

The `rewind()` function rewinds the stream designated by the `ioptr` parameter.

```
rewind(ioptr)
    FILE  *ioptr;
```

`rewind()` is not useful for an output file, since it is still open for writing after the rewind has been performed. If a file needs to be rewound for reading, use the `freopen()` function (described above).

7

Character I/O

7

Character I/O

This section describes those macros and functions which are concerned with reading and writing characters from and to streams.

`getc()` Macro — Get a Character from a File

The `getc()` macro gets a character from a file. The definition is:

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

The `getc()` macro obtains the next character from the stream designated by `ioptr`. `ioptr` is a file descriptor such as is returned by the `fopen()` function, or is a name such as `stdin()`.

When the end of file is reached, the integer `EOF` is returned. The character is a valid character from `getc()`.

Note that `getc()` is a macro, not a function.

Figure 7-1 *Example of Using* `getc()`"

```
main (argc, argv)

int
 argc;

char
 *argv [];
{
    FILE  *fd;
    int  num_char = 0;

    if ((fd = fopen (argv [1], 'r')) == NULL)
        prinf("Can't open %s\n", argv [1]);
    else
                        /*  count characters in a file  */
        while (getc(fd) != EOF)
            num_char++;

}     /*  end of the count function  */
```

`fgetc()` Function — Get Character from File

The `fgetc()` function obtains a single character from a file. The interface definition is:

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

`fgetc()` obtains the next character from the stream designated by `ioptr`. `ioptr` is a file descriptor such as is returned by the `fopen()` function, or is a name such as `stdin()`.

When the end of file is reached, the integer `EOF` is returned. The character `\0` is a valid character from `fgetc()`.

`fgetc()` is a genuine function, as opposed to the `getc()` macro. This means that `fgetc()` can be pointed to, passed as an argument to another function, and so on.

Figure 7-2 *Example of Using* `fgetc()`

```
main (argc, argv)
    int  argc;
    char  *argv [];
{
    FILE  *fd;
    char  ch;
    int  num_line = 0;

    if ((fd = fopen (argv [1], 'r')) == NULL)
        prinf("Can't open %s\n", argv [1]);
    else
                        /*  count lines in a file  */
        while ((ch = fgetc(fd)) != EOF)
            if (ch == '\n')
                num_line++;

}     /*  end of the count function  */
```

Remember that `getc()` normally buffers its input; terminal I/O will not be properly synchronized unless this buffering is defeated. For input, see `setbuf` in Section 5.1.

`getchar()` Macro — Get a Character from Standard Input

The `getchar()` macro obtains a single character from the standard input. The interface to `getchar()` is:

```
int getchar()
```

The `getchar()` macro is a shorthand notation for

```
getc(stdin)
```

Note that `getchar()` is a macro, not a function.

Figure 7-3 *Example of Using* `getchar()`

```
main ()
{
    int  ch;
    int  num_nums = 0;

                          /*  count digits in a file  */
    while ((ch = getchar()) != EOF)
        if (ch >= '0' && ch <= '9')
            num_nums++;

}     /*  end of the count function  */
```

`fgets()` — Read a String from a File

The `fgets()` function reads a string from a specified file. The interface definition is:

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

The `fgets()` function reads up to *n-1* characters from the stream designated by `ioptr` into the character array pointed to by `s`. The read terminates when a newline character is read. The newline character is placed in the buffer. The last character read is always followed by a null character in the character array.

The `fgets()` function returns its first argument, or `NULL` if an error or an end of file was encountered.

Figure 7-4 *Example of Using* `fgets()`

```
main (argc, argv)
    int  argc;
    char  *argv [];
{
    FILE  *fd;
    char  line [256];
    int  num_line = 0;

    if ((fd = fopen (argv [1], 'r')) == NULL)
        prinf("Can't open %s\n", argv [1]);
    else
                            /*  count lines in a file  */
        while ((fgets(line, 255, fd)) != NULL)
            num_line++;

}     /*  end of the count function  */
```

`ungetc()` — Push a Character Back on a Stream

The `ungetc()` function pushes a single character back onto a stream. The interface definition is:

```
ungetc(c, ioptr)
    char  c;
    FILE  *ioptr;
```

The `ungetc()` function pushes the character argument `c`, back onto the input stream designated by `ioptr`.

Only one character may be pushed back between two reads.

Figure 7-5 *Example of Using* `ungetc()`

```
main ()
{
    int  ch;
    char  digits [256];
    int  idx = 0;

      /*  read number from standard input  */

    while ((ch = getchar()) != EOF)
        if (ch >= '0' && ch <= '9')
            digits [idx++] = ch;
        else  {
            ungetc (ch, stdin);
            break;
        }

}     /*  end of the read number function  */
```

`putc()` Macro — Put a Character to a File

The `putc()` macro puts a single character to a specified file. The interface definition is:

```
putc(c, ioptr)
    char  c;
    FILE  *ioptr;
```

The `putc()` macro writes the character `c` onto the output stream designated by `ioptr`, where `ioptr` is a file descriptor such as is returned by the `fopen()` function, or is a name such as `stdout()` or `stderr()`.

The character `c` is normally returned as a value from the macro, but if an error occurs during the transfer, the value `EOF` is returned.

Note that `putc()` is a macro, not a function.

```
main ()
{
    char  ch;

                          /*  copy stdin to stdout */
    while ((ch = getchar()) != EOF)
        putc(ch, stdout);

}     /*  end of the copy function  */
```

Remember that `putc()` normally buffers its output; terminal I/O will not be properly synchronized unless this buffering is defeated. For output, use `fflush`.

`fputc()` Function — Put a Character to a File

The `fputc()` function outputs a single character to a specified file. The interface definition is:

```
fputc(c, ioptr)
    char  c;
    FILE  *ioptr;
```

The `fputc()` function writes the character `c` onto the stream designated by `ioptr`, where `ioptr` is a file descriptor such as is returned by the `fopen()` function, or is a name such as `stdout()` or `stderr()`.

The character `c` is normally returned as a value from the function, but if an error occurs during the transfer, the value `EOF` is returned.

`fputc()` is a genuine function, as opposed to the `putc()` macro. This means that `fputc()` can be pointed to, passed as an argument to another function, and so on.

Figure 7-6 *Example of Using* `fputc()`

```
main ()
{
    char  ch;

                          /*  copy stdin to stdout */
    while ((ch = fgetc(stdin)) != EOF)
        fputc(ch, stdout);

}     /*  end of the copy function  */
```

`putchar()` Macro — Put a Character to Standard Output

The `putchar()` macro puts a single character to the standard output file. The interface definition is:

```
putchar(ch)
    char  ch;
```

The `putchar()` macro is a shorthand notation for

```
putc(stdout)
```

Note that `putchar()` is a macro, not a function.

Figure 7-7 *Example of Using* `putchar()`

```
main ()
{
    char  ch;

                          /*  copy stdin to stdout */
    while ((ch = getchar()) != EOF)
        putchar(ch);

}     /*  end of the copy function  */
```

`fputs()` — Put a String to a File

`fputs()` writes a character string to a file. The interface definition is:

```
fputs(s, ioptr)
    char  *s;
    FILE  *ioptr;
```

The `fputs()` function writes the null-terminated character string `s` (which is a character array) to the stream designated by `ioptr`.

`fputs()` does not append a newline to the string.

`fputs()` does not return a value.

Figure 7-8 *Example of Using* `fputs()`

```
main ()
{
    char  line [256];

              /*  copy lines from stdin to stdout */
    while ((fgets(line, 255, stdin)) != NULL)
        fputs (line, stdout);

}     /*  end of the copy function  */
```

`feof()` — Test for End Of File

The `feof()` function checks for an end of file on a specified file. The interface definition is:

```
feof(ioptr)
    FILE  *ioptr;
```

The `feof()` function returns a nonzero value if an end-of-file has occurred on the stream designated by `ioptr`.

7.1. Formatted Input and Output

The C run-time library provides extensive facilities for formatted conversions of character strings to numeric data, and for the formatted conversion of numeric data to character strings. Conversions can be done between the standard input or standard output, an arbitrary file, or strings in memory. The subsections to follow give detailed descriptions of these facilities.

Formatted Output Conversions

There are three variations of the formatted output functions: They are all similar in their actions, the only difference being the destination of the formatted string.

```
prinf(format, arg1, . . .)
    char  *format;
```

`prinf()` writes the formatted string to the standard output.

```
fprinf(ioptr, format, arg1, . . .)
    FILE  *ioptr;
    char  *format;
```

`fprinf() writes the formatted string to the file designated by ioptr .`

```
sprinf(s, format, arg1, . . .)
    char  *s;
    char  *format;
```

`sprinf()` stores the formatted string into a character string (character array) in memory.

Formatted Input Conversions

The `scanf()`, `fscanf()`, and `sscanf()` functions are the equivalents of the `prinf()` functions described above, except that the `scanf()` functions perform conversions from character strings to data in the computer memory. They are thus used for reading formatted information instead of writing it.

There are three variations of the `scanf()` function:

```
scanf(format, arg1, . . .)
    char  *format;
```

`scanf()` reads the formatted string from the standard input.

```
fscanf(ioptr, format, arg1, . . .)
    FILE  *ioptr;
    char  *format;
```

`fscanf()` reads the formatted string from the file designated by `ioptr`.

```
sscanf(s, format, arg1, . . .)
    char  *s;
    char  *format;
```

`sscanf()` gets the formatted string from a character string (character array) in memory.

The Format Control Templates

All six `print` and `scan` functions accept a `format` argument, followed by zero or more arg_n arguments.

The `format` argument is a template, in the form of a character string. The `format` character string consists of two kinds of objects:

- It can contain fixed parts which are sent to the destination unchanged (for formatted output) or match characters in the input source (for formatted input).
- It can also contain conversion specifications, which indicate how the following arg_n are to be converted and placed into the final formatted output string, or recognized in the input, and converted to internal form and placed in the arg_n.

Conversion Specifications

A *conversion specification* is marked by a percent sign `%`, and ends with a conversion character. In between the `%` sign and the conversion character, there can be modifiers. These modifiers are described after the descriptions of the conversion characters. Any character in a format that is not part of a conversion specification is passed or recognized as is.

Here is a `prinf()` call with a simple string template and no conversion specifications:

```
prinf("Calling occupants of interactive space\n");
```

This example simply prints the quoted string on the standard output.

The following paragraphs describe the effects of the conversion characters. There are also modifiers for the conversion specifications, and these are described later.

d — Decimal Conversion

A conversion character of d specifies that the associated argument is converted to (or from) decimal notation.

Figure 7-9 *Example of* d *Format Specification*

```
main ()
{
    int  data = -25;

    prinf("The value of data is: %d\n", data);

}     /*  End of the program  */
```

When the above program is run, it generates the result:

```
The value of data is: -25
```

o — Octal Conversion

A conversion character of o specifies that the associated argument is converted to (or from) unsigned octal notation. The resulting output string does not contain a leading zero. It is the responsibility of the programmer to insert the leading zero "manually" as part of the format string, if that is what is required.

Figure 7-10 *Example of* o *Format Specification*

```
main ()
{
    int  data = 25;

    prinf("The value of data is: 0%o\n", data);

}     /*  End of the program  */
```

When the above program is run, it generates the result:

```
The value of data is: 031
```

Note that the program explicitly places the digit "0" in the generated number.

x — Hexadecimal Conversion

A conversion character of x specifies that the associated argument is converted to (or from) unsigned hexadecimal notation. The resulting output string does not contain a leading "0x". It is the responsibility of the programmer to insert the leading "0x" "manually", as part of the format string, if that is what is required.

Figure 7-11 *Example of* `x` *Format Specification*

```
main ()
{
    int  data = 25;

    prinf("The value of data is: 0x%x\n", data);

}     /*  End of the demo function  */
```

When the above program is run, it generates the result:

```
The value of data is: 0x19
```

Note that the programmer explicitly coded the "0x" in front of the generated number.

`h` — Short Conversion on Input Only

A conversion character of `h` is used only for formatted input, and specifies that the associated argument is a pointer to a `short int` data item.

`u` — Unsigned Decimal Conversion

A conversion character of `u` specifies that the associated argument is converted to (or from) unsigned decimal notation.

Figure 7-12 *Example of* `u` *Format Specification*

```
main ()
{
    int  data = -25;

    prinf("The value of data is: %u\n", data);

}     /*  End of the demo function  */
```

When the above program is run, it generates the result:

```
The value of data is: 4294967271
```

`c` — Character Conversion

A conversion character of `c` specifies that the associated argument is to be converted to (or from) a single character.

Figure 7-13 *Example of* `c` *Format Specification*

```
main ()
{
    static char  data [10] = "Hi there!";

    prinf("Parts of data are: %c %c %c\n",
                        data[0], data[8], data[4]);

}     /*  End of the demo function  */
```

When the above program is run, it generates the result:

```
Parts of data are: H ! h
```

`s` — String Conversion

A conversion character of `s` specifies that the associated argument is a string. Characters from the string are printed until a null character is found, or until the number of characters indicated by the precision specification (see below) are used up.

Figure 7-14 *Example of* `s` Format Specification"

```
main ()
{
    static char  data [] = "Hello, World!";

    prinf("The value of data is: '%s'\n", data);

}     /*  End of the demo function  */
```

When the above program is run, it generates the result:

```
The value of data is: 'Hello, World!'
```

`e` — Exponential Floating Conversion

A conversion character of `e` specifies that the associated argument is assumed to be a `float` or a `double`. It is converted to (or from) a decimal exponential notation of the form

```
[-]m.nnnnnnnE[±]xx
```

where the length of the string of *n*'s is specified by the precision. The default precision is six decimal places.

Figure 7-15 *Example of* `e` *Format Specification*

```
main ()
{
    float  data = 123.456;

    prinf("The value of data is: %e\n", data);

}     /*  End of the demo function  */
```

When the above program is run, it generates the result:

```
The value of data is: 1.234560e+02
```

`f` — Fractional Floating Conversion

A conversion character of `f` specifies that the associated argument is assumed to be a `float` or a `double`. It is converted to (or from) a notation`f`loatingdecimal

```
[-]mmm.nnnnnn
```

where the length of the string of `n`'s is specified by the precision. The default precision is six decimal places. The precision does not determine the number of digits printed in `f` format, but the number of decimal places displayed.

Figure 7-16 *Example of* `f` *Format Specification*

```
main ()
{
    float  data = 123.456;

    prinf("The value of data is: %f\n", data);

}     /*  End of the demo function  */
```

When the above program is run, it generates the result:

```
The value of data is: 123.456001
```

g — Adaptable Floating Conversion

A conversion character of g specifies that the associated argument is converted to (or from) either e or f format, depending upon which is the shorter. Non-significant zeros are not printed in g format. This is similar to FORTRAN's G format conversion.

Figure 7-17 *Example of g Format Specification*

```
main ()
{
    float
 data = 123.456;

    prinf("The value of data is: %g\n", data);

}     /*  End of the demo function  */
```

When the above program is run, it generates the result:

```
The value of data is: 123.456
```

Literal Character Output

If the character which follows the % sign is not a conversion character, that character is printed verbatim. Thus, to print a % sign, use a format conversion of %%.

Figure 7-18 *Example of Literal Character Output*

```
main ()
{
    int
 data = 25;

    prinf("The value of data is: %y %%\n", data);

}     /*  End of the demo function  */
```

When the above program is run, it generates the result:

```
The value of data is: y %
```

The two percent signs are displayed as one, and the unknown conversion character (y) is output verbatim. The value of the data variable in the output list is simply ignored, since no conversion specification in the format required data.

Optional Format Modifiers

Between the % sign and the format conversion letters as defined above, there may be some optional information. The characters which may appear in these positions are described below.

Left Justify Field

A minus sign (-) appearing before the conversion character specifies that the argument is to be left-justified in the output field. The minus sign is optional.

After the minus sign can appear width and precision specifications, as described next.

Minimum Field Width and Precision Specifications

The form of the optional field width and precision specifications are:

- a digit string, which specifies a minimum field width. The converted number is printed in a field at least this wide, and wider if required. If the converted argument has fewer characters than the field width, it is padded on the left (or on the right, if a minus sign was given) with enough padding characters to make up the specified field width. The padding character is normally a space. If the field width is specified with a leading zero, it does not mean an octal field width, rather it means that the output field is to be padded with zeros.
- a period character, which separates the field width from the next digit string.
- a digit string, which is the precision. The precision means one of two things. In the case of a `float` or a `double` argument, the precision is the number of digits to be printed to the right of the decimal point. In the case of a string argument, the precision is the number of characters to be printed from the string.

The examples below show the way that the justification, width, and precision specifications apply to string values when they are output. The value to be printed is the string "Wizard", which is six characters long. It is printed in a variety of format specifications, and there are vertical bands at either end of the field to show the extent of the field.

Figure 7-19 *Example of Field Width Specifications*

```
main ()
{
    static char data [] = "Wizard";

    prinf("data in %%4s format is: |%4s:|\n", data);
    prinf("data in %%-4s format is: |%-4s:|\n", data);
    prinf("data in %%10s format is: |%10s:|\n", data);
    prinf("data in %%-10s format is: |%-10s:|\n", data);
    prinf("data in %%10.4s format is: |%10.4s:|\n", data);
    prinf("data in %%-10.4s format is: |%-10.4s:|\n", data);
    prinf("data in %%.4s format is: |%.4s:|\n", data);

}     /*  End of the demo function  */
```

When the above program is run, it generates the results:

```
data in %4s format is: |Wizard|
data in %-4s format is: |Wizard|
data in %10s format is: |    Wizard|
data in %-10s format is: |Wizard    |
data in %10.4s format is: |      Wiza|
data in %-10.4s format is: |Wiza      |
data in %.4s format is: |Wiza|
```

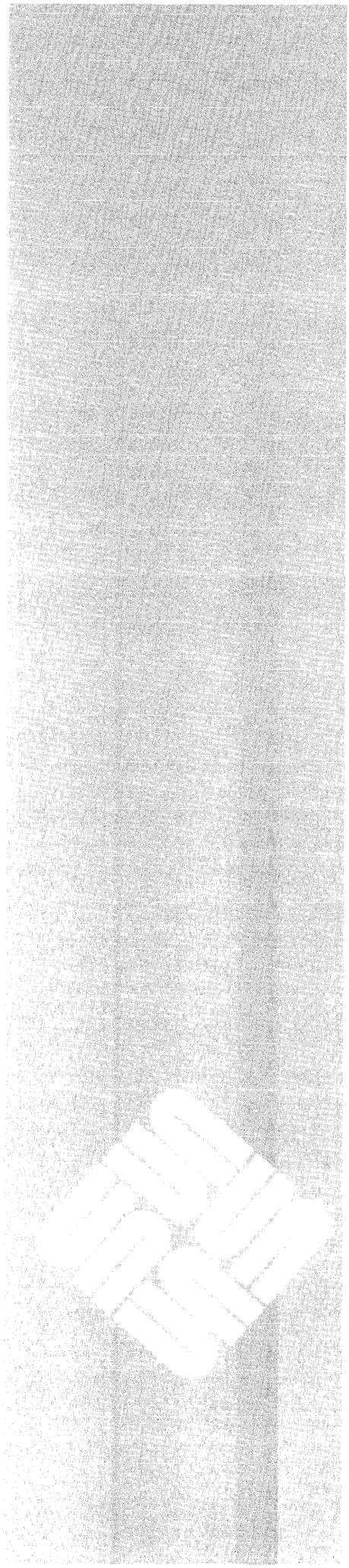
Length Modifier

If the conversion specification is preceded by a `lx`, it means that the associated argument is a `long` and `lf` indicates a `double`. If no length modifier precedes the conversion specification, the associated argument is assumed to be an `int`. instead of an `int`. A lone `l` preceding the conversion specification is ignored in Sun C because `int`s and `long`s are the same.

On `scanf()`, arguments are pointers. Sizes in `%` specifiers must be correct: `%f` for floats and `%lf` for doubles.

8

String-Handling Functions



8

String-Handling Functions

The C programming language has no language-defined facilities for manipulating character string data. The C library does, however, provide a fairly rich set of primitives for manipulating character string data.

This chapter contains three major areas relating to string handling:

- Macros for classifying characters (is a character, uppercase, letter, digit, and such), plus macros for doing some minimal conversions (convert uppercase to lowercase).
- Functions for handling null-terminated strings.
- Functions for handling bit strings and byte strings.

8.1. Character Classification

These macros classify ASCII-coded integer values. Each is a predicate returning nonzero for true, zero for false. `isascii()` is defined on all integer values; the rest are defined only where `isascii(c)` is true and on the single non-ASCII value EOF(see *stdio*(3S)).

You should have the line:

```
#include <ctype.h>
```

in any program unit that uses these macros.

`isalpha()` — Is Character Alphabetic

`isalpha(c)` *c* is a letter — `a` thru `z` or `A` thru `Z`.

`isupper()` — Is Character Uppercase Letter

`isupper(c)` *c* is an upper case letter — `A` thru `Z`.

`islower()` — Is Character Lowercase Letter

`islower(c)` *c* is a lower case letter — `a` thru `z`.

`isdigit()` — Is Character Decimal Digit

`isdigit(c)` *c* is a digit — `0` thru `9`.

`isxdigit()` — **Is Character Hexadecimal Digit**

`isxdigit(c)` *c* is a hexadecimal digit — `0` thru `9`, `a` thru `f`, or `A` thru `F`.

`isalnum()` — **Is Character Letter or Digit**

`isalnum(c)` *c* is an alphanumeric character, that is, *c* is a letter or a digit.

`isspace()` — **Is Character Whitespace**

`isspace(c)` *c* is a space, tab, carriage return, newline, or formfeed.

`ispunct()` — **Is Character Punctuation**

`ispunct(c)` *c* is a punctuation character (neither control nor alphanumeric)

`isprint()` — **Is Character Printable**

`isprint(c)` *c* is a printing character, such as ASCII characters 0x20 (space) through 0x7E (tilde).

`iscntrl()` — **Is Character Control Character**

`iscntrl(c)` *c* is a delete character (0x7F) or an ordinary control character (less than 0x20).

`isascii()` — **Is Character an ASCII Character**

`isascii(c)` *c* is an ASCII character less than 0x80.

`isgraph()` — **Is Character a Visible Graphic**

`isgraph(c)` *c* is a visible graphic character, and ASCII character code from 0x21 (exclamation mark) through 0x7E (tilde).

8.2. Character Conversion Macros

These macros perform simple conversions on single characters.

`toupper()` — **Convert Lowercase to Uppercase**

`toupper(c)` converts *c* to its upper-case equivalent. Note that this *only* works if *c* is known to be a lower-case character to start with (presumably checked by `islower()`).

`tolower()` — **Convert Uppercase to Lowercase**

`tolower(c)` converts *c* to its lower-case equivalent. Note that this *only* works if *c* is known to be an uppercase character to start with (presumably checked by `isupper`).

`toascii()` — **Ensure Character is ASCII**

`toascii(c)` masks *c* with the correct value so that *c* is guaranteed to be an ASCII character in the range 0 thru 0x7F.

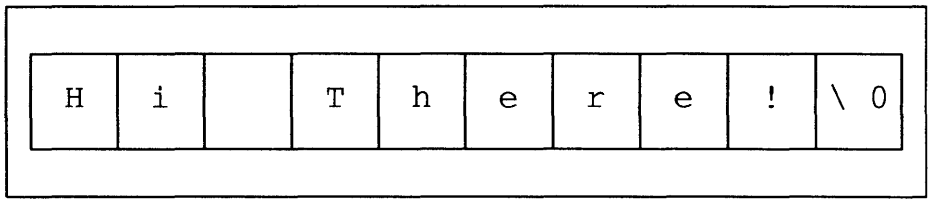
8.3. Functions for Handling Null-Terminated Strings

Null-terminated strings are arrays of characters. A correctly formed string has a zero (ASCII NUL) byte at the end to act as a terminator. All string handling routines and I/O routines conform to these semantics. C builds in this notion when a programmer writes a string constant — the compiler correctly adds the null byte at the end of the string. Suppose you have this declaration in your program:

```
char  greetings[] = "Hi There!";
```

Such a string appears in memory as:

Figure 8-1 *Layout of Null-Terminated String in Memory*



Functions described in this section operate on null-terminated strings. They do not check for overflow of any receiving string.

You must have the line:

```
#include <strings.h>
```

in any program unit that uses the functions described here.

Null Pointers versus Null Strings

On Sun workstations (and on most other machines), you *cannot* use a zero pointer to indicate a null string. Dereferencing a null pointer is an error and results in aborting the program. If you wish to indicate a null string, you must have a pointer that points to an explicit null string.

Programmers using `NULL` to represent an empty string should be aware that such programs work by coincidence rather than by intent and should be aware that testing for zero pointers is inherently nonportable.

`strlen()` — Find Length of String

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

`strlen()` returns the number of non-null characters in *s*.

`strcmp()` and `strncmp()` — Compare Strings

```
strcmp(string_1, string_2)
  char *string_1, *string_2;
```

```
strncmp(string_1, string_2, n)
  char *string_1, *string_2;
```

`strcmp()` compares its arguments and returns an integer greater than, equal to, or less than 0, according as *string_1* is lexicographically greater than, equal to, or less than *string_2*.

`strncmp()` makes the same comparison but looks at at most *n* characters.

`strcmp()` uses native character comparison, which is signed on Sun workstations.

`strcpy()` and `strncpy()` — Copy Strings

```
char *strcpy(string_1, string_2)
  char *string_1, *string_2;
```

```
char *strncpy(string_1, string_2, n)
  char *string_1, *string_2;
```

`strcpy()` copies string *string_2* to *string_1*, stopping after the null character has been moved. `strncpy()` copies exactly *n* characters, truncating or null-padding *string_2;* the target may not be null-terminated if the length of *string_2* is *n* or more. Both return *string_1*.

`strcat()` and `strncat()` — Concatenate Strings

```
char *strcat(string_1, string_2)
  char *string_1, *string_2;
```

```
char *strncat(string_1, string_2, n)
  char *string_1, *string_2;
```

`strcat()` appends a copy of string *string_2* to the end of string *string_1*.

`strncat()` copies *n* characters at most. Both return a pointer to the null-terminated result.

`index()` and `rindex()` — Find Character in String

`index()` returns a pointer to the *first* occurrence of character *c* in string *s*, or zero if *c* does not occur in the string.

`rindex()` returns a pointer to the *last* occurrence of character *c* in string *s*, or zero if *c* does not occur in the string.

```
char *index(s, c)
  char *s, c;
```

```
char *rindex(s, c)
  char *s, c;
```

8.4. Byte String and Bit String Functions

Functions described in this section operate on byte strings and bit strings. They do *not* recognize null-terminated strings as do the functions described in Section 8.3 .

`bcmp()` — Compare Byte Strings

```
bcmp(b1, b2, length)
    char *b1, *b2;
    int length;
```

`bcmp()` compares byte string *b1* against byte string *b2*, returning zero if they are identical, nonzero otherwise. Both strings are assumed to be *length* bytes long.

`bcopy()` — Copy Byte Strings

```
bcopy(b1, b2, length)
    char *b1, *b2;
    int length;
```

`bcopy()` copies *length* bytes, in left-to-right order, from string *b1* to string *b2*. Overlapping strings are handled correctly.

Note: The order of arguments is backwards from that of `strcpy()` — that is, `bcopy()` copies from its first argument to its second argument, while `strcpy()` copies from its second argument to its first argument.

`bzero()` — Clear Byte String to Zero

```
bzero(b, length)
    char *b;
    int length;
```

`bzero()` places *length* 0 bytes in the string *b*.

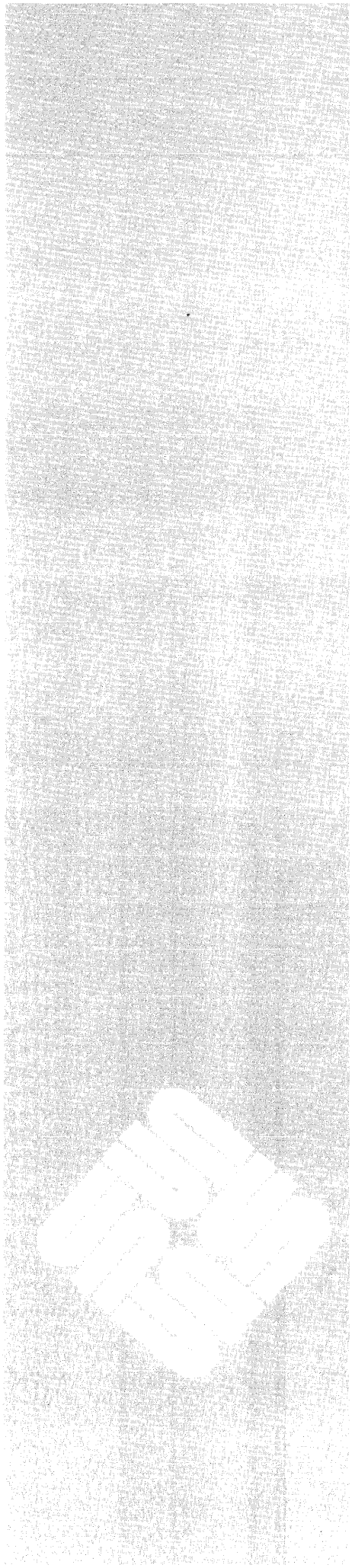
`ffs()` — Find First Bit Set

```
ffs(i)
    int i;
```

`ffs()` finds the first bit set in the argument passed it and returns the index of that bit. Bits are numbered starting at 1 from the right. A return value of –1 indicates the value passed is zero.

A

Low-Level File I/O



A

Low-Level File I/O

This appendix describes the bottom level of I/O on the SunOS system. The lowest level of I/O in SunOS provides no buffering or any other services except moving data; it is in fact a direct entry into the operating system. You are entirely on your own, but on the other hand, you have the most control over what happens. And since the calls and usage are quite simple, this isn't as bad as it sounds.

A.1. File Descriptors

In the SunOS operating system, all input and output is done by reading or writing files, because all peripheral devices, even the user's terminal, are files in the file system. This means that a single, homogeneous interface handles all communication between a program and peripheral devices.

In the most general case, before reading or writing a file, it is necessary to inform the system of your intent to do so, a process called 'opening' the file. If you are going to write on a file, it may also be necessary to create it. The system checks your right to do so — does the file exist? Do you have permission to access it? And, if all is well, returns a small positive integer called a *file descriptor*. Whenever I/O is to be done on the file, the file descriptor is used instead of the name to identify the file. This is roughly analogous to the use of `READ(5,...)` and `WRITE(6,...)` in FORTRAN. All information about an open file is maintained by the system; the user program refers to the file only by the file descriptor.

File pointers are similar in spirit to file descriptors, but file descriptors are more fundamental. A file pointer is a pointer to a structure that contains, among other things, the file descriptor for the file in question.

Since input and output involving the user's terminal are so common, special arrangements exist to make this convenient. When the command interpreter (the 'shell') runs a program, it opens three files, with file descriptors 0, 1, and 2, called standard input, standard output, and standard error output. All of these are normally connected to the terminal, so if a program reads file descriptor 0 and writes file descriptors 1 and 2, it can do terminal I/O without opening the files.

If I/O is redirected to and from files with < and >, as in

```
tutorial% prog < infile > outfile
```

the shell changes the default assignments for file descriptors 0 and 1 from the terminal to the named files. Similar observations hold if the input or output is associated with a pipe. Normally file descriptor 2 remains attached to the terminal, so error messages can go there. In all cases, the file assignments are changed by the shell, not by the program. The program does not need to know where its input comes from nor where its output goes, so long as it uses file 0 for input and 1 and 2 for output.

A.2. `read()` and `write()`

All input and output is done by two functions called `read()` and `write()`. The first argument for both of these functions is a file descriptor. The second argument is a buffer in your program where the data is to come from or go to. The third argument is the number of bytes to be transferred. The calls are

```
n_read = read(fd, buf, n);
n_written = write(fd, buf, n);
```

Each call returns a byte count which is the number of bytes actually transferred. On reading, the number of bytes returned may be less than the number asked for, because fewer than **n** bytes remained to be read. When the file is a terminal, `read()` normally reads only up to the next newline, which is generally less than what was requested. A return value of zero bytes implies end of file, and `-1` indicates an error of some sort. For writing, the returned value is the number of bytes actually written; it is generally an error if this isn't equal to the number supposed to be written.

The number of bytes to be read or written is quite arbitrary. The two most common values are 1, which means one character at a time ('unbuffered'), and 1024, corresponding to the physical blocksize on many peripheral devices. This latter size will be most efficient, but even character-at-a-time I/O is not inordinately expensive.

Putting these facts together, we can write a simple program to copy its input to its output. This program will copy anything to anything, since the input and output can be redirected to any file or device.

```
#define BUFSIZE 1024

main()  /* copy input to output */
{
    char    buf[BUFSIZE];
    int n;

    while ((n = read(0, buf, BUFSIZE)) > 0)
        write(1, buf, n);
    exit(0);
}
```

If the file size is not a multiple of BUFSIZE, some read() will return a smaller number of bytes, and the next call to read() after that will return zero.

It is instructive to see how read() and write() can be used to construct higher-level routines like getchar(), putchar(), etc. For example, here is a version of getchar() which does unbuffered input.

```
#define CMASK   0xff    /* for making char's > 0 */

getchar()   /* unbuffered single character input */
{
    char c;

    return((read(0, &c, 1) > 0) ? c & CMASK : EOF);
}
```

c *must* be declared char, because read() requires a character pointer. The character being returned must be masked with 0xff to ensure that it is positive; otherwise sign extension may make it negative. The constant 0xff is appropriate for Sun workstations but not necessarily for other machines.

The second version of getchar() does input in big chunks, and hands out the characters one at a time:

```
#define CMASK   0xff    /* for making char's > 0 */
#define BUFSIZE 1024

getchar()   /* buffered version */
{
    static char buf[BUFSIZE];
    static char *bufp = buf;
    static int  n = 0;

    if (n == 0) {   /* buffer is empty */
        n = read(0, buf, BUFSIZE);
        bufp = buf;
    }
    return((--n >= 0) ? *bufp++ & CMASK : EOF);
}
```

A.3. open(), creat(), close(), unlink()

Other than the default standard input, output and error files, you must explicitly open files in order to read or write them. There are two system entry points for this, open() and creat().

open() is rather like the fopen() discussed in the previous section, except that instead of returning a file pointer, it returns a file descriptor, which is just an int.

```
int fd;

fd = open(name, rwmode);
```

As with `fopen()`, the `name` argument is a character string corresponding to the external file name. The access mode argument is different, however: `rwmode` is 0 for read, 1 for write, and 2 for read and write access. `open()` returns `-1` if an error occurs; otherwise it returns a valid file descriptor.

It is an error to try to `open()` a file that does not exist. The entry point `creat()` is provided to create new files, or to rewrite old ones.

```
fd = creat(name, pmode);
```

returns a file descriptor if it could create the file called `name`, and `-1` if not. If the file already exists, `creat()` will truncate it to zero length; it is not an error to `creat()` a file that already exists.

If the file is new, `creat()` creates it with the *protection mode* specified by the `pmode` argument. In the SunOS file system, there are nine bits of protection information associated with a file, controlling read, write and execute permission for the owner of the file, for the owner's group, and for all others. Thus a three-digit octal number is most convenient for specifying the permissions. For example, 0755 specifies read, write and execute permission for the owner, and read and execute permission for the group and everyone else.

To illustrate, here is a simplified version of the SunOS utility *cp*, a program which copies one file to another. The main simplification is that our version copies only one file, and does not permit the second argument to be a directory:

```
#define NULL 0
#define BUFSIZE 1024
#define PMODE 0644 /* RW for owner, R for group & others */

error(s1, s2)   /* print error message and die */
char *s1, *s2;
{
    printf(s1, s2);
    printf("\n");
    exit(1);
}

main(argc, argv)    /* cp: copy f1 to f2 */
int argc;
char *argv[ ];
{
    int f1, f2, n;
    char    buf[BUFSIZE];

    if (argc != 3)
        error("Usage: cp from to", NULL);
    if ((f1 = open(argv[1], 0)) == -1)
        error("cp: can't open %s", argv[1]);
    if ((f2 = creat(argv[2], PMODE)) == -1)
        error("cp: can't create %s", argv[2]);

    while ((n = read(f1, buf, BUFSIZE)) > 0)
        if (write(f2, buf, n) != n)
            error("cp: write error", NULL);
    exit(0);
}
```

As noted above, there is a limit (typically 64) on the number of files which a program may have open simultaneously. Accordingly, any program which intends to process many files must be prepared to reuse file descriptors. The routine `close` breaks the connection between a file descriptor and an open file, and frees the file descriptor for use with some other file. Program termination through `exit` or return from the main program closes all files it had open.

The function `unlink(filename)` removes the file `filename` from the file system.

A.4. Random Access — `lseek()`

File I/O is normally sequential: each `read()` or `write()` takes place at a position in the file right after the previous one. When necessary, however, a file can be read or written in any arbitrary order. The system call `lseek()` provides a way to move around in a file without actually reading or writing:

```
lseek(fd, offset, origin);
```

forces the current position in the file whose descriptor is `fd` to move to position `offset`, which is taken relative to the location specified by `origin`. Subsequent reading or writing will begin at that position. `offset` is a `long`; `fd` and `origin` are `int`'s. `origin` can be 0, 1, or 2 to specify that `offset` is to be measured from the beginning, from the current position, or from the end of the file, respectively. For example, to append to a file, seek to the end before writing:

```
lseek(fd, 0L, 2);
```

To get back to the beginning ('rewind'),

```
lseek(fd, 0L, 0);
```

Notice the `0L` argument; it could also be written as `(long) 0`.

With `lseek()`, it is possible to treat files more or less like large arrays, at the price of slower access. For example, the following simple function reads any number of bytes from any arbitrary place in a file.

```
get(fd, pos, buf, n) /* read n bytes from position pos */
int fd, n;
long pos;
char *buf;
{
    lseek(fd, pos, 0);  /* get to pos */
    return(read(fd, buf, n));
}
```

A.5. Error Processing

The routines discussed in this section, and in fact all the routines which are direct entries into the system can incur errors. Usually they indicate an error by returning a value of −1. Sometimes it is nice to know what sort of error occurred; for this purpose all these routines, when appropriate, leave an error number in the external variable `errno`. The meanings of the various error numbers are listed in *intro*(2) in the Sun *System Interface Manual* so your program can, for example, determine if an attempt to open a file failed because it did not exist or because the user lacked permission to read it. Perhaps more commonly, you may want to display the reason for failure. The routine `perror` displays a message associated with the value of `errno`; more generally, `sys_errno` is an array of character strings which can be indexed by `errno` and displayed by your program.

B

Binary I/O

B

Binary I/O

The binary I/O facilities of the C library provide for record-oriented sequential access to files.

WARNING ***Using these routines may result in imcompatabilities when porting programs to or from some other machines. See the description of Sun's External Data Representation (XDR) standard for creating portable code as described in*** *Network Programming*

`fread()` — Read Data from File

The `fread()` function reads some number of objects into a block, from a specified file. The interface to `fread()` is:

```
fread(pointer, sizeof *pointer, items, stream)
    char *pointer;
    int  items;
    FILE *stream;
```

The arguments to `fread()` have the following meanings:

pointer is a pointer to a block of objects.
items is a count of the number of objects of a data type determined by the type of whatever "pointer" points to.
stream is the named input stream.

The value of the `fread()` function is the number of objects actually read.

`fwrite()` — Write Data to File

The `fwrite()` function writes some number of objects from a block, onto a specified file. The interface to `fwrite()` is:

```
fwrite (pointer, sizeof *pointer, items, stream)
    char *pointer;
    int  items;
    FILE *stream;
```

The arguments to `fwrite()` have the following meanings:

pointer is a pointer to a block of objects.

items is a count of the number of objects of a data type determined by the type of whatever "pointer" points to.

stream is the named output stream.

The value of the `fwrite()` function is the number of objects actually written to the named stream.

C

Memory Management

C

Memory Management

These routines provide a general-purpose memory allocation package. They maintain a table of free blocks for efficient allocation and coalescing of free storage. When there is no suitable space already free, the allocation routines call `sbrk` (see `brk`(2)) to get more memory from the system.

Each of the allocation routines returns a pointer to space suitably aligned for storage of any type of object. They return a null pointer if the request cannot be completed.

C.1. `malloc()` — Allocate Memory

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

allocates `num` bytes. The pointer returned is aligned so as to be usable for any purpose. `NULL` is returned if no space is available. The result of `malloc(0)` is undefined.

C.2. `free()` — Free Allocated Memory

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

`free()` frees up memory previously allocated by `malloc()`. Disorder can be expected if the pointer was not obtained from `malloc()`.

C.3. `calloc()` — Allocate Memory for C Objects

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

allocates space for `num` items, each of size `size`. The space is guaranteed to be set to 0 and the pointer is aligned so as to be usable for any purpose. `NULL` is returned if no space is available.

C.4. `cfree()` — Free Allocated Memory

```
(void) cfree(ptr, num, size)
    char *ptr;
    unsigned num;
    unsigned size;
```

Space is returned to the pool used by `calloc()`. Disorder can be expected if the pointer was not obtained from `calloc()`.

C.5. `realloc()` — Change Size of Allocated Block

`realloc()` changes the size of the block referenced by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes. For backwards compatibility, `realloc()` accepts a pointer to a block freed since the most recent call to `malloc()`, `calloc()`, `realloc()`, `valloc()`, or `memalign()`. Note that using `realloc()` with a block freed *before* the most recent call to `malloc()`, `calloc()`, `realloc()`, `valloc()`, or `memalign()` is an error.

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

C.6. `memalign()` — Allocate to Alignment Boundary

`memalign()` allocates *size* bytes on a specified alignment boundary, and returns a pointer to the allocated block. The value of the returned address is guaranteed to be an even multiple of *alignment*. Note that the value of *alignment* must be a power of two, and must be greater than or equal to the size of a word.

```
char *memalign(alignment, size)
    unsigned alignment;
    unsigned size;
```

`realloc()`, `valloc()`, and `memalign()` return NULL and set *errno* if arguments are invalid, or if there is insufficient available memory, or if the heap has been detectably corrupted, for example, by storing outside the bounds of a block.

C.7. `valloc()` — Allocate Memory on a Page Boundary

`valloc(size)` is equivalent to `memalign(getpagesize(), size)`.

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

`realloc()`, `valloc()`, and `memalign()` return NULL and set *errno* if arguments are invalid, or if there is insufficient available memory, or if the heap

has been detectably corrupted, for example, by storing outside the bounds of a block.

C.8. `alloca()` — Allocate Memory on Stack

`alloca()` allocates *size* bytes of space in the stack frame of the caller, and returns a pointer to the allocated block. This temporary space is automatically freed when the caller returns.

```
char *alloca(size)
    int size;
```

C.9. Memory Allocation Debugging

More detailed diagnostics can be made available to programs using the memory management routines described in this chapter by including a special relocatable object file at link time. This file also provides routines for control of error handling and diagnosis, as defined below. Note that these routines are *not* defined in the standard library.

`malloc_debug()` — Set Debug Level

```
int malloc_debug(level)
int level;
```

`malloc_debug()` sets the level of error diagnosis and reporting during subsequent calls to `malloc()`, `calloc()`, `realloc()`, `valloc()`, `memalign()`, `cfree()`, and `free()`. The value of *level* is interpreted as follows:

0 `malloc()`, `calloc()`, `realloc()`, `valloc()`, `memalign()`, `cfree()`, and `free()` behave the same as in the standard library.

1 `malloc()`, `calloc()`, `realloc()`, `valloc()`, `memalign()`, `cfree()`, and `free()` abort with a message to *stderr* if errors are detected in arguments or in the heap. If a bad block is encountered, its address and size are included in the message.

2 Same as level 1, except that the entire heap is examined on every call to `malloc()`, `calloc()`, `realloc()`, `valloc()`, `memalign()`, `cfree()`, and `free()`.

`malloc_debug()` returns the previous error diagnostic level. The default level is 1.

`malloc_verify()` — Check Storage Allocation Heap

```
int malloc_verify()
```

`malloc_verify()` attempts to determine if the heap has been corrupted. It scans all blocks in the heap (both free and allocated) looking for strange

addresses or absurd sizes, and also checks for inconsistencies in the free space table. `malloc_verify()` returns 1 if all checks pass without error, and otherwise returns 0. The checks can take a significant amount of time, so it should not be used indiscriminately.

C.10. Errors from Memory Management Routines

`malloc()`, `calloc()`, `realloc()`, `valloc()`, `memalign()`, `cfree()`, and `free()` set *errno* if:

`EINVAL` is true — an invalid argument was given. The value of *ptr* given to `free()`, `cfree()`, or `realloc()` must be a pointer to a block previously allocated by `malloc()`, `calloc()`, `realloc()`, `valloc()`, or `memalign()`. `EINVAL` is also true if the heap is found to have been corrupted. More detailed information may be obtained by enabling range checks using `malloc_debug()`.

`ENOMEM` is true — *size* bytes of memory could not be allocated.

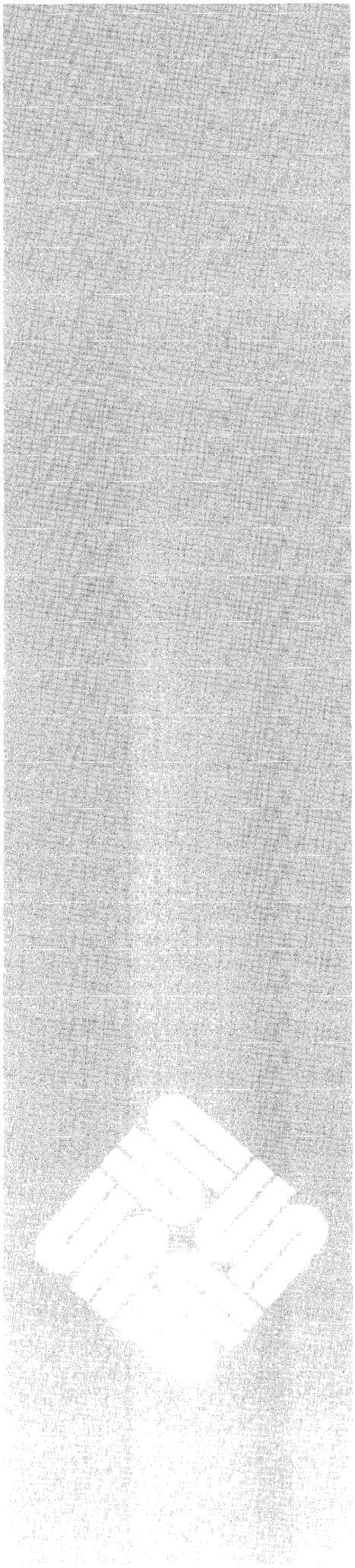
C.11. Notes on the Memory Management Routines

The file `/usr/lib/debug/malloc.o` contains the diagnostic versions of `malloc()`, `free()`, etc.

`alloca()` is both machine- and compiler-dependent; its use is strongly discouraged.

D

Sun-2, -3, and -4 Data Representations



D

Sun-2, -3, and -4 Data Representations

This appendix describes how Sun C represents data in storage and the mechanisms for passing arguments to functions. This chapter is intended as a guide to programmers who wish to write or use modules in languages other than C and have those modules interface to C code.

D.1. Storage Allocation

This section describes how storage is allocated to variables of various types.

In general, any *word* value is always aligned on a two-byte boundary. Anything larger than a word is also aligned on a two-byte boundary. Values that can fit into a single byte are aligned on a byte boundary.

Table D-1 *Storage Allocation for Data Types*

Data Type	*Internal Representation*
char elements	a single 8-bit byte.
short integers	one word (two bytes or 16 bits), aligned on a two-byte boundary.
int and long	32 bits (four bytes or two words), aligned on a two-byte boundary. On a Sun-4, they are aligned on 4-byte boundaries.
float	32 bits (four bytes or two words), aligned on a two-byte boundary. A `float` has a sign bit, 8-bit exponent and 23-bit fraction. On a Sun-4, they are aligned on 4-byte boundaries.
double	64 bits (eight bytes or four words), aligned on a word boundary. A `double` element has a sign bit, an 11-bit exponent and a 52-bit fraction. On a Sun-4, they are aligned on 8-byte boundaries.

D.2. Data Representations

Whatever the size of the data element in question, the most significant bit of the data element is always in the lowest numbered (leftmost) byte of however many bytes are required to represent that object. The tables below describe the various representations.

Integer Representations

There are three integer types used in Sun C: `short`, `int`, and `long`.

Table D-2 *Representation of* `short`

Bits	*Content*
8-15	Byte 0
0-7	Byte 1

Table D-3 *Representation of* `int` *and* `long`

Bits	*Content*
24-31	Byte 0
16-23	Byte 1
8-15	Byte 2
0-7	Byte 3

`float` and `double` Representation

`float` and `double` data elements are represented according to the ANSI IEEE 754-1985 standard. The tables below describe the representation.

Table D-4 `float` *Representation*

Bits	*Name*	*Content*
31	Sign	1 iff number is negative.
23-30	Exponent	Eight-bit exponent, biased by 127. Values of all zeros, and all ones, reserved.
0-22	Fraction	23-bit fraction component of normalized significand. The "one" bit is "hidden".

Table D-5 `double` *Representation*

Bits	*Name*	*Content*
63	Sign	1 iff number is negative.
52-62	Exponent	Eight-bit exponent, biased by 1023. Values of all zeros, and all ones, reserved.
0-51	Fraction	52-bit fraction component of normalized significand. The "one" bit is "hidden".

A `float` or `double` number is represented by the form:

$$(-1)^{Sign}\ 2^{(exponent-bias)}\ 1.f$$

where "1.f" is the significand and "f" is the bits in the significand fraction.

Extreme Number Representation

Table D-6 *Extreme Number Representation*

Number	*Description*
zero (signed)	is represented by an exponent of zero, and a fraction of zero.
subnormal numbers	are nonzero numbers with an exponent of zero. The form of a denormalized number is: $(-1)^{Sign}\ 2^{(exponent-bias+1)}\ 0.f$ where `f` is the bits in the fraction.
signed infinity	(that is, affine infinity) is represented by the largest value that the exponent can assume (all ones), and a zero fraction.
Not-a-Number (NaN)	is represented by the largest value that the exponent can assume (all ones), and a non-zero fraction. The sign is usually ignored.

Normalized `float` and `double` numbers are said to contain a "hidden" bit, providing for one more bit of precision than would otherwise be the case.

Hexadecimal Representation of Selected Numbers

Value	***float***	***double***
+0	00000000	0000000000000000
-0	80000000	8000000000000000
+1.0	3F800000	3FF0000000000000
-1.0	BF800000	BFF0000000000000
+2.0	40000000	4000000000000000
+3.0	40400000	4008000000000000
+Infinity	7F800000	7FF0000000000000
-Infinity	FF800000	FFF0000000000000
NaN	7F8xxxxx	7FFxxxxxxxxxxxx

Pointer Representation

A pointer in C occupies four bytes. The `NULL` value pointer is equal to zero.

Array Storage

Arrays are stored with their elements in a specific storage order. The elements are actually stored in a linear sequence of storage elements.

C arrays are stored in row major order, such that the last subscript in a multi-dimensional array varies fastest.

String data types are simply arrays of `char` elements.

Arithmetic Operations on Extreme Values

This subsection describes the results derived from applying the basic arithmetic operations to combinations of extreme and ordinary floating-point values.

No traps or any other exception actions are taken.

All inputs are assumed to be positive. Overflow, underflow, and cancellation are assumed not to happen. In all the tables below, the abbreviations have the following meanings:

Table D-7 *Extreme Values Usage*

Abbreviation	***Meaning***
`Num`	Subnormal or Normalized Number
`Inf`	Infinity (positive or negative)
`NaN`	Not a Number
`Uno`	Unordered

The tables that follow describe the types of values that result from arithmetic operations performed with combinations of different types of operands.

Table D-8 *Addition and Subtraction Results*

Addition and Subtraction				
Left Operand	Right Operand			
	0	Num	Inf	NaN
0	0	Num	Inf	NaN
Num	Num	Num	Inf	NaN
Inf	Inf	Inf	Note	NaN
NaN	NaN	NaN	NaN	NaN

Note:" Inf + Inf = Inf; Inf - Inf = NaN

Table D-9 *Multiplication Results*

Multiplication				
Left Operand	Right Operand			
	0	Num	Inf	NaN
0	0	0	NaN	NaN
Num	0	Num	Inf	NaN
Inf	NaN	Inf	Inf	NaN
NaN	NaN	NaN	NaN	NaN

Table D-10 *Division Results*

Division				
Left Operand	Right Operand			
	0	Num	Inf	NaN
0	NaN	0	0	NaN
Num	Inf	Num	0	NaN
Inf	Inf	Inf	NaN	NaN
NaN	NaN	NaN	NaN	NaN

Table D-11 *Comparison Results*

Comparison				
Left Operand	Right Operand			
	0	Num	Inf	NaN
0	=	<	<	Uno
Num	>		<	Uno
Inf	>	>		Uno
NaN	Uno	Uno	Uno	Uno

Note: NaN compared with NaN is Unordered, and also results in inequality. +0 compares equal to -0.

D.3. Argument Passing Mechanism

This section describes how arguments are passed in Sun C.

All arguments to C functions are passed by value.

Actual arguments are pushed onto the stack in the reverse order from which they are declared in a function declaration.

Actual arguments which are expressions are evaluated before the function reference. The result of the expression is then pushed onto the stack.

Functions return their results in register `D0`, or in registers `D0` and `D1` when the result is a `float` or `double` value.

All arguments, except `doubles`, are passed as four-byte values; a `double` is passed as an eight-byte value. All `float` values are passed as `doubles`.

Upon return from a function, it is the responsibility of the caller to pop arguments from the stack.

D.4. Referencing Data Objects in C

This section describes how variables of different types are actually accessed (or referenced). The method and notations of access, of course, differ depending on whether the object is a simple variable, an array, a structure, or a union.

Referencing Simple Variables

A plain variable (of simple scalar type) is acessed by its identifer. Since such a simple variable has no structure, its identifier alone is enough to reference it.

Figure D-1 *Examples of Simple Variable References*

```
                    /*  Declare some simple variables  */
int  egress;
float  lightly;
char  coal;
extern double sin();

                   /*  Now reference those variables  */
egress = 10;       /*  Set the int to a constant  */

printf ("%f", sin (lightly));  /*  Pass it as argument  */

putc (coal);       /*  Write it to the standard output  */
```

Referencing With Pointers

A variable can also be declared as a pointer to another object. In this case, the reference to the object must be done with the pointer notation. Placing an asterisk character `*` in front of an identifier uses that identifier as a pointer to an object, and the thing that is read from or written to is the object that the identifier points to.

Figure D-2 *Examples of Pointer References*

```
                    /*  Declare some pointer variables  */
int  *egress;
float *lightly;
char  *coal;
extern double sin();

                    /*  Now reference those variables  */
*egress = 10;       /*  Set it to a constant  */

printf ("%f", sin (*lightly));  /*  Pass it as argument */

putc (*coal);       /*  Write it to the standard output  */
```

Referencing Array Elements

When an identifier of an array type appears in an expression, the identifier is converted to a pointer to the first member of the array.

The subscript operation `[]` is interpreted such that

```
E1 [E2]
```

is equivalent to the construct

```
*((E1) + (E2))
```

Figure D-3 *Examples of Array Variable References*

```
                /*  Declare some array variables  */
int  egress[10];
float  lightly[5][5];
char  coal[100];
extern double sin();

               /*  Now reference those variables  */
for (idx = 0;  idx < 10;  idx++)
    egress [idx]  = 10;      /*  Set int to a constant  */

for (idx = 0;  idx < 5;  idx++)
    for (idy = 0;  idy < 5;  idy++)
        printf ("%f", sin (lightly[idx][idy]));

for (idx = 0;  idx < 100;  idx++)
    putc (coal[idx]);      /*  Write to standard output  */
```

Referencing Structures and Unions

There are only two operations which may be done on a structure or a union:

1. A member of the structure or union can be referenced by means of the . or -> operator,
2. The address of the entire structure or union can be taken, with the & operator.
3. One structure can be copied to another of the same type.

The . operator is used in contexts where the structure or union identifier is available directly to the expression. The -> operator is used when the identifier for the structure or union is a pointer to the object.

Figure D-4 *Examples of Accessing Members of Structures*

```
demo (wanted)
    char  *wanted;
{
                    /*  Declare a couple of structures  */
struct  {    /*  This one is fairly simple  */
    int  level;
    char  *cp;
    char  pbuffer[MAXLEN];
    }  putter;

struct vallist {  /*  This one is a linked list  */
    char  *name;
    char  valtype;
    int  value;
    struct vallist *nextval;
    }  *valhead, *valtail;

struct vallist  *pointer;
                /*  Now access the members  */
    putter.level = 10;
    for (i = 0;  i < MAXLEN;  i++)
        putter.pbuffer [i] = *putter.cp;

                /*  Access members through pointers  */
    for (pointer = valhead;
              pointer != NULL;
                 pointer = pointer->nextval)
        if (strcmp (pointer->name, wanted) == 0)
            return (pointer);

}     /*  End of the demo function  */
```

E

Sun386i Data Representation

E

Sun386i Data Representation

This appendix describes how Sun C represents data in storage and the mechanisms for passing arguments to functions on the Sun386i. This chapter is intended as a guide to programmers who wish to write or use modules in languages other than C and have those modules interface to C code.

E.1. Storage Allocation

This section describes how storage is allocated to variables of various types.

The Sun386i C compiler aligns data on *natural boundaries*. This means that bytes are aligned on byte boundaries, words (16 bits) on word boundaries, and doublewords on doubleword boundaries. Anything larger than a doubleword (32 bits) is also aligned on a doubleword boundary. In bit fields, data are aligned beginning at the *least signigicant bit* of the word.

Table E-1 *Storage Allocation for Data Types*

Data Type	*Internal Representation*
char elements	a single 8-bit byte.
short integers	one word (two bytes or 16 bits), aligned on a two-byte boundary.
int and long	32 bits (four bytes or two words), aligned on a doubleword boundary.
float	32 bits (four bytes or two words), aligned on a doubleword boundary. A `float` has a sign bit, 8-bit exponent and 23-bit mantissa.
double	64 bits (eight bytes or four words), aligned on a doubleword boundary. A `double` element has a sign bit, an 11-bit exponent and a 52-bit mantissa.

Note that the Sun386i alignment scheme differs from the Sun-3 scheme, in which characters are aligned on byte boundaries and everything else, regardless of size, is aligned on word boundaries. Consequently, reading with one type of system from a disk or over the network data created by the other type can cause errors because of the different alignment schemes. See the *Sun386i Developer's Guide* for further discussion of this topic.

E.2. Data Representations

On the Sun386i, whatever the size of the data element in question, the *least significant* bit of the data element is always the lowest numbered (rightmost) byte of however many bytes are required to represent that object. The tables below describe the various representations.

Integer Representations

There are three integer types used in Sun C: `short`, `int`, and `long`.

Table E-2 *Representation of* `short`

Bits	*Content*
8-15	n+1
0-7	n

Table E-3 *Representation of* `int`

Bits	*Content*
24-31	n+3
16-23	n+2
8-15	n+1
0-7	n

Table E-4 *Representation of* `long`

Bits	*Content*
16-31	n+2
0-15	n

`float` and `double` Representation

A `float` or `double` number is represented by the form

$$(-1)^{Sign}\ 2^{(exponent-bias)}\ 1.f$$

according to the ANSI IEEE 754-1985 standard. In the tables below,

s = sign (1 bit)

e = biased exponent (11bits)

f = fraction (23 bits)

u = unsigned

Table E-5 `float` *Representation*

Bits	*Name*	*Content*
31	Sign	1 iff number is negative.
23-30	Biased Exponent	Eight-bit exponent, biased by 127. Values of all zeros, and all ones, reserved.
0-22	Fraction	23-bit fraction component of normalized significand. The "one" bit is "hidden".

Table E-6 `double` *Representation*

Bits	*Address*	*Content*
63	n+4	Sign
55-62	n+4	Exponent
32-54	n+4	Significand fraction - msb
0-31	n	Significand fraction - lsb

where "1.f" is the significand and "f" is the bits in the significand fraction.

Extreme Number Representation

Table E-7 *Extreme Number Representation*

Number	*Description*
zero (signed)	is represented by an exponent of zero, and a fraction of zero.
subnormal numbers	are nonzero numbers with an exponent of zero. The form of a denormalized number is: $(-1)^{Sign}\ 2^{(exponent-bias+1)}\ 0.f$ where `f` is the bits in the fraction.
signed infinity	(that is, affine infinity) is represented by the largest value that the exponent can assume (all ones), and a zero fraction.
Not-a-Number (NaN)	is represented by the largest value that the exponent can assume (all ones), and a non-zero fraction. The sign is usually ignored.

Normalized `float` and `double` numbers are said to contain a "hidden" bit, providing for one more bit of precision than would otherwise be the case.

Table E-8 *Extreme* `float` *Representations*

normalized number (0<e<255):	$(-1)^{Sign}\ 2^{(exponent-127)}\ 1.f$
denormalized number (e=0, f!=0):	$(-1)^{Sign}\ 2^{(exponent-126)}\ 1.f$
zero (e=0, f=0):	$(-1)^{Sign}\ 0$
signaling NaN Quiet Nan Infinity	s=u, e=255(max); f=.0uuu-uu (at least one bit must be nonzero) s=u, e=255(max); f=.1uuu-uu s=u, e=255(max); f=.0000-00 (all zeroes)

Table E-9 *Extreme* `double` *Representations*

normalized number (0<e<2047):	$(-1)^{Sign}\ 2^{(exponent-1023)}\ 1.f$
denormalized number (e=0, f!=0):	$(-1)^{Sign}\ 2^{(exponent-1022)}\ 1.f$
zero (e=0, f=0):	$(-1)^{Sign}\ 0$
signaling NaN Quiet Nan Infinity	s=u, e=2047(max); f=.0uuu-uu (at least one bit must be nonzero) s=u, e=2047(max); f=.1uuu-uu s=u, e=2047(max); f=.0000-00 (all zeroes)

Other Extreme Representations

A signaling NaN is a value where the sign bit is undefined, the exponent is 255 or less for `float` data and 1023 or less for and`double`data, significand is of the form f = .0uuu-uu (at least one bit must be nonzero).

A quiet NaN is a value where the sign bit is undefined, the exponent is 255 or less for `float` data and 1023 or less for `double` data, and the fractional part of the significand is of the form f = .1uuu-uu.

An infinity is represented by a value where the sign bit is undefined, the exponent is 255 or less for `float` data and 1023 or less for `double` data, and the fractional part of the significand is of the form f = .0000-00 (all zeros).

Hexadecimal Representation of Selected Numbers

Value	*float*	*double*
+0	00000000	0000000000000000
-0	80000000	8000000000000000
+1.0	3F800000	3FF0000000000000
-1.0	BF800000	BFF0000000000000
+2.0	40000000	4000000000000000
+3.0	40400000	4008000000000000

Table E-9 *Extreme* `double` *Representations— Continued*

Value	*float*	*double*
+Infinity	7F800000	7FF0000000000000
-Infinity	FF800000	FFF0000000000000
NaN	7F8xxxxx	7FFxxxxxxxxxxxx

Pointer Representation

A pointer in C occupies four bytes. The `NULL` value pointer is equal to zero.

Array Storage

Arrays are stored with their elements in a specific storage order. The elements are actually stored in a linear sequence of storage elements.

C arrays are stored in row major order, such that the last subscript in a multi-dimensional array varies fastest.

String data types are simply arrays of `char` elements.

Arithmetic Operations on Extreme Values

For information on arithmetic operations, see the *80387 Programmer's Reference Manual* from Intel. See also IEEE Standard 754.

E.3. Argument Passing Mechanism

This section describes how arguments are passed in Sun C.

All arguments to C functions are passed by value.

Actual arguments are pushed onto the stack in the reverse order from which they are declared in a function declaration.

Actual arguments which are expressions are evaluated before the function reference. The result of the expression is then pushed onto the stack.

On the Sun386i, integer functions return their results in register `eax`. Floating point functions return their results on the top of the `FP` stack (register `st(0)`).

All arguments, except `doubles`, are passed as four-byte values; a `double` is passed as an eight-byte value. All `float` values are passed as `doubles`.

Upon return from a function, it is the responsibility of the caller to pop arguments from the stack.

E.4. Referencing Data Objects in C

This section describes how variables of different types are actually accessed (or referenced). The method and notations of access, of course, differ depending on whether the object is a simple variable, an array, a structure, or a union.

Referencing Simple Variables

A plain variable (of simple scalar type) is acessed by its identifer. Since such a simple variable has no structure, its identifier alone is enough to reference it.

Figure E-1 *Examples of Simple Variable References*

```
                    /*  Declare some simple variables  */
double sin();
int  egress;
float  lightly;
char  coal;

                    /*  Now reference those variables  */
egress = 10;        /*  Set it to a constant  */

printf ("%f", sin (lightly));  /*  Pass it as argument  */

putc (coal);        /*  Write it to the standard output  */
```

Referencing With Pointers

A variable can also be declared as a pointer to another object. In this case, the reference to the object must be done with the pointer notation. Placing an asterisk character * in front of an identifier uses that identifier as a pointer to an object, and the thing that is read from or written to is the object that the identifier points to.

Figure E-2 *Examples of Pointer References*

```
                    /*  Declare some pointer variables  */
double sin();
int  *egress;
float *lightly;
char  *coal;

                    /*  Now reference those variables  */
*egress = 10;       /*  Set it to a constant  */

printf ("%f", sin (*lightly));  /*  Pass it as argument */

putc (*coal);       /*  Write it to the standard output  */
```

Referencing Array Elements

When an identifier of an array type appears in an expression, the identifier is converted to a pointer to the first member of the array.

The subscript operation [] is interpreted such that

```
E1 [E2]
```

is equivalent to the construct

```
*((E1) + (E2))
```

Figure E-3 *Examples of Array Variable References*

```
                /*  Declare some array variables  */
double sin();
int  egress[10];
float  lightly[5][5];
char  coal[100];

                /*  Now reference those variables  */
for (idx = 0;  idx < 10;  idx++)
    egress [idx]  = 10;     /*  Set it to a constant  */

for (idx = 0;  idx < 5;  idx++)
    for (idy = 0;  idy < 5;  idy++)
        printf ("%f", sin (lightly[idx][idy]));

for (idx = 0;  idx < 100;  idx++)
    putc (coal[idx]);     /*  Write to standard output  */
```

Referencing Structures and Unions

There are only three operations which may be done on a structure or a union:

1. A member of the structure or union can be referenced by means of the . or -> operator,
2. The address of the entire structure or union can be taken, with the & operator.
3. One structure may be copied to another of the same type.

The . operator is used in contexts where the structure or union identifier is available directly to the expression. The -> operator is used when the identifier for the structure or union is a pointer to the object. Structures can also be passed as parameters, returned from functions, or assigned to variables of the same structure or union type.

Figure E-4 *Examples of Accessing Members of Structures*

```
demo (wanted)
    char  *wanted;
{
                    /*  Declare a couple of structures  */
struct  {    /*  This one is fairly simple  */
    int  level;
    char  *cp;
    char  pbuffer[MAXLEN];
    }  putter;

struct vallist {  /*  This one is a linked list  */
    char  *name;
    char  valtype;
    int  value;
    struct vallist *nextval;
    }  *valhead, valtail;

struct vallist  *pointer;
                /*  Now access the members  */
    putter.level = 10;
    for (i = 0;  i < MAXLEN;  i++)
        putter.pbuffer [i] = *putter.cp;

                /*  Access members through pointers  */
    for (pointer = valhead;
            pointer != NULL;
               pointer = pointer->nextval)
        if (strcmp (pointer->name, wanted) == 0)
            return (pointer);

}     /*  End of the demo function  */
```

Index

Notes

Notes

Notes

Notes

Notes

Notes

Notes